Partial Evaluation of Computer Programs; an Approach to Interpreter Compiling

by

YOSHIHIKO FUTAMURA

Central Research Laboratory, Hitachi, LTD.

Kokubunji, TOKYO, JAPAN

and

JOHN K. DIXON

Heuristics Laboratory
Division of Computer Research and Technology
National Institutes of Health
Department of Health, Education and Welfare
Bethesda, Maryland, 20014

KEY WORDS AND PHRASES: LISP, partial evaluation, heuristic procedures, compiler, semantic compiler, semantics, lambda calculus, specialization, pattern languages, predicate calculus, debugging, optimization, ~~data compiling~~, programming style, pure LISP, algorithm, programming language, interpreter, compiler-compiler

CR CATEGORIES: 5.24 4.12 3.66 4.12 5.23

Abstract

This paper presents a new type of generalized compiler and compiler-compiler based on the technique of partial evaluation. Several algorithms for partial evaluation are presented. Their properties and limitations are discussed. It is shown how a partial evaluation program can serve as a generalized compiler. This is done by partially evaluating an interpreter which provides a semantic and syntatic definition of the language which is to be compiled. It is also shown how a partial evaluation program can partially evaluate itself along with an interpreter to generate a compiler for any arbitrary language. (IE, A COMPILER-COMPILER)

One partial evaluation program called the specializer was written to process LISP. Thespecializer is shown to have desirable theoretical properties such as a freedom from infinite loops and output code that is always correct. Examples of the use of the specializer to compile several simple languages are given. The resulting output code is shown. The future problems and possibilities of the partial evaluation technique are discussed.

I. INTRODUCTION

This paper presents an unusual idea, a generalized (multi-language) compiler based on the principle of partial evaluation of computer programs. Several such generalized compilers called specializers have been written. Examples of their practical use are given. The theoretical ~~properties of one specializer~~ problems of partial evaluation are also discussed.

Consider a simple example of partial evaluation. Suppose we have a FORTRAN statement:

A = B + 2 (C*D)

and suppose that the values C = 2, D =3 are known but no value has yet been assigned to the variable B. Then the FORTRAN statement can be simplified or specialized or partially evaluated to A = B + 12.

It is easy to see how the concept embodied in this simple FORTRAN example can be extended. For example partial evaluation of a conditional jump statement might eliminate some alternatives. Of course, languages other than FORTRAN can be simplified or partially evaluated in an analogous way.

Think of a computer program P as a mechanism which takes a number of input items, A1, A2, A3, . . ., Am, each one being an element of data, then performs some processing on these data and finally produces an output vector, V. If some of the input itmes are known (say A1 and A2) at an

early time, then it is possible to partially evaluate P to produce a new simplified or specialized program P′. Then at a later time when the missing input itmes, (A3, ...Am), become available, the new program can be evaluated (the final evaluation) to producethe same output vector V. We would naturally expect the final evaluation of P′ to take less computer time than evaluation of P since part of the work has already been done.

At this point, it should be plausible to the reader that rigorous rules for partial evaluation of any computer language could be cmbodied in a program which would then be able to partially evaluate or specialize any statement in that language. We will call such a program a specializer.

We will now define some informal notation which will make it easier to discuss the properties of the specializer. Assume that a specializer is written for some base language, L. Normally, a statement, E, written in the the language L would be evaluated by an interpreter called EVAL.

V = (EVAL E AL)

Where AL is an association list of variable-value pairs which assigns values to all the variables which appear in E. V is the value or output of E after normal evaluation.

Now suppose a psecializer, called SP, is written for the language L. To specialize the statement E, we first partition AL into two parts so that AL1 assigns values to variables which are known at specialization time, and AL2 assigns values to the remaining variables. Thus AL = AL1 + AL2

Now the statement E can be specialized and then final-evaluated as follows:

E' = (SP E AL1)

V = (EVAL E' AL2)

Where E' is the specialized version of E and V is the final value which will be identical to the value obtained earlier in normal evaluation of E.

But ~~how do we use~~ can also be used a specializer to compile some arbitrary language, L1. First ~~we write~~ must be written an interpreter, INL1, which is capable of evaluating any statement in L1. INL1 will thus implicitly define the syntax and semantics of L1. Any statement, E1, written in L1 may then be evaluated as follows:

V1 = (INL1 E1 AL)

If INL1 is written in L, the language which the psecializer is designed to accept, then we can use SP as a compiler as follows:

E1' = (SP "(INT E1 NIL)"NIL)

V1 = (EVAL E1' AL)

Where NIL indicates an empty list, and E1' is the compiled version of E1. E1' will be expressed in the base

language L, and final evaluation of E1' will produce the same value V1 as before. In thiscase we have used SP as a universal compiler.

But there is also another, more elegant, way of compiling with a specializer. We can use the specializer to specialize itself and thus produce a special purpose compiler for the language L1 as follows:

CPL1 = (SP"(SP(INT NIL NIL) NIL)"NIL)

CPL1 can now be used to compile any particular statement, E1, in the language L1:

E1' = (CPL1 E1 NIL)

V1 = (EVAL E1' AL)

In this case SP is used as a compiler-compiler. We might Call SP a semantic compiler, since it uses the semantic definition of a language which is implicit in an interpreter. These ideas may be useful since it is often convenient and easy to express a new language in the form of an interpreter.

These ideas are not entirely new. McCarthy first suggested expressing the syntax and semantics of a computer language by an interpreter [13, 14, 15]. The LISP 1.5 programmers Manual [15] contains an example. Lombardi and Raphael [12] actually wrote a program to partially evaluate LISP; however they did not view it as a generalized compiler, but instead as a step toward an "Incremental Computer" Also, partial evaluation is an optional feature of the POP-2 language [2].

