TR-166

Affinity between Meta Interpreters and Partial Evaluation

by
Akikazu Takeuchi

April, 1986

AFFINITY BETWEEN META INTERPRETERS AND PARTIAL EVALUATION

Akikazu Takeuchi
ICOT Research Center
Institute for New Generation Computer Technology
1–4–28, Mita, Minato-ku, Tokyo 108 Japan

A fine affinity between generality of meta interpreters and specialization by partial evaluation is remarked. Two open problems with this view are pointed out and discussed. One relates to the semantics of partial evaluation of nondeterministic parallel languages and the other concerns meta description on parallel computation.

1. AFFINITY

Partial evaluation has been investigated theoretically and has been recognized as an important methodology for software development {1, 2}. Many researchers explored partial evaluation in practical applications {3, 4, 5, 6, 7}. These applications are mainly in compiler generation and optimization.

The general idea of partial evaluation is specialization. It is dual to generalization. To explore the power of partial evaluation, it is better to consider partial evaluation with something general, since, whenever something is general, then there is room for partial evaluation and the resultant specialized program is usually more efficient. Recent reports, {8, 9, 10}, differ from the previous ones in that they explore partial evaluation accompanied with exploration of meta description in terms of a meta interpreter, which is essentially a *general* concept.

As we proceed to the stage where the problem solving ability of computer software is the main theme, we often require not only the domain specific (object-level) knowledge, but also the meta-level knowledge for making use of the object-level knowledge more flexibly and hence enriching problem solving power {11, 12, 13, 14, 15, 16}. The separation of object and meta levels is often fruitful for designing and understanding programs. However, meta knowledge implemented by a meta interpreter results in run-time inefficiency because of its generality.

However, generality in meta level description matches the condition of applicability of partial evaluation. Meta level description can be specialized when an object level program is fixed. The resultant program has functionalities of both meta and object programs, since it is a specialized version of meta description with respect to some object program. Thus, there is a fine affinity between higher order general description in a meta interpreter and specialization in partial evaluation. Owing to this fine affinity, the technique can be applied naturally without any refinements or extensions. This fine affinity implies the new programming approach taken in {8, 9, 10} where a program is clearly described as layers of descriptions and executed after combining all the layers to form one layer by partial evaluation. We believe that the new approach is useful for general programming methodology including A.I. where meta-level knowledge plays a central role. The new approach is similar to general program transformation. However, it is more practical since

1) it is straightforward, just specializing the meta interpreter with respect to an object program. In program transformation the target is usually abstract, that is, increased program efficiency,
2) program transformation requires many heuristics to control transformation, while in partial evaluation only partial evaluation techniques of a language are required, which are easier to establish compared with general heuristics for program transformation.

However, several questions arise, such as: What is the formal meaning of partial evaluation of a meta interpreter with respect to an object program? Where does the resulting program belong? What kinds of techniques can be used in partial evaluation, especially for FCP? Other questions concerns the limits of this approach: What can and what cannot be expressed as a meta description? What kinds of meta descriptions are useful in parallel and nondeterministic languages such as FCP? These questions should be solved to verify the new approach.

The following sections are devoted to discussion of the above two types of question, i.e., section 2 deals with issues in semantics and section 3 with meta description on parallel computation.

2. SEMANTICS OF PARTIAL EVALUATION

The general semantics of partial evaluation is theoretically analyzed {1, 2}. Here we first give model theoretic semantics for partial evaluation of a meta interpreter for pure Prolog. It is useful for an intuitive understanding of specialization of a meta interpreter.

Although we assume that a syntactically identical language is used for description of both object level and meta level as usual, we have to distinguish the two languages conceptually. At the meta level, language constructs of an object language are all represented by constants of the meta language. From the object level, meta level description can be regarded as higher order description. Meta level description in the form of an interpreter is general in the sense that (meta) variables in the meta description can range over any language construct which can be expressed at the object level. For example, in the description of the program which can detect deadlock in {10}, a universally quantified meta variable is used to denote an object level goal. Specialization by partial evaluation is regarded as restricting the domain of such meta variables in the meta description to constants naming object level individuals, and executing parts of a meta program which become ready for execution. Formally the resultant program is still a meta level program which has smaller model than before. It is, however, easy to translate it back to the object level, since the same language is used for object and meta languages. In fact, the resultant programs in the examples in {10} can be regarded as programs obtained by translating the results of partial evaluation to the object level.

