PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

ALGEBRAIC META-LEVEL PROGRAMMING IN PROLOG

Georges Louis (glouis@prlb2.UUCP)
Marc Vauclair (mvauclair@prlb2.UUCP)
Philips Research Laboratory Brussels
Avenue E. 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 specialised 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.

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, suitable for the task at hand, e.g. debugging or 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.

When compared to direct execution, meta-interpretation entails a loss of efficiency. First, 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-level program itself, and will not take advantage of its peculiarities.

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 [Sestoft and Søndergaard 1988].

Partial evaluation is a general program transformation technique whose benefits are well established [Sestoft and Søndergaard 1988].

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' (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-level program analysis, which is performed during the partial evaluation process and not any more during execution by the meta-interpreter.

In this paper, we introduce a more direct approach where program transformation is specified explicitly 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 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. 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.

The rest of this paper is organised 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 Section 2. Sections 4 to 6 present a number of examples to illustrate several aspects of algebraic meta-programming. More precisely, in Section 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 MANY-SORTED WORD ALGEBRAS

The use of concepts of universal algebra to specify compilers is not new [Burstall and Landin 1969]. The application of these concepts to program structuring, although apparently little known, is presented in [Burstall 1980]. Algebraic concepts are also used to define language semantics [Goguen et al. 1977].

The basic concepts of universal algebra needed in the sequel will now be recalled.

To compare different algebras, it is convenient to define a *many-sorted signature* Σ as a collection of sorts, $S = \{s_1, s_2, ...\}$, and a set, the *operator domain* Ω, with a mapping $a : \Omega \rightarrow S^* \times S$. The elements of Ω are called *operators* (more precisely *operator symbols*). For $\omega \in \Omega$, $a(\omega)$ is the *(generalised) arity* of ω; it indicates the sorts of the arguments of ω and the sort of its result.

A many-sorted Σ–algebra (for an introduction see [Burstall and Goguen 1982]), A_Σ (A for short), is an S-indexed family (the *carrier* of the algebra) of sets (denoted A_s with s in S) with a collection of functions in correspondence with the operators of Ω: for each $\omega \in \Omega$, the algebra has a corresponding operation ω_A. If ω has arity $([s_1, s_2, ...], s)$, ω_A maps $A_{s_1} \times A_{s_2} \times ...$ to A_s.

An homomorphism, h, between S-sorted Σ–algebras A_Σ and B_Σ is an S-indexed family of functions $h = \{h_s : A_s \rightarrow B_s | s \in S\}$ such that $h_s(\omega_A(a_1, \ldots, a_n)) = \omega_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ $(\omega \in \Omega, a(\omega) = ([s_1, ..., s_n], s))$.

Given a signature, Σ, and an S-indexed family of sets, X (with $\cup_s X_s \cap \Omega = \phi$), whose elements are called *generators*, the many-sorted *Σ–word algebra on* X, $W_\Sigma(X)$ (also named the *Σ–algebra freely generated by* X), is the set of syntactically correct terms that can be constructed using generators in X and operators in Ω. Given an operator ω of arity $([s_1, ..., s_n], s)$ in Ω, and n terms $T_1, \ldots, T_n$, in $W_\Sigma(X)$, of respective sorts $s_1, ..., s_n$, the term $\omega(T_1, \ldots, T_n)$ is taken as the result, of sort s, of the application of ω 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 $W_\Sigma(X)$ is uniquely analysable as the application of an operator (the *principal operator* or *principal functor* of the term) to subterms. This precludes operator overloading.

Many-sorted Σ–word algebras enjoy the following *universal* property: given a many-sorted Σ–algebra A_Σ and a S-indexed family of sets of generators, $X = \{X_s | s \in S\}$, any S-indexed family of functions $f = \{f_s : X_s \rightarrow A_s | s \in S\}$ has a unique homomorphic extension $f^* = \{f_s^* : W_\Sigma(X)_s \rightarrow A_s | s \in S\}$. f_s^* is defined by induction on the structure of terms: if term t is a generator of sort s in X_s, then $f_s^*(t) = f_s(t)$[1], otherwise, t is of the form $\omega(T_1, \ldots, T_n)$ for some ω of arity $([s_1, ..., s_n], s)$ and $f_s^*(t) = \omega_A(f_{s_1}^*(T_1), \ldots, f_{s_n}^*(T_n))$.