Moreover, the attempt to find a normal form for a lambda-calculus expression is closely related to partial evaluation [5, 11].

The novelty of this paper is that all these ideas have been pulled together, a particular Algorithm for the specialization of LISP [15, 18] has been written, its theoretical properties have been investigated, and examples of its practical use in compiling various languages are given.

In addition, what Allen [1] calls "folding" is a simple form of partial evaluation.

II. The PURELISP Specializer

For convenience in disucssing the specializer, PURELISP is now defined. PURELISP is a simplified version of LISP 1.5 [15] designed by the author. PURELISP is quite similar to the formal mathematical system of pure LISP defined by McCarthy in Section I of [15].

PURELISP includes the five primitive functions CAR, CDR, CONS, ATOM, and EQ. COND, LAMBDA, and QUOTE expressions are also included. Arbitrary atoms (EXPRATOM's) may be used as functions if they are defined equivalent to a PURELISP LAMBDAFORM.

PURELISP is a simple but powerful language. The EXPRFORM allows one to build up recursive functions by nesting the primitives. It is possible to write almost any symbolic processing program in PURELISP, provided the side effects are not needed.

The semantics of PURELISP are defined by the interpreter in [6, 7]. The syntax of the language is also implicit in this interpreter. This interpreter has actually been implemented in LISP 1.6 [18] and used to interpret statements in PURELISP. It is also called by the PURELISP specializer.

The CAR or CDR of an atom is not allowed in PURELISP. This is explicitly indicated in the interpreter by the function (ERR). This is the LISP 1.6 error function.

The syntax of LISP specifies a tree-like structure. By starting at the outside of a PURELISP form and peeling off the functions, one at a time, as the interpreter dos, one can construct a tree. For example, the form:

```
((LAMBDA (X Y)
         (COND ((ATOM X) (QUOTE T))
               (Y (CONS X Y))   ))
         (QUOTE A)
         (QUOTE B))
```

expands into the tree shown in Figure .

This PURELISP tree has several properties which should be noted:

1. All terminal points ofthe tree are data, either quoted constants or variables.

2. Each non-terminal node represents some operation on the data (several nodes areconsidered to be associated with LAMBDA and COND).

3. Each non-terminal node has oneor more lines entering from below, which representinput dataand exactly one upward line indicating output from the operation.

4. All upward communication of datais indicated by the lines shown on the tree.

Roughly speaking, specialization of PUREBISP involves two operations: 1) pruning a branch ofthe tree which can be evaluated and replacing it with a quoted

value, 2) removing a branch of a conditional expression (which may be evaluated or not) since it is not needed.

A specialization algorithm for PURELISP is given in Table I. A simplified summary of the algorithm is given in Figure 1.

This algorithm has two useful properties:

Theorem I: Programs produced by the algorithm are always correct; that is, they produce the same value, upon evaluation, before or after specialization.

Theorem II: A Specialized program always runs as fast or faster than the original.

Detailed proofs of these theorems are given in [6,7]. To the reader who is familiar with LISP, it will be plausible that these properties hold. In reading the algorithm remember that in LISP, NIL=() = F and each of these symbols means both "the empty list" and also "FALSE."

III: Examples of Specialization

Actually LISP is a particularly good base language for specialization since (1) LISP programs and data are interchangeable and, (2) the notation used in LISP is expecially convenient.

Simple examples may give the impression that partial evaluation is a rather trivial process. It would appear that a specialized program is simply a shortened and simplified version of the original; and it is true that each of the six basic rules for specialization of LISP expression is quite simple (see Table I). However, when these simple rules are applied to a multiply recursive LISP program the resulting specialized version can be so different in form and structure from the original, that the two programs appear to be entirely unrelated (see Fig. 1). In such a situation it is perhaps appropriate to say that the specializer has written a new program.

A good example of the use of SP as a semantic compiler is shown in Figure 2. In this case, E stands for the LISP function TRANS and its two subfunctions FIN1 and ELEP. TRANS is a LISP function which applies an arbitrary transformation to an S-expression A (first argument of TRANS). TRANS is a bit like an editor. It

can also insert new elements. The resulting new expression is the value of TRANS.

The particular transformation which will be applied is specified by B and C (2nd and 3rd arguments of TRANS). Thus we can consider B and C, taken together, to represent a particular statement (B C) in a transformation language L1. TRANS, then, is an interpreter for L1. In the example of Figure 2, B is (XY(QR) Z) and C is (Z X (R Q) NEW). B and C indicate a transformation which is to be applied to A, the data. It is assumed that A will be an S-expression iwht a structure corresponding to B. The transformation specified by these particular values of B and C could be expressed in English as follows. "First interchange the two sub-parts (Q R) of the third element of A, then delete thesecond element (Y) of A, then move the last element (Z) of A to the front, finally add a new element "NEW" to the end."

In Figure 2, DO is the LISP function which is a specialized version of TRANS for the particular values of B and C. In other words, TRANS has been specialized for A variable, B and C fixed.

In studying this example, certain significant facts should be noted:

1. DO performs exactly the same action as TRANS on any data, A, assuming that B and C have the values given.

3. DO has only one argument, A, although TRANS had 3.

4. The missing arguments B and C are implicit in the structure of DO.

5. DO is simpler than TRANS andits subfunctions; and it occupies less storage space (This is not an algorithmic property of specialization, DO could be larger or smaller than TRANS depending on the size of B and C).

