ELl PARTIAL E..LUATOR

(Progress Report)

Submitted to

Dr. Ben Wegbreit

Center for Research in Computing Technology

Harvard University

Cambridge, Massachusetts

By

Yoshihiko Futamura

Applied Mathematics 260

January 24, 1973

1 Introduction

The aim of this report is to provide information concerning work done, problems encountered and difficulties overcome while the project for the ELl partial evaluator has been under way since October, 1972.

The goal of the project is to implement a partial evaluator $\mathcal{A}$ for ELl which has the properties described below.

$\mathcal{A}$ is a partial evaluator such that:

$$\mathcal{A}(\text{int},s)(r)=\text{int}(s,r) \qquad (1)$$

for arbitrary program int and its arguments s and r. We can derive following equations from (1).

$$\mathcal{A}(\text{int},s)(r)=\mathcal{A}(\mathcal{A},\text{int})(s)(r)$$
$$=\mathcal{A}(\mathcal{A},\mathcal{A})(\text{int})(s)(r).$$

If we consider int, s, and r as an interpreter, a source program and runtime data respectively, then $\mathcal{A}(\text{int},s)$, $\mathcal{A}(\mathcal{A},\text{int})$, $\mathcal{A}(\mathcal{A},\mathcal{A})$ can be considered as an object program, a compiler and a compiler-compiler respectively. In order to be useful, $\mathcal{A}$ has to have following two properties.

1) $\mathcal{A}$ should evaluate as many portions of programs (e.g. int, $\mathcal{A}$) as plssible in order to save time during final evaluation.

2) $\mathcal{A}$ shoule avoid working on portions of the program not needed in final evaluation in order to save partial evaluation time.

Work done is:

(1) finding an efficient procedure to terminate loops during a partial evaluation which is caused by the partial evaluation of equal programs in equal environments.

(2) formalizing a class of partial evaluations and partly designing a partial evaluator for the class.

(3) coding a part of the partial evaluator in EL1.

The result of (1) and (2) will be described in Section 2 of this report.

Problems encountered are:

(1) terminating a loop during partial evaluation time.

(2) backtracking the environment for the partial evaluation of conditionals.

(3) embeding the partial evaluator in EL1.

(4) avoiding the execution of an assignment which has undesirable side-effect.

Difficulties over come are, probably, (1), (2), and (3) above. The solution to (1) and (2) are described in section 2 of this paper. The solution to (3) is to consider a partial evaluator as a special case of the closure in EL1.[1]

Problem (4) has not been solved yet. It will take a little bit more time to solve the problem. Coding of the partial evaluator will be another big problem because of the complexity of the algorithm.

---

[1] Ben Wegbreit "Procedure closure in EL1" Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, March, 1972.

2 Partial evaluator

This section describes techniques for implementing EL1 partial evaluator. The partial evaluation algorithm described in the literature[1] can be constructed for EL1 based on the EL1 evaluator[2]. The closure in EL1[3] will specify an initial set of p-variables and their values.

2.1 Simple partial evaluation

The concept of a partial evaluation has been described in the literature[1] informally. We know that there are infinitely many possible partial evaluation algorithms. However the algorithm which is desirable should have properties listed below.

(1) It should be efficient (i.e. runs fast).

(2) Object programs produced by partial evaluation should be executed much faster than the source program.

(3) Object programs should be always correct.

It is usual that optimizers change the logics of a source program[4]. However in a partial evaluation, where the logics of a source program play very important role, only a small change of the logic will cause the production of an erroneous ogject program.

---

[1] Yoshihiko Futamura "Partial evaluation of computation process, an approach to a compiler-compiler" J.Inst.Elect.& Com.Eng. of Japan Vol. 54-6,No.8, August, 1971.

[2] Ben Wegbreit "Studies in extensible programming languages" ESD-TR-70-297,Harvard University,Cambridge,Massachusetts,May,197o.

[3] ---------- "Procedure closure in EL1" op. cit.

