TR-148

Application of Partial Evaluation to the Algebraic Manipulation System and its Evaluation

by
Toshiaki Takewaki, Akikazu Takeuchi
Susumu Kunifuji and Koichi Furukawa

December, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Application of Partial Evaluation to the Algebraic Manipulation System and its Evaluation

Toshiaki Takewaki, Akikazu Takeuchi
Susumu Kunifuji and Koichi Furukawa

ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F
4-28 Mita 1-chome, Minato-ku, Tokyo, 108 Japan

ABSTRACT

This algebraic manipulation system solves an input expression by meta-level inference [Takewaki 85]. It is shown that using meta-level inference drastically reduces search spaces and, leads to clear object and meta-level programs.

In this system, meta-level inference is realized by a meta programming approach. Meta-programming is a widely used programming technique in logic programming. The advantages of the meta-programming approach are (1) it is easy to distinguish clearly an object-level programs and meta-level programs, (2) it is easy to understand both meta level and object-level program and (3) it is also easy to modify. A disadvantage is that execution speed is slow. Partial evaluation [Takeuchi 85] can overcome this problem. This paper presents an application of partial evaluation to the system and an evaluation of a specialized program by partial evaluation.

Keywords: Algebraic manipulation system, demo predicate, meta-level inference, meta-programming, partial evaluation

1. Introduction

Expert systems are special cases of problem solving systems where the emphasis is on domain specific knowledge. Algebraic manipulation systems are problem solving system which use algebraic knowledge. The system presented here solves input expressions by applying a refined method, which employs rewriting rules.

Meta-level inference is used for selection from the set of rewriting rules. Meta-level inference is controlled by meta-knowledge which guides the system towards the best way of solving an expression. The unlimited use of rules for

mathematical problem solving will result in an explosion of computation volume and exhaustion use of search space. Consequently it becomes extremely difficult to solve the problem in real time. Furthermore, it cannot be generally said that the search is oriented always in the optimum direction. There is, therefore, a need to minimize wasteful search by a selective use of search space and directing search in what is believed to be the best direction. Object-level inference consists of the application of rewriting rules to input expressions.

The system solves equations of elementary functions of one variable, and differentiates and integrates single variable elementary functions. The expressions are largely taken from high school textbooks and examination papers in Japan. Some typical expressions are shown in Figure 1.

In section 2, the system is introduced and its main features and guiding philosophy are described. In section 3 we discuss the meta-programming approach. The principles of meta-inference using meta-knowledge in this system. are given in section 4. Finally, section 5 shows how partial evaluation is applied in this system and describes the evaluation of the program by partial evaluation.

2. Overview of the system

The system consists of an object-level program and a meta-level program. The object-level program consists of set of rewriting rules and a program to apply them. These rules are called **methods**. The meta-level program controls selection from the set of rewriting rules using meta-level knowledge.

$$4 \cdot X^4 - 17 \cdot X^2 + 4 = 0 \qquad \text{(Expression 1)}$$

$$2 \cdot \log_2 X - 3 \cdot \log_X 2 + 5 = 0 \qquad \text{(Expression 2)}$$

$$3 \cdot \cos^2 X + 5 \cdot \sin X - 1 = 0 \qquad \text{(Expression 3)}$$

$$\left(\log \left| X + (X^2 + A) \right| \right)' \qquad \text{(Expression 4)}$$

$$\int X^2 \cdot \arcsin X \, dx \qquad \text{(Expression 5)}$$

$$\int X \cdot e^{-X^2} dx \qquad \text{(Expression 6)}$$

Figure 1. Some typical expressions handled by the system

Expressions are solved by applying methods. The system uses methods such as isolation, collection, attraction, change of unknown, factorization and so on. Most basic methods are similar to those of PRESS [Bundy 81] at the University of Edinburgh. For example, the collection method is designed to reduce the number of occurrence of the unknown variable in an expression. Some typical rewriting rules of the collection method are

```
U·V+U·W          →    U·(V+W)
sin U·cos U      →    1/2·sin(2·U)
```

all of which collect with respect to U.

Meta-level control of object-level programs reduces the number of applicable rewriting rules. Because the search space at meta-level inference is usually much smaller than the search space at the object-level inference it is controlling and this helps overcome the combinatorial explosion.