6. DO has a structure quite different than that of TRANS and its subfunctions. For example, there are no conditional statements in DO.

7. The structure of DO ismuch like that of C. Thus one might say that the data(B and C) have been compiled.

Figure 3 is another example of specialization. The function specialized here is MATCH which tells if an arbitrary S-expression, D, matches a pattern P. P is expressed in ZIP, a pattern language. Thus one may consider MATCH an interpreter for the ZIP language. (ZIP is similar to FLIP [19]) This example contains statements which a human programmer would certainly simplify. For example:

(EVAL (LIST (QUOTE $A) (QT (CAR A2))))

can be simplified to:

($A (CAR A2))

($A is a function name in ZIP and is equivalent to ATOM)

This example emphasizes that the specializer is not an optimizer and suggests that it might be useful to process specialized code with an optimizer.

The MATCH example has all the same properties as the TRANS example except that the specialized code is larger, in this case, than the original.

A third example ofspecialization is given in Figure 5. Here an axiom expressed in first order predicate calculus was compiled or translated into LISP. This was part of an effort to prove theorems more efficiently by means of compiled axioms [6,8].

PURELISP has been used as a convenience in illustrating basic principles. But for practical purposes, it would be desirable to have a specializer which accepts more of the features usually available in LISP. It isprobably possible to extend the specializer to handle all of the features of LISP 1.5 or LISP 1.6 and still maintain Theorems I and II. In some cases this is rather easy; in other cases, harder.

The functions which change existing list structure such as RPLACA, RPLACD, NCONC, etc. are a difficult problem. If they are executed, there is no trouble; but if they are not executed, then all variables which point to list structure which would have been changed become uncertain. There is no direct method in LISP to find out which variables are pointing at a particular bit of list structure. A garbage-collector-like search is one possible solution.

EVAL and functional arguments are a most difficult problem. If the specializer encountered an EVAL statement with an uncertain argument, there is no easy way to know that the side effects might be. Thus to maintain

the truth of Theorem I, it must beassumed that all variables, and all data pointed to from the A-list have become uncertain. The same applies to functional arguments.

A practical specializer ~~which uses many of the above techniques~~ has been written and has been in use since April, 1969. This program is called SPZR. Some shortcuts have been taken in the difficult cases so that Theorems I and II are true only for a certain subset of LISP 1.6. Many programs of practical interest have been specialized by SPZR; and the results have always been correct and have always executed faster than the original.

Often the pattern or format of a particular variable is known without knowing the exact value. For example, one may know that the value of a variable X is always a list of exactly 7 objects, the first one a non-numeric atom and the rest numbers. A specializer which could make use of such pattern-type information would be more powerful than the ones discussed previously since it could do everything they could do plus more.

A pattern specializer based on the ZIP pattern language was programmed and debugged in September 1969. This program has proven to be substantially more powerful than the non-pattern specializers. An example of pattern specialization is given in Figure 4.

IV: GENERAL PROBLEMS OF SPECIALIZATION

There are many possible algorithms for partial evaluation of any language. Different algorithms have different advantages and disadvantages. In particular, the handling of conditional jump statements is a key question in the formulation of a partial evaluation algorithm. This question will now be discussed in very general terms, without reference to a specific programming language.

The Partial Evaluation of conditional jump statements (i.e. branching points or flow of control decisions) raises some interesting problems. During total evaluation the predicate of a jump statement is evaluated and a jump is either made or not made. In other words, there is a two-valued logic (T, F). However, during partial evaluation, there is a third possibility; that the predicate cannot be assigned any definite value. Thus, in partial evaluation, there is a three valued logic (T , ? , F). Treatment of the T and F cases is quite clear; they simply become unconditional jumps. The '?' case is a different matter. Here we meet two conflicting requirements:

1. A good partial evaluator should specialize every branch needed in the final evaluation in order to save time during final evaluation, ~~time~~.

2. A good partial evaluator should avoid working on branches of the program not needed in final evaluation in order to save partial evaluation time.

The troulbe is that we do not know if '?' branches are needed or not. Thus, there is danger that the partial evaluator will enter branches which would not be entered during full evaluation. There is not only the question of saving time, but there is also the problem of endless looping during partial evaluation.

A partial evaluation algorithm will now be presented. This algorithm is in very general terms and is presented to illustrate the problems involved in the partial evaluation of conditional jump statements. In discussing this algorithm, the objectprogram will be represented as a graph as in Figure 6.

In Figure 6, nodes (o) represent conditional branching points, branches (arrows) represent fragments of the program not containing a branching point and the leaves (o) represent the termination points of the program. All the nodes and branches are named n_i and b_j (a different one is subscripted by a different number), respectively. Let B_1 express the entry branch, and let m denote the total number of branches.

The Generalized Partial Evaluation Algorithm α,

At each stage of the partial evaluation, the set of all variables is partitioned into two subsets; the 'p' variables, which have definite, known values during partial evaluation, and the 'r' variables whose values

are unknown, and which therefore must remain in the object program until final evaluation.

At the start of the partial evaluation, values are assigned to certain variables; these become 'p' variables and all others become 'r' variables. Thereafter, assignment statements may change the status of any variable by assigning either a definite or an indefinite value to that variable.

The algorithm is shown by the following five operations (1) -(5). (In the description of the algorithm, integer variables g, j(1),...,j(m) and a list variable L are used. Also a block of memory spaces, MS, is reserved for storing the results of the partial evaluation. $a_g^{j(g)}$ denotes the address in this space where the result of partial evaluation of branch b_g on the $j(g)^{th}$ iteration are stored.)