It is unclear what kinds of partial evaluation techniques are used in the partial evaluation of FCP programs. When the author developed the partial evaluator for Prolog {9}, the main techniques adopted were instantiation of some of the arguments and unfolding of a goal by clauses, the heads of which are unifiable with the goal. These techniques are known to guarantee preservation of the meaning of a program. Tamaki *et al.* {17} established the equivalence property of unfolding/folding procedures for Prolog in terms of the minimum model semantics. As far as we know, no theorem which guarantees the equivalence property of the unfolding procedure for FCP has been proved yet. But it is necessary to establish that the pair of a meta interpreter and an object program is equivalent to the partially evaluated program.

It is well known that, for a pure logic program, the success set, the minimum model and the least fixpoint of the function associated with the program are equivalent {18}. The success set is a set of ground literals that can be finitely derived from the program. It is possible to imagine the success set of a FCP program. A success set of a FCP program corresponds to a set of literals which are final forms of queries after the computation successfully terminates. Usually it is called a set of total histories, since the final form of a query represents in itself all the inputs received and all the output sent. However, it is known that a set of total histories is insufficient for the semantics of nondeterministic data flow languages {19}. The example used to prove this proposition is also valid in FCP. Hence, the set of total histories is insufficient for the semantics of FCP. We illustrate this using the example in {19}.

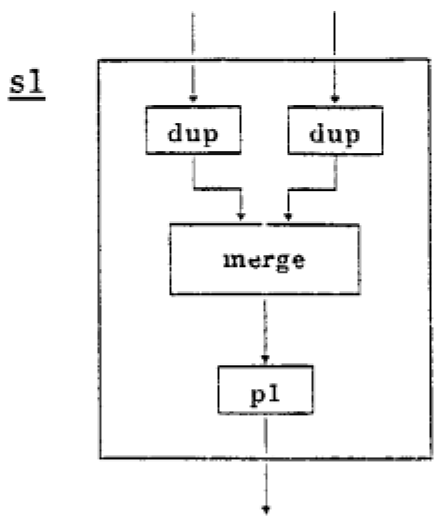

Fig. 1 Structure of s1

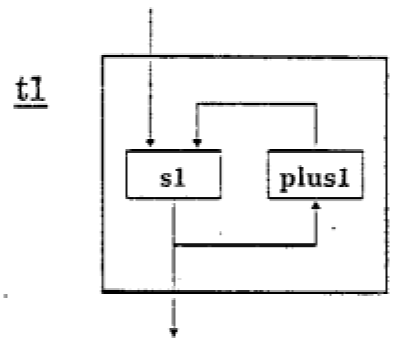

Fig. 2 Structure of t1

```
p1([A|In],[A|Out]) :-
     true | p11(In?,Out).
p11([A|In],[A]) :- true | true.

dup([A|I],[A,A]) :- true | true.

merge([A|Ix],Iy,[A|Out]) :-
     true | merge(Ix?,Iy,Out).
merge(Ix,[A|Iy],[A|Out]) :-
     true | merge(Ix,Iy?,Out).
merge(Ix,[],[]) :- true | true.
merge([],Iy,Iy) :- true | true.

s1(Ix,Iy,Out) :- true |
     dup(Ix?,Ox), dup(Iy?,Oy),
     merge(Ox?,Oy?,Oz), p1(Oz?,Out).


t1(In,Out) :- true |
     s1(In?,Mid?,Out), plus1(Out?,Mid).

plus1([A|In], [A1]) :- A1 := A+1 | true.
```