In this system meta-level control of the object-level program is called **meta-level inference**. Meta-level inference is the heart of the system. It significantly increases the system's efficiency.

Meta-level inference employs meta-knowledge. The system used a variety of meta-knowledge for inference strategies and to select methods. The general solution procedure use meta-level knowledge about the expression to find an appropriate method to solve it.

The kinds of knowledge about expressions used by the system are the number of occurrences of the unknown variable in the expression, the argument positions at which they occur,classification of functions in it, degree of polynomial, operators in the expression and others.

This system uses a meta-programming approach and a partial evaluation technique to realize a two-level structure. Meta-programming improves readability and maintainability of program while partial evaluation increases execution speed.

3. Meta-programming approach

Meta-programming is often used in logic programming. For example, the algorithmic program debugging system [Shapiro 83] involves a lot of meta-programming and, APES [Hammond 83] is a tool for building logic based expert systems, which utilizes meta programming. A meta-program can be defined informally in the following way. First, the meta-program handles a program as data. Second, the meta-program handles data as a program and evaluates it. Third, the meta-program treats the result of a program as data. The most well known example of a meta-program is the demo predicate of Bowen and Kowalski [Bowen 83]. In this system, the demo predicate is called

solve.

A meta-programming approach enables the construction of powerful programming environments as most meta-programs are realized as meta-interpreters of object-programs. Lisp and Prolog make it easy to write interpreters for the languages themselves. For instance, an explanation facility can be realized with meta-programming by adding an extra argument to the predicate solve two arguments.

Figure 2 is a meta-interpreter with an explanation facility. The predicate **solve** contains three arguments. The first argument is the domain world name, the second argument is a goal to be solved and the last argument is trace information on the goal when it is solved.

The first and second clauses provide the solution procedures when a goal is in the form of conjunctions and disjunctions, respectively. The third clause is assumed to be true, if **Goal** is true in the domain world. The fourth clause actually solves the goal using the predicate clause_world. The fifth clause is used when a goal is a built-in system predicate in Prolog.

This meta-interpreter can be easily modified to handle the cut operator , and to execute a Prolog interpreter in Prolog.

The meta-programming approach has several advantages. First, it is easy to distinguish clearly an object-level program and meta-level program. This clear separation of object program and meta program makes it easy to understand the system (readability), and system modification is also easy (maintainability). However, execution speed is slow.

```
solve(World, (Goal1,Goal2),(Proof1,Proof2)) :-
             solve(World,Goal1,Proof1),
solve(World,Goal2,Proof2).
solve(World, (Goal1;Goal2),Proof) :-
             solve(World,Goal1,Proof);
solve(World,Goal2,Proof).
solve(World, true, true).
solve(World, Goal,implies(Proof,Goal,Body)) :-
             clause_world(World,Goal,Body),
             solve(World, Body,Proof).
solve(World, Goal,system) :-
             system(Goal),
             Goal.
```

Figure 2. Meta-interpreter with an Explanation Facility

4. Control of the system

Meta-level inference is controlled by meta-knowledge which guides the system toward the best way of solving an expression. Figure 3 shows a part of the predicate **strategy** for selecting a method on the basis of such meta-knowledge. The predicate **strategy** contains five arguments. The first argument is the input expression, the second is the selected method name, and the third is the unknown variable. The predicate **strategy** determines a method to be applied using meta knowledge.

- The first clause determines that the isolation method will be used when the right hand side is free of unknown variables (predicate **free_of**), there is a single occurrence of an unknown variable in the left hand side (predicate **singleocc**), and its position is known (predicate **position**).

- The second clause determines that the factorization method will be selected when the main operator in the left hand side is multiplication (predicate **main_op_ mult**) and the right hand side is zero (predicate **zero**).

The process of solving expressions is controlled the predicate **method_demo**. Figure 4 shows the predicate **method_demo** used in this control. Argument of **method_demo** indicate the input expression, the unknown variable, the control information for the input expression, the past history and the solution.