(1) Set each of g, j(1),...,j(m) to 1, and set L to a null.

(2) Allocate the first address of the space MS. Namely, enter the triplet $(b_g, S_g^{j(g)}, a_g^{j(g)})$ in list L. Where, $S_g^{j(g)}$ denotes the set of 'p' variables and their current values, and $a_g^{j(g)}$ denotes the address of the space in which the product of j(g)th partial evaluation of b_g is generated.

3. Evaluate the portions of b_g which can be evaluated only with 'p' variables and constants. Let $H_g^{j(g)}$ express the new program fragment generated by this

operation. ($b_g^{j(g)}$ is a new computation process generated from b_g, and b_g is left intact.)

Increment the value of j(g) by 1 and perform operation (4).

(4) If the next process to b_g (i.e., the arrowhead of b_g) is a termination symbol (o), then stop the partial evaluation.

If the next process to b_g is a conditional branching point $n_{k(i)}$, then perform (4.1) or (4.2).

(4.1) If $n_{k(i)}$ can be evaluated, then select the branch indicated by the value of $n_{k(i)}$. Let b_p express the branch selected. Set the value ofg to p, and perform operation (5).

(4.2) If $n_{k(i)}$ cannot be evaluated, then it is left intact. Let b_p and b_q express two branches following $n_{k(i)}$. Set the value of g to p, and perform operation (5). After that, set the valueof g to q and perform operation (5) again.

(5) Search list L fora triplet whose first and second terms are the same as the current b_g and $S_g^{j(g)}$. (if one if found, it means that the same branch is being entered with the same 'p' variable values as before. Thus a nonterminating partial evaluation loop exists). Go to (2) if no such a triplet exists. Otherwise insert into the newly generated program a jump statement to the position indicated by the third term a_g^x ofthe triplet. Then stop partial evaluation.

This completes a discription of the generalized algorithm α_1. Now some examples will be given of some different possible ways the algorithm might act on the program II represented in Figure 6.

~~(5.2) If there is no one, then perform operation (1).~~

Example 1 Suppose that the conditional branching points n_1, n_3, and n_6 can be evaluated only with 'p' variables and constants, and that each evaluation of n_1, n_3, and n_6 selects the branch b_3, b_7, and b_{12} respectively. Then, Γ is transformed by α_1, into the program described in Figure 7.

Example 2 Consider the case in which n_1 and n_6 can be evaluated only with 'p' variables, and the value of n_3 depends on the values of 'r' variables. Let n_1 always select branch b_3, and let n_6 select the branch b_{13} at first time and select the branch b_{12} at second time. Then, Γ is transformed by α_1, into the program described in Figure 8.

Example 3 In example 2, if n_6 selects b_{13} forever, α_1, does not always terminate its computation and generate such an infinite graph as described in Figure 9. However, if n_6 always selects b_{13} simply because the partial evaluation variables of b_7 cyclically take the same values, the computation of α_1, is terminated by operation (5) and produces the result described in Figure 10 in the case when the values of 'p' variables of b_7 do not change.

In partially evaluating an interpreter with respect to a source, programs which contain loops or recursive calls, the above case occurs. Therefore, operations (2) and (5) are essential for generating a compiler.

Example 4 In Figure 11, let us assume that n_1 depends on the 'r' variables. In this case, if the iterative partial evaluation of b_3 does not produce the same S_3^x more than once, then an infinite graph will be generated. But in total evaluation, it is possible that after b_3 has been computed several times, n_1 selects b_2 and the computation will terminate. If b_3 does not contain 'r' variables but contains an infinite loop; and if n_1 always selects b_2 in total evaluation, then it is a trivial example of a program whose total evaluation terminates but whose partial evaluation does not terminate.

This problem can be avoided by the following procedure: The '?' portions of a program are not evaluated at partial evaluation time, but values are substituted for the 'p' variables. This procedure is necessary not only for the saving of partial evaluation time but also for protecting the printing of error messages included in an interpreter, input-output operations, and other portions of the object program having a side effect which do not have to be evaluated at partial evaluation time.

The reason why we make an exception of conditional branching points in the foregoing procedure is to reduce the number of nodes and branches, by evaluating as many

conditional branching points at partial evaluation time as possible. If the portions of a program following a '?' branching point contains 'r' variables, then α_1, is recursively applied to those portions. It is based on the idea that because the portions of a program containing 'r' variables often include recursive calls for an interpreter, it is worth taking a risk with partial evaluation of those portions. Therefore, functions, procedures and pseudo-variables which do not have to be evaluated at partial evaluation time must be marked and must be handled exceptionally.

However, if we describe an interpreter carefully, we can avoid such a meaningless loop as the one described in the last case of Example 4. Then α_1, can be modified as follows: α_1 evaluates all parts of a program except for those portions marked as unnecessary to be evaluated at partial evaluation time.

A partial evaluation algorithm has been described in the preceding discussion, but the details of the algorithm are not described. The details are quite different for each programming language.

Example 5 Partial evaluation of ALGOL program

Let a and b represent lists of integers (i.e. integer array). $a(0)$ and $b(0)$ contain the length of each list respectively. $a(1)$, $a(2)$,..., $a(a(0))$, $b(1)$, $b(2)$,..., $b(b(0))$ contain the elements of the lists. The program concatenating lists a and b is described below. (bigm denotes the upper bound of array a).

```
begin if a(0) + b(0) > bigm then goto overfl;
for k:=1 step 1 until b(0) do a(K + a(0)) := b(k) ;
a(0) := a(0) + b(0) ;
end
```

The result of the partial evaluation of the above program with respect to b at b(0) =2, b(1) = 10, b(2) = 20 is described below.

```
begin if a(0) + 2 > bigm then goto overfl;
a(1) + a(0) = 10; a(2 + a(0)) =20; a(0) = a(0) + 2;
end
```

V. Discussion

Some of the fundamental problems associated with partial evaluation have been solved and others are still open questions.