[4] E.S.Lowry and Medlock "Object code optimization" Comm.ACM 12,1(January, 1969) 13-22.

So we must be very careful about the point (3) above. Notice that the more powerful a partial evaluator is, the more erronneous object programs it produces. For example, if a partial evaluator only substitutes given values for corresponding variables like a macro processor, it will not do any harm. If a partial evaluator evaluates all portions of a source programs which can be evaluated based on given values and if the source program contains operatoms which have side effect, the object program will be almost always erroneous except a trivial case.

What we are going to do now is to define a partial evaluator which runs fast, always produces a correct and reasonably efficient object program. This partial evaluator will do the simple partial evaluation defined in this section.

We use the terms listed in Figure 1 in the following discussion.

term	meaning
source program	procedure to be partially evaluated
object program	result of a partial evaluation
p-time	time at which a partial evaluator runs
r-time	time at which an object program runs
p-variable	variable in source program to which data are assigned at p-time
r-variable	variable in source program other than p-variable
p-value	value of p-variable
p-object	data which are processed at p-time (p-object will be p-value)
p-environment	system which defines values ofp-variables in a source program. This may be changed by assignments of a p-object to a p-variable during p-time.

Figure 1: Summary of the meaning of terms

Definition 1: A partial evaluation is simple iff it does not contain execution of

(1) operators which destroy p-objects

(2) I/O operations

and

(3) code procedures with a free variable which is not compatible with declared variables in a source program.

GENSYM in LISP and a machine coded random number generator are the kind of operators mentioned in (3).

We will abbriviate a simple partial evaluation as SEP in the rest of this paper. SPE of a part of a source program s is represented by SPE(s).

This definition of SPE matches the requirements for a procedure call to be evaluated during compilation described by Wegbreit[1].

In theory, we can construct a partial evaluator for SPE as follows:

Before executing one step of a partial evaluation, check the following conditions.

(1) If an assignment is going to be executed , check whether the target of the assignment can be reached from p-variables by tracing p-objects (as a grabage collector does). If it can be reached, don't execute the assignment but generate an assignment statement in order to be executed at r-time. If it can't, execute the assignment.

---

[1] Ben Wegbreit "Procedure closure in EL1" op.cit. page 22.

(2) If an I/O operation or a code procedure with a free variable is going to be executed, don't execute the operation but generate the operation in order to be executed at r-time.

However operation (1) is too expensive to be practical. We will discuss the practical simple partial evaluator later.

Equivalence problem of SPE is discussed here.

Definition 2: PENV(s) is a p-environment at the entrance of a part of a source program s.

It is clear that two SPE(s)'s are equal if PENV(s)'s are equal.

Definition 3: A local p-environment (LPE) of a SPE of a part of a source program s is a set of ordered pairs (V,O) such that:

" V is a p-variable which appears in the course of SPE(s) and has the p-object pointed to by O as its value when it is refered for the first time."

We represent a LPE of SPE(s) as LPE(s). Intuitively, LPE(s) is a set of initial values of p-variables contained in s and refered by SPE(s).

Note that the construction of LPE(s) starts at the entrance of s and finishes at the end of s. If s is a recursive procedure or a loop, the construction of a new LPE(s) starts before the old LPE(s) has been completed. This situation can be illustrated graphically in Figure 2.

Let s^i represent the i-th entry of a recursion or a loop to s.

Definition 4: $LPE_i(s)$ is a set of ordered pairs (V^i, O^i) such that:

" V^i is a p-variable which appears in the course of $SPE(s^i)$ before $SPE(s^{i+1})$ starts. O^i is a pointer to the value of V^i when it is refered to for the first time."

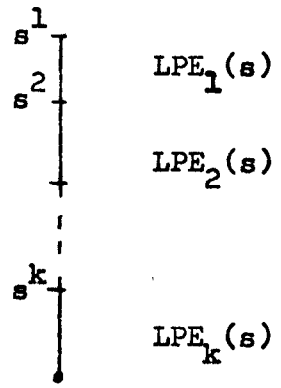


Figure 2: Recursive call (or loop) and the construction of LPE.