Thus, if a set of finite trees can be represented as $W_\Sigma(X)$ by an appropriate choice of Σ and X, a function over these trees will be specified by:

- a family of functions $(f_s, s \in S)$ for the computations at the leaves;
- the operators of a Σ-algebra for the computations at the non-leaf nodes;
- the carriers of the Σ-algebra for the ranges of the values of the function being specified.

The universal property then uniquely defines the functions over trees. For example, take Ω to contain only operator + of arity $([int, int], int)$, X_{int} to be the set of integer numerals, A_Σ to be the integers with addition corresponding to +, and f to be the function yielding the value of a numeral. $W_\Sigma(X)$ will then be the set of additive expressions, and f_{int}^* the function yielding the value of such expressions.

For language semantics in general, and transformational meta-programming in particular, the language sentences will be viewed as words in $W_\Sigma(X)$. Some language constructs (e.g. the atomic goals in Prolog) will be selected for X, while others (e.g. logical connectives in Prolog) will form Ω. The specification of semantics –or of the target of the transformation– amounts to the definition of a Σ-algebra, and a (family of) mappings from X into it.

If the language syntax is specified by a context-free grammar, the simplest approach is to define operators in the signature for the syntactic rules in the grammar,

[1] To be formal, one should not identify t as an element of X_s with t as an element of $W_\Sigma(X)_s$, and use $\eta_s : X_s \rightarrow W_\Sigma(X)_s$ as an injection: $f_s^*(\eta_s(t)) = f_s(t)$. This remark is not pure pedantry: it is sometimes convenient in a program to represent x in X_s differently from x in $W_\Sigma(X)_s$, and functions η_s are then needed. In this paper, we attempt to keep such formal details to a minimum, sometimes at the price of full rigour. See [Burstall and Goguen 1982] for a formal treatment.

although it is often preferred to take an abstract syntactic form as the basis of the definition. 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

$$(\Sigma, \mathbf{X}, A_\Sigma, f = \{f_s : \mathbf{X}_s \to A_s | s \in S\}).$$

However, in general, the language to be given semantics together with the range of applications determine the choice of the signature and of the family of generators, and in the sequel an (algebraic) translation scheme is simply a pair (f, A_Σ) when Σ and $\mathbf{X}$ are understood. For Prolog, such a scheme will map constructs viewed as words in $W_\Sigma(\mathbf{X})$ into other constructs viewed as elements of A_Σ, thus specifying a translation from Prolog to Prolog.

3 ALGEBRAIC META-LEVEL PROGRAMMING IN PROLOG

In this section, the framework of Section 2 will be applied to meta-programming in Prolog. There are two aspects to it: the implementation of algebraic translation schemes in Prolog itself, and the definition of translation schemes applicable to object-level Prolog programs.

Once a signature and a set of generators has been selected, an algebraic translation scheme can be implemented by a Prolog program by defining:

- `'ω_A'(+V1,+V2,...,+Vn,-V)`[2] to implement $V = \omega_A(V_1, V_2, ..., V_n)$, for each ω in the signature[3].
- `f(+S,+G,-V)` to implement $V = f_s(G)$ for $G \in \mathbf{X}_s$ and $s \in S$.

A final predicate, `val(+S,+W,-V)` is needed to implement $V = f_s^*(W_s)$, i.e. the semantic function proper. The clauses defining `val` are constructed from the signature only: there is one clause for each operator ω in Ω, which defines the value of terms whose principal functor is ω, e.g. if the signature contains `+/([int,int],int)`, there will be a clause:

```
val(int,W1+W2,V) :- !,
     val(int,W1,V1),
     val(int,W2,V2),
     '+_A'(V1,V2,V).
```

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

```
val(Sort,G,V) :- f(Sort,G,V).
```

[2]Where the notation `p(+P,-Q)` indicates that upon call, the argument for `P` is known (not an uninstantiated variable), i.e `P` is an input parameter. Conversely, `Q` is an output parameter, i.e. its argument is an uninstantiated variable.

