PARTIAL EVALUATION AS A MEANS FOR INFERENCING DATA STRUCTURES IN AN APPLICATIVE LANGUAGE: A THEORY AND IMPLEMENTATION IN THE CASE OF PROLOG

H. Jan Komorowski
Software Systems Research Center
Linköping University
S-581 83 Linköping, Sweden

ABSTRACT

An operational semantics of the Prolog programming language is introduced. Meta-IV is used to specify the semantics. One purpose of the work is to provide a ***specification of an implementation*** of a Prolog interpreter. Another one is an application of this specification to a formal description of program optimization techniques based on the principle of ***partial evaluation***.

Transformations which account for pruning, forward data structure propagation and opening (which also provides backward data structure propagation) are formally introduced and proved to preserve meaning of programs. The so defined transformations provide means to inference data structures in an applicative language. The theoretical investigation is then shortly related to research in rule-based systems and logic.

An efficient well-integrated partial evaluation system is available in Qlog - a Lisp programming environment for Prolog.

1.0 INTRODUCTION

It is very likely that a large part of future programming will be programming in increasingly higher level languages. In such languages more attention will be paid to efficient problem solving, whereas this efficiency need not reflect the requirements of an efficient computation. In these circumstances program transformation tools will play the central role in making the efficiency realistic. It is also felt that the tools should be interactive. One reason is that they are to support the user in the immanently interactive activities of programming. The other one is that due to the complexity of some transformations the user's support might be indispensable in some points.

The growing interest in applicative languages programming has also given much impulse to research concerned with Prolog (which is a good example of an applicative and rather high level programming language). At the same time the perceived inefficiency in execution of many applicative languages has been an obstacle to their wide-spread acceptance. Consequently, algorithms are often coded for efficient execution at the expense of clarity. This compromises the applicative style which is the prime advantage of such languages.

We argue, that high-level program transformations can relieve the programmer from concern for efficiency in many cases. Several authors have considered the optimizing transformations and their applications, while relatively little attention has been paid to the so called ***partial evaluation*** transformations. Unfortunately, even if partial evaluation seems to be a very powerful and useful tool it has not been given any precise definition.

In this paper we investigate partial evaluation of Prolog programs as a part of a theory of interactive, incremental programming. The goal of this investigation is to provide formally correct, interactive programming tools for program transformation. Moreover, partial evaluation as introduced in the paper is not only a means to improve program efficiency but also a means for inferencing data structures in an applicative language.

The rest of this paper is organized as follows. First, an informal introduction of partial evaluation and a short discussion of related research is presented. Second, after the preliminaries which establish the conventions and notation, an abstract Prolog machine is introduced. The machine is then extended to account for partial evaluation transformations. A partial evaluation system is implemented in the Qlog system according to the specification. Finally, a brief discussion of relations between logic, partial evaluation and research in rule-based systems follows.

2.0 PARTIAL EVALUATION: AN INTRODUCTION

An informal introduction to partial evaluation is presented here and illustrated with a simple example.

The goal of partial evaluation is to transform programs into more efficient ones. The improved efficiency is obtained at the expense of the generality of the programs. The restrictions on generality are usually introduced by setting

some global parameters, or by deciding upon values of certain formal arguments of the top-level procedure(s). The partial evaluation transformations aim to exclude checks shown to be unsuccessful, to open calls if the calls can be proved non-recursive, and to do some other improvements, all that with respect to the introduced restrictions. In partial evaluation the target language is the same as the source one. Thus, one can describe partial evaluation as a case of a sophisticated source-to-source compilation.

Much of the research in partial evaluation has been concerned with processing of programs written in weak-type languages. In such languages functions (or procedures) are responsible for data type checking. The prime attempt of partial evaluation is then a reduction in the number of run-time checks, since the regular compilers cannot reduce them.

Therefore, one would ultimately like the transformations to automate specializations of program data structures within a particular restricted domain. However, instead of providing type declarations in every procedure of the program one would think of an overall data structures propagation performed by a partial evaluation system. The last requirement is difficult to satisfy directly in a language like Lisp [McCarthy 66] but seems to be rather plausible in Prolog [Kowalski 74].

To illustrate the idea of partial evaluation and especially the requirement of the data structure propagation let us consider the following example of a general plus program. It can add numbers, vectors of numbers and matrices. The restricted application will be addition of integers. The propagated data structures are written in lowercase. (For the syntax of Qlog see the appendix.)

```
14_PP GEN-PLUS
([GEN-PLUS :X1 :X2 :RES]
    (OPER-2 + :X1 :X2 :RES))

([OPER-2 :OP (:TYPE . :A1) (:TYPE . :A2) :RES]
    (ACT :TYPE :OP :A1 :A2 :RES))

([ACT INTEGER + (:N1) (:N2) (INTEGER . :RES)]
    (QIPLUS :N1 :N2 :RES))
([ACT REAL + (:N1) (:N2) (REAL . :RES)]
    (QFPLUS :N1 :N2 :RES))
([ACT VECTOR + :V1 :V2 (VECTOR . :RES)]
    (ADDVECTS :V1 :V2 :RES))

([ADDVECTS NIL NIL NIL])
([ADDVECTS (:HV1 . :TV1) (:HV2 . :TV2) (:HRES . :TRES)]
    (OPER-2 + :HV1 :HV2 :HRES)
    (ADDVECTS :TV1 :TV2 :TRES))

([QIPLUS :I1 :I2 :RES]
    (IS :RES (IPLUS :I1 :I2)))

([QFPLUS :I1 :I2 :RES]
    (IS :RES (FPLUS :I1 :I2)))
```

And the call to the partial evaluator:

```
15_(GEN-PLUS (INTEGER . :X) (INTEGER . :Y) :RESULT)
Pruning...
    ([QIPLUS :I1 :I2 :RES]
        (IS :RES (IPLUS :I1 :I2)))
    ([ACT INTEGER + (:N1) (:N2) (INTEGER . :RES)]
        (QIPLUS :N1 :N2 :RES))
    ([OPER-2 :OP (:TYPE . :A1) (:TYPE . :A2) :RES]
        (ACT :TYPE :OP :A1 :A2 :RES))
    ([GEN-PLUS :X1 :X2 :RES]
        (OPER-2 + :X1 :X2 :RES))
Should I propagate data structures?... Yes

    ([QIPLUS :x :y :res]
        (IS :res (IPLUS :x :y)))
    ([ACT INTEGER + (:x) (:y) (INTEGER . :res)]
        (QIPLUS :x :y :res))
    ([OPER-2 + (integer . :x) (integer . :y) :result]
        (ACT integer + :x :y :result))
    ([GEN-PLUS (integer . :x) (integer . :y) :result]
        (OPER-2 + (integer . :x) (integer . :y) :result))

More?... Yes
FAILURE
```

Annotations

Interaction 15_: the top loop of the partial evaluator accepts Qlog forms as the interpreter does. We note that the actual parameters are only partially specified. They are known to be ***integers***.

The material displayed below ***Pruning...*** is a stack of procedure calls. The procedures are shown with their respective bodies. The growing end is the top of the stack.

The last (partially) evaluated call on the stack is the call to ***QIPLUS***. The call to ***IS*** (the assignment in Prolog DEC10 [Pereira 78]) cannot be evaluated since the arguments of IPLUS are variables. Such a call is "suspended" and the evaluation continues.

It is the last evaluation. The system displays the sequence of the calls and asks whether to ***propagate*** the data structures. The propagated program is displayed in the form of a stack. The propagated data structures are written in lower case letters. For example, we can observe that ***OPER-2*** gets the representation of the ***:TYPE*** variable set to ***integer***. Also the type in call to ***ACT*** becomes ***integer***. However, the variable name ***:result*** is not propagated beyond ***OPER-2***. This propagation transformation is a case of the ***forward*** data structure propagation.

