Partial Computation of Programs

Yoshihiko Futamura

Central Research Laboratory, HITACHI,LTD.
Kokubunji, Tokyo, Japan

Abstract

This paper attempts to clarify the difference between partial and ordinary computation. Partial computation of a computer program is by definition "specializing a general program based upon its operating environment into a more efficient program". It also shows the usefulness of partial computation. Finally, the formal theory of partial computation, technical problems in making it practical, and its future research problems are discussed.

The main purpose of this paper is to make partial computation effectiveness widely known. However, two new results are also reported:

(1) a partial computation compiler, and

(2) a tabulation technique to terminate partial computation.

1. Introduction

Research on automating partial computation has been conducted in Sweden, Russia, the United States, Italy, Japan etc. for some time. Rapid progress has been made in this field in the last decade. Recent research achievements have indicated the practical utility of partial computation.

Partial computation of a computer program is by definition "specializing a general program based upon its operating

environment into a more efficient program" (This is also called mixed computation[14] or partial evaluation [6, 16, 20, 23 etc.]). For example, let us assume that a programmer has written program f having two input data k and u. Then let f(k,u) be the computation performed by f. Let us further assume that f is often used with k=5. In this case, it is more efficient to produce a new program ,say f_5, by substituting 5 for k in f and doing all possible computation based upon value 5, as well as iteratively compute $f_5(u)$, than to compute f(5,u) iteratively changing values of u.

Partial computation is an operation automatically producing f_5 when f(5,u) is given. Contrary to partial computation, ordinal computation described, for example, in FORTRAN or COBOL does not produce f_5 but produces an error message "u is undefined". Therefore, generating f_5 from f(5,u) must be done by a FORTRAN or COBOL programmer by himself. Thus, present computer's lack of partial computation ability forces programmers to do partial computation by themselves to write efficient programs. Since general or standard program specialization is often performed in everyday program production activity, automating the specialization is expected to greately improve program productivity.

This paper attempts to clarify the difference between partial and ordinary computation. It also shows the usefulness of partial computation. Finally, the formal theory of partial computation, technical problems in making it practical, and its future research problems are discussed.

2. Partial Computation Algorithm

Usually, a computer program is designed to produce desired result based upon given data. To make discussion easier, let us consider program f processing only two input data k (for known data) and u (for unknown data). Ordinary computation can start only when both k and u are given. This is called total computation (see Fig 1).
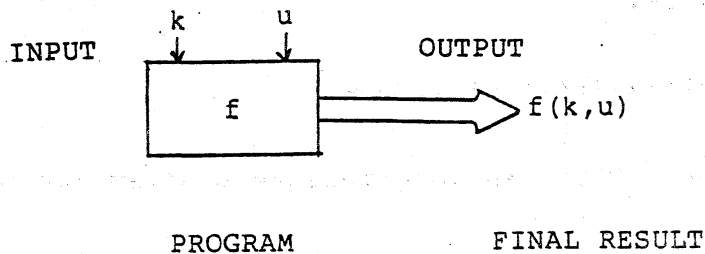

Fig 1: Total computation produces a final result.

On the contrary, partial computation can start when one of the input data, say k, is given. Content of the computation is as follows:

All computation in f that can be performed based upon k are finished. All the computation that cannot be performed without u are left intact. Thus, a new program is produced (see Fig 2).
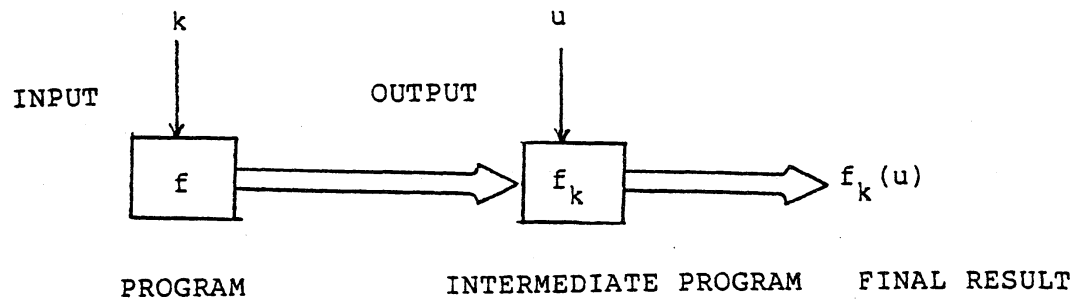
Fig 2: Partial computation produces intermediate program f_k based upon f and partial data k.

For example, let

$$f(k,u) = k*(k*(k+1)+u+1) + u*u$$

and k=2 be given. Thus, an intermediate program (or projection of f at k=2) is

$$f_2(u) = 2*(7+u) + u*u \ .$$

When data u is given to f_2, the $f_2(u)$ result is equal to that of f(2,u). Thus,the following equation holds for f_k and f:

$$f_k(u) = f(k,u) \quad -----E1$$

Both total computation and computation via partial computation give identical results. This f_k is called a projection of f at k.

Now, let us consider the reason partial computation is useful. Actually, there is a case in which partial computation is not useful.

When f is computed just once with k=2 and u=1 in the above example, computation via f_2 is less efficient than direct

computation of f(2,3).

However, when f(k,u) is iteratively computed with k fixed to 2, and u varied from 1 to 1000 by 1, the situation may be reversed.

The reason is obvious:

Assume that each addition and multiplication takes time a, and producing an intermediate program f_2 takes time p (see Fig 3).
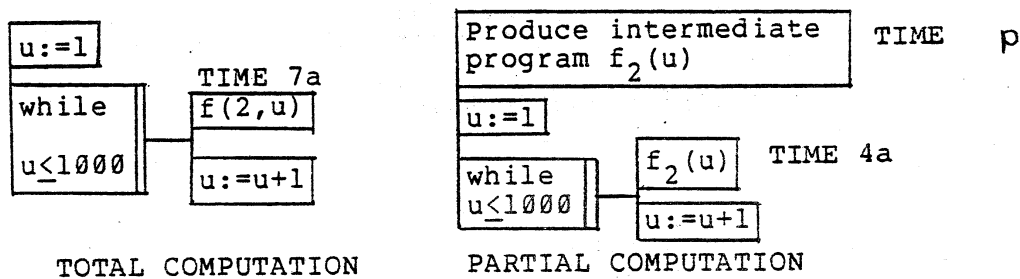

TOTAL COMPUTATION   PARTIAL COMPUTATION

Fig 3: Partial computation is often useful when f is computed repeatedly.(Programs in this figure are described in PAD. See Appendix.1).

time for computing f(2,u) = 7a

time for computing f_2(u) = 4a

Therefore, if computation is repeated 1000 times it takes 7000a for total computation,and p + 4000a for partial computation. Thus, if p is less than 3000a, partial computation is more efficient.

As described above, partial computation may be more efficient when a program is computed iteratively with a part of the input data fixd and the remaining data changed.

- 5 -

A little more complex ·example of partial computation is given below to clarify its difference from algebraic manipulation and symbolic execution.

A program that computes u^5 and stores its value to z is shown in Fig 4. The partial computation result of f at k=5, i.e. an intermediate program, is shown in Fig 5.
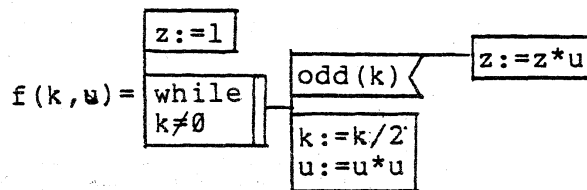
$$f(k,u)= \boxed{\begin{array}{l} \boxed{z:=1} \\ \boxed{\begin{array}{l}\text{while}\\ k\neq 0\end{array}} \end{array}} \quad \boxed{\text{odd}(k)} \quad \boxed{z:=z*u} \\ \boxed{\begin{array}{l}k:=k/2\\ u:=u*u\end{array}}$$

Fig 4: f computes $z:=u^k$

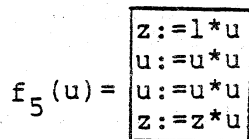$$f_5(u)= \boxed{\begin{array}{l} z:=1*u \\ u:=u*u \\ u:=u*u \\ z:=z*u \end{array}}$$

Fig 5: $f_5(u)= f(5,u)$

The author hopes readers are aware that the result of partial computation (i.e. projection) is a program while that of algebraic manipulation or symbolic execution is an algebraic expression.