The halting problem during specialization can be solved by the brute force technique of setting an arbitrary limit on the number or depth of recursions or iterations. Other more elegant methods can also be used while still retaining the arbitrary limit as a backup. Note that this is not the same as the general halting problem which cannot be solved.

Open ended language features such as functional arguments will probably have to be restricted in a practical specializer. The restriction needed is that all inputs and outputs of the unknown function must be made known to the specializer. In LISP a sufficient restriction would be that free variables and subfunctions which change existing list structure must not be used. In other words, ~~the~~ a specializer cannot be expected to correctly specialize a program when an important part of that program is not given.

The most serious disadvantage of the specializer as described up to this point is the fact that it makes no effort to conserve free storage space.

If one specializes only carefully selected programs,

the use of storage space can be kept within reasonable bounds. However, it is not difficult to find examples of LISP functions which use up large amounts of storage space when specialized.

At the present time, ~~our~~ a solution to the space problem is to think through how the specializer will expand a given problem and then permit it to expand only those subfunctions which will expand neatly. ~~SPEER~~ One practical way takes a list of ~~EXPR~~ functions names which may be expanded. ~~EXPR's not on the list are not expanded.~~

The specializer would be more useful if it could make such decisions by itself. Some suggestions about how this might be done are given in [6, 7] .

Most of our discussion of partial evaluation has been based on the assumption that some variables in the object program have definite values and others are entirely unknown. But this is sometimes an awkward way of describing our partial knowledge of the inputs to the program. Finer shadings are possible. The ultimate technique is perhaps to describe all that we know about the program and its inputs in predicate calculus and then use theorem proving techniques to specialize the program. This approach is discussed in [4]. The pattern specializer mentioned earlier is a step in this direction.

Now let us consider how a specializer can be used.

Perhaps the most obvious application of the specializer is the compilation of special languages. ~~The specializer~~ One of the first applications of a specializer was first developed by one of us (Dixon) as a means of

compiling clauses written in first order predicate calculus (see Fig. 6). After this was successfully done, it was realized that the specializer could also be used to compile any other "language" provided that the language was defined by an interpreter written in LISP. For example, the ZIP language was defined long before the specializer was conceived. ZIP was first implemented by an interpreter called MATCH which was quite slow.

Since this interpreter was so slow, a compiler called DEFPAT was written to compile ZIP patterns into recognition functions. DEFPAT was a conventional hand-made compiler. It was a rather laborious programming job.

After the development of the specializer, it was found that patterns could be compiled just by specializing MATCH with the pattern as quoted data. The LISP code produced by the specialization of MATCH was similar, in quality, to that produced by DEFPAT.

In contrast, the compilation of predicate calculus required a new interpreter. The original one, a resolution program based on Robinson's unification algorithm, would not specialize well with one argument known. Consequently, an entirely new program had to be written to get practical specialization.

It is clear that the specializer can be used as a general compiler and that such use is practical and convenient in at least some cases.

It is also clear that, in theory at least, the specializer can be used to partially evaluate itself and thus function as a compiler-compiler. This has not actually been done yet. But there does not seem to be any great problem in doing so.

It is possible that the partial evaluation technique will ultimately prove to be the best technique for general compiling and compiler generation. This possibility is based on the following assumptions.

1. A sophisticated specializer followed by a good optimizer will prove to be a highly efficient generator of code. (Optimization is discussed in [1, 3, 17])

2. The writing of an interpreter in some convenient base language is a very handy way to define the syntax and semantics of an arbitrary new language.

3. There is considerable freedom possible in the choice of a base language because the specializer-optimizer may be followed by a conventional compiler so that the language ultimately generated is different than the base language. This was actually done in [6, 8]. The sequence here was (1st order predicate calculus) ⟶ (LISP) ⟶ (MACHINE LANGUAGE)

4. It is possible to write a specializer-optimizer which has the desirable property of always producing correct code and to prove that it has this property. (This has been done for one case.) [6, 7].

5. It is not difficult to write an interpreter which correctly describes a new language without error. For one thing, writing an interpreter automatically forces resolution of semantic ambiguities. [9].

6. If properties (5) and (6) are attained, then the code generated will be free of error, for any proper statement in the new language.

There are still a great many questions to be answered about specialization. Perhaps the logical next step would be to design an improved specializer for LISP. Three areas seem quite fruitful for the immediate future:

1. Heuristic rules to ~~reduce~~ control expansion and hold down the size of the specialized program.
2. Optimization either integrated with or following the specializer.
3. Improved capabilities for dealing with partial knowledge about the values of variables. The pattern specializer is a step in this direction.

V: CONCLUSIONS

1. A new technique, partial evaluation, for designing a generalized compiler has been presented.

2. The same technique can also be used as a compiler-compiler.

3. Some algorithms for partial evaluation have been described.

4. One algorithm has been proved to be correct.

5. The good and bad points of the various algorithms have been discussed.

6. A program to implement one such algorithm has been written.

7. Several examples were given of the use of this program as a general compiler.

8. An extention of this technique, the pattern specializer, has also been implemented.

9. The practical value of the specializer in at least some applications has been established.

10. The future possibilities of this technique were discussed.

1 Ales, F. E.) "[illegible]" pp 239-307 [illegible] 1967

2 Burstall, R. M. and Popplestone, R. J., POP-2 reference manual. Machine Intelligence 2, 1968, pp 205-249.

3 Busam, Vincent A. and Englund, Donald E.: Optimization of Expressions in FORTRAN. Com. of ACM, Vol. 12, No. 12, December, 1969, 666-674.