Since in this computation there is no more alternative computation the evaluation is finished. The word ***FAILURE*** informs about it.

There are possibilities for other partial evaluation transformations too. We come back to this example in the sequel.

3.0 RELATED RESEARCH

Source-to-source transformations have been investigated by a number of researches. However, partial evaluation has not been yet formalized sufficiently. Almost all the literature is concerned with either special cases of transformations, or with informally described implementations of partial evaluation systems. In fact,

partial evaluation is to a large extent a craft technique existing in the folklore, especially in the weak-typed language communities like the Lisp one.

The idea of partial evaluation can be already found in Kleene's S-m-n theorem [Kleene 52]. An informal version of this theorem is often applied as a definition of partial evaluation. In the context of a language with functions one usually formulates it as follows.

Suppose that F is a function with the parameters $x_1, x_2, ..., x_n$. If the values of some of the parameters are known, say $x_1 = a_1, ..., x_k = a_k$, where a_i are constant values, a specialized version F' of F can be generated such that:

$$F'(x_{k+1}, ..., x_n) = F(a_1, ..., a_k, x_{k+1}, ..., x_n)$$

for all values of x_j, $j = k+1, ..., n$.

The definition, although reflects some of the intuitions concerned with partial evaluation, has several drawbacks. For example, it unnecessarily limits the known parameters to be constants.

Partial evaluation in connection with programming is first explicitly mentioned by Futamura [Futamura 71]. He also recognizes the relation between interpretation and compilation by means of partial evaluation. An early overview of partial evaluation applications can be found in [Beckman 76]. Other references to source-to-source transformations in general can be found in [Arsac 79], [Burstall 78], [Cheatham 79], [Kieburtz 81], [Loveman 77], [Standish 76].

A partial evaluation approach to program generation which uses Lisp and logic can be found in the work of A Haraldsson and E Sandewall on the PCDB system [Haraldsson 74], [Sandewall 71]. In this work partial evaluation was a tool used to generate procedures for maintaining a first order Predicate Calculus Data Base. The procedures were compiled into Lisp from the axioms stored in the data base. Much of that experience has then been incorporated in the Redfun system [Haraldsson 77, 80].

Some theoretical investigation has been done by Ershov and his group [Ershov 77, 80]. They call partial evaluation ***mixed computation***. An important characterization introduced by the group is the split of computation into a suspended and an executable part. They have also tried to set up a theoretical framework for partial evaluation. However, in that work mixed computation is not based on a semantics of a language and many formal considerations are somewhat fuzzy. It is also unclear how a mixed evaluation system could be implemented for languages of the type considered by them. There is, however one interesting work by [Ershov and Itkin 77] where partial evaluation is investigated for a small language with assignments, conditionals and goto's. For that language an operational definition is given which is then extended to account for a partial evaluator. Such an approach is similar to that taken in this paper. However, Prolog compared to that language is much a richer language since it has procedures.

Among larger applications of partial evaluation are works by P Emanuelson [Emanuelson 80] and B Ostrovsky [Ostrovsky 80]. Emanuelson implemented in Lisp an interpreter of a pattern match language. A compilation of programs written in the pattern match language to the regular Lisp is done by means of partial evaluation. Ostrovsky uses the same idea to generate a specialized parser from a general one and a given syntax.

In conclusion, although the partial evaluation research is quite active there is not any widely accepted formal definition of partial evaluation. This is also the reason why it has been necessary to begin with an informal introduction of partial evaluation. Consequently, we propose in this paper a definition of partial evaluation in Prolog with an aim to establish it.

4.0 A SPECIFICATION OF A PROLOG MACHINE

The method of the description chosen here is influenced by the Vienna Development Method [Björner and Jones 78], although it is not followed literally. In particular, we pattern our meta-language on Meta-IV. The reason for choosing this meta-language is twofold. First, it is a language designed to support formal definitions of high-level languages. Second, Meta-IV has been exclusively used to define imperative languages (eg. Algol-60 [Henhapl 78]) in the spirit of the denotational semantics [Gordon 79], [Stoy 77]. Therefore we would like to experiment and use Meta-IV in a definition of a purely applicative language in a operational style.

4.1 Conventions

To get started we set up a certain notation. If D_1 and D_2 denote sets (in the sequel the word ***domain*** is rather used), then:

$D_1\ D_2$	denotes the Cartesian product
$D_1 \rightarrow D_2$	denotes the set of all total functions from D_1 to D_2
$D_1 \xrightarrow{m} D_2$	denotes continuous finite-domain functions
$D_1 \xrightarrow{\sim} D_2$	denotes the set of all partial functions from D_1 to D_2
D_1^*	denotes the set of all (possibly empty) strings in D_1
D_1^+	denotes the set $D_1^* - \varepsilon$
$D_1 + D_2$	denotes the disjoint union

Some common operations on strings of objects are defined:

first. $x_1...x_n \triangleq x_1$
tail. $x_1...x_n \triangleq x_2...x_n$

We usually write ε for the empty sequence, except for the traditional logic programming $\square$ used to denote the empty clause and the end of proofs.

The identical substitution is denoted by id.

$\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\equiv$ are used for logical conjunction, disjunction, implication, negation and equivalence, respectively.

The symbols $\exists$, $\forall$ are used for the quantifiers.

The following function notations are used:

f. x	stands for f(x)
b → v_1, v_2	stands for **if** b **then** v_1 **else** v_2
cases v: v_1 → e_1 ... v_n → e_n	stands for case statement; if v_n is complementary to v_1, ..., v_{n-1} then it can be written T.
^	stands for concatenation
[d'/d]	stands for a simultaneous substitution

For the purpose of local definitions the **let** notation is introduced:

let x = ... **in** e(x) ≜ (λx. e(x))(...)

Instead of writing f : D_1 ... D_{n-1} → D_n its ***curried*** version is used and denoted with the same name:

f : D_1 → ... → D_n

Objects are described by an abstract syntax. Abstract syntax classes are defined as follows:

Name { = | :: } ***Rule***

By convention the first character of ***Name*** is in upper case. The choice between = and :: and the form of ***Rule*** dictate the kind of objects which are to be associated with ***Name***. If one defines strings of objects then = is used. For defining tuples (or trees) :: is applied. For a detailed description see the manual of Meta-IV [Jones 78].

While dealing with sets it is assumed that a predicate which tests for membership in a particular class of objects is implicitly defined for each ***Name***. Thus

is-***Name***. t ≡ t ∈ ***Name***

It is necessary to have a means of combining instances of trees into new trees and to be able to recognize and decompose them. Trees are built by ***constructor functions***. These functions have the property of uniqueness. The names of constructor functions are always formed by the prefix ***mk-*** and the name of the relevant abstract syntax rule.

In the below definitions short explanations to the mnemonic abbreviations are written in italics.