A very important point is that partial computation procedure itself can be a computer program. This partial computation program is called projection machine, partial computer or partial evaluator. By letting it be α, we get the below equation:

$$\alpha(f,k) = f_k \quad \text{-----E2}$$

Now, consider αf, which is the result of partial

computation of α at f.

From E1 we get $\quad \alpha_f(k) = \alpha(f,k)$

From E2 we get $\quad \alpha(f,k) = f_k$

Therefore

$$\alpha_f(k) = f_k \quad \text{-----E3}$$

This E3 is called the basic equation of partial computation.

The author hopes that readers have grasped the rough idea of partial computation thus far. It is further hoped readers understand that "Since computation performed by a computer usually contains repetitions, partial computation would seem useful".

To clarify the usefulness of partial computation, examples of its applications are described in the next chapter.

3. Applications of Partial Computation: Examples

Since a computer program usually contains repetitions, computation efficiency of a program can often be improved through partial computation. Therefore, the author believes that the basic comuptation mechanism of a computer should be established based on partial computation. He also believes that present computer's lack of partial computation ability is one of the biggest obstacle to program development. It forces programmers to do partial computation by themselves to write efficient programs. This is basically inefficient.

As Kahn pointed out in [23], program production using automatic partial computation mechanism, i.e. projection machine, could be made much easier than it is now. For readers

262

to understand the power of partial computation,four examples are shown below. They are selected due to their clearness in application areas and effectiveness (See [3] and [12] for other examples).

3.1 Automatic theorem proving

Theorem prover is a program automaticlly proving a given theorem based upon a given set of axioms.

Let theorem prover be P, axioms be A, and theorems be T. Generally, a large number of theorems are proved based upon a fixed set of axioms, e.g., axioms of Euclidian geometry or the group theory. That is, compute $P(A,T)$ with A fixed and T changed. In this case, generating P_A by partially computing P at A and iteratively computing $P_A(T)$ is more efficient than computing $P(A,T)$ iteratively.

For example, consider theorem proving in Euclidian geometry. Program P_A is then a specialized theorem prover for Euclidian geometry $(P_A = \alpha(P,A))$. Theorems are proved by using this P_A.

Projection machine α for specializing P automatically was first developed by Dixon in 1971 to the best of the author's knowledge [5]. His α can partially compute programs written in pure LISP [38].

3.2 Pattern matching

Text editors like word processors have simple pattern matching functions. For example, consider the case when a text string like

THIS IS A BOOK.

is searched in order to substitute PEN in the pattern and to get the string

THIS IS A PEN.

This sort of operation for locating where a pattern is in a text string is pattern matching.

Let M be a pattern matching program, P be a pattern and T be a text string. Pattern matching is thus represented as M(P,T).

When a pattern, say BOOK, repeatedly appears in a text and when all the instances of the pattern are replaced by the other string ,say PEN, or when a pattern is used frequently, it is useful to partially compute M at the pattern ,say P, to get M_p, and to perform $M_p(T)$ repeatedly.

Emanuelson[6] implemented such a projection machine α to produce M_p by performing α (M,P) for a complicated pattern matcher M written in LISP.

3.3 Syntax analyzer

In language processing, a program called syntax analyzer is necessary for parsing a given sentence based upon a given grammar. Let S be a syntax analyzer, G be a language grammar and T be a given text. Syntax analysis then can be represented as S(G,T).

When grammar G is fixed and T is varied, it is not efficient to compute A(G,T) repeatedly. Specialized analyzer A_G can be produced by partially computing A at G. The author discussed the partial computation method for BTN(Basic Transition Network) analyzer in [8]. Ershov's group also partilly computed LR(1)

264

analyzer S at PASCAL grammar G[28].

Let G be FORTRAN grammar and T be a FORTRAN program. Syntax analysis in FORTRAN compiler can then perform the same function as S(G,T). Note that G is fixed and only T is varied in this case. Thus, $S_G(T)$ can be used instead of S(G,T).

Generally speaking, projection machine α produces less efficient syntax analyzer S_G than a good human partial computer nowadays. Therefore, very efficient S_G for popular programming languages like FORTRAN and COBOL are implemented by human programmers at the cost of a lot of man-hours. However, in the future it may be possible that S_G produced by α (S,G) will become more efficient than that made by human programmers.

3.4 Compiler generation

The three examples described above are generation of intermediate programs from data like axioms, patterns, or grammars. Therefore, they may be called compilation of axioms, patterns, or grammars instead of partial computation.

In this section, problems in automatically generating compilers are discussed. A compiler is a program to transform a program in some language (this is called source language) into a program in another language (this is called object language).

We often hear "A BASIC program runs slowly because it is executed by an interpreter. If we have a good BASIC compiler as FORTRAN, it will run much faster".

What is the interpreter in this case?

Interpreter,say I, is a program to perform specified computations that analyze the meanings of a given program,say P,

- 10 -

based upon given data, say D. Thus, running a program by interpreter means performing I(P,D). While a compiler, say C, translates a source program, say P, to an object program C(P). When this C(P) runs with data D, the result is the same as that of I(P,D). This produces the following equation:

$$C(P)(D) = I(P,D)$$

While $I_p = \alpha(I,P)$. Thus, we get the below equation .

$$I_p = C(P) -----E4$$

(First equation of partial computation)

Therefore I_p can be regarded as an object program of P.

From the above discussion, it is clear that the repetitive computation of I(P,D) with P fixed and D varied is less efficient than that of $I_p(D)$ after producing I_p by compiler C.

Now if we compute I(P,D) once with both P and D fixed, is a compiler still useful ? Generally, partial computation is not useful when a computation is performed just once. However, it is also true that generally a program runs faster after it is compiled even if it is executed only once. Why is this ?

The reason is that a program P itself contains repetitive computation (i.e. loops). Usually, a program contains a loop for performing repetitive, say 10 or 10000 times, computation. When this program is executed by an interpreter, the interpreter repeats analysis of a loop. However, partial computation of interpreter I at program P (i.e. $\alpha(I,P)$) finishes the part of meaning analysis which can be performed without knowing data D, e.g. analysis of a program structure. So, when a loop in an object program I_p is performed repetitively, there remains just a small part of computation performed by an interpreter I. Thus,

even when I(P,D) is computed just once for given data P and D, compiling P is often useful. If P contains no loops, compiling P is useless.

From the discussion above, readers may see the significance of a compiler. The remaining part of this section deals with a compiler generation method based upon partial computation.

If P is an object program, then can α which produces I_P from I and P be regarded as a compiler ? The answer is no. Generally, α is a complicated program and computation of α (I,P) takes a long time. Therefore, regarding α as a compiler is not practical.

Now, noting that α itself is a program with two data, let us look at α_I which is produced by partially computing α at I. From equation E3, equation E5 is derived by substituting each f and I for k and P, respectively.

$$\alpha_I(P) = I_P \text{-----E5}$$

(Second equation of partial computation)

Since I_P is an object program and α_I transforms a source program P to I_P, α_I has the same function as that of a compiler. Furthermore, in α_I, computation of α concerning I has been finished. Thus, it is not wrong to regard α_I as a compiler. Generally, implementing an interpreter is much easier than that of a compiler [16]. Therefore, if we have a good partial computer α, a compiler can be made automatically through $\alpha(\alpha, I)$ after easily implementing I [16].

Now, what is the result of partial computation of α at itself (i.e. α_α) ?

Equation E6 below is derived from equation E3 by

substituting each α and I for f and k, respectively.

$$\alpha_\alpha(I) = \alpha_I \text{-----E6}$$

(Third equation of partial computation)

Since α_I is a compiler, from E6 above, α_α is regarded as a compiler-compiler which generates a compiler α_I from an interpreter I.