[3]There is a not yet fully explored opportunity to handle operator overloading by the addition of an extra argument to `'ω_A'`: `'ω_A'([s1,...sn],s),+V1,+V2,...,+Vn,-V)` if $\alpha(\omega) = ([s_1, ..., s_n], s)$.

The construction of the clauses for `val` from Σ 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 `'ω_A'` of arity $n+1$ for each ω in Ω, and `f` of arity 3 for the f_s.

Remark, that neither the carriers nor the generators have an explicit representation in the meta-program itself, however, predicate `f` should succeed when its second argument is a generator.

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

```
f(int,V,V).
'+_A'(V1,V2,V) :- V 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 $\mathbf{X}$ 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, the correspondence with the theoretical framework is straightforward and the signature will consist of operators used in the syntax.

As the examples in the next sections will show, the translation is most often performed clause by clause. Observe that facts as syntactic sugar for clauses whose bodies reduce to `true` will not be handled correctly by a straightforward translation scheme, and should be submitted to `val` in their extended form, i.e. as clauses with a `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, e.g.:

```
concat([],L,L) :- true
&
concat([H|T],L,[H|TL]) :- 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 `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 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 Ω where:

$S = \{h, b, r\}$
$\Omega = \{$':-'/([h, b], r),','/([b, b], b),...$\}$.

This signature will be used throughout the rest of the paper.

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. Definition of `val` is given by:

```
val(b,(G1,G2),V) :- !,
     val(b,G1,VG1),
     val(b,G2,VG2),
     ',_A'(VG1,VG2,V).
val(r,(H :- Body),V) :- !,
     val(h,H,VH),
     val(b,Body,VB),
     ':-_A'(VH,VB,V).
val(Sort,G,V):- f(Sort,G,V).
```

There will be definitions of `f(h,...)` and `f(b,...)` for 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 target program consists of a collection of the translations of all the clauses in the source 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 parts of the target program under construction. This means that carrier A_r for clauses should consist of clauses, while the carrier A_b can be arbitrarily selected.

Further sections will present specific illustrative examples. As usual, we select simplified examples, for the sake of clarity and brevity. We mostly restrict ourselves to programs which consist of pure Horn clauses.

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

Each program clause of the form

$p(\mathcal{T}_1, \mathcal{T}_2, ...) :- \mathcal{B}$

should be translated into

$p(\mathcal{X}_1, \mathcal{X}_2, ...) :-$
$\quad$ `unify(`$p(\mathcal{X}_1, \mathcal{X}_2, ...), p(\mathcal{T}_1, \mathcal{T}_2, ...)), \mathcal{B}$

where the $\mathcal{X}_i$'s are new variables not occurring in the original clause. Thus, the clause defining the concatenation of difference lists:

```
concat(X-Y,Y-Z,X-Z) :- true.
```

would be translated as:

```
concat(V1,V2,V3) :-
 unify(concat(V1,V2,V3),
       concat(X-Y,Y-Z,X-Z)),
 true.
```

The code for the translation consists of the definition of predicates `',_NSU'` (NSU for 'non-standard unification' is the name of the semantic algebra), `':-_NSU'`, and `f`:

```
',_NSU'(Goal1,Goal2,(Goal1,Goal2)).
':-_NSU'(Head,
         Body,
         (Template :-
            unify(Template,Head),Body)) :-
  functor(Head,Name,Arity),
  functor(Template,Name,Arity).
f(_,Goal,Goal).
```

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

```
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

```
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` is applied later instead.