4.2 Syntactic Domains

x	:	Ide			*identifier*
t	:	Ter	=	Ide | Lit	*term*
A	:	Lit	::	Ide | Ter*	***literal***
H	:	Head	=	Lit	***head of a clause***
B	:	Body	=	Lit*	***body of a clause***
C	:	Cla	::	Head Body	***clause or procedure***
P	:	Pro	=	Cla*	***program***

4.3 Semantic Domains

conf	:	Conf	::	Pro Body Subs	***configuration***
dir	:	Direct	=	backward | forward	***direction***
stk	:	Stack	=	Conf*	
stt	:	State	::	Stack Direct	

4.4 Functions

1. θ, Δ	:	Subs	=	Ide	$\xrightarrow[m]{}$	Ter	***substitution***
2. unif	:	Lit	→	Lit	→	Subs + {nil}	***unification***
3. subster	:	Ter	→	Ter			***subs. on terms***
4. subsbody	:	Body	→	Body			***subs. on bodies***
5. subscla	:	Cla	→	Cla			***subs. on clauses***
6. name	:	Lit	→	Ide			***literal's name***
7. is-def	:	Lit →	Pro	→	Conf	+ {break, failmatch}	
8. is-uni	:	Lit →	Pro	→	Conf	+ {failmatch}	
9. apm	:	Pro	→	State	⇒	State	***Abstract Prolog Machine***

4.5 Definitions of the Functions

1. **Note.** In fact every θ is a total function defined on ***Ide***, but since D(θ) = {x : θ. x ≠ x} is always finite, θ is defined as a function on D(θ). Consequently the ***id***entical substitution is a function defined on the empty domain.

2. ***Unif*** is defined in [Robinson 65]. Its result is a ***subs***titution which, when performed on the arguments, makes them equal. In its implementation it is essential that the names of variables (which are defined in the concrete syntax) are renamed prior to unification. It is true that is-subs(unif(A, H)) = is-subs(unif(H, A)), but it is not true that unif(A, H) = unif(H, A) for any literals A and H. For the purpose of the theorems' proofs it is assumed that if an identifier in A is unified with an identifier in H then A's identifier is substituted for H's.

3. subster. θ. t ≜

   is-Ide. t → θ. t,
   let mk-Lit(x, tt) = t **in** mk-Lit(x, subster. θ. tt)

4. subsbody. θ. B ≜

   B = [] → ε,
   subster. θ. (first. B)^subsbody. θ. (tail. B)

5. subscla. θ. mk-Cla(H, B) ≜
   mk-Cla(subster. θ. H, subsbody. θ. B)

Note. In order to make the definitions less loaded with symbols the names of the substitution functions are omitted in the sequel. For example, it is written θ. B rather than subsbody. θ. B

6. name. mk-Lit(x, tt) ≜ x

7. is-def. A. P ≜

```
.1   P = ε                          → break,
.2      let mk-Cla(H, B) = first. P in
.3      name. H = name. A     → is-uni. A. P,
                                 is-def. A. (tail. P)
```

Annotations

.1 if the program P is empty then ***break***. It means that there is no definition for A in P. This case is significant in the definition of ***apm*** if the direction is ***forward***.

.2 else, select the first ***cla***use in the program P and

.3 if the head H has the same ***name*** as A then call ***is-uni*** else, check if there ***is*** a ***def***inition for A in the ***tail*** of P.

8. is-uni. A. P ≜

```
.1   P = ε                          →  failmatch,
.2      let mk-Cla(H, B) = first. P in
.3      name. H = name. A  →
.4            (let  Δ = unif. A. H in
.5            is-subs. Δ      →  mk-Conf(tail. P, B, Δ),
                 is-uni. A. (tail. P) ),
.6          is-uni. A. (tail. P)
```

Annotations

.1 if the program is empty then ***failmatch***. This branch is reached only if there are definitions for A (at least one) but none of them unifies with A.

.2 else, select the first ***cla***use in the program P and

.3 if the respective ***name***s are equal then

.4 let Δ be the result of the call to ***unif*** and

.5 if Δ is a substitution, that is the unification of H and A exists, then make a ***conf***iguration which contains: the not yet used rest of the program (cf .10 and .16 in the definition of ***apm***), the body B of the selected clause, and the (increment) unification substitution Δ; otherwise call ***is-uni*** for A and for the ***tail*** of P.

.6 else, call ***is-uni*** for A and for the ***tail*** of P.

9. apm. P. mk-State(stk, dir) ≜

```
.1   let mk-Conf(P_I, B_I, θ_I) = first. stk in
.2   cases dir:
.3      forward →
.4         (B_I = []    →  mk-State(tail. stk, backward),
.5            cases is-def. (first. B_I). P:
.6               break      →  undefined
.7               failmatch  →
                       mk-State(tail. stk, backward)
.8               mk-Conf(P', B', Δ)          →
.9      mk-State(mk-Conf(P, Δ. (B'^tail. B_I), Δ. θ_I)^
.10           mk-Conf(P', B_I, θ_I)^tail. stk, forward) )
.11     backward →
.12        cases is-def. (first. B_I). P_I:
.13               mk-Conf(P', B', Δ)          →
.14     mk-state(mk-Conf(P, Δ. (B'^tail. B_I), Δ. θ_I)^
.15           mk-Conf(P', B_I, θ_I)^tail. stk, forward)
.16               T   →  mk-State(tail.stk, backward)
```

Annotations

.1 The top ***conf***iguration of the ***stack*** is selected

.2 If the ***dir***ection is

.3 ***forward***, that is the last step was successful, then

.4 If the ***b***ody is empty then pop the top element of the ***stack*** and make a new ***state*** with the ***backward*** direction. This is the case we are most interested in. The definition of semantics given in the sequel refers to it, and especially to the current substitution θ_1.

.5 otherwise, call ***is-def*** (selection and unification) for the first element of the body. The selection is to be made from the global program P. If the value is:

.6 ***break***, then ***apm*** is ***undefined***. This must be so, if the calls to the undefined procedures are to be catched.

.7 ***failmatch***, then pop the top element of the ***stack*** and make a new ***state*** with the ***backward*** direction. The unification was unsuccessful and a backtrack occurs.

.8 a ***conf***iguration then

.9 make a new ***state*** such that two configurations are concatenated to the ***tail*** of the ***stack***. The top configuration's elements are: the global program P, the Δ substituted concatenation of a new ***b***ody B' with the ***tail*** of the previous body B_1, and the substitution Δ added to θ.

.10 The top but one configuration is unchanged except that the previous program P_1 is replaced by P'. It is done to memorize the rest of the program not yet used by ***is-def***. This program is to be used if due to a failure a next choice must be made. The direction is ***forward***, which tells us that the step was successful.

.11 ***backward***, that is the previous step was a failure, then

.12 call ***is-def***. Now the selection is to be made from the local program P_1 stored on the stack. It memorizes the rest of the program as left by the previous call to is-def (compare with the comment on .10). If the value return by the (current) call to ***is-def*** is:

.13 a ***conf***iguration, then

.14, .15 perform as in .9 and .10

.16 otherwise pop the top element of the ***stack*** and make a new state with the ***backward*** direction. We can see that since the machine is in the backward direction (cf .11) the break and failmatch values returned by ***is-def*** cause the same action because there is either no more clause to select (but there ***was*** some) or none of the selected ones can be unified.

A few comments on the way ***apm*** is defined may be useful here.

We have chosen to have a program P as a parameter since it makes it easy to extend the definition to account for side-effect procedures which change programs (eg assertions). It is also convenient to have this parameter explicit when the semantic function (defined in the sequel) is applied to different programs.

A superficial analysis of the definition of ***apm*** may lead to a conclusion that it is unnecessarily complicated. The usual objection is as to why even the previous top configuration

is modified while a new one is pushed on the stack (.15). The explanation is that it must contain the "history" of selecting a clause from the program. (And at the same time such a solution was preferred to the use of a side-effect which could be introduced by the ***is-uni*** procedure.) Should a backtrack occur (ie popping an element from the stack) then a new trial must be made in that very program. An alternative approach which carries the history forward and stores it in the top configuration requires popping two configurations while backtracking. The reader is encouraged to write such a version of ***apm***.