Fig. 1 and fig. 2 illustrate s1 and t1, respectively. The first argument of p1 is used as an input port and the second argument as an output port. The first two elements of a list received are output one by one as they are received. s1 has two input ports, the first and second arguments, and one output port, the third argument. Internally it invokes four goals, two dup's, merge and p1. dup outputs doubleton, the elements of which are duplicates of the first element of the list received. The total history set of s1 is:

```
{s1([X|Ix],_,[X,X]). s1(_,[Y|Iy],[Y,Y]).
 s1([X|Ix],[Y|Iy],[X,Y]).
 s1([X|Ix],[Y|Iy],[Y,X])}
```

where X and Y denote the first element of lists received at the first and second arguments, respectively. Note that the total history set reveals several possibilities of outputs for the same input, which results from the nondeterminism of the merge operator. t1 consists of s1 and plus1 where the output of s1 is connected to the second input port of s1 through plus1. plus1 outputs a singleton at the second argument, the element of which is equal to the value of the first element of the first argument plus one. The total history set of t1 when it receives [5] is:

```
{t1([5],[5,5]). t1([5],[5,6])}
```

Consider the partial evaluation of p1. If we unfold p11 in the body part of the clause of p1, then the following new clause is obtained for p1.

```
p1([A,B|In],[A,B]) :- true | true.
```

The total history set of s1 is not changed by this unfolding. However, the total history set of t1 changes to

```
{t1([5],[5,5])}.
```

This unfolding does not affect the semantics of the intermediate module, i.e., s1, but the semantics of the outer module, i.e., t1, including this intermediate module does change in the sense of the total history set. Thus the unfolding procedure above does not preserve the equivalence property of a program, even if it seems safe locally. The difficulty in modelling the computation of FCP and nondeterministic data flow languages results from their nondeterminism introduced by, for example, the merge operation. Although it is not clear that the techniques used in the partial evaluation in {10} include the unfolding in the situation mentioned above, it is clear that, when techniques for partial evaluation are introduced, they have to be accompanied with the model of computation by which we can discuss the equivalence property of each technique.

3. LIMIT OF META DESCRIPTION IN DISTRIBUTED ENVIRONMENT

Another point related to enriching the new approach concerns the functionalities of meta interpreters. The important question is what we can express in terms of a meta interpreter.

As mentioned above, a meta language has language constructs of an object language as constants. A meta interpreter written in the meta language is regarded as the description of the simulation of the object level computation at the meta level. Applications of meta interpreters presented so far can be classified into two categories, though they are closely mixed in real systems. One concerns observation and the other control.

Meta level description for observation of the object level computation is used in various areas such as debugging, expert system shells which produce explanation, deadlock-detecting meta interpreters and so on. Descriptions in these meta interpreters refer to the computation state or history of the object level computation simulated by the meta interpreter. Meta level description for control is used to modify and augment computation rules for programs written in object level languages. {8} explored several meta interpreters which adopt different strategies for resolving given goals. A meta interpreter handling certainty factors {20} is also an example of this category since it prunes search space with low certainty. We do not know of any other kinds of meta description useful to application. There is yet unexplored power in meta interpreters that is well worth investigating since the new approach makes it practical in real applications.

The problem arises in observation of computation. As is well known in the field of distributed processing, finite delay of data flow is inevitable in parallel programs running in a distributed environment. Hence it is generally difficult or even impossible to define an observable computation state meaningfully. The difficulty seems to appear in the program of deadlock-detecting meta interpreter in {10}. However, it is rather easy to recognize deadlock, since deadlock is a global and stable state. Owing to the side-effect free property of the language, once an event happens and leaves some traces, the event will be eventually observed since no computation can erase or overwrite the traces it has left.

If a meta interpreter has the description *"do something when some transient state S exists"*, it is difficult to implement a method for precisely determining whether the state still exists or not when it is observed. The difficulty originates in the delay between the moment when an event happens and the moment when it is observed, because of the finite delay of data flow. It is generally impossible to predict the delay between the real event and observation by a meta interpreter. The direct implication of this is that, in parallel languages, the definition and implementation of the concept of unboundness of an object variable is difficult {21}, since unboundedness is a transient property of a variable. Therefore, the definition of copy, that is, making a variant of a term, becomes difficult because it requires distinguishing whether a variable in the term is bounded or not.

Where does this observation lead us? The conclusion can be summarized as follows. There is no problem if meta description concerns pure observation, since what happened will eventually become observable. The problem may arises when meta description includes actions to be taken when observing some states. But it is still safe if the states are something like dead-end. The most difficult things to describe in meta level are actions to be performed when knowing some transient states to exist.

What we should do now is to investigate unexplored power of meta description on parallel computation after due consideration of these problems.