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 run-time, each goal of the form $p(T_1, T_2, \ldots)$ should be translated into

```
clause(X_H, X_B),
unify(X_H, p(T_1, T_2, ...)),
call(X_B)
```

where X_H and X_B are new variables.
This translation applies to atomic goals in bodies and not to heads. The clause defining `f` in the previous translation scheme, should be replaced by:

```
f(h,Head, Head).
f(b,Goal,(clause(Head,Body),
          unify(Head,Goal),
          call(Body))).
```

and `':-_NSU'` should just reconstruct a clause:

```
':-_NSU'(Head,Body,(Head:-Body)).
```

The general observation is that simple meta-interpreters can 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 loss which can be recovered at the price of some complexity by introducing filter code to benefit from static cases. Conversely, simple translation schemes produce reasonably efficient programs but must be made more complex to apply to 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 define `f` to translate cuts by cuts.
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. Start from a program, $\mathcal{P}$, to which a query $\mathcal{Q} = p(T_1, T_2, \ldots)$ is submitted. The intention is to arrive at a new program, call it $\mathcal{P}_{\mathrm{pl}}$, which is equivalent to $\mathcal{P}$ (it delivers the same answers), but computes the length of each proof necessary to arrive at each answer. The queries for $\mathcal{P}_{\mathrm{pl}}$ will accommodate a new variable for the proof length.
A first possibility is to modify $\mathcal{Q}$ as follows: $\mathcal{Q}_{\mathrm{pl}} = p(T_1, T_2, \ldots)$ `?` X_{pl} with a suitable operator definition for '`?`'.

Each atomic goal in $\mathcal{P}$ will be modified to construct $\mathcal{P}_{\mathrm{pl}}$: $f(\mathcal{G}) = \mathcal{G}$ `? Pl.` The semantic domains PL_b and PL_h (PL for 'proof length' is the name of the semantic algebra) consists of Prolog terms of the form:

```
lambda([Count],G)
```

where `Count` is an arithmetic expression yielding the length of the proof of $\mathcal{G}$.
Function $,_{PL}$ combines two such terms as follows:

```
lambda([C_1],G_1) ,_PL lambda([C_2],G_2)
 = lambda([C_1+C_2],(G_1,G_2)).
```

The choice of `lambda` as functor name is justified since this construct indeed resembles λ-expressions. β-reduction can be defined as follows:

```
beta(Lambda,Argument,Reduct):-
   copy_term(Lambda,
             lambda(Argument,Reduct)).
```

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

Observe how predicate `beta` generalises β-reduction: whereas the β-reduct is obtained by substitution

$$(\lambda P.E)(A) =_\beta E[P \leftarrow A],$$

`beta` specifies the unification of parameters with arguments. The λ-expressions here behave in that respect as Prolog predicates, and the unification can instantiate arguments. It is also possible to invoke λ-expressions as (anonymous) Prolog predicates:

```
call(Lambda,Arguments):-
   beta(Lambda,Arguments,Goal),
   call(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 λ-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 λ-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'(Lambda_1,
       Lambda_2,
       lambda([Pl_1+Pl_2],(Body_1,Body_2)))
:- beta(Lambda_1,[Pl_1],Body_1),
   beta(Lambda_2,[Pl_2],Body_2).
```

However, a simplified form of β-reduction without copy is sufficient, since λ-expressions are only β-reduced once. The translation scheme thus reads:

```
',_PL'(lambda([Pl_1],Body_1),
       lambda([Pl_2],Body_2),
       lambda([Pl_1+Pl_2],
              (Body_1,Body_2))).
```

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

```
':-_PL'(lambda([Pl_h],Head),
        lambda([Pl_b],Body),
        (Head:-Body,Pl_h is Pl_b+1)).

f(_,Goal,lambda([Pl],Goal ? Pl)).
```

There is a slight difficulty with built-in predicates, of course. A simple solution is, if built-ins count for one logical inference, to add the following clauses to the definition of `f`:

```
f(b,true,lambda([0],true)):- !.
f(b,Built_in,lambda([1],Built_in)) :-
 built_in(Built_in), !.
```