From the above discussion, the relationship between language processors like interpreter, compiler and compiler-compiler is clarified (see [15]).

Let I-language be a language defined by an interpreter I. From equation E6, I-language compiler can be produced automaticlly through $\alpha_\alpha(I)$. Now, let us substitute α for I in E6 or consider α as an interpreter. Then what is α-language ?

Let f be a program in α-language and k be data. From E1, the result of performing f with data k is:

$$\alpha(f,k) = f_k .$$

Substituting α for I in E6, we get:

$$\alpha_\alpha(\alpha) = \alpha_\alpha .$$

This means α_α is an α-language compiler. Therefore, $\alpha_\alpha(f)$ is an object program of f and we get equation E7 below.

$$\alpha_\alpha(f)(k) = f_k \text{-----E7}$$

(Forth equation of partial computation).

From the discussion above, we see that, in general, partial computation of f at k can be done more efficiently through compiling f by α_α than by directly computing $\alpha(f,k)$ (However, when $f=\alpha$, computing $\alpha_\alpha(k)$ is sufficient). Therefore α_α is the partial computation compiler desired. However, the author has not heard of a report concerning the execution of $\alpha(\alpha,\alpha)$

- 13 -

to produce α_α for practical α..

The author implemented an α and an interpreter I in LISP for ALGOL-like language, and tried to generate ALGOL compiler α_I in 1970 [16]. The generated compiler was not efficient enough to be used practically. Similar, but more advanced, experiment was conducted by Haraldsson [20]. He implemented partial computation program REDFUN in lisp [3,20]. REDFUN was used by Emanuelson to implement a pattern compiler [6].

Based upon partial computation and Prolog interpreter on LISP machine (LM-Prolog [37]), Kahn [23] tried to automatically generate Prolog compiler. The compiler translates a Prolog program into a LISP program.

To do so ,from the discussion in this section, it is sufficient to have a LISP program α which can partially compute a LISP program. However, good α is very hard to write (Haraldsson told in [20] his partial evaluator REDFUN-2 was 120 pages in prettyprinted format). Therefore, Kahn tried to write his α in Prolog [35,37,39]. Since Prolog has a powerful pattern matching ability and theorem proving mechanism, it seems easier to write complicated α in Prolog than in LISP. The outline of Kahn's method is described below.

First, Kahn implemented the following two programs:

(1) L : Prolog interpreter written in LISP.

Let D be database and S be a statement in Prolog, then L performs L(D,S). Interpreter L is similar to a theorem prover described in Section 2.1. Database D and S correspond to axioms and a theorem, respectively.

(2) P : LISP partial computation program (projection

. machine) written in Prolog.

Note that P is database for L. Let f be a LISP program, k and u be its input data, and [f,k] be a Prolog statement describing f and k. Then the equation below holds.

$$f_k = L(P, [f,k]) -----E8$$

He then gets L_p by performing $L(P, [L,P])$ from E8 which is again a LISP program.

From E1:

$$L_p([f,k]) = L(P, [f,k]) -----E9$$

From E8 and E9, $L_p([f,k]) = f_k$ is derived. Therefore, L_p is a LISP program producing f_k from f and k. This means that L_p is a LISP partial computer described in LISP.

Let L_p be α. Then from equation E5, by performing $\alpha(\alpha, L)$, we can get a Prolog compiler α_L translating a Prolog program into a LISP program.

Kahn's method of producing a Prolog compiler seems very promising because:

(1) A partial computation method of a LISP program is becoming clearer through its long time research.

(2) A very convenient language like Prolog for describing partial computer is becoming available.

4. Theory of Partial Computation

It was in the 1930's that Turing, Church, Kleene, etc. proposed several computation models and clarified the mathematical meanings of mechanical procedure. A computation model, in plain language, is a sort of programming language.

270

Typical computation models include the Turing machine, lambda expression and the partial recursive function.

At that time, the main research interest concerned computability, i.e. computational power of the models, not computational complexty or efficiency. Since partial computation was the same as ordinary computation, as far as computability was concerned, it did not attract research attention.

However, equation El in Chapter 2 appeared in Kleene's S-m-n theorem (It is also called the parametarization theorem or iteration theorem) of the 1930's[41,45]. A procedure to produce f_k by fixing k in f was also described in the proof of the theorem. That procedure was just like putting assignment statement k:=(value of k), i.e. k:=5, in front of f(5,u):

$$f_5(u) == \boxed{\begin{array}{|c|} \hline \texttt{k:=5} \\ \hline \texttt{f(k,u)} \\ \hline \end{array}}$$

Furthermore, the partial computation of α itself, which appears in the third equation of partial computation, was also used in the proof of Kleene's Recursion Theorem[41](1938). However, he only used partial computation in its simplest form to fix the value of a variable.

While Turing machines and partial recursive functions were formulated to describe total computation, Church's lambda expression was based upon partial computation. Inputs to a lambda expression are also lambda expressions, and the result is, again, a lambda expression. This computation is called lambda conversion. Furthermore, when a lambda expression corresponding to f(5,u),for example, is computed with u undefined, the result

is a lambda expression corresponding to $f_5(u)$ in which computation concerning k=5 is finished. Lambda conversion of $f_5(u)$ with, again for example, u=6 produces the same result as f(5,6) (by the Church-Rosser Theorem). That is to say, $f_k(u)=f(k,u)$. Thus lambda conversion is partial computation.

As described above, the concept of partial computation already existed in the 1930's. However, implementation of a projection machine and its application to real world problems started in the 1960's after the programming language LISP began to be widely used (see Appendix 2).

Problems in making a practical projection machine will be discussed in the next chapter. The rest of this chapter deals with the theory behind the projection machine.

The relationships between a projection machine and a language processor, i.e. equations E4, E5 and E6, have been discussed by Futamura[16,17], Beckman[3], Ershov[8,12,14] and others.

Formal treatment of a projection machine and its proof of correctness were presented by Ershov in [9,13,14]. He dealt with ALGOL-like programs that included assignment, conditional and GOTO statements in [9] and a recursive program schema similar to pure LISP in [13]. The author believes [9] and [13] are landmark papers that establish the theoretical foundations for a projection machine.

It is important for projection machine α to satisfy the following three conditions:

(1) Correctness: Program α should satisfy the equation $\alpha(f,k)(u)=f(k,u)$.

272

(2) Efficiency improvement: Program α should perform as much computation as possible based upon given data k.

(3) Termination: Program α should terminate on partial computation of as many programs as possible. Termination at α (α,α) is most desirable.

In condition (2), the meaning of "as much computation as possible" is not mathematically clear. This can be rephrased as "all possible computation" for a simple language as a recursive program schema. However, for a language with assignment statements or side effects, it is not easy to make computations concerning k only without spoiling condition (1). Therefor, the author believes that it is of primary importance to establish a projection machine satisfying conditions (1), (2) and (3) for a recursive program schema that makes theoretical discussion easiest to do.

Ershov[13] has established α satisfying (1) and (2) for a recursive program schema. This paper attempts to describe a new α that satisfies all three conditions based upon Ershov[13]. The description is rather informal in order for readers to understand α intuitively.

