UDC 681.326.06 : 800.92-56

計算過程の部分評価
――コンパイラ・コンパイラの一方法――

正員　二村　良彦†

Partial Evaluation of Computation Process, An Approach to a Compiler-Compiler

Yoshihiko FUTAMURA†, *Member*

あらまし　インタープリタを用いて形式的に記述されたプログラミング言語のセマンティクスと現実のコンパイラとの関係およびインタープリタからコンパイラを自動的に作成する方法について述べる．この方法は計算過程の部分評価の一種である．この方法を応用してできるコンパイラ・コンパイラと既存のコンパイラ・コンパイラの相違は，プログラミング言語のセマンティクスを記述するさいに，既存のものが翻訳過程を記述しなければならないのに対して，本方式によるものは評価手順を記述すればよいことである．

Summary　This paper reports the relationship between formal description of semantics (i.e., interpreter) of a programming language and an actual compiler. The paper also describes a method to automatically generate an actual compiler from a formal description, which is, in some sense, the partial evaluation of a computation process.

The compiler-compiler inspired by this method differs from a conventional one in the point that the compiler-compiler based on our method can describe an evaluation procedure (interpreter) in defining the semantics of a programming language, while the conventional one describes a translation process.

1. ま え が き

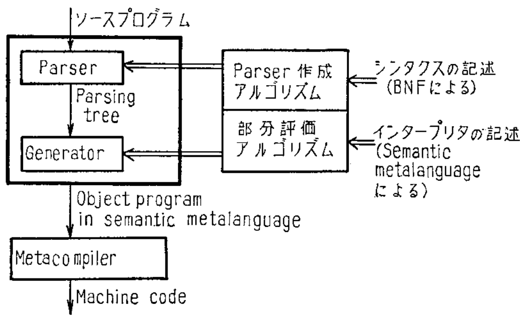
プログラミング言語のセマンティクス（意味）を形式的に記述する方法は，大ざっぱに言って二とおりあると言われている．一つは，プログラミング言語をすでに意味の知られている言語に翻訳する手順を記述する，すなわち翻訳過程の記述である．もう一つは，ソースプログラムがどんな結果をもたらすかという評価手順を記述する，すなわちインタープリタの記述である．

既存のコンパイラ・コンパイラでは，プログラミング言語のセマンティクスの記述には翻訳過程の記述が用いられている．すなわち，コンパイラ・コンパイラのユーザは，プログラミング言語のセマンティクスを定義する際に，ある種の翻訳過程記述用言語を用いて翻訳過程（トランスレータまたはコンパイラ）を記述しなければならない．翻訳過程を書く際には，翻訳のためにやる仕事と翻訳されたものがやる仕事をはっきりと区別しなければならない．この区別は，プログラミング言語のコンパイラの初心者を当惑させる困難な問題である．

一方，プログラミング言語のセマンティクスをインタープリタを用いて記述する際には，そのような区別をする必要がない．したがって，翻訳過程の記述に比べると相当わかりやすい方法と思われる．事実，ALGOL 60 の方法書[1]や一般のプログラミング言語の文法書においてもインタープリタによる記述が暗黙のうちに用いられている場合が多い．また，PL/I および ALGOL 60 のような複雑なプログラミング言語に対してもインタープリタが形式的に記述されている[2],[3]．

しかし，インタープリタをそのままで言語処理に用いると，いわゆるコンパイラを用いたときと比べて数

† 日立製作所中央研究所，国分寺市
Central Research Laboratory, Hitachi Ltd., Kokubunji-shi, Japan 185
論文番号：昭 46-1362 [C-513]



太枠で囲まれたブロックが生成されたコンパイラである．このコンパイラのオブジェクトプログラムはインタープリタを記述するのと同じメタ言語（sematic meta-language）に属し，メタコンパイラによって機械コードに翻訳される．

図 1 コンパイラ・コンパイラの構造

Fig. 1—Structure of compiler-compiler.