Bibliography

4 CHANG, C.L., LEE, R. C. T., and Dixon, J. K. "The Specialization of Programs By Theorem Proving" to be [illegible]

5 Church, A.: The Calculi of Lambda-Conversion, Princeton University Press, Princeton, N. J., 1941.

6 Dixon, J.K., "An Improved Method for Solving Deductive Problems on a Computer by Compiled Axioms," Doctoral Dissertation, University of Calif. Davis, Dept. of Applied Science Engineering, Sept. 1970, (available from University Microfilm, Ann Arbor, Mich.)

7 Dixon, J.K., "The Specializer: A Method of Automatically Writing Computer Programs," (to be published).

8 Dixon, J.K., "Experiments with a Z-Resolution Program," (to be published).

9 1) Feldman, J. A., A formal semantics for computer-oriented languages. Comput. Ctr., Carnegie Institute of Technology, 1964.

Futamura, Y. "Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler" J. Inst. [illegible] Engineers of Japan (in Japanese, to be published)

[illegible] "[illegible] between ALGOL 60 and [illegible] notation." CACM [illegible] 1965, pp 89-101, pp [illegible]

12 Lombardi, L.A. and Raphael, Bertram: "LISP as the Language for an Incremental Computer." The Programming Language LISP: Its Operation and Applications. The M.I.T. Press, Cambridge, Mass., 1964, 204-220

13 McCarthy, J. "Towards a Mathematical Science of Computation." Proc. IFIP Congress 1962

14 McCarthy, J.: A Basis for a Mathematical Theory of Computation. Computer Programming and Formal Systems (Eds. Braffort & Hirschberg), Amsterdam, North Holland, 1963, 33-70.

15 McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I.: LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1962.

16 McCarthy, J. and Painter, J.: Correctness of a Compiler for Arithmetic Expressions. Stanford Artificial Intelligence Project Memo #40, April, 1966.

17 Nievergelt, J.: On the Automatic Simplification of Computer Programs. Com. ACM, June 1965, Vol. 8, No.6, 366-7.

18 Quam, Lynn H.: Stanford LISP 1.6 Manual, Stanford Artificial Intelligence Project, Stanford, California, Dec. 31, 1968.

19 Teitelman, W.: FLIP - A Format List Processor. MIT Project MAC Memo M-263, Cambridge, Mass., Sept. 1, 1965.

TABLE I
Specialization Algorithm for PURELISP

1. QUOTEFORM — If E = (QUOTE V), a PURELISP QUOTEFORM then E' = (QUOTE V)

2. VARIABLE — If E = X, a PURELISP VARIABLE and if the A-list contains (X QUOTE V); then E' = (QUOTE V) otherwise E' = X

3. SUBRFORM — If E = (FN A_1 A_2 A_3 ... A_n), a PURELISP SUBRFORM, then first, all the arguments are specialized, then if any A'_i is not a QUOTEFORM,
E' = (FN A'_1 A'_2 A'_3 ... A'_n) otherwise E' = (QUOTE E^*)

4. EXPRFORM — If E = (FN A_1 A_2 A_3 ... A_n) and if the length of the A-list is greater than some constant DMAX then E is treated like SUBRFORM above. Otherwise the EXPRATOM, FN, is replaced by the LAMBDA function assigned to FN and the resulting LAMBDAFORM is treated as shown below.

5. LAMBDAFORM — If E = ((LAMBDA(X_1 X_2 ... X_n) E_1) A_1 A_2 ... A_n) first all the arguments (A_1 A_2 ... A_n) are specialized then E_1 is specialized using an A-list which includes the variables X_1 through X_n paired with the corresponding arguments. If any A'_i is not a QUOTEFORM
E' = ((LAMBDA (X_i X_j ... X_n) E'_1) A'_i A'_j ... A'_m)
where (A_i A_j ... A_m) are all of the A's excepting those which are QUOTEFORM's and those which are variables identical to the corresponding X. However, if all A'_i are excluded then: E' = E'_1.

6. CONDFORM If E = (COND(P_1 V_1) (P_2 V_2) ... (P_n V_n)) each PV pair is specialized and action taken as indicated below

	VALUES P'	V'	ACTION Current Pair	Rest of List
1	NIL	-	drop	continue
2	T	V'	keep	drop
3	T	ERR	drop	drop
4	?	V'	keep	continue
5	?	ERR	drop	continue
6	ERR	-	drop	drop

where NIL means (QUOTE NIL)
T means some quoted value other than NIL
? means some unquoted value
ERR means an error message
- means anything (not necessary to specialize)
V' means anything but an error message

If two or more pairs are kept during the above processing then

$$E' = (\text{COND}(P'_i\ V'_i)\ (P'_j\ V'_j)\ \ldots\ (P'_m\ V'_m))$$

which includes only the kept pairs. If only one pair (P'_i V'_i) is saved $E' = V'_i$. If no pairs are kept then E' = ERR.