Roughly speaking, a recursive program schema is a program consisting of only three different components:
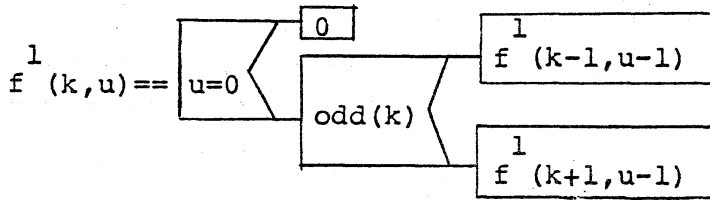
(1) Condition:
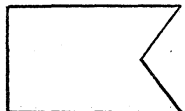
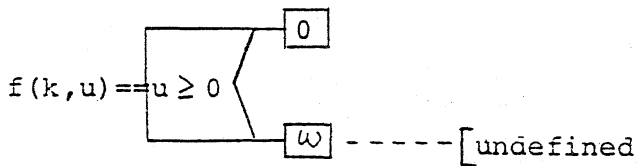(2) Expression:
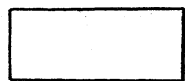
(3) function definition:

$$\underline{\underline{\qquad}} \quad or \quad \xrightarrow{\text{def}}$$

Six examples of recursive program schema are described below:

Example 1 :

$$f^1(k,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{odd(k)} \Big\langle \begin{array}{l} \boxed{f^1(k-1,u-1)} \\ \boxed{f^1(k+1,u-1)} \end{array} \end{array}$$

This means a program that takes value 0 if u>=0 or loop indefinitely if u<0; i.e.

$$f(k,u) == \boxed{u \geq 0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{\omega} \end{array} -----[\text{undefined}$$
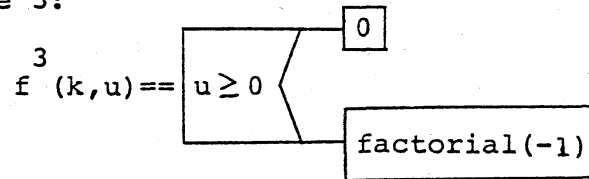
Following Examples 2 to 4 describe this same function in different forms.

Example 2:

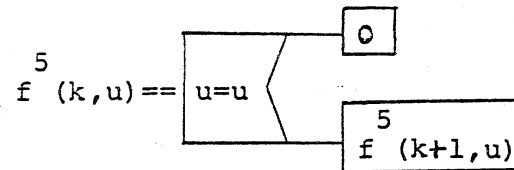$$f^2(k,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{f^2(k+],u-1)} \end{array}$$

274

Example 3:

$$f^3(k,u) == \boxed{u \geq 0} \begin{cases} \boxed{0} \\ \boxed{\text{factorial}(-1)} \end{cases}$$

$$\text{factorial}(x) == \boxed{x=0} \begin{cases} \boxed{1} \\ \boxed{x*\text{factorial}(x-1)} \end{cases}$$

Example 4:

$$f^4(k,u) == \boxed{u \geq 0} \begin{cases} \boxed{0} \\ \boxed{f^4(u+1,u)} \end{cases}$$

Example 5:

$$f^5(k,u) == \boxed{u=u} \begin{cases} \boxed{0} \\ \boxed{f^5(k+1,u)} \end{cases}$$
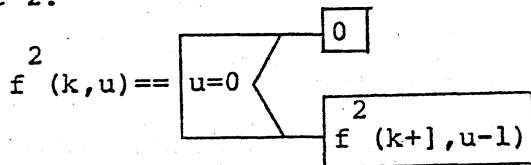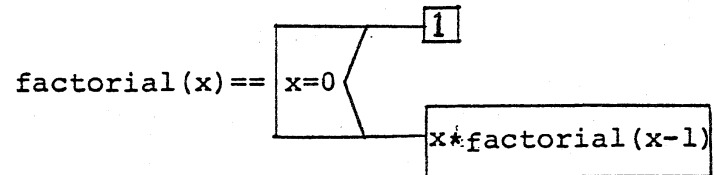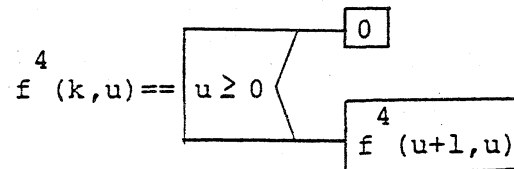
The value of f^5 is always 0.

Example 6:

$$f^6[u;k] == \boxed{\text{null}[u]} \begin{cases} \boxed{k} \\ \boxed{\text{cons}[\text{car}[u];f^6[\text{cdr}[u];k]]} \end{cases}$$
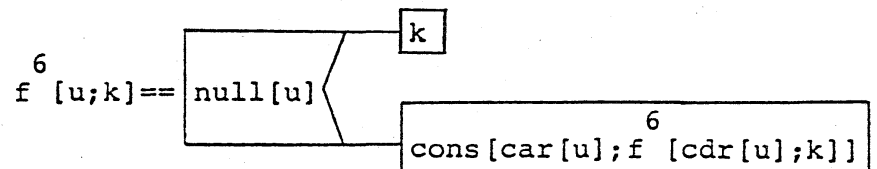
This is equal to append[u;k] in LISP.

As described in [42], a computation rule for a recursive program schema is repetition of (1) Rewriting and (2) Simplification. The computation terminates when there is no longer occasion to apply the rewriting rule.

Partial computation of f at k is performed by repetition of the following three rules:

(1) Rewriting: Replace f(k,u) by its definition body (This f(k,u) is called a semi-bound call by Ershov[13]). This rule is different from the rewriting rule for ordinary computation because it can only be applied when u is undefined.

(2) Simplification: Perform possible simplification to a program produced by the rewriting rule. This is not very different from the simplification rule for ordinary computation.

(3) Tabulation: Let E be the expression derived from the above two rules.

   (3.1) If there are semi-bound calls, say f'(k',u'), in E, then each one is replaced by $f'_{k'}(u')$. Let the resulting expression be E'. Then define f_k as E', i.e.

$$f_k \overset{def}{=} E'.$$

   (3.1.1) If there is an undefined $f'_{k'}$ in E', repeat (1), (2) and (3) for each f' and k' (i.e. partially compute f' at k').

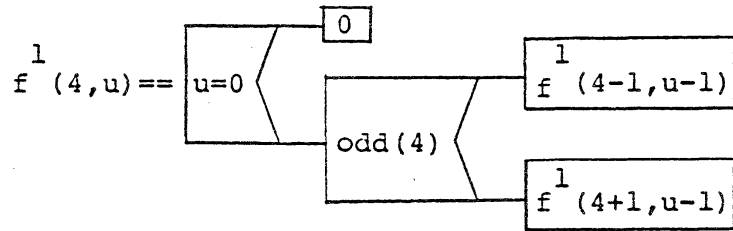   (3.1.2) If there is no undefined $f'_{k'}$ in E', terminate the computation.

   (3.2) If there is no semi-bound call in E', define f_k as E' and terminate the computation.

Thus, the discriminating characteristics of partial

computation are the semi-bound call and tabulation. Six examples are shown to illustrate the above rules:

Example 1: Partially compute $f^1(4,u)$.

(1) Rewriting: Replace k by 4 in the $f^1(k,u)$ definition.

$$f^1(4,u) == \boxed{u=0} \Bigg\langle \begin{array}{l} \boxed{0} \\[2mm] \boxed{odd(4)} \Bigg\langle \begin{array}{l} \boxed{f^1(4-1,u-1)} \\[2mm] \boxed{f^1(4+1,u-1)} \end{array} \end{array}$$

(2) Simplification: Perform computation and decide what can be done based upon the value of k, i.e. 4.

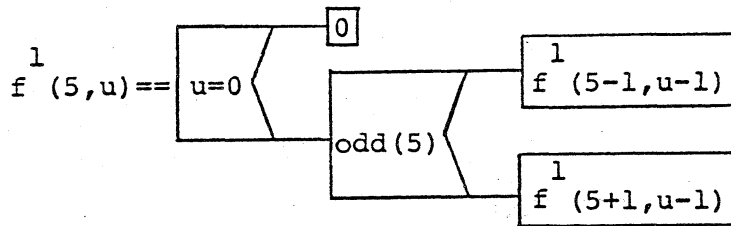$$f^1(4,u) == \boxed{u=0} \Bigg\langle \begin{array}{l} \boxed{0} \\[2mm] \boxed{f^1(5,u-1)} \end{array}$$
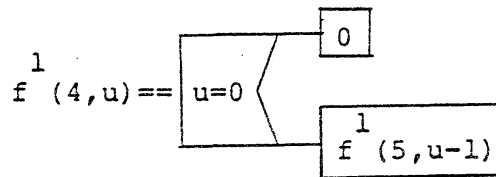
(3) Tabulation: Replace semi-bound call $f^1(5,u-1)$ by $f^1_5(u-1)$. The resulting expression is defined as f^1_4

$$f^1_4(u) \overset{def}{==} \boxed{u=0} \Bigg\langle \begin{array}{l} \boxed{0} \\[2mm] \boxed{f^1_5(u-1)} \end{array}$$
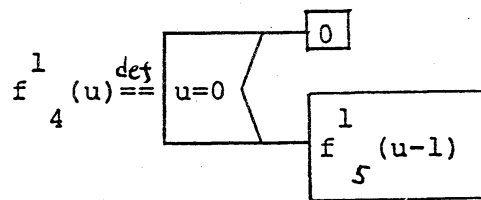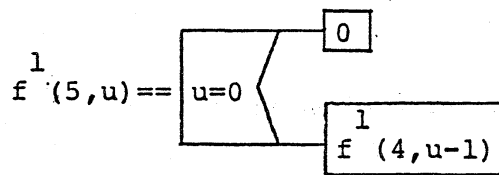
Since f^1_5 is undefined, define $f^1_5(u)$ through partial computation of $f^1(5,u)$, i.e.:

(4) Rewriting:

$$f^1(5,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{odd(5)} \Big\langle \begin{array}{l} \boxed{\; ^1\!f\;(5-1,u-1)} \\ \boxed{\; ^1\!f\;(5+1,u-1)} \end{array} \end{array}$$

(5) Simplification:

$$f^1(5,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{\; ^1\!f\;(4,u-1)} \end{array}$$

(6) Tabulation:

$$f^1_5(u) \overset{def}{==} \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{\; ^1\!f_4\;(u-1)} \end{array}$$
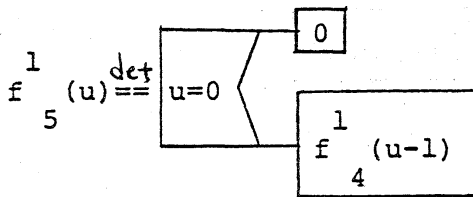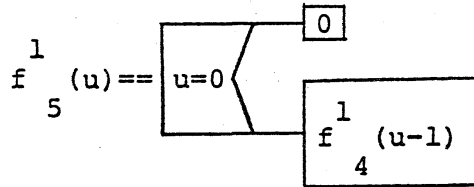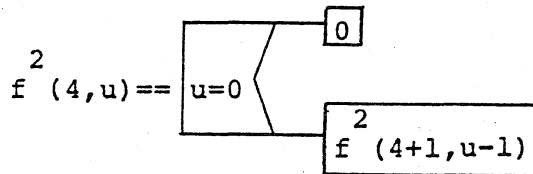
Since f^1_4 has been defined by previous tabulation, there is neither a semi-bound call nor an undefined function in the above expression. Thus, partial computation terminates. The results are the following two equations:
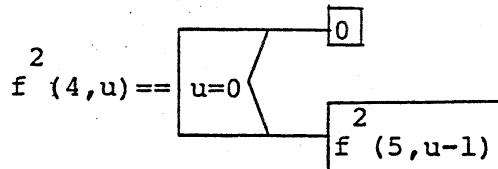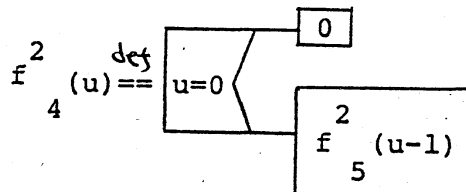
$$f^1_4(u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{\; ^1\!f_5\;(u-1)} \end{array}$$

278

$$f^1_5(u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\[4pt] \boxed{f^1_4(u-1)} \end{array}$$
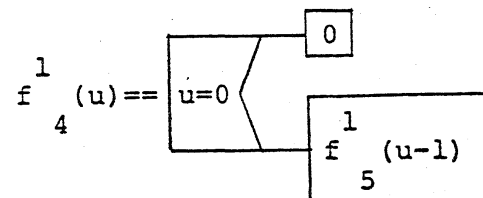
Example 2: Partially compute $f^2(4,u)$.

(1) Rewriting:

$$f^2(4,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\[4pt] \boxed{f^2(4+1,u-1)} \end{array}$$

(2) Simplification:

$$f^2(4,u) == \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\[4pt] \boxed{f^2(5,u-1)} \end{array}$$

(3) Tabulation:

$$f^2_4(u) \stackrel{def}{==} \boxed{u=0} \Big\langle \begin{array}{l} \boxed{0} \\[4pt] \boxed{f^2_5(u-1)} \end{array}$$

(4) Rewriting:

$$f^2(5,u) == \begin{cases} u=0 & \boxed{0} \\ & \boxed{f^2(5+1,u-1)} \end{cases}$$

(5) Simplification:

$$f^2(5,u) == \begin{cases} u=0 & \boxed{0} \\ & \boxed{f^2(6,u-1)} \end{cases}$$

(6) Tabulation:

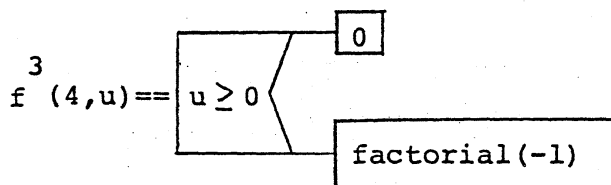$$f^2_5(u) \overset{\text{def}}{==} \begin{cases} u=0 & \boxed{0} \\ & \boxed{f^2_6(u-1)} \end{cases}$$
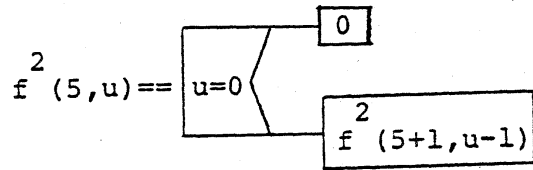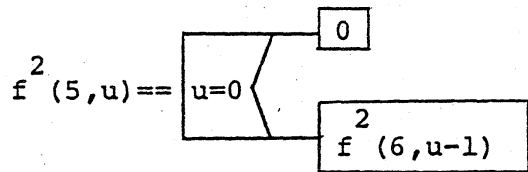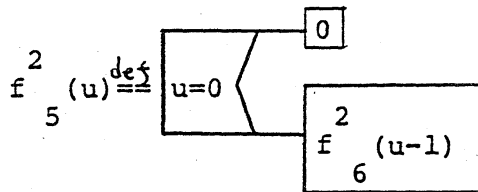
Undefined functions $f^2_6, f^2_7, \ldots$ appear indefinitely and partial computation does not terminate. Thus, f^2_4 cannot be defined.

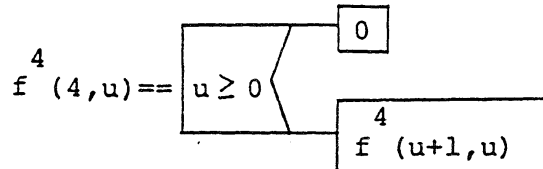Example 3: Partially compute $f^3(4,u)$.

(1) Rewriting:

$$f^3(4,u) == \begin{cases} u \geq 0 & \boxed{0} \\ & \boxed{\text{factorial}(-1)} \end{cases}$$

(2) Simplification: According to the definition of the

280

factorial function, factorial(-1) does not terminate. Thus, this partial computation does not terminate.
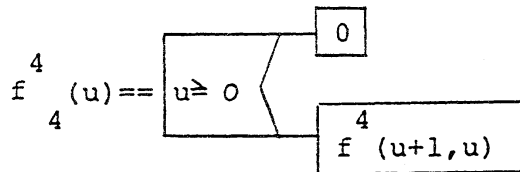
Example 4: Partially compute $f^4(4,u)$.

(1) Rewriting:

$$f^4(4,u) == \left[u \geq 0 \left\langle \begin{array}{l} 0 \\ f^4(u+1,u) \end{array} \right. \right.$$

(2) Simplification:

The same expression as the above.