The ***apm*** function can be simply made recursive as well. Informally speaking, it is sufficient to encapsulate right sides of the conditionals (except undefined (.6)) in a call to apm (and change its type definition) and add a check if the stack is not empty.

4.6 Semantics

Definition 1 ***Computation Sequence***

Given the definition of ***apm***, a ***computation*** (sequence) of a literal A determined by a program P is defined recursively as follows:

1^0 stt_0 = mk-State(mk-Conf(P, A, id), forward)

2^0 If apm. P. stt_i is defined then stt_{i+1} = apm. P. stt_i

Before we proceed to the definition of the semantics, an additional function sem^{α} is introduced.

Definition 2

Sem^{α} : Pro → Lit → $State^{\infty}$

where sem^{α}. P. A is the longest computation of A determined by P.

Definition 3 ***Semantics***

The semantics ***sem*** of A determined by P is defined in the following way:

1^0 sem : Pro → Lit → $Subst^{\infty}$

2^0 Let A and P be given and established. From sem^{α}. P. A a subsequence is selected which contains all the elements such that:

stt_i = mk-State(mk-Conf(P_i, B_i, θ_i)^stk_i, dir_i), where B_i = □, dir_i = forward.

Then

sem. P. A ≙ {θ_{i1}, θ_{i2}, ...}

Such a subsequence is called a ***successful computation sequence*** (for A determined by P) and abbreviated s.c.s.

5.0 PARTIAL EVALUATION

Informally speaking, a partial evaluation machine is an extended abstract Prolog machine which, while computing, labels those clauses (parts of a program) which contributed to a successful computation. Those labelled clauses form a subsequence P' of P such that the denotation of A in P is preserved in the new program P'. P' is called a ***pruned*** version of P (with respect to A) and will be subject to further transformations.

In partial evaluation one also has to take a different approach to the so called under-defined programs. The partial evaluation machine should be total. However, instead of popping the stack and switching to the backward state in the case of undefined procedures (ie. backtracking, as it was the case in the regular interpreter) we rather assume that a definition will be provided later, and thus the forward state is preserved. These concepts are formalized in the following way.

The essential change is in the definition of ***configuration***. It is now defined as follows:

con : Conf = Pro Body Subs Cla

The change induces the following natural modification in the definition of ***is-uni***. (The ***is-def*** function's text remain unchanged, although the function is defined on a new domain).

```
8. is-uni. A. P ≙
.1    P = ε                          →  failmatch,
.2       let mk-Cla(H, B) = first. P in
.3       name. H = name. A  →
.4              (let Δ = uni (H, A) in
.5              is-subs. Δ      →
                    mk-Conf(tail. P, B, Δ, mk-Cla(H, B)),
                    is-uni. A. (tail. P) ),
.6          is-uni. A. (tail. P)
```

The more significant change is in the definition of ***apm***. In the case of partial evaluation this function is ***total***.

```
apm     :   Pro    →  State  →  State
                      Abstract p.e. Prolog Machine
```

```
9. apm. P. mk-State(stk, dir) ≙
.1    let mk-Conf(P1, B1, θ1, C1) = first. stk in
.2    cases dir:
.3       forward →
.4          (B1 = □    →  mk-State(tail. stk, backward),
.5              cases is-def. (first. B1). P:
.6                  break                    →
.6a      mk-State(mk-Conf(P, tail. B1, θ, ε)^
.6b            mk-Conf(ε, B1, θ1, C1)^tail. stk, forward)
.7                  failmatch                →
                          mk-State(tail. stk, backward)
.8                  mk-Conf(P', B', Δ, C')       →
.9       mk-State(mk-Conf(P, Δ. (B'^tail. B1), Δ. θ1, C')^
.10            mk-Conf(P', B1, θ1, C1)^tail. stk, forward) )

.11      backward →
.12         cases is-def. (first. B1). P1:
.13                 mk-Conf(P', B', Δ, C')       →
.14      mk-state(mk-Conf(P, Δ. (B'^tail. B1), Δ. θ1, C')^
.15            mk-Conf(P', B1, θ1, C1)^tail. stk, forward)
.16                 T   →  mk-State(tail.stk, backward)
```

Annotations

We annotate below only the significantly modified part of the ***apm's*** definition.

.6 ***break***, then a new state is created such that the elements of the top configurations are: the global program P, the ***tail*** of previous body, the previous

substitution, and the empty clause (which labels no clause of the program); in the configuration next to the top the *program* is empty as to denote the fact that no more choices can be made there; the description of the other configurations does not substantially differ from the previous one. Such a top configuration is called ***suspended***.

.8, .9, .10, .13, .14, .15 are modified as to reflect the change in the definition of the configuration.

Similarly, an obvious modification in the definitions of the ***sem*** functions is introduced to account for that change. Namely,

1^0 $\quad stt_0$ = mk-State(mk-Conf(P, A, id, ε), forward)

We can now turn to the partial evaluation theorems. In order to simplify the proofs of this chapter we assume that:

sem. P. A = $\{\theta_{i1}\}$

which says that there is only one s.c.s. for A determined by P.

The proofs of the theorems in the sequel require the following lemma characterizing unification:

Lemma 1

If θ. A = θ. H and A' = θ'. A, where θ, θ' $\in$ Subs, and A, H $\in$ Lit, then

$\exists$ ($\Delta_1 \in$ Subs) (Δ_1. A' = Δ_1. H) $\equiv$ $\exists$ ($\Delta_2 \in$ Subs) (Δ_2. A' = Δ_2. θ. A)

We can now state our theorems. Let us put sem. P. A = $\{\theta\}$. From the definition there exists a s.c.s. Let P_A' be the subsequence of P consisting of those $C_i \in$ P which appear in the s.c.s.

Theorem 1 ***Pruning, or dead code elimination***

P_A' preserves the semantics of A i.e.

sem. P. A = sem. P_A'. A

Moreover, the semantics is preserved for every instance A' of A, that is:

sem. P. A' = sem. P_A', A'

where A' = θ'. A.

Proof. It is sufficient to prove that the substitutions determined by both programs are equal. In order to do so one can prove a stronger fact that the respective s.c.s.'s are equal. (Perhaps with the exception for *Program* coefficients). The proof is by contradiction. For the lack of the space we omit this somewhat lengthy, although simple proof. It can be found in [Komorowski 81a]. □

The program P_A' is called the ***prunned*** version of P with respect to A. Usually, the A index is omitted.

Such a pruned program is the subject of the next theorem called the memorization theorem. Let us put sem. P. A = $\{\theta\}$ where P = P_A' and assume that A' = θ. A for some θ' $\in$ Subs. We assume that the instantiation is the proper one, that is it substitutes identifiers by terms which themselves are not identifiers. Otherwise it would be only a renaming of identifiers, which is not of interest here.

Theorem 2A ***Memorization***

sem. P. A' = θ'. θ

where the application of substitution θ' is understood as follows:

Given θ. A and θ' perform all its substitutions on θ. A; if there is any substitution which cannot be performed because the respective identifier does not appear in θ. A then the right side is undefined and A' does not have a semantics determined by P.