Figure X Summary of PURELISP Specialization

1. VARIABLE:

$$x \rightarrow x \mid (\text{QUOTE } x^*)$$

2. LAMBDAFORM

$$((\text{LAMBDA } (x\ y)\ E)\ A_x\ A_y) \rightarrow ((\text{LAMBDA}(x\ y)\ E')\ A_x'\ A_y') \mid ((\text{LAMBDA } (x)\ E')\ A_x') \mid E' \mid (\text{QUOTE } E^*)$$

3. QUOTEFORM

$$(\text{QUOTE } V) \rightarrow (\text{QUOTE } V)$$

4. CONDFORM

$$(\text{COND}(P_1\ V_1)\ (P_2\ V_2)\ (P_3\ V_3)\ \ldots) \rightarrow (\text{COND}(P_1'\ V_1')(P_2'\ V_2')(P_3'\ V_3')\ldots \mid (\text{COND}(P_1'\ V_1')(P_3'\ V_3')\ \ldots) \mid \ldots \mid V_n' \mid (\text{QUOTE } V_n^*)$$

5. EXPRFORM

$$(\text{FN } A_1\ A_2\ \ldots) \rightarrow (\text{FN } A_1'\ A_2'\ \ldots) \mid$$

same as LAMBDAFORM

6. SUBRFORM

$$(\text{FN } A_1\ A_2\ \ldots) \rightarrow (\text{FN } A_1'\ A_2'\ \ldots) \mid (\text{QUOTE } E^*)$$

Figure 1 Specialization of TRANS

```
(DEFPROP TRANS
         (LAMBDA(A B C)
          (COND ((NULL C) F)
                ((ATOM C) (COND ((ELEP C B) (FIN1 A B C)) (T C)))
                (T (CONS (TRANS A B (CAR C)) (TRANS A B (CDR C)))))
          (NOTE*
               *
               (TRANSFORM A AS B WAS MADE INTO C)
               (B & C ARE LINKED BY COMMOM ATOMS)
               (A & B ARE LINKED BY COMMON STRUCTURES)))
         EXPR)
(DEFPROP ELEP
         (LAMBDA(E L)
          (COND ((ATOM L) (EQ E L))
                ((EQUAL E L) T)
                (T (OR (ELEP E (CAR L)) (ELEP E (CDR L)))))
          (NOTE *
                *
                (PREDICATE * TELLS IF E IS AN ELEMENT)
                (IN ANY PART OF ARBITRARY EXPRESSION L)))
       EXPR)
(DEFPROP FIN1
         (LAMBDA(A B C)
          (COND ((NULL B) F)
                ((ATOM B) (COND ((EQ B C) A) (T F)))
                ((ELEP C (CAR B)) (FIN1 (CAR A) (CAR B) C))
                (T (FIN1 (CDR A) (CDR B) C)))
          (NOTE *
                *
                (FIN1 RETURNS THE ELEMENT OF A WHICH)
                (CORRESPONDS TO THE ATOM C IN B)
                (A AND B HAVE PARALLEL STRUCTURE)))
         EXPR)
```

TEST ARGUMENTS: A = (A B (K L) (Z X)) B = (X Y (Q R) Z)
C = (Z X (R Q) NEW)

(TRANS A B C) = ((Z X) A (L K) NEW) (time = 150 ± 5 MS)

(DO A) = ((Z X) A (L K) NEW) (time = 8 ± 3 MS)

DO is a specialization of TRANS for B & C fixed, variable specialization time: 15550 MS specialization time ratio 15550/140 = 60

```
(DEFPROP DO
         (LAMBDA(A)
          (CONS (CAR (CDR (CDR (CDR A))))
                (CONS (CAR A)
                      (CONS
                       (CONS
                        (CAR (CDR (CAR (CDR (CDR A)))))
                        (CONS (CAR (CAR (CDR (CDR A))))
                              (QUOTE NIL)))
                       (QUOTE (NEW))))))
       EXPR)
```

Figure 2 Specialization of MATCH

```
(DEFPROP MATCH
        (LAMBDA(P D)
         (COND ((ATOM P) (MAT1 P D))
               ((ATOM D) F)
               (T
                (AND (MATCH (CAR P) (CAR D))
                     (MATCH (CDR P) (CDR D)))))
        (NOTE *
              *
              (MATCH IS A PREDICATE WHICH RETURNS T)
              (IFF THE PATTERN P MATCHES THE DATA D)))
      EXPR)
(DEFPROP MAT1
         (LAMBDA(G E)
          (COND ((GET G (QUOTE PATF)) (EVAL (LIST G (QT E))))
                ((GET G (QUOTE PATFG))
                 (EVAL (LIST G (QT F) (QT E))))
                ((GET G (QUOTE PATT))
                 (MATCH (GET G (QUOTE PATT)) E))
                (T (EQ G E)))
          (NOTE *
                *
                (MAT1 IS A PREDICATE WHICH RETURNS T)
                (IFF THE PATTERN ELEMENT G MATCHES THE)
                (DATA ELEMENT E (G MUST BE AN ATOM))))
        EXPR)
```

TEST ARGUMENTS: P = ($A B $) D = (A B C)

(MATCH P D) = T (time: 117 ± 5 MS)
(KP D) = T (time: 10 ± 2 MS)

KP is a specialization of MATCH for P fixed D variable.

Specialization time 3167 ± 50 MS specialization time ratio = 3167/107 ≃ 30

```
(DEFPROP KP
         (LAMBDA(A2)
          (COND ((ATOM A2) (QUOTE NIL))
                (T
                 (AND (EVAL (LIST (QUOTE $A) (QT (CAR A2))))
                      (COND
                       ((ATOM (CDR A2)) (QUOTE NIL))
                       (T
                        (AND (EQ (QUOTE B) (CAR (CDR A2)))
                             (COND
                              ((ATOM (CDR (CDR A2))) (QUOTE NIL))
                              (T
                               (AND
                                (EVAL
                                 (LIST (QUOTE $)
                                       (QT (CAR (CDR (CDR A2))))))
                                (EQ (QUOTE NIL)
                                    (CDR (CDR (CDR A2))))))))))))))
      EXPR)
```

Figure 4 Specialization of ELEP when the Second Argument is a Pattern.