Then $LPE(s^i)=g_i(k)$

where $g_i(1)=LPE_i(\varepsilon)$

$$g_i(n+1)=\{(V_j^{n+1},O_j^{n+1})\in LPE_{n+1}(s) \mid V_j^{n+1}\neq V^n \text{ for all } (V^n, O^n) \text{ in } g_i(n)\}\cup g_i(n),$$

Note that the value of a p-variable V can be get from the p-environment. The p-environment has an usual stack structure and keeps the values of variables (both p-variables and r-variables) appeared in the course of SPE. The p-environment associates NOTHING with r-variables and p-objects with p-variables. When NOTHING is assigned to a p-variable w or to a part of the p- object which is a value of w, w becomes r-variable.

Definition 5: $LPE_i^c(s) \equiv \{(V,O') \mid (V,O) \in LPE_i(s)$ and O' is the pointer to the value of p-variable V at the (i+1)-st entry to s.$\}$

$LPE_i^c(s)$ represents the current values of p-variables, which were referred to in the $SPE(s^i)$, at the entrance of $SPE(s^{i+1})$.

Theorem: Let s be a recursive procedure or a loop. If it happens to be $LPE_i(s)=LPE_i^c(s)$ for some i in $SPE(s^1)$ then $SPE(s^i)=SPE(s^{i+1})$ and the SPE will never terminate.

Proof: Intuitively, $LPE_i(s)=LPE_i^c(s)$ means that the initial values of p-variables at the i-th entry of s and that of at the (i+1)-st entry of s are equal. Since SPE has no side effect, $SPE(s^i)$ and $SPE(s^{i+1})$ will do the same computation and loop. More formal proof is given

below.

Assume that $LPE_i(s)$ has been constructed in the order $(V_1,O_1),\ldots,(V_m,O_m)$ and that the construction of $LPE_{i+1}(s)$ advances in the order $(v_1,o_1),\ldots,(v_n,o_n)$. If $V_i=v_i$ and $O_i=o_i$ for all i then $SPE(s^{i+1})$ does the same computation as $SPE(s^i)$. Because SPE has no side effect which destroys p-objects and every two SPE(s)'s which have the same initial pointers to p-values produce the same object program.

Assume that there is some i such that $(V_i,O_i)\neq(v_i,o_i)$.

If $V_i=v_i$ then $O_i=o_i$ by $LPE_i(s)=LPE_i^c(s)$.

So $V_1=v_1,O_1=o_1,\ldots,V_{i-1}\neq v_{i-1},O_{i-1}=o_{i-1}$ and $V_i\neq v_i$.

Since both $SPE(s^i)$ and $SPE(s^{i+1})$ refer to the same variable-values until they refer to V_i and v_i, they should do the same computation by that time and those variables should be on the same path of s. So the p-value of v_i has been set outside of $SPE(s^{i+1})$.

(a) If the p-value of v_i was set in $SPE(s^i)$ then there is some j such that $V_j=v_i,O_j\neq o_i$. However this is against $LPE_i(s)=LPE_i^c(s)$.

(b) If the p-value of v_i was set before $SPE(s^i)$ starts then $V_i=v_i$.

Therefore there cannot be any i such as assumed above.

So $LPE_i(s)=LPE_{i+1}(s)$ and $SPE(s^i)$ and $SPE(s^{i+1})$ do the same computation and produce the same object program. If a recursive call or a loop repeat the same computation, it will never terminate.

2.2 Simple partial evaluator

This section roughly describes the algorithm for SPE of the three types of forms in EL1.

2.2.1 Procedure application

Assume that

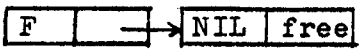
(1) s:F(A1,...,An); is called (s is a label)

(2) Ai1,...,Aip are known (p-values)

(3) Aj1,...,Ajq are unknown

(4) X1,...,Xn are formal parameters of F

Then push (Xi1,Ai1),...,(Xip,Aip),(Xj1,NOTHING),...,(Xjq,NOTHING) on the top of PENV(s). This new PENV is PENV(F).