Adopting this solution for built-ins, the `reverse` procedure:

```
reverse([],[]) :- !.
reverse([X|Tail],LiatX) :-
   reverse(Tail,Liat),
   append(Liat,[X],LiatX).
```

is translated into:

```
reverse([],[]) ? Pl :- !, Pl is 0 + 1 . (1)
reverse([X|Tail],LiatX) ? Pl :-
   reverse(Tail,Liat) ? Pl_1,
   append(Liat,[X],LiatX) ? Pl_2,
   Pl is Pl_1 + Pl_2 + 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). 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 [Sterling and Beer 1986]: 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

$$p(T_1,\ T_2,\ \ldots)\ ?\ X_{pl},$$

X_{pl} is a meta-level variable, and should be "pushed" as a new argument of predicate p: $p(T_1,T_2,\ldots,X_{pl})$. This idea translates simply in our case: it is sufficient to redefine the last clause for `f` so that meta-arguments are pushed:

```
f(_,Goal,lambda([Pl],NewGoal)) :-          (2)
   push_args(Goal,[Pl],NewGoal).

push_args(Goal,Args,NewGoal) :-
   Goal =.. List,
   append(List,Args,NewList),
   NewGoal =.. NewList.
```

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 `reverse` procedure becomes:

```
reverse([],[],Pl) :- !, Pl is 0 + 1 .    (3)

reverse([X|Tail],LiatX,Pl) :-
  reverse(Tail,Liat,Pl_1),
  append(Liat,[X],LiatX,Pl_2),
  Pl is Pl_1 + Pl_2 + 1 .
```

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

Note 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 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 [Sterling and Beer 1986]. Such a separate transformation, an *argument pusher* can also be defined here, if convenient: here is an argument pusher to go from the translated clauses (1) to clauses (3):

```
',_AP'(Goal1,Goal2,(Goal1,Goal2)).

':-_AP'(Head,Body,(Head:-Body)).