- The first clause checks for an infinite loop in **History**. If the analysis is positive, a **Find infinite loop** message is returned and operation is suspended.

```
strategy( LHS=RHS,isolation,Unknown,Control,
            [position(P)|Control]) :-
        free_of(Unknown, RHS),
        singleocc(Unknown,LHS),
        position(Unknown,LHS,P).
strategy(LHS=RHS,factorization, Unknown, Control,Control) :-
        main_op_mult(LHS),
        zero(RHS).
           :
```

Figure 3. Part of The Predicate "strategy"

```
method_demo(Expression,_,_,History,'Find infinite loop') :-
          loop_check(Expression,History).
method_demo(Expression,Unknown,Control,History,Answer) :-
          end_check(Expression,Unknown,Control,Ans).
method_demo(Expl or Exp2, Unknown,Control,
                                  History,Ansl or Ans2) :-
          method_demo(Expl,Unknown,Control,History,Ansl),
          method_demo(Exp2,Unknown,Control,History,Ans2).
method_demo(Expression,Unknown,Control,History,Answer) :-
     strategy(Expression,World,Unknown,Control,NewControl),
     solve(World,
           apply(Expression,Unknown,New_Exp,NewUnknown),
           CutMark,Explanation),
     explanation(Explanation),
     method_demo(New_Exp,New_Unknown,New_Control,
             [Expression|History],Answer).
```

Figure 4. Predicate method__demo

- The second clause breaks down the problem into its component expressions. Component expressions are created when the system applies the factorization method.

- The third clause judges whether the expression was solved. The judgment is made by checking the condition contained in the Control. For example, given the request to solve equation $X^2-4=0$, the answers are $X=2$ and $X=-2$. But, if a condition $X>0$ is stipulated, then $X=-2$ will not be a solution.

- The fourth clause determines the applicable rewriting rule with the predicate **strategy** and applies the rewriting rule when using the predicate solve. The predicate **explanation** has the system explain the reason of the object-level program (a rewriting rule), and the predicate **method__demo** processes the expression recursively.

The predicate solve is a meta-interpreter with the cut operator and solves the goal **apply(Expression, Unknown, New__ Exp, NewUnknown)** in the domain world.

5. Application of Partial Evaluation

As stated above, the meta-programming approach makes it easy to develop a system, but execution speed is slow. Partial evaluation can remove this problem when used in translating a meta-program to a efficient program.

Partial evaluation of programs involves making the original program more specific using information about the run time environment. The meta-interpreter plus the original object program is converted into the

```
solve(World,(P;Q),V) :- (solve(World,P,V);solve(World,Q,V)).
solve(World,(!,Q),V) :- cut(V),
        (V==cut, !; solve(World,Q,V)).
solve(World,(P,Q),V) :- P\== !,
        solve(World,P,V), solve(World,Q,V).
solve(World,!,CUT) :- cut(CUT).
solve(World,true,_).
solve(World,not(P),_) :- !, if(solve(World,P,_),fail,true).
solve(World,P,V) :-
        clause_world(World,P,Q),
        solve(World,Q,V),
        (V==cut, !, fail; true).
solve(World,P,V) :- sub_system(P), P.
solve(World,P,V) :- system(P), P.

cut(_).
cut(cut).

if(If,Then,Else) :- If, !, Then.
if( _,   _,Else) :- Else.

clause_world(World,P,Q) :- X=.. [World,R], X,
        clause_body(R,P,Q).

clause_body((P :- Q),P,Q) :- !.
clause_body(P,P,true).

sub_system(tidyl(_,_)).
sub_system(condition_check(_)).

system(_ is _).
system(_ =:= _).
```

Figure 5. (a) Meta-interpreter

specialized object program by partial evaluation and program obtained is executed much faster than the original program. In fact, since an object program can be regarded as input data from a meta-program, the meta-program can be specialized by partial evaluation if the object program is given. Thus, the specialized program has no interpretive code, which improves the efficiency of the program.