(ELEP x1 x) x = (K L A B) x1 = B

evaluation time: 17 ± 4 MS

specialization for x = ($A $A A B)

specialization time = 2300 ± 100 MS

(EP x1 x) x = (K L A B) x1 = B

evaluation time: 10 ± 3 MS

specialization ratio $\frac{2300}{7} \sim 300$

```
(DEFPROP EP
         (LAMBDA(X1 X)
          (COND ((EQUAL X1 X) (QUOTE T))
                (T
                 (OR (EQ X1 (CAR X))
                     (COND
                      ((EQUAL X1 (CDR X)) (QUOTE T))
                      (T
                       (OR (EQ X1 (CAR (CDR X)))
                           (COND
                            ((EQUAL X1 (QUOTE (A B))) (QUOTE T))
                            (T
                             (OR (EQ X1 (QUOTE A))
                                 (COND
                                  ((EQUAL X1 (QUOTE (B))) (QUOTE T))
                                  (T
                                   (OR (EQ X1 (QUOTE B))
                                       (EQ X1
                                           (QUOTE NIL)))))))))))))))
    EXPR)
```

Figure 5 Symmetry Axiom of EQ S-compiled

```
(DEFPROP RDG
        (LAMBDA (LA LB)
         (COND
          ((NEG LA)
           (COND ((NEG LB) (QUOTE NIL))
                 (T
                  (COND
                   ((EQ PA (QUOTE EQ))
                    (COND
                     ((EQ PB (QUOTE EQ))
                      (OR
                       (RES4
                        (CONS
                         (CAR  (CDR  EA))
                         (CONS  (CAR EA)  (QUOTE NIL)))
                        EB)))
                     (T (QUOTE NIL))))
                   (T (QUOTE NIL))))))
          (T
           (COND
            ((NEG LB)
             (COND
              ((EQ PA  (QUOTE EQ))
               (COND
                ((EQ PB  (QUOTE EQ))
                 (OR
                  (RES4
                   (CONS  (CAR  (CDR  EA))
                          (CONS  (CAR EA)  (QUOTE NIL)))
                   EB)))
                (T (QUOTE NIL))))
              (T (QUOTE NIL))))
            (T (QUOTE NIL))))))
        EXPR)
```

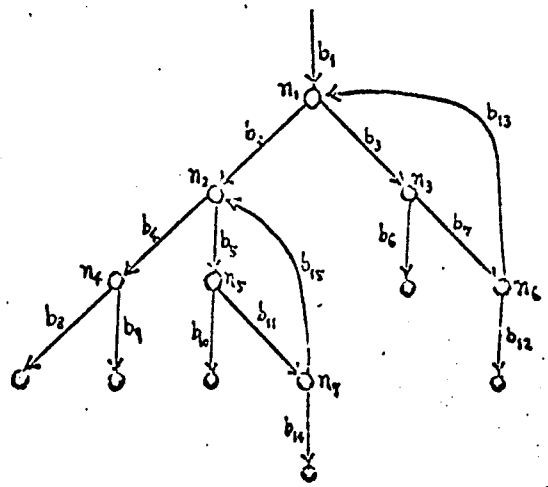

Fig. 6 — Graph representation of computation process π.

$n_1, \ldots, n_7$ are names given to the nodes. $b_1, \ldots, b_{15}$ are names given to the branches.

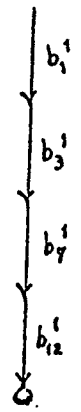

Fig. 7 — Example 1

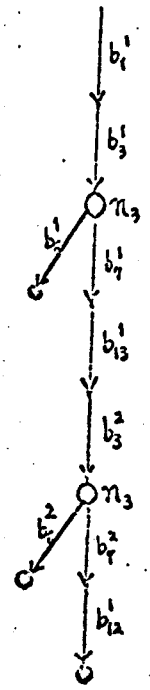

Fig. 8 — Example 2

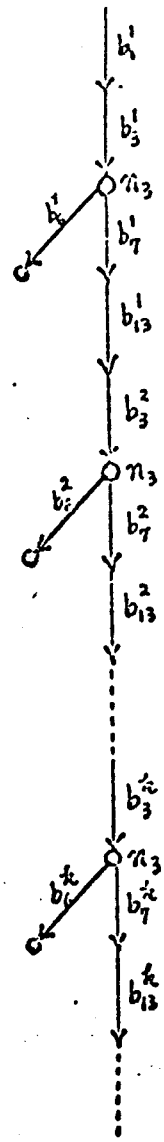

Fig. 9 — Example 3 (non-terminating)

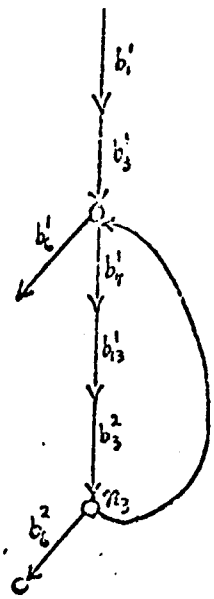

Fig. 10 — Example 3 (terminating)

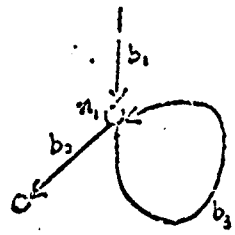

Fig. 11 — Example 4