f(_,MetaGoal,NewGoal) :-
  MetaGoal = ObjGoal ? MetaArgs
  -> push_args(ObjGoal,MetaArgs,NewGoal)
  ;  NewGoal = MetaGoal.
```

We are thus faced with two approaches to meta-argument pushing:

- the direct pushing approach (clauses (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 [Sterling and Beer 1986] 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 one 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 $W(X)$. 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 Σ-algebras is itself a Σ-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

$$\begin{aligned}&\texttt{lambda}(\overline{\mathcal{M}}_1,\mathcal{G}_1) \ {}_{,A}\ \texttt{lambda}(\overline{\mathcal{M}}_2,\mathcal{G}_2)\\ &= \texttt{lambda}(\overline{\mathcal{M}},\ (\mathcal{B}_1[\overline{\mathcal{M}}_1,\overline{\mathcal{M}}_2,\overline{\mathcal{M}}],\\ &\qquad \mathcal{G}_1,\mathcal{G}_2,\\ &\qquad \mathcal{B}_2[\overline{\mathcal{M}}_1,\overline{\mathcal{M}}_2,\overline{\mathcal{M}}]))\end{aligned}$$

$$\begin{aligned}&\texttt{lambda}(\overline{\mathcal{M}}_\mathcal{H},\mathcal{H}) \ :-_A\ \texttt{lambda}(\overline{\mathcal{M}}_\mathcal{B},\mathcal{B})\\ &= \mathcal{H} \ :-\ (\mathcal{B}_3[\overline{\mathcal{M}}_\mathcal{H},\overline{\mathcal{M}}_\mathcal{B}],\mathcal{B},\mathcal{B}_4[\overline{\mathcal{M}}_\mathcal{H},\overline{\mathcal{M}}_\mathcal{B}])\end{aligned}$$

$$f_A(\mathcal{G}) = \texttt{lambda}(\overline{\mathcal{M}}, push(\mathcal{G},\overline{\mathcal{M}}_p))$$

where $\overline{\mathcal{M}}$ ranges over lists of meta-level terms (terms in which the only variables are meta-level ones), $\overline{\mathcal{M}}_p$ denotes a subset of $\overline{\mathcal{M}}$, $push(\mathcal{G},\overline{\mathcal{M}})$ denotes the pushing of terms in $\overline{\mathcal{M}}$ as new arguments to goal $\mathcal{G}$, $\overline{\mathcal{M}}_1 \parallel \overline{\mathcal{M}}_2$ denotes the concatenation of two such lists, and $\mathcal{B}[\overline{\mathcal{M}}_1,\overline{\mathcal{M}}_2,...]$ indicates that the compound goal $\mathcal{B}$ instantiates only variables occurring in $\overline{\mathcal{M}}_1$, $\overline{\mathcal{M}}_2$, ...

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

$$\begin{aligned}&\texttt{lambda}(\overline{\mathcal{M}}_1 \parallel \overline{\mathcal{M}}'_1,\mathcal{G}_1) \ {}_{,A+A'}\\ &\texttt{lambda}(\overline{\mathcal{M}}_2 \parallel \overline{\mathcal{M}}'_2,\mathcal{G}_2)\\ &= \texttt{lambda}(\overline{\mathcal{M}} \parallel \overline{\mathcal{M}}',\\ &\qquad (\mathcal{B}_1[\overline{\mathcal{M}}_1,\overline{\mathcal{M}}_2,\overline{\mathcal{M}}],\mathcal{B}'_1[\overline{\mathcal{M}}'_1,\overline{\mathcal{M}}'_2,\overline{\mathcal{M}}'],\\ &\qquad \mathcal{G}_1,\mathcal{G}_2,\\ &\qquad \mathcal{B}_2[\overline{\mathcal{M}}_1,\overline{\mathcal{M}}_2,\overline{\mathcal{M}}],\mathcal{B}'_2[\overline{\mathcal{M}}'_1,\overline{\mathcal{M}}'_2,\overline{\mathcal{M}}']))\end{aligned}$$