(3) Tabulation:

$$f^4_{4}(u) == \left[u \geq 0 \left\langle \begin{array}{l} 0 \\ f^4(u+1,u) \end{array} \right. \right.$$
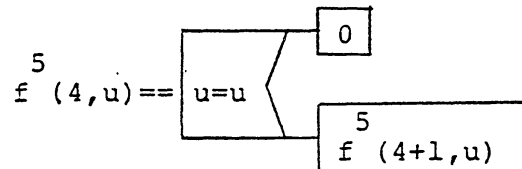
Since neither semi-bound calls nor undefined functions are included in the expression, the partial computation terminates.
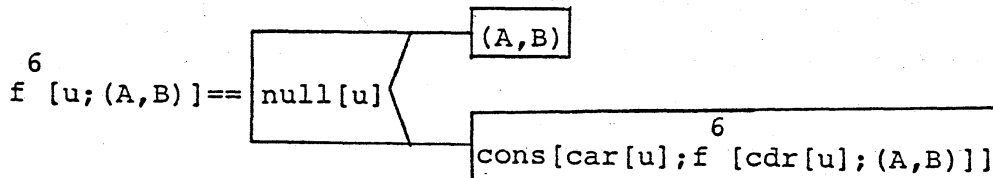
Example 5: Partially compute $f^5(4,u)$.

(1) Rewriting:

$$f^5(4,u) == \left[u=u \left\langle \begin{array}{l} 0 \\ f^5(4+1,u) \end{array} \right. \right.$$

(2) Simplification:

If this rule is powerful enough to find that u=u always holds, then partial computation terminates and the result is 0 (i.e. $f^5(4,u)=0$). However, if boolean expression u=u is not evaluated because u is unknown, partial computation of f^5 at k=5 is performed. Thus, the same as in Example 2, partial computation will not terminate.

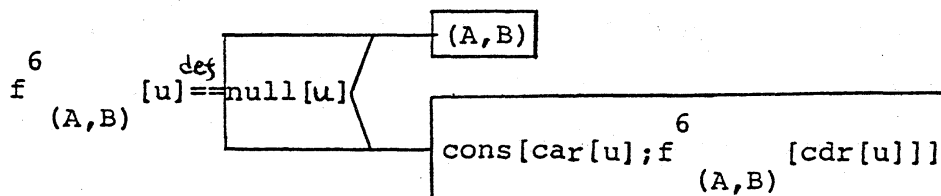Example 6: Partially compute $f^6[u;(A,B)]$.

(1) Rewriting:

$$f^6[u;(A,B)]== \text{null}[u] \begin{cases} (A,B) \\ \text{cons}[\text{car}[u];f^6[\text{cdr}[u];(A,B)]] \end{cases}$$

(2) Simplification:

The same as above.

(3) Tabulation:

$$f^6_{(A,B)}[u] \overset{def}{==} \text{null}[u] \begin{cases} (A,B) \\ \text{cons}[\text{car}[u];f^6_{(A,B)}[\text{cdr}[u]]] \end{cases}$$

Partial computation is terminated and projection $f^6_{(A,B)}$ is produced.

In the above examples, partial computation of f^2 and f^3 do not terminate, but total computation terminates, at u>=0 (see Table 1). The reasons are as follows:

(1) The value of known variable k changes indefinitely (for

- 27 -

f^2).

(2) Infinite computation that can be performed based upon known variable k and constants is included (for f^3).

Table 1: Summary of Examples.

	Total computation	Partial computation
Terminating	$f^1, \cdots, f^4, f^5, f^6$ ($u \geq 0$)	f^1, f^4, f^5, f^6
Non-terminating	$f^1, \cdots, f^4$ ($u < 0$)	f^2, f^3, f^5

As described in Chapter 3, partial computation can be applied to such data of finite structure as a pattern, a set of axioms, a grammar or a program. Therefore, case (1) above will not happen in practical application. Furthermore, case (2) will not happen either, if the programs to be partially computed are written carefully. Thus, projection machine α described above seems powerful enough to be practical.
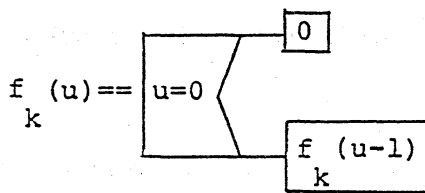
Tabulation is a unique feature of α (Tabulation technique for ALGOL-like programs was described in [18]). The first paper to mention the importance of tabulation in partial computation was probably [16]. Kahn[23] also mentioned a powerful tabulation method using Prolog.

Ershov[13] and REDFUN[6] devised mechanisms to terminate partial computation in such special cases as when the value of k does not change. For example, their α can terminate for partial

computation of f(k,u), where

$$f(k,u) == \boxed{u=0} \left\langle \begin{array}{l} \boxed{0} \\ \boxed{f(k,u-1)} \end{array} \right.$$
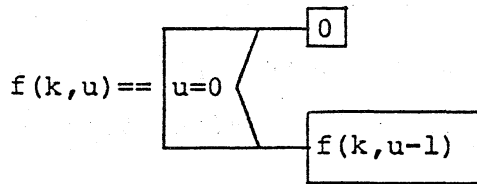
and the projection $f_k(u)$ is

$$f_k(u) == \boxed{u=0} \left\langle \begin{array}{l} \boxed{0} \\ \boxed{f_k(u-1)} \end{array} \right.$$

However, their α's do not terminate for partial computation of f^1 in Example 1 above.

5. Technical Problems in Making α Practical

One of the most energetic groups persuing how to make partial computation practical is Professor Sandewall's group in Sweden, to the best of the author's knowledge. In that group, Beckman[3], Haraldsson[20], Emanuelson[6], Komorowski[25] and Kahn[23,24] have been working on the subject for a long time. The recent achivements of Emanuelson[6] and Kahn[23] lead the author to believe that a practical projection machine can be implemented in the near future.

The group has been trying to partilly compute LISP programs. LISP is one of the best real-world languages for partial computation research because its interpreter is simple and clear, its programs can be easily manipulated as data, and its program

structure is similar to a recursive program schema. Therefore, the Sandewall group's approach of establishing α for LISP as their first step seems very sound.

As described in the previous chapter, partial computation is a repetition of (1) rewriting, (2) simplification and (3) tabulation.

Rewriting is like so-called macro expansion and procedure integration[30] in the optimization techniques of a compiler. This operation is often performed in combination with simplification (see Fig.6).

ORIGINAL PL/I
PROCEDURES

```
P:PROC(A);
  B=5;
  C=A*B;
  CALL Q(A,B);
RETURN(C);
END;

Q:PROC(X,Y);
  X=X*Y;
END;
```

AFTER INTEGRATION

```
P:PROC(A);
  B=5;
  C=A*B;
  A=A*B;
  RETURN(C);
END;
```

AFTER SIMPLIFICATION

```
P:PROC(A);
  A=A*5;
  RETURN(A);
END;
```

Fig.6 Example of the integration and simplification [30].

Partial computation is useful because it automatically improves program efficiency. However, since α produces a projection by simple rules, there still remain redundant parts in the projection, such as car[cons[x;y]] for y with no side-effect. These redundancy should be avoided by program optimization, e.g. car[cons[x;y]] should be x. This operation is algebraic manipulation of a program that is similar to transformation of $\sin^2 x + \cos^2 x$ into 1 in symbol manipulation (A more complex example

of program manipulation[23] is shown in Appendix 3).

Therefore, simplification is not a process unique to partial computation but a common one for program optimization[30]. This process is most important to the efficiency of a generated program (i.e. a projection). Program manipulation will become easier through advancement of Prolog-like languages.

Tabulation is a process used to associate program f and known data k to projection f_k. Essentially, it builds a table, like that shown below, and searches through it.

function	known data	projection
f	k1	f_{k1}
f	k2	f_{k2}
g	k3	g_{k3}
g	k4	g_{k4}
f	k5	f_{k5}
.	.	.
.	.	.
.	.	.

For example, when there is a semi-bound call f(k2,u+1), it is replaced by f_{k2}(u+1) and the table is searched for the entry of f, k2 and f_{k2}.

If the entry is found, partial computation of f(k2,u+1) is not performed again. If the entry is not found, f, k2 and the unfinished definition of f_{k2} are entered into the table and partial computation of f(k2,u+1) is performed. (A more detailed discussion of this is conducted in Chapter 4).