This can be best explained by example. Figure 5 is a general meta interpreter in Prolog with the cut operator for all methods and an object program of isolation method. **sub_system** and **system** are recognizer predicates that succeed when they are the basic predicate in an algebraic manipulation system and a built-in predicate, respectively. Each clause of Figure 5 (b) is expressed fact in Prolog as **isolate(Clause)**. Figure 6 lists the instructions for partial evaluation specified by an user. Figure 7 is the specialized program of the original by partial evaluation. The general meta-

```
apply(L=R,Ctrl,Ans,N_Ctrl,Hist) :-
      member(unknown(Unknown),Ctrl),
      remove(position(P),Ctrl,Ctrl_1),
      isolate(P,L=R,Ans1,Cont,Hist),
      tidy1(Ans1,Ans),
      append(Cont,Ctrl_1,N_Ctrl),!.

member(X,[X|_]) :- !.
member(X,[_|Y]) :- member(X,Y).

remove(X,[X|Y],Y) :- !.
remove(X,[Y|Z],[Y|W]) :- remove(X,Z,W).

isolate([Car|Cdr],FX=W,RHS,N_Ctrl,[(RuleNo,Wait)]) :-
        isolax(RuleNo,Wait,Car,FX=W,RHS,Ctrl,Condition),
        condition_check(Condition),
        add_information(Cdr,Ctrl,N_Ctrl).

append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

add_information([],Ctrl,[end|Ctrl]).
add_information(P,C,[next(isolate),position(P)|C]) :- P\==[].

isolax(iso02,1,1,V+U=W,V=W-U,
      [explanation([Explanation])],true).
```

Figure 5. (b) Object Program of Isolation Method

interpreter is transformed into an exclusive meta-interpreter for the isolation method.

Table 1 is a comparison of the execution times of various programs. Execution times are compared with the execution time required by the program **p1**.

The program **p1** is the meta-interpreter plus object program, that is the original program produced with the meta-programming approach (two level structure program). Program **p2** is the specialized program generated by partial evaluation of program **p1**. Lastly, program **p3** is written with the one-level structure approach, that is a mixture of meta-level and object-level program combined as an object-level program.

Program **p1** has good readability and maintainability of the program but its execution speed is slow. Program **p3**'s execution speed is fast, but

```
type(A=B,  t) :- var(A),var(B),!.
type(A=cut,t) :-!.
type(A=B,  s) :- ( nonvar(A) ; nonvar(B) ).
type(!,          t).
type(system(A), e).
type(call(A),    t).
type(A \== !,    e).
type(cut(A),     t).
type(A==cut,     t).
type(fail,       t).
type(sub_system(A),       e).
type(clause_world(A,B,C),e).
type(solve(World,(B,C), D),g).
type(solve(World,true,  B),e).
type(solve(World,fail,  B),t).
type(solve(World,!,     A),g).
type(solve(World,A\==[],B),g).
type(solve(World,B,     C),g) :- sub_system(B).
type(solve(World,apply(B,C,D,E,F),       G),g) :- nonvar(World).
type(solve(World,member(unknown(A),B),   C),g).
type(solve(World,remove(position(A),B,C),D),g).
type(solve(World,isolate(A,B=C,D,E,F),   G),g).
type(solve(World,isolax(A,B,C,D=E,F,G,H),I),g).
type(solve(World,add_information(A,B,C), D),g).
type(solve(World,append(A,B,C),          D),g).

inhibit_unfolding(solve(World,(!,Goal),                C)).
inhibit_unfolding(solve(World,!,                       A)).
inhibit_unfolding(solve(World,member(unknown(A),B),    C)).
inhibit_unfolding(solve(World,remove(position(A),B,C),D)).
inhibit_unfolding(solve(World,isolate(A,B=C,D,E,F),    G)).
inhibit_unfolding(solve(World,isolax(A,B,C,D=E,F,G,H),I)).
inhibit_unfolding(solve(World,add_information(A,B,C), D)).
inhibit_unfolding(solve(World,append(A,B,C),           D)).
```