ACKNOWLEDGEMENT

We would like to thank Kazunori Ueda, Toshihiko Miyazaki, Hirohisa Seki, Koichi Furukawa for their many helpful discussions.

REFERENCES

{1} Y. Futamura, Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler, *Systems, Computers, Controls*, vol. 2, no. 5, 1971, 721-728.

{2} A. P. Ershov, Mixed Computation: Potential Applications and Problems for Study, *Theoretical Computer Science*, vol. 18, 1982, 41-67.

{3} P. Emanuelson, *Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation*, Linkoping Studies in Science and Technology Dissertations, no. 55, Software Systems Research Center, Linkoping University, 1980.

{4} J. Komorowski, *A Specification of Abstract Prolog Machine and Its Application to Partial Evaluation*, Linkoping Studies in Science and Technology Dissertations, no. 69, Software Systems Research Center, Linkoping University, 1981.

{5} K. Kahn, *A Partial Evaluator of Lisp Written in a Prolog Written in Lisp Intended to be Applied to the Prolog and Itself which in turn is Intended to be Given to Itself Together with the Prolog to Produce a Prolog Compiler*, UPMAIL, Dept. of Computing Science, Uppsala University, 1982.

{6} N. D. Jones, P. Sestoft and H. Sondergaard, *An Experiment in Partial Evaluation: The Generation of a Compiler Generator*, DIKU RAPPORT, NR: 85/1, University of Copenhagen, 1985.

{7} R. Venken, A Prolog Meta-Interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query-Optimisation, In *Proc. of ECAI-84*, North-Holland, 1984, 91-100.

{8} J. Gallagher, *Transforming Logic Programs by Specialising Interpreters*, Dept. of Computer Science, Trinity College, University of Dublin, 1984.

{9} A. Takeuchi and K. Furukawa, Partial Evaluation of Prolog Programs and Its Application to Meta Programming, In *Proc. IFIP-86 Congress*, Elsevier Science Publishers, 1986.

{10} E. Shapiro and S. Safra, Meta Interpreters for Real, In *Proc. IFIP-86 Congress*, North-Holland, 1986.

{11} R. Davis, Meta-rules: Reasoning bout Control, *Artificial Intelligence*, vol. 15, 1980, 179-182.

{12} A. Bundy and B. Welham, Using Meta-level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation, *Artificial Intelligence*, vol. 16, 1981, 189-212.

{13} H. Gallaire and C. Lasserre, Meta-Level Control for Logic Programs, In *Logic Programming*, K. Clark and S. Tarnlund (ed.), Academic Press, 1982, 173-185.

{14} K. Bowen and R. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, In *Logic Programming*, K. Clark and S. Tarnlund (ed.), Academic Press, 1983, 153-172.

{15} K. Bowen and T. Weinberg, *A Meta-Level Extension of Prolog*, Tech. Report CIS-85-1, Syracuse University, 1985.

{16} L. Sterling, Logical Levels of Problem Solving, In *Proc. The Second Inter. Conf. on Logic Programming*, Uppsala University, 1984, 231-242.

{17} H. Tamaki and T. Sato, Unfold/Fold Transformation of Logic Programs, In *Proc. The Second Inter. Conf. on Logic Programming*, Uppsala University, 1984, 127-138.

{18} K. R. Apt and M. H. van Emden, Contributions to the Theory of Logic Programming, *J. ACM*, vol. 29, no. 3, 1982, 841-862.

{19} J. D. Brock and W. B. Ackermann, Scenario: A Model of Nonderminate Computation, In *Formalization of Programming Concepts*, J. Diaz and I. Ramos (ed.), Lecture Notes in Computer Science, vol. 107, Springer-Verlag, 1981, 252-259.

{20} E. Shapiro, Logic Programs with Uncertainties: A Tool for Implementing Rule-based Systems, In *Proc. IJCAI-83*, 1983, 529-532.

{21} K. Ueda, *Guarded Horn Clauses*, ICOT Tech. Report TR-103, Institute for New Generation Computer Technology. Also to appear in *Logic Programming'85*, Eiichi Wada (ed.), Lecture Notes in Computer Science, no. 221, Springer-Verlag, Berlin Heidelberg, 1986.