If there is an entry in the Function-stack shown in figure 3, check whether $LPE_i(F)=LPE_i^c(F)$. If it is true then the value of SPE(F) is in object-table(k) as shown in Figure 4. If it is false then push 

on the top of the function-stack and do SPE of the body of F.

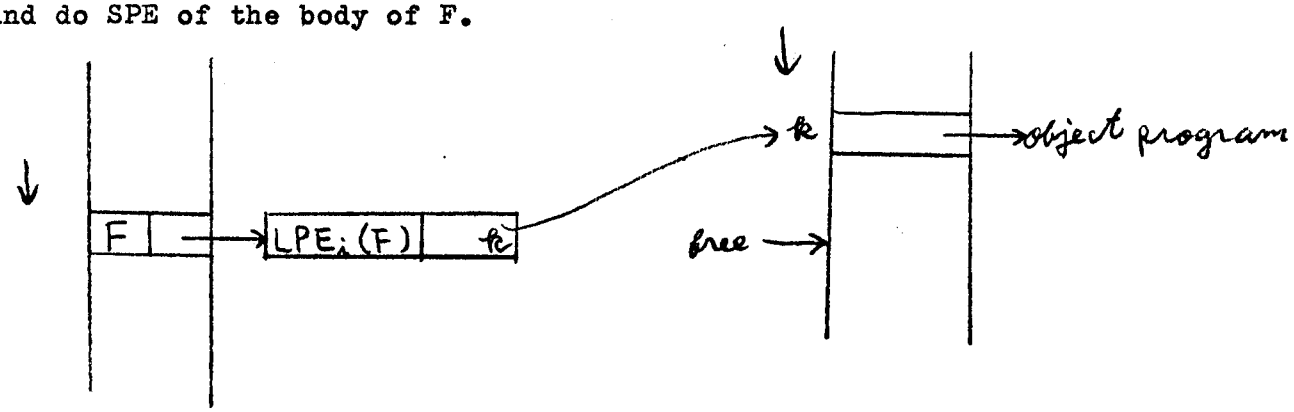


Figure 3: Function-stack
This stack keeps procedure names, their LPE_is, and the address of the object program.

Figure 4: Object-table

2.2.2 Conditionals

Conditionals appear in the following context:

s:BEGIN ... p⇒ e; c1;c2;... END:

s1;s2;...

(1) If the value of p is known then generate

(a) BEGIN ...;SPE(e) END; SPE(s1);SPE(s2);... if p is true.

(b) BEGIN ...;SPE(c1);SPE(c2);... END;SPE(s1);SPE(s2);...

if p is false.

(2) If the value of p is unknown then generate

v1 ←o1; v2 ←o2;...

BEGIN ...;SPE(p)⇒ SPE(e);SPE(c1);... END;

SPE(s1);SPE(s2);...

If a value of p-variable vi in PENV(s) will be changed in SPE(e),... END; then assignment vi ←(value of vi); should be generated in order to be executed at run time. And PENV(s) should be changed so that vi will be r-variable. This generation of assignments avoids the necessity of generating SPE(s1);SPE(s2);... twice for both SPE(e) and SPE(c1);SPE(c2);...END.

2.2.3 Iteration

s:FOR i ← form1, {form2 },..., form3 {test} DO form;

(a) If form1,form2, and form3 are known then do the SPE of conditionals test=> form; for each i.

(b) If one of them is unknown then generate

v1 ←o1; v2 ←o2;...

FOR i ←SPE(form1), {SPE(form2),}...,SPE(form3){, SPE(test)} DO

SPE(form);

If a value of p-variable vi in PENV(s) is going to be changed in SPE(form) then assignment vi ← (value of vi); should be generated in order to be executed at r-time. And PENV(s) should be changed so that vi will be r-variable.