Figure 6. Instructions for Partial Evaluation

readability and maintainability are poor. Thus, advantages and disadvantages of both programs are complementary. This complementary relation reflects

```
solve(isolate,apply(A=B,C,D,E,F),G) :-
        solve(isolate,member(unknown(H),C),G),
        solve(isolate,remove(position(I),C,J),G),
        solve(isolate, isolate(I,A=B,K,L,F), G),
        call(tidy1(K,D)),
        solve(isolate,append(L,J,E),G),
        solve(isolate,!,G),
        (G==cut, !, fail; true).

solve(isolate,member(unknown(A),[unknown(A)|B]),C) :-
        solve(isolate,!,C),
        (C==cut, !, fail; true).
solve(isolate,member(unknown(A),[B|C]),D) :-
        solve(isolate,member(unknown(A),C),D),
        (D==cut, !, fail; true).

solve(isolate, remove(position(A),[position(A)|B],B), C) :-
        solve(isolate,!,C),
        (C==cut, !, fail; true).
solve(isolate, remove(position(A),[B|C],[B|D]), E) :-
        solve(isolate,remove(position(A),C,D),E),
        (E==cut, !, fail; true).

solve(isolate, isolate([A|B],C=D,E,F,[(G ,H)]), I) :-
        solve(isolate, isolax(G,H,A,C=D,E,J,K), I),
        call(condition_check(K)),
        solve(isolate,add_information(B,J,F),I),
        (I==cut, !, fail; true).

solve(isolate,add_information([],A,[end|A]),B) :-
        (B==cut, !, fail; true).
solve(isolate,
       add_information(A,B,[next(isolate),position(A)|B]), C) :-
        call(A\==[]),
        (C==cut, !, fail; true).

solve(isolate,append([],A,A),B) :-
        solve(isolate,!,B),
        (B==cut, !, fail; true).
solve(isolate,append([A|B],C,[A|D]),E) :-
        solve(isolate,append(B,C,D),E),
        (E==cut, !, fail; true).

solve(isolate,
      isolax(iso02,1,1,A+B=C,A=C-B,[explanation([D])],true),
      E) :-
        (E==cut, !, fail; true).

solve(isolate,!,A) :- cut(A).
```

Figure 7. Specialized program

Table 1. Comparison of Execution Times

	P1	P2	P3
Execution speed-up	1.00	5.01	6.21
Readability, Maintainability	○	○	△

p1: Meta-interpreter + Object Program
p2: Specialized Program by Partial Evaluation
p3: One-level Structure Program

the trade-off between efficiency in **p1** and readability and maintainability in **p3**. Program **p2** overcomes the trade-off by partial evaluation, it has the advantages of both programs with out the disadvantages of either. This is because **p1** is used to modify the program, while **p2** is used in its execution.

6. Conclusion

we described an application of partial evaluation to the algebraic manipulation system using meta programming. Meta-programming plays an important role in logic programming because of its expressive power. Partial evaluation will make meta-programming more practical by improving the execution efficiency of meta-programs.

Building on this research, we will seek to develop an algebraic manipulation system in a parallel logic programming language such as Guarded Horn Clauses (KL1-c) [Furukawa 85, Ueda 85]. Naturally, it will use meta-programming and partial evaluation in the parallel logic programming language.

ACKNOWLEDGMENTS

The authors wish to express their thanks to Kazuhiro Fuchi, Director of the ICOT Research Center, who provided us with the opportunity to pursue this research in the FGCS Project at ICOT. We would also like to thank the members of the First Research Laboratory at ICOT for their valuable comments.

REFERENCES

[Bundy 81] Bundy, A. and Welham, B. : Using Meta-level Inference for Selective Application of Multiple Rewrite Rule Sets in Algebraic Manipulation,

Artificial Intelligence No.16, pp.189-212 (1981).

[Bowen 83] Bowen, K. and Kowalski, R. : Amalgamating Language and Meta-language in Logic Programming, In Clark, K. and Tarnlund, S. (eds.) Logic Programming, Academic Press (1983).

[Furukawa 84] Furukawa, K., Kunifuji, S., Takeuchi, A., and Ueda, K. : The Conceptual Specification of the Kernel Language Version 1, ICOT TR-054, (1984).

[Hammond 83] Hammond, P. et al : A Prolog Shell for Logic Based Expert Systems, Imperial College (1983).

[Shapiro 83] Shapiro, E. : Algorithmic Program Debugging, The MIT Press (1983).

[Takeuchi 85] Takeuchi, A. and Furukawa, K. : Partial Evaluation of Prolog programs and its Application to Meta Programming, Proc. of the Logic Programming Conference'85, ICOT (1985).

[Takewaki 85] Takewaki, T., Miyachi, T., Kunifuji, S. and Furukawa, K. : An Algebraic Manipulation System Using Meta-level Inference Based on Human Heuristics. To appear in ICOT TR (1985).

[Ueda 85] Ueda,K. : Guarded Horn Clauses, Proc. of the Logic Programming Conference'85, ICOT (1985).