Proof. Let us denote the (only) s.c.s. for A

stk = (P, [], θ, C_{in})...(P_2, B_2, Δ_2. id, C_{i2}) (P_1, A, id, ε)

and consider a s.c.s. for A'

stk' = (P, [], θ'', C_{in}') ... (P_2', B_2', Δ_2'. id, C_{i2}') (P_1', A', id, ε)

If such a sequence exists then it must be

stk $|_{Cla}$ = stk' $|_{Cla}$

(ie only the ***cl****ause* coefficients are compared)

Namely, let us consider the second configuration of ***stk*** and ***stk'*** and suppose that $C_{i2} \neq C_{i2}$'. From the definition of a s.c.s. there is

Δ_2'. A' = Δ_2. H_{i2}'

but since A' is an instance of A, ie A' = θ'. A then of course there exists Δ_2'' $\in$ Subs such that

Δ_2''. A = Δ_2''. H_{i2}'

But it contradicts the assumption that there is only s.c.s. for A, thus C_{i2} = C_{i2}' follows. And consequently, Δ_2'. H_{i2} is an instance of Δ_2. H_{i2} where

Δ_2'. H_{i2} = θ'. Δ_2. H_{i2} (cf note on the definition of ***unif***)

Thus in particular

Δ_2'. B_{i2} = θ'. Δ_2. B_{i2} and θ'. (first. B_2) = first. B_2'

By repeating this reasoning for an arbitrary k = 2, 3, ..., n it is obtained that

C_{ik} = C_{ik}' and θ'. (first. B_k) = first. B_k'

Consequently,

θ. θ' = θ'. Δ_n. Δ_{n-1}. ... Δ_2. id = Δ_n'. Δ_{n-1}'. ... Δ_2'. id □

Informally, one can say that the semantics of an instance of a literal is the instance of the semantics of the literal if such exists.

The memorization, although epistemologically very interesting, has a limited application in the case of partial evaluation. Its epistemological value is that it is sufficient to store the result of sem. P. A, ie the substitution, and then instead of computing sem. P. A' one only has to find the unifier of A' and θ. A. Note that there are unification algorithms linear in time! (eg [Martelli 76]) Unfortunately, such a global memorization has among others a disadvantage of neglecting the suspended computations, ie the configurations created in the case of ***break*** (cf .6, .6a, .6b). As soon as definitions for those literals are provided the global substitution shall generally be changed.

(Memorization may also lead to a large storage requirement since there usually is a large number of semantic substitutions.) In order to give the theorem a more practical value it is improved as to account for such cases. Namely, it is proposed to distribute the component semantic substitutions over the program instead of storing its global result. The idea is formalized as follows.

As previously, it is assumed that sem. P. A = {θ} and P = P_A'. From the definition of ***sem*** there exists a s.c.s. for A determined by P:

stk = (P, □, θ_n, C_{in}) ... (P_2, B_2, θ_2, C_{i2}) (P_1, A, id, ε)

where $\theta_n = \theta$ and $\theta_k = \Delta_k . \theta_{k-1}$, k = 2, ..., n.

Let us also assume that $C_{ij} = C_{ik} \Rightarrow j = k$, for j, k = 2, ..., n, and let $P_A'' \triangleq \theta . P$ where θ. P is defined as follows:

$\theta . P \triangleq [\Delta_2 . C_{i2} / C_{i2}, ..., \Delta_n . C_{in} / C_{in}] . P$

Then the following theorem holds:

Theorem 2B ***Data Structure Propagation***

sem. P. A = sem. (θ. P). A

Proof. The theorem is a consequence of the fact that if unif(A, H) = Δ ∈ Subs then there also is unif(A, Δ. H) = Δ. The s.c.s. for A as determined by θ. P is:

stk'' =
(P'', □, θ_n, $\Delta_n . C_{in}$)...(P_2'', B_2, θ_2, $\Delta_2 . C_{i2}$) (P_1'', A, id, ε)
□

The theorem generalizes for A' = θ'. A in a natural way. It is enough to repeat arguments from the proof of the memorization theorem to obtain the following corollary.

Corollary

sem. P. A' = sem. (θ. P). A'

This corollary accounts for the so called ***forward*** data structure propagation since the substitutions generated in, say, k-th configuration have no effect on an i-th configuration where i < k, but can have it on any l-th configuration where l > k. This situation can be compared to an inference of the admissible data types (represented as data structures) where the types carried by a call filter the types of the procedure. (We return to the issue of inferencing data types in the sequel.) Moreover, the unification mechanism of variable binding has a property of substituting variables backwards. It is known in Prolog programming under the name ***logical variable***. We are going to use this unique property in the forthcoming theorem which accounts for ***backward*** data structure propagation. Before we embark on that theorem two additional notions are introduced.

Definition 4 ***Deterministic Procedure***

A clause C = mk-Cla(H, B) ∈ P is called a ***deterministic procedure*** in a program P iff

∀ (C' = mk-Cla(H', B') ∈ P) (name. H' = name. H ⇒ C' = C)

Definition 5 ***Deterministic Call***

A ***deterministic call*** in a procedure C = mk-Cla(H, B) ∈ P is a literal A such that A ∈ B and there exists a deterministic procedure C' ∈ P such that C' = mk-Cla(H', B') where name. H' = name. A.

Theorem 3 ***Backward DS Propagation***

Let P = P' and A_k be a deterministic call in a procedure C_i = mk-Cla(H_i, B_i) ∈ P where $B_i = A_1 \ldots A_k A_{k+1} \ldots A_m$ and let C_j = mk-Cla(H_j, B_j), i ≠ j, be the corresponding deterministic procedure. If C_i is replaced by the clause

$C_i' = \theta_A .$ mk-Cla(H_i, B_i')

where

$\theta_A . A_k = \theta_A . H_j$

and

$B_i' = A_1 \ldots B_j A_{k+1} \ldots A_m$

then the semantics of any literal A determined by $P' = C_1 \ldots C_i \ldots C_n$ is equal with the semantics determined by $P''' = C_1 \ldots C_i' \ldots C_n$

Proof. The proof follows immediately from the definition of semantics and s.c.s.
□

We have chosen P = P' and not removed C_j from the program in order to point out that P''' is now eligible for pruning. Let us also notice that since C_j has not been removed the semantics of ***any*** literal determined by P''' is the same as determined by P.

Definition 6 ***Partial Evaluation***

The introduced transformations: pruning, forward data structure propagation and backward data structure propagation are called ***the partial evaluation transformations***.

6.0 IMPLEMENTATION

The specification of the abstract Prolog machine is directly implementable in, for example, Lisp. One can just compile it into Lisp in an "off the cuff" manner, or use a Lisp compiler translating a subset of Meta-IV into Lisp [Ollongren 80]. For the details of the first implementation see [Komorowski 81a]. However, since we do not refine this specification, such implementations must be inefficient (there is no provision for, eg structure-sharing [Boyer and Moore 72]) and they are not well-integrated into a programming environment. Therefore, it is preferred to overcome these drawbacks, even for the price of a very formal strictness of an implementation.

Let us also note that the usual "build-on-top" approach to an implementation of an embedded system - known from Lisp or Prolog experience - is not good either. It also partly cuts an access to the programming environment and severely worsens the efficiency because of a double interpretation.

There is, however, a possibility to directly implement the specified extensions in a Prolog interpreter provided a convenient access to its source. The Qlog system [Komorowski 79, 80, 81] (see also the appendices), which is an advanced programming environment for Prolog in Lisp, makes an introduction of the specified extensions rather

straightforward. The basic extension to the Lisp code of the Prolog interpreter requires just two lines of code and the other parts which implement storage of a transformed program, an interface to the user, the opening theorem, etc, take less than 50 lines. In other implementations of Prolog such changes are usually impossible because the source code is either unavailable or written in a language which discourages any experiments with the source. The price to pay is that it is impossible to formally prove the correctness of such an implementation, since the Qlog system implements a Prolog machine different from the specified one.