$$\begin{aligned}&\texttt{lambda}(\overline{\mathcal{M}}_\mathcal{H} \parallel \overline{\mathcal{M}}'_\mathcal{H},\mathcal{H}) \ :-_{A+A'}\\ &\texttt{lambda}(\overline{\mathcal{M}}_\mathcal{B} \parallel \overline{\mathcal{M}}'_\mathcal{B},\mathcal{B})\\ &= \mathcal{H} \ :-\ (\mathcal{B}_3[\overline{\mathcal{M}}_\mathcal{H},\overline{\mathcal{M}}_\mathcal{B}],\mathcal{B}'_3[\overline{\mathcal{M}}'_\mathcal{H},\overline{\mathcal{M}}'_\mathcal{B}],\\ &\qquad \mathcal{B},\\ &\qquad \mathcal{B}_4[\overline{\mathcal{M}}_\mathcal{H},\overline{\mathcal{M}}_\mathcal{B}],\mathcal{B}'_4[\overline{\mathcal{M}}'_\mathcal{H},\overline{\mathcal{M}}'_\mathcal{B}])\end{aligned}$$

$$\begin{aligned}&f_{A+A'}(\mathcal{G})\\ &= \texttt{lambda}(\overline{\mathcal{M}} \parallel \overline{\mathcal{M}}', push(\mathcal{G},\overline{\mathcal{M}}_p \parallel \overline{\mathcal{M}}'_p))\end{aligned}$$

For example, combining the proof length translation scheme with a proof tree construction translation scheme [Sterling and Beer 1986] produces the following translation scheme:

$$\begin{aligned}&f_{PL+PROOF,b}(!)\\ &= \texttt{lambda}([0,!],push(!,[]))\\ &f_{PL+PROOF,b}(\texttt{true})\\ &= \texttt{lambda}([0,\texttt{true}],push(\texttt{true},[]))\\ &f_{PL+PROOF,b}(\mathcal{G})\\ &= \texttt{lambda}([Pl,Proof],push(\mathcal{G},[Pl,Proof]))\\ &f_{PL+PROOF,h}(\mathcal{G})\\ &= \texttt{lambda}([Pl,Proof,\mathcal{G}],push(\mathcal{G},[Pl,Proof]))\end{aligned}$$

$$\begin{aligned}&\texttt{lambda}([Pl\_1,Proof\_1],\mathcal{G}_1) \ {}_{,PL+PROOF}\\ &\texttt{lambda}([Pl\_2,Proof\_2],\mathcal{G}_2)\\ &= \texttt{lambda}([Pl\_1+Pl\_2,Proof],\\ &\qquad (\texttt{true},\\ &\qquad \texttt{true},\\ &\qquad \mathcal{G}_1,\mathcal{G}_2,\\ &\qquad \texttt{true},\\ &\qquad Proof = \texttt{and}(Proof\_1,Proof\_2)))\end{aligned}$$

$$\begin{aligned}&\texttt{lambda}([Pl\_h,Proof\_h,Code\_h],H) \ :-_{PL+PROOF}\\ &\texttt{lambda}([Pl\_b,Proof\_b],B)\\ &= H \ :-\ \texttt{true},\texttt{true},\\ &\qquad B,\\ &\qquad Pl\_h \ \texttt{is}\ Pl\_b + 1,\\ &\qquad Proof\_h = \texttt{rule}(Code\_h,Proof\_b)\end{aligned}$$

This form of combination solves the problems encountered with the successive application of meta-interpreters [Sterling and Beer 1986] 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 resulting translation for the example is:

```
reverse([],[],Pl_h,Proof_h) :-
    true,
    true,
    !,
    Pl_h is 0 + 1,
    Proof_h = rule(reverse([],[]),!).
reverse([X|Tail],LiatX,Pl_h,Proof_h) :-
    true,
    true,
    true,
    true,
    reverse(Tail,Liat,Pl_1,Proof_1),
    append(Liat,[X],LiatX,Pl_2,Proof_2),
    true,
    Proof_b = and(Proof_1,Proof_2),
    Pl_h is Pl_1 + Pl_2 + 1,
    Proof_h = rule(reverse([X|Tail],LiatX),
                   Proof_b).
```

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 characterised as follows:

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

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

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 localised at the semicolons.

Programs in such a form are simpler to handle than programs in the general form, e.g. for compilation.

First, the normalisation 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 a goal implies its replacement by a possibly different goal, and the production of new clauses, meant to be part of the normal form program. To simplify goals, a supply of 'new' predicate symbols not occurring in the program must be available.

- 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 $\wp(\mathcal{X}_1, \ldots)$ with $\wp$ a new predicate symbol and the $\mathcal{X}_i$ all the variables occurring in $\mathcal{G}_1$ and $\mathcal{G}_2$. A new clause is produced:

$$\wp(\mathcal{X}_1, \ldots) \text{ :- } \mathcal{G}_1, \mathcal{G}_2$$

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

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

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

Again, this will imply both goal replacement and the production of new clauses: the normalisation 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 $(\mathcal{S}, C)$, where $\mathcal{S}$ is a simplifiable goal and C a set of clauses. Normalisation can be described as follows:

$$f_h(\mathcal{H}) = \mathcal{H}$$

$$f_b(\mathcal{G}) = (\mathcal{G}, [])$$

$$(\mathcal{S}_1, C_1),_{nf} (\mathcal{S}_2, C_2) = ((\mathcal{S}_1', \mathcal{S}_2'), (C_1 \cup C_2 \cup C_1' \cup C_2'))$$

$$(\mathcal{S}_1, C_1);_{nf} (\mathcal{S}_2, C_2) = ((\mathcal{S}_1'; \mathcal{S}_2'), (C_1 \cup C_2 \cup C_1' \cup C_2'))$$

where the $\mathcal{S}_i'$ are the respective translations of the $\mathcal{S}_i$ with the production of clauses $C_i'(i = 1, 2)$.

The normalisation of a clause $\wp(T_1, \ldots) \text{ :- } \mathcal{S}$ with a normalised (hence simplifiable) body, is a list of clauses:

$$\wp(\mathcal{T}_1, \ldots) :-_{nf} (\mathcal{S},\mathcal{C}) = [(\wp(\mathcal{X}_1,\ldots):-\wp'(\mathcal{X}_1,\ldots)), (\wp'(\mathcal{T}_1,\ldots):-\mathcal{S}) \mid \mathcal{C}]$$

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

Finally, the semantics for & is as follows:

$$[(\wp(\mathcal{X}_1,\ldots) :- \mathcal{S}) \mid \mathcal{C}] \;\&_{nf}\; [(\wp(\mathcal{X}_1,\ldots) :- \mathcal{S}') \mid \mathcal{C}'] = [(\wp(\mathcal{X}_1,\ldots):-\mathcal{S}_0;\mathcal{S}'_0) \mid \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'']$$

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 predicates $\wp$ is defined by a clause $\wp(\mathcal{X}_1, \ldots) :- \mathcal{S}$ which will unify with any goal invoking $\wp$. In this way, the second characterization of the normal form is satisfied. To lift this restriction it is sufficient to modify the specification for function f as follows:

$$f_b(\texttt{true}) = (\texttt{true},[])$$
$$f_b(\wp(\mathcal{T}_1, \ldots)) = (\wp'(\mathcal{T}_1,\ldots),[\wp'(\mathcal{X}_1,\ldots):-\wp(\mathcal{X}_1,\ldots)]) \quad \text{if } \mathcal{G} \neq \texttt{true}$$

where $\wp'$ 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 concur 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[4] in order to minimise 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[5]. 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 has 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 implementations that favour some Prolog constructs over other apparently equivalent ones; this can introduce a factor of more than 2 in the ratios!

9 CONCLUDING REMARKS

An algebraic framework for meta-level programming is presented. Its main aspects are illustrated by a number of examples. Comparisons are 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.

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

[5] As in [Sterling and Beer 1986][Sterling and Shapiro 1987], some authors seem to define the efficiency ratio by comparing an *interpreted* meta-interpreter with a *compiled version* of its partial evaluation.

[6] Ratios in parentheses indicates the additional gain of argument pushing.

Table 1.

	Vanilla unify	Occur check unify	Proof[6] length	Description
rev	1.54	1.03	1.60 (×2.12)	Naive reverse of 30 elements
lips	5.50	1.83	2.38 (×1.11)	200 deterministic propositional calls
lipsconj	11.00	2.65	4.66 (×1.18)	Lots of propositional conjunctions
lipsback	4.50	1.79	1.12 (×1.03)	Heavy backtracking
succ	1.72	1.07	1.64 (×2.07)	Factorial of s(s(s(s(0))))

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 following features help the structuring of algebraic translation schemes:

- 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 optimisation 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 translations 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 optimisation 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.

REFERENCES

[Burstall and Landin 1969]
R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer, D. Michie, and F. Michael Swann, editors, *Machine Intelligence 4*, chap. 2, pp 17–43, Edinburgh University Press, 1969.

[Burstall 1980]
R. Burstall. Electronic category theory. In P. Dembiński, editor, *Mathematical Foundations of Computer Science 1980, 9th Symposium held in Rydzyna, Poland*, Springer-Verlag, 1980.

[Burstall and Goguen 1982]
R. Burstall and J. Goguen. Algebras, theories and freeness: an introduction for computer scientists. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pp 329–349, D. Reidel, 1982.

[Goguen et al. 1977]
J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24(1):68–95, January 1977.

[Sestoft and Søndergaard 1988]
P. Sestoft and H. Søndergaard. A Bibliography on Partial Evaluation. *SIGPLAN*, 23(2):19–27, February 1988.

[Sterling and Beer 1986]
L. Sterling and R. D. Beer. Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proceedings 1986 Symposium on Logic Programming, Salt Lake City, Utah*, pp 20–27, IEEE, 1986.

[Sterling and Shapiro 1987]
L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1987.