The tabulation technique is important not only in partial

computation but also in ordinal computation. It has been studied for some time in the field of recursive programs[31,32]. Tabulation can be performed efficiently if a powerful hashing mechanism is implemented[33].

6. Conclusion

In the last decade, the author never believed that a practical projection machine could be implemented within 10 years. However, he is now more optimistic.

The achievements of Ershov, Emanuelson, Kahn and others establish the basis for the theory and practice of partial computation. Furthermore, the comming of a commercial LISP machine and inauguration of the 5th generation computer project in Japan will encourage research in this field.

7. References

Those papers not cited in this manuscript are also included.

[Theory and practice of partial computation]

[1] Babich,G.Kh., The DecAS algorithmic language for the solution of problems with incomplete information and its interpreting algorithms. Kibernetika, No., 1974, 61-71 (Russian).

[2] Babich,G.Kh. et al., The Incol algorithmic language for computations over incomplete information. Programmirovanie,

No.4, 1976,24-32 (Russian).

[3] Beckman,L. et al., A partial evaluator and its use as a programming tool. Datalogilaboratoriet, Memo 74/34, Uppsala University, Uppsala, 1974 (also: Artificial Intelligence,vol.7, 1976, 319-357).

[4] Chang,C,-L. and Lee, R., Symbolic Logic and Mechanical Theorem -Proving, (Chapter 10.9) Academic Press, 1973.

[5] Dixon,J., The SPECIALIZER, a method of automatically writing computer programs. Div. of computer Res. and Technology, NIH: Bethesda, Md

[6] Emanuelson, P., Performance enhancement in a well-structured pattern matcher through partial evaluation. Linkoping Studies in Science and Technology Dissertations,No.55, Software Systems Research Center,Linkoping University, 1980.

[7] Ershov., A.P., Problems in many-language programming syatems. Language hierarchies and interfaces. LNCS, vol. 46, Springer-Verlag, Heidelberg, 1976,358-428.

[8]------------ , On the partial computation principle. Information Processing Letters, No. 2, 1977.

[9]------------ and Itkin, V.E., Correctness of mixed computation in Algol-like programs. LNCS,Springer-Verlag, Heidelberg, 1977.

[10]----------- and Grushetsky, V.V., An implementation-oriented method for describing algorithmic languages. Proc. of IFIP 77, 1977.

[11]-----------,A theoretical principle of system programming, Soviet Math. Dokl. Vol. 18 (1977), No. 2.

[12]-----------,On the essence of compilation. IFIP Working

288

Conference on formal description of programming concepts, August 1977.

[13]------------,Mixed computation in the class of recursive program schema. Acta Cyberneticca, Tom. 4, Fosc. 1, Szeged, 1978.

[14]------------,Mixed computation: Potential applications and problems for study, Theoretical Computer Science 18(1982) 41-67.

[15]------------,On Futamura projection. bit, Vol.12, No.14, 1980 (Japanese).

[16] Futamura, Y., Partial evaluation of computation process: an approach to a compiler-compiler. Systems Computers Controls, Vol. 2, No. 5, 1971.

[17]------------,EL1 partial evaluator. Term paper manuscript, AM260, DEAP, Harvard University, 1972.

[18]------------,Compilation of Basic Transition Networks. Term paper manuscript, AM221, DEAP, Harvard University, 1972.

[19]------------,Partial computation of computer programs. AL78-80, Inst. Electronics Comm. Engrs. Japan, 1978 (Japanese).

[20] Haraldsson, A., A program manipulation system based on partial evaluation. Linkoping Studies in Science and Technology Dissertations, No.14, Department of Mathematics, Linkoping University, 1977.

[21]------------, Experiences from a program manipulation systems. Informatic Laboratory, Linkoping University, 1980.

[22] Heuderson,P. and Morris,J.H., Jr., A lasy evaluator, Techn. Report No. 75, January 1976, Computing Laboratory, The University of Newcastle upon Tyne (also: 3rd ACM Symposium on

principle of programming languages, January , 1976.

[23] Kahn, K., A partial evaluator of Lisp written in a Prolog written in Lisp intended to be applied to the Prolog and itself which in turn is intended to be given to itself together with the Prolog to produce a Prolog compiler. UPMAIL, Dept. of Computing Science, Uppsala University, March 1982.

[24]--------, The automatic translation of Prolog programs to Lisp via partial evaluation. UPMAIL, Dept. of computing Science , Uppsala University, P.O. Box 2059, Uppsala, Sweden.

[25] Komorowski, H.J., A specification of an abstract Prolog machine and its application to partial evaluation. Linkoping Studies in Science and Technology Dissertations, No. 69, Software Systems Research Center, Linkoping University, 1981.

[26] Lombardi,L.A. and Raphael, B., LISP as the language for an incremental computer. In E. Berkley and D. Bobrow (Eds), The programming language LISP: Its operation and application, MIT Press, Cambridge, 1964.

[27]-------------, Incremental computation. Advances in computers, Vol. 8, 1967.

[28] Ostrovsky, B.N., Obtaining language oriented parser systematically by means of mixed computation, in I.V. Pttosin, Ed., Translation and Program Models (Computing Center, Novosibirsk, 1980) 68-80 (in Russian).

[29] Turchin, V.F., Equivalent program transformation in REFAL. The automated system for construction control. Trans. of the CNIIPIASS institute, issue 6, M., 1974, 36-68 (Russian).

[Program optimization, recurtion removal and tabulation]

290

[30] Allen,F.E. et al., The experimental compiling system. IBM J ,RES DEVELOP. Vol. 24, No. 6, November 1980.

[31] Bird, R.S., Tabulation techniques for recursive programs. Computing Surveys, Vol. 12, No. 4, December 1980.

[32] Goto,E, Monocopy and associative algorithms in an extended Lisp. Report of the FLATS Project, Vol. 1, October 1978, Information Science Laboratory, The Institute of Physical and Chemical Research, Wako-Shi , Saitama 351, Japan.

[33] Keller, R.M. and Sleep, M.R., Applicative caching: Programmer control of object sharing and lifetime in distributed implementation of applicative languages. Proc. Functional Programming Language and Computer Architecture, 131-140, Dec. 1981.

[34] Rish, T., REMREC - A program for automatic recursion removal in LISP. Datalogilaboratoriet, Uppsala University, DLU 37/24, 1973.

[LISP, Prolog and PAD]

[35] Fuchi, K., Programming languages based on predicate logic. Inf. Processing journal of Japan, Vol. 22, No. 6, 588-591, June 1981 (Japanese).

[36] Futamura, Y. et al., Development of computer programs by PAD (Problem Analysis Diagram). Proc. of the Fifth International Conference on Software Engineering (New York: IEEE Computer Society, 1981), 325-332.

[37] Kahn, K., Unique Features of Lisp Machine Prolog. UPMAIL, Dept. of Computing Science, Uppsala University, March 1982.

[38] McCarthy, J. et al., LISP 1.5 Programmer's manual. M.I.T. Press, Cambridge, Massachusetts, 1962.

[39] Nakashima, H., Prolog/KR User's Manual for Version C-2. Wada Laboratory, Information Engineering Course, University of Tokyo, August 1981.

[40] Weinreb, D. and Moon, D., Lisp machine manual. MIT AI Laboratory, March 1981.

[Mathematical theory of computation]

[41] Kleene, S.C., Introduction to Meta-Mathematics. North-Holland Publishing Co., Amsterdam, 1952.

[42] Manna, Z., Mathematical Theory of Computation. McGraw-Hill, New York, 1974.

[43] Nakajima, R.,Introduction to Mathematical Information Science, Asakura-Shoten, Tokyo, 1982 (Japanese).