Finally, we note that two very characteristic properties of Prolog - unification and backtracking - significantly contribute to the relative ease of defining and implementing a partial evaluator.

Unification offers two things. One is the ability of an errorless handling of uninstantiated variables (in other languages one would say unbound), the other is the possibility of substituting such variables later in the computation, (known as the logical variable).

One can also add here that due to the weak-typing of unification there are no constraints on the type a variable can accept. It contributes to a worse efficiency, but it also provides a greater flexibility of a program, which is especially important in the development phase. And eventually, it is the unification based binding mechanism which makes it possible to define forward and backward data structure propagation.

7.0 SAMPLES OF PARTIAL EVALUATIONS

In this chapter we present samples of interactive sessions with a partial evaluation system. These are excerpts from script files. The user's input is written in bold print and the comments added later to the files are written in italics. The main feature of the partial evaluation system is that it appears to the user as the regular interpreter.

7.1 Open-macro Example

In the following example the data structure propagation is performed by an application of the opening theorem.

The procedure ***arch*** checks if its argument is an arch. As a result of providing data access procedures, the initially generic type procedure ***arch*** is specialized to a specific abstract data type. First, the procedures are pretty printed, (hence the abbreviation PP).

15_PP ARCH

```
([ARCH :X]
    (PIER :U)
    (ARCHITRAVE :V)
    (PIER :W)
    (ON :U :V)
    (ON :W :V)
    (TOP :X :V)
    (LEFT :X :U)
    (RIGHT :X :W))
```

ARCH

16_PP RIGHT

```
([RIGHT (A :P1 :AR :P2) :P2])
```

RIGHT

17_PP LEFT

```
([LEFT (A :P1 :AR :P2) :P1])
```

LEFT

18_PP TOP

```
([TOP (A :P1 :AR :P2) :AR])
```

LEFT

The call is:

19_(ARCH :Z)

Pruning...

```
    ([RIGHT :X :W])
    ([LEFT :X :U])
    ([TOP :X :V])

    ([ARCH :X]
        (PIER :U)
        (ARCHITRAVE :V)
        (PIER :W)
        (ON :U :V)
        (ON :W :V)
        (TOP :X :V)
        (LEFT :X :U)
        (RIGHT :X :W))
```

Pruning has no effect, nor will the forward propagation have

Should I propagate data structures?... Yes

```
    ([RIGHT (a :p1 :ar :p2) :p2])
    ([LEFT (a :p1 :ar :p2) :p1])
    ([TOP (a :p1 :ar :p2) :ar])

    ([ARCH :z]
        (PIER :U)
        (ARCHITRAVE :V)
        (PIER :W)
        (ON :U :V)
        (ON :W :V)
        (TOP :z :V)
        (LEFT :z :U)
        (RIGHT :z :W))
```

More? Yes
FAILURE

The ***forward*** data structure propagation gives no interesting result since the top-level restriction is very weak: it is a variable.

On the other hand an application of the opening of the deterministic calls to the ***right***, ***left***, and ***top*** procedures results in this program:

```
([ARCH (a :p1 :ar :p2)]
      (PIER :p1)
      (ARCHITRAVE :ar)
      (PIER :p2)
      (ON :p1 :ar)
      (ON :p2 :ar))
```

Although the opening in the case of Prolog partial evaluation seems similar to the opening in the regular compilers, it is in fact more effective than, for example, in Interlisp [Teitelman 78] or Prolog DEC10 [Pereira 78]. It flattens the hierarchy of procedure calls and, this is unique, it lifts to a higher level the data type checks of the opened procedure. Thus the efficiency benefit has two sources. Moreover, in this way the data structures are inferenced from lower-level procedures. The opening and forward propagation are an alternative to declarations in weak-typed language with a pattern-matching variable binding.

7.2 Gen-plus Example Revisited

One more transformation can be performed on the example introduced in Chapter 2.0: the opening of the ***deterministic*** calls. The result is:

```
([GEN-PLUS (INTEGER . :X) (INTEGER . :Y) (integer . :res)]
      (IS :RES (IPLUS :X :Y)))
```

The opening of the call to *ACT* is essential here. Consequently, ***(integer . :res)*** is propagated up to the top call and the type of the result can be determined. One can repeat here the same comment as in the above example on the ***arch*** procedure.

7.3 A Large Example

A test on a medium size Prolog program which was a part of the author's master thesis [Komorowski 76] has been performed. The program implements a QA system with an interface in a natural-like language. It accepts input about family relationships. A sample session can be as follows:

>Henry is the father of Jan.
OK, continue...

>George is the father of Henry.
OK, continue...

>Who is a grandfather of Jan?
A grandfather of Jan is... George.

>Violette is the mother of Jan.
OK, continue...

>Who is a daughter-in-law of George?

A daughter-in-law of George is... Violette.

...

The system has 56 procedures of 93 clauses where 10 procedures process input and are common for both updating and querying. Three experiments have been performed. The first one has been an extraction of this part of the code which is responsible for querying, the second one - for updating, and the last one has been an unrestricted partial evaluation of the entire program.

According to the expectations, the first partial evaluation has produced a program of 53 procedures where in total 11 clauses have been removed. It is a result comparable to a manual extraction which, although possible, is almost always erroneous. Above that, two procedures have been found deterministic and simplified.

The second partial evaluation has given a program consisting of 32 procedures where 34 clauses have been removed and six deterministic calls have been found.

The third result is rather surprising. The partial evaluation has found some clauses never used by the system which were left in the file by mistake and were the dead-code indeed. Moreover, it actually has improved the hand-coded program! Eight deterministic calls have been found and simplified. The performed simplifications are in fact a good practice of an experienced Prolog programmer who prefers unification to a procedure call.

Comments on the implementation

Ease of implementation: The emulation of partial evaluation in the Qlog system requires some knowledge about the internal structures of the interpreter as to how and where make the incisions. It is a certain disadvantage. However, it is still a conceivable task even for a user unacquainted with the interpreter, since an access to a Lisp programming environment is provided.

A significant advantage of the emulation approach is the access to the internal structures of Qlog. For example, one can separate variable names from their bindings, change user's interface, or introduce other modifications in the standard interpreter. All that is a practice in Lisp/Qlog but unknown or rather hardly possible in other implementations of Prolog.

Integration in a programming environment: The emulation is also advantageous with respect to the integration in a programming environment. In fact, it is the very same interpreter which after a few extensions also acts as a partial evaluator. Thus all the programmer tools which are available for a user of the Qlog interpreter can serve a user of the partial evaluator (eg break, trace, editor, file-librarian, etc). This is basically the same principle as was used to obtain the Qlog programming environment from an existing Lisp's [Komorowski 81].

8.0 REPRESENTING KNOWLEDGE ABOUT PROGRAMS

A Prolog program can be seen as a specification of some rules, and a Prolog interpreter - as an interpreter of a rule based system. This observation suggests that research in rule-based systems can contribute to research concerned with Prolog. Our special interest is in the so called meta-rules systems where meta-rules aim at improving the efficiency of execution eg [Davis 80], [Gallaire 80], [Dincbas 80].

Unfortunately, the limit of the space makes it impossible to elaborate this issue. Let us only mention that one can write rules (which are Prolog programs themselves) which can control execution. Such rules are generalizations of the recommendation lists (cf [Sussman 72]). It is also possible to write rules which can perform transformations of programs in the partial evaluation style, ie transformations which use current restrictions. More details can be found in [Komorowski 81a].