[44] Wegner, P., Programming, Languages, Information science and Computer Organization. McGraw-Hill, New York, 1968.
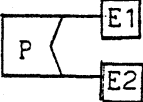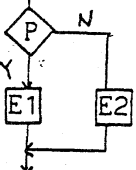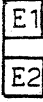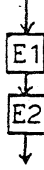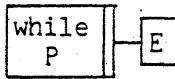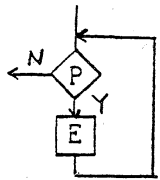
[45] Yasuhara, A., Recursive Function Theory and Logic. Academic Press, New York, 1971.
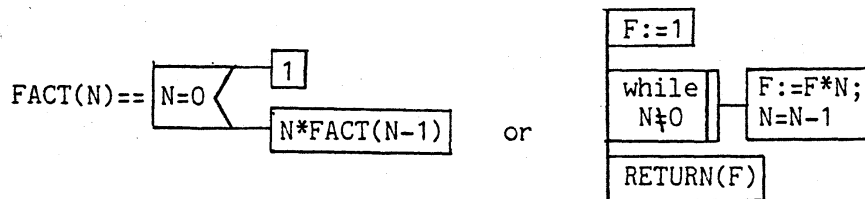

[5th generation computer project]

[46] Motooka, S., Overview of the 5th generation computer. Information Processing Journal of Japan, 426-432, Vol. 23, No. 5, May 1982 (Japanese).

Appendix 1: PAD (Problem Analysis Diagram) [36]

PAD is a new program schema for a substitute for both flow chart and recursive program schema. Correspondence between PAD, LISP and flow charts is described below.

	PAD	LISP	FLOW CHARTS
processing	E	E (LISP form)	E
selection	P ⟨ E1 / E2	[P->E1;T->E2]	P (Y→E1, N→E2)
definition	f(x)==E(f,x)	f[x]==E(f,x)	
sequencing	E1 / E2	E1;E2 (in prog feature)	E1 → E2
repetition	while P — E		while P (N, Y→E)
comment	-----[comment		-----[comment

Examples

FACT(N)== N=0 ⟨ 1 / N*FACT(N-1)

or

F:=1

while N≠0 — F:=F*N; N=N-1

RETURN(F)

Appendix 2: Early Research in the Field (Cited from [3])

Partial evaluation has been used by several researchers and for a variety of purposes. We have identified the following applications (in an attemted chronological order):

Lombardi and Raphael used it as a modified LISP evaluation which attempted to do as much as it could with incomplete data[26].

Futamura studied such programs for the purpose of defining compilers from interpreters[16].

Dixon wrote such a program and used it to speed up resolution[5].

Sandewall reports the use of such a program for code generation in [S71].

Deutsch uses partial evaluation in his thesis[D73], but had written a partial evaluation program much earlier (personal communication, December 1969). His idea at that time was to use it as an alternative to the ordinary eval, in order to speed up some computations.

Boyer and Strother Moore use it in a specialized prover for theorems about LISP functions[B74]. Somewhat strangely, they call it an "eval" function.

Hardy uses it in a system for automating induction of LISP functions[H73].

Finally, the language ECL[W74] can evaluate, during compilation, a procedure call only containing "frozen" or constant arguments. Most of these authors seem to arrive at the idea independently of each other, and at least number 2 through 5

at roughly the same time.

[B74] Boyer, R. and Moore, S., Proving theorems about LISP functions, Third International Joint Conference on Artificial Intelligence, Stanford Research Institute, 1974.

[D73] Deutch, P.,An interactive program verifier. Ph.D. thesis, Xerox, Palo Alto Research Center, 1973.

[H73] Hardy,S., Automatic induction of LISP functions. Essex University, Dec. 1973.

[S71] Sandewall,E. A., Programming tool for management of predicate-calculus-oriented data bases. Proc. of Second International Joint Conference on Artificial Intelligence, British Computer Society, 1971.

[W74] Wegbreit, B., The treatment of data types in EL1. Communications for the Association of Computing Machinery, 5, 251-64, 1974.

Appendix 3: An example of program manipulation [23]

Example: Produce an efficient new program combining two programs.
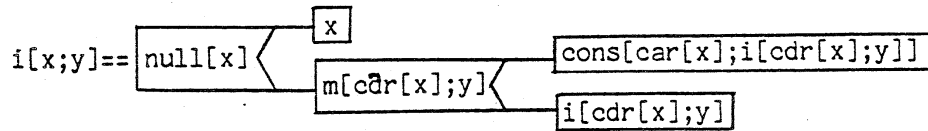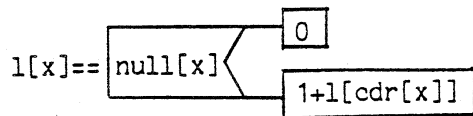
Given programs:

l[x]: Compute the length of list l.

i[x;y]: Produce the intersection of lists x and y.

New program to be produced:

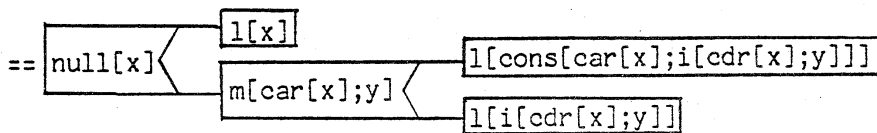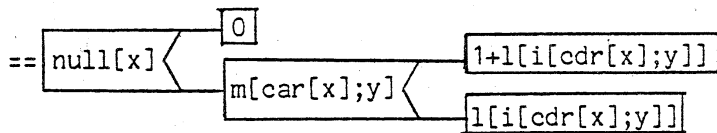li[x;y]: An efficient program to compute the length of intersection of x and y.

Definitions:

$$l[x] == \boxed{null[x]} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{1+l[cdr[x]]} \end{array}$$

$$i[x;y] == \boxed{null[x]} \Big\langle \begin{array}{l} \boxed{x} \\ \boxed{m[cdr[x];y]} \Big\langle \begin{array}{l} \boxed{cons[car[x];i[cdr[x];y]]} \\ \boxed{i[cdr[x];y]} \end{array} \end{array}$$

where   m[x;y] is a predicate to give T if x is a member of y else
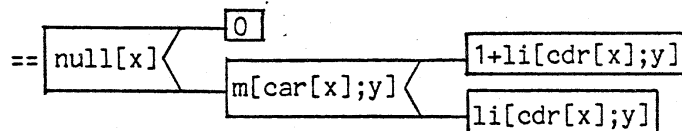it gives NIL, i.e.   m[x;y]=member[x;y].

Algebraic manipulation:

li[x;y]==l[i[x;y]]

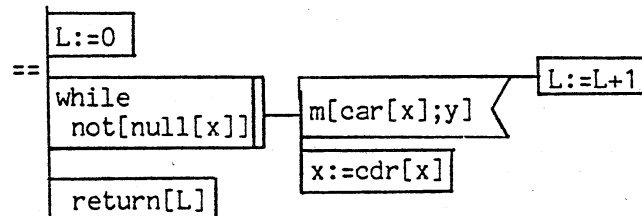$$== \boxed{null[x]} \Big\langle \begin{array}{l} \boxed{l[x]} \\ \boxed{m[car[x];y]} \Big\langle \begin{array}{l} \boxed{l[cons[car[x];i[cdr[x];y]]]} \\ \boxed{l[i[cdr[x];y]]} \end{array} \end{array}$$

(From distribution over conditional expressions)

$$== \boxed{null[x]} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{m[car[x];y]} \Big\langle \begin{array}{l} \boxed{1+l[i[cdr[x];y]]} \\ \boxed{l[i[cdr[x];y]]} \end{array} \end{array}$$

(From the definition of l)

$$== \boxed{null[x]} \Big\langle \begin{array}{l} \boxed{0} \\ \boxed{m[car[x];y]} \Big\langle \begin{array}{l} \boxed{1+li[cdr[x];y]} \\ \boxed{li[cdr[x];y]} \end{array} \end{array}$$

(From the definition of li)

$$== \begin{array}{l} \boxed{L:=0} \\ \boxed{\begin{array}{l} while \\ \ not[null[x]] \end{array}} \boxed{m[car[x];y]} \Big\langle \begin{array}{l} \boxed{L:=L+1} \\ \boxed{x:=cdr[x]} \end{array} \\ \boxed{return[L]} \end{array}$$

(From recursion removal)