It is believed that such an approach to improved efficiency is better than annotations of Prolog programs used in IC-Prolog [Clark 80]. The meta-level approach to regular interpretation and partial evaluation has the following advantages:

- it provides a clear, unannotated representation of object programs;
- it provides a better modularization of programs and a means to express their static and dynamic optimizations;
- it amplifies the power of search by using inference for controlling it, and thus helps to control the exponential explosion;
- it is perspicuous in that it allows one to study object level and strategic meta-level programming separately.

CONCLUSIONS

In the introduction it was pointed out that partial evaluation can be considered to be a kind of a source-to-source compilation. However, the presented formalization, and even more the implementation, show that it is rather a case of a modified interpretation. This observation was first made by Futamura and our experiment confirms it strongly. Ershov and Itkin also obtain a similar result, however for a much weaker language.

The introduction of the implementation-oriented specification of Prolog was a software engineering requirement since the existing semantics of Prolog have been either not implementation-oriented and idealized [Kowalski 74], [Emden 76], [Apt 81], or unacceptable for a formal treatment (ie machine implementations of the language). However, the logic-related semantics of logic programming provide several useful paradigms which can help in a better understanding of the foundations and laws of partial evaluation.

Syntax and Semantics. The syntax and semantics are very strongly correlated in logic. In the extreme case of a complete system, the syntactic consequence is equivalent with the semantic one. Thanks to this relation any syntactic change in a program, ie. theory, can have an immediate and predictable result in the semantics.

First Order Logic is monotonic. If, in some axiom of a theory, a term is substituted for a variable, then the new theory has a non-greater set of consequences. It is stressed that any term, not only a constant, can be substituted. This fact provides a kind of a "smooth" reduction of the set of the consequences. Speaking informally, there is a lattice of terms ordered by specificity. For example, ***:x***, ***(a :y)***, ***(a :p1 :ar1 :p2)***, ... or ***(a :x)***, ***(a (t :x1 :x2))***, ***(a (t (t1 :y1)(t2 :y2)))***, ... In the model-theoretic sense such substitutions lead to a non-greater number of models, and can often give a smaller one. This property legitimates the ***data structure propagation*** theorems and provides a generalization of the definition of partial evaluation patterned on Kleene's theorem. The refinement allows arbitrary terms, not only constants, to carry the restrictions. Partial evaluation, thus defined, operates on abstract data types represented by terms.

The ***pruning*** theorem can be justified by the following fact. If a set of axioms and a restriction on a class of theorems to be proved from that set are given, then the irrelevant axioms can be ***pruned***; where irrelevant means: not appearing in the syntactic proof.

In summary, partial evaluation aims at a better efficiency by an elimination of axioms which are irrelevant to a restricted theory, and by specialization of axioms if their restricted domain can be determined. The efficiency is improved after partial evaluation because fewer axioms are tried while searching for a proof, and because the instantiated axioms contribute to faster unification. At the same time the partial evaluation transformations are proofs about possible data types represented as data structures and acceptable by a theory (ie program).

FUTURE RESEARCH

Some directions of future research following up issues addressed in this paper are suggested here.

The standard specification of Prolog. Since there has been no mathematically acceptable ***and*** implementation-oriented definition of the Prolog language, we propose the definition introduced here to be established as the standard.

Further development of the specification. The extensions to the definition which account for side-effect, input/output and other system procedures are straightforward. In particular, the specification of the so called ***cut*** is a matter of tagging a ***configuration*** which introduces a ***body*** containing a ***cut***. Then an execution of a ***cut*** is merely a replacement of all ***pro***gram coordinates in the ***configuration***s of the stack by the empty clause until a tag is found (searching from the growing end of the stack). This specification may be further refined to include structure sharing technique [Boyer and Moore 72], incremental indexing and other optimizations. Eventually, the specification may serve as a basis for the development of a compiler.

Partial evaluation in other languages. Although it is not possible to directly apply the same technique to other languages, a basis is established for the definition and implementation of partial evaluation transformations in other languages. It is also certain that any implementation will require designing several tools which are granted for free in Prolog, eg theorem proving. Probably Prolog itself could partly support systems for other languages. The experience from this research and from the work by Ershov and Itkin shows that partial evaluation should be based on an operational definition.

Problems to be solved. One would like to see better tools for limiting the growth of a partially evaluated program. At

present there are only almost manual means for restraining the growth. A promising solution is the application of meta-rules.

Future program transformation. Partial evaluation is a case of program transformation. It attempts to improve efficiency of program execution by eliminating run-time checks and performing as much computation in advance as possible. However, it does not essentially modify algorithms. We believe that general program transformation systems should include partial evaluation as one of the tools and be accompanied by a number of other transformations. Several optimizing transformations may achieve better results if applied on partially evaluated programs.

Partial and lazy evaluation. Partial and lazy evaluation should be examined carefully. It seems that they are related in an interesting way.

Partial evaluation and an automatic generation of semantics-directed compilers are another promising area. This research should be pursued very strongly.

Acknowledgements

I have had fruitful interactions with Andrzej Blikle, Anders Haraldsson and Erik Sandewall. I have also received helpful suggestions from many colleagues at the Software Systems Research Center in the course of the design and writing my thesis of which parts are included in this paper. Dines Björner, Jan Maluszynski, Jörgen Fisher Nilsson, Alexander Ollongren, Jerzy Tiuryn and Joseph Stoy have read manuscripts of the thesis in various stages and contributed with several comments. Gerald Jay Sussman has encouraged me to work on partial evaluation and provided me with a good deal of enthusiasm.

The research has been sponsored by the Swedish Board for Technical Development under the grant Dnr 80-3918 and by the Swedish Natural Science Research Council under the grant F 4170-100.

APPENDIX A The Qlog Programming Environment

The Qlog programming environment is an implementation of the Prolog language in a portable Lisp. It exists in main Lisp dialects: Interlisp, Maclisp, and Lisp Machine Lisp. Throughout this thesis the Interlisp [Teitelman 78] version of Qlog is used. The Qlog interpreter is equally efficient as the Prolog DEC-10 interpreter. The new system provides a unique Prolog programming environment in a class of the host Lisp. By virtue of the implementation technique called *functional embedding* Qlog inherits most of the major components of the Lisp programming environment at very low cost, and obtains a high quality programming environment. Some of the inherited facilities are: breakpoints and procedure traces, editor, file librarian, Interlisp's spelling correction, and history list. Interfacing the existing Interlisp facilities to Prolog required 30 to 50 times less code than the Lisp facilities themselves require. Some new facilities were developed, not found in any other Prolog, to provide appropriate displays in breakpoints of the control and data in a pattern-matching, backtracking language.

APPENDIX B The Concrete Syntax of Qlog

The concrete syntax of Qlog is written in a modified BNF. <Lispatom> including the special atom nil, and <Lispnumber> are the objects defined in a host Lisp of Qlog. We leave them undefined here. The dot used in the definition is that of Lisp.

```
<Ide1>      :: =  <Lispatom>
<Ide>       :: =  <Ide1> | <Lispnumber>
<Variable>  :: =  :<Ide>
<Ter>       :: =  <Variable> | <Ide> | (<Ide1> . <Ter>)
<Lit>       :: =  (<Ide1> . <Ter>)
<Head>      :: =  [<Ide1> . <Ter>] | (<Ide1> . <Ter>)
<Body>      :: =  (<Lit>) | (<Lit> . <Body>)
<Cla>       :: =  (<Head> . <Body>)
<Pro>       :: =  {<Cla>}
```

Note. The "[" and "]" are a syntactical sugar used in the Qlog-Interlisp pretty printer. Both representations are equal to the standard input routines. In other Lisps those brackets might be "<" and ">", respectively. The "{" and "}" denotes a set.

REFERENCES

[Arsac 79] J J Arsac, ***Syntactic Source to Source Transforms and Program Manipulation***, Communications of the ACM, Vol 22, No 1, January 1979.

[Apt 81] K Apt, M H Emden, ***Contributions to the Theory of Logic Programming***, Erasmus University, The Netherlands, 1981.

[Beckman 76] L Beckman, A Haraldsson, Ö Oskarsson, E Sandewall, ***A Partial Evaluator and its Use as a Programming Tool***, Artificial Intelligence Journal 7, 1976, pp. 319-357.

[Björner and Jones 78] D Björner, C B Jones, eds, ***The Vienna Development Method: The Meta-Language***, Lecture Notes in Computer Science, Vol 61, Springer-Verlag, 1978.

[Boyer and Moore 72] R S Moore, J S Moore ***The Sharing of Structure in Theorem Proving Programs***, in Machine Intelligence 7, Meltzer and Michie, eds, Edinburgh 1972.

[Burstall 78] R M Burstall, M S Feather, ***Program Development by Transformations: An Overview***, Toulouse CREST Course of Programming, Toulouse, 1978.

[Cheatham 79] T Cheatham, G Holloway, J Townley, ***Symbolic Evaluation and the Analysis of Programs***, IEEE Transactions on Software Engineering, Vol SE-5, No 4, July 1979.

[Clark 80] K Clark, F McCabe, ***IC-Prolog - Language Features***, in: [Tärnlund 80].

[Davis 80] R Davis, ***Meta-rules: Reasoning about Control***, Artificial Intelligence Journal Vol 15, 1980.

[Dincbas 80] M Dincbas, ***The METALOG Problem Solving System: An Informal Presentation***, in: [Tärnlund 80].

[Emanuelson 80] P Emanuelson, ***Performance enhancement in a well-structured pattern-matcher through partial evaluation***, Ph.D. Thesis, Linköping University, 1980.

[Emden 76] M V Emden, R Kowalski, ***The Semantics of Predicate Logic as a Programming Language***, J of ACM, Vol 23, No 4, October 1976.

[Ershov 77] A P Ershov, ***Correctness of Mixed Computation in Algol-like Programs***, Proceedings of the 6th MFCS Symposium, in: Lecture Notes in Computer Science, Vol 53, Springer-Verlag, 1977.

[Ershov 80] A P Ershov, ***Mixed Computation: Potential Applications and Problems for Study***, Computing Center, Siberian Branch, USSR Ac. Sci. Novosibirsk 630090, USSR.

[Ershov and Itkin 77] A P Ershov, V E Itkin, ***Corectness of Mixed Computation in an Algol-like programs***, in Lecture Notes in Computer Science, vol 53, Springer Verlag, 1977.

[Futamura 71] Y Futamura, ***Partial Evaluation of Computer Programs: An Approach to A Compiler-compiler***, J. Inst. Electronics and Communication Engineers, 1971.

[Gallaire 80] H Gallaire, C Lasserre, ***A Control Metalanguage for Logic Programming***, in: [Tärnlund 80].

[Gordon 79] M J C Gordon, ***The Denotational Description of Programming Languages, An Introduction***, Springer-Verlag, 1979.

[Haraldsson 74] A Haraldsson, ***PCDB - A Procedure Generator for A Predicate Calculus Data Base***, IFIP-74 Information Processing, North-Holland, 1974.

[Haraldsson 77] A Haraldsson, ***A Program Manipulation System Based on Partial Evaluation***, Ph.D. Thesis, Linköping University, Sweden 1977.

[Haraldsson 80] A Haraldsson, ***Experiences from a program manipulation system***, Linköping University, July 1980, LiTH-MAT-R-80-24.

[Henhapl 78] W Henhapl, C B Jones, ***A Formal Definition of Algol-60 as described in the modified Report***, in [Björner and Jones 78]

[Jones 78] C B Jones, ***The Meta-language: A Reference Manual***, in: [Björner and Jones 78].

[Kieburtz 81] R Kieburtz, J Shultis, ***Transformations of FP Program Schemes***, in: Invited Papers, Symposium on Functional Languages and Computer Architecture, Chalmers University of Technology, Gothenburg, 1981.

[Kleene 52] S C Kleene, ***Introduction to Metamathematics***, Van Nostrand, New York 1952.

[Komorowski 76] H J Komorowski, ***Familia - A Question-Answering System in Prolog with Natural Language Interface***, M Sc Thesis, University of Warsaw, 1976. In Polish.

[Komorowski 79] H J Komorowski, ***Qlog Interactive Programming Environment - The Experience form Embedding A Generalized Prolog in Interlisp***, paper read at International Summer Seminar on Artificial Intelligence, Dubrovnik, August 1979. Also, Informatics Laboratory, Linköping University, LiTH-MAT-R-79-19.

[Komorowski 80] H J Komorowski, ***Qlog - The Software for Prolog and Logic Programming***, in: [Tärnlund 80].

[Komorowski 81] H J Komorowski, J W Goodwin, ***Embedding Prolog in Lisp: An Example of A Lisp Craft Technique***, SSRC, Linköping University, March 1981, LiTH-MAT-R-81-2. Paper read at Symposium on Functional Programming Languages and Computer Architecture in Gothenburg, June 1981, Sweden.

[Komorowski 81a] H J Komorowski, ***A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation***, Ph.D. thesis no 69, Linköping University, Sweden 1981.

[Kowalski 74] R Kowalski, ***Predicate Logic as a Programming Language***, IFIP 74 Information Processing, North-Holland 1974.

[Loveman 77] D B Loveman, ***Program Improvement by Source to Source Transformation***, Journal of the ACM, Vol 24, No 1, January 1977.

[Martelli 76] A Martelli, U Montanari, ***Unification in Linear Time and Space: A Structured Presentation***, Consiglio Nazionale delle Recierche, Istituto di Elaborazione della Informazione, Nota Interna B76-16, Pisa, 1976.

[McCarthy 66] J McCarthy et al, ***Lisp 1.5 Programmer's Manual***, MIT Press, 1966.

[Ollongren 80] A Ollongren, ***On the Implementation of Parts of Meta-IV in Lisp***, SSRC, Linköping University, LiTH-MAT-R-81-7, April 1981.

[Ostrovsky 80] B N Ostrovsky, ***Obtaining Language Oriented Parser Systematically by Means of Mixed Computation***, in: ***Translation and Program Models***, Ed. I V Pottosin, Computing Center, Novosibirsk, 69-80. In Russian.

[Pereira 78] L M Pereira et al, ***User's Guide to DECsystem-10 Prolog***, September 1978.

[Robinson 65] J A Robinson, ***A Machine-Oriented Logic Based on the Resolution Principle***, Journal of ACM Vol 12, No 1, 1965.

[Sandewall 71] E Sandewall, ***A Programming Tool for Management of a Predicate Calculus-Oriented Data Base***, Proceedings of the 2nd IJCAI, London, 1971.

[Standish 76] T A Standish, ***An Example of Program Improvement Using Source-to-Source Transformations***, Computer Science Conference, Anaheim, February 1976.

[Stoy 77] J E Stoy, ***Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory***, MIT Press, 1977.

[Sussman 72] G J Sussman, T Winograd, E Charniak, ***Micro- Planner Reference Manual***, AI Memo 203a, AI Lab, MIT, 1971.

[Teitelman 78] W Teitelman, ***Interlisp Reference Manual***, Xerox, PARC, 1978.

[Tärnlund 80], S-Å Tärnlund, ***Proceedings of the Logic Programming Workshop***, John von Neumann Society, Debrecen, Hungary, 1980.

□