倍から数十倍能率が悪くなる可能性がある（その理由は本文 **3**．で述べられる）．

本文では，インタープリタをコンパイラに自動的に変換するアルゴリズムおよびそのコンパイラ・コンパイラへの応用について述べる．そのアルゴリズムは部分評価アルゴリズムの一種である（図1参照）．

計算過程の部分評価自体は決して新しい概念ではない．プログラミング言語においても POP-2[4] において partial application という名前で現われている．しかし，それをコンパイラ・コンパイラに応用しようという試みは本研究が最初であると思われる．コンパイラ・コンパイラに応用できるためにはどんな部分評価アルゴリズムが必要であるか，その性質を調べることが本文の目的である．

2. トランスレータとインタープリタ

2.1 相　　違

実用的プログラミングシステムとしてのトランスレータとインタープリタは明確に区別されていないし，区別することにもあまり意味があるとは思えない．プログラミング言語を特定の計算機で実現する場合には，そのプロセッサがインタープリーティブになる度合は言語自身の性質，言語が使われる状況および作成の条件などの兼合いによって決定される．

プログラミング言語のセマンティクスを形式的に定義するために用いられるトランスレータによる方法とインタープリタによる方法は，Bandat[5] によってつぎのように述べられている．

「トランスレータによる方法は完全に定義された言語 L の存在を必要とする．定義しようとしている言語 L' で書かれた任意の構文的に正しい(well-formed) プログラムを，言語 L の中に翻訳するトランスレータを設計できるならば，言語 L の定義と L' を L に翻訳するトランスレータとは，L' のセマンティクスを完全に定義する．厳密であいまいさがないためには，プログラムを翻訳する規則は，それ自身完全に定義されたメタ言語で書かれなければならない．

インタープリタによってセマンティクスを定義する方法は，通常，任意に与えられた入力データ集合と任意に与えられたプログラムに対して，出力データを作り出す関数またはプロセスを指定する．このことは，定義しようとしている言語を機械語として持つ抽象機械を設計することによって達成される．プログラミング言語の完全な論理的記述は，抽象機械のとり得る状態の記述およびプログラムを解釈することによってその状態が他の状態に変わる方法からなっている」

二つの方法の相違をさらに明らかにするために Mc Carthy と Painter[6] における例を引用する．

いま定義しようとしている言語の文は常数（constant)，変数（variable）および和（sum）からなる数式であり，つぎの (1)，(2) が仮定されているものとする．（1）おのおのの構文は述語 isconst (e)，isvar (e) および issum (e) によって判定される．（2）和は二つの加数からなりそれらは関数 $s1(e)$ および $s2(e)$ によってとり出すことができる．

このとき，そのセマンティクスをつぎのインタープリタ value で与えることができる．

value (e, ξ) = <u>if</u> isconst (e) <u>then</u> val (e) <u>else</u> <u>if</u> isvar (e) <u>then</u> c (e, ξ) <u>else</u> <u>if</u> issum (e) <u>then</u> value $(s1(e), \xi)$ + value $(s2(e), \xi)$

ただし，<u>if</u> <u>then</u> <u>else</u> は ALGOL 60 におけるのと同様の意味を持ち，ξ は変数とその対応する値を記憶するテーブル（状態ベクトルと呼ばれる）である．val (e) は常数を表わす式 e の値を与え，c (e, ξ) は状態ベクトル ξ における変数 e の値を与える関数である．＋は数の和を表わす二元演算子である．

［例］ val $(3+x, \xi)$ = val (3) + c (x, ξ)

数式のセマンティクスをトランスレータで定義するためには，まず，目的言語を定義しなければならない．目的言語の文はつぎの四つの命令からなるプログラムであるとする．

li α : (load immediate) 常数 α を累算器 (accumulator) ac にロードする．すなわち，$\alpha \rightarrow$ ac

sto x：(store) 累算器 ac の内容を x 番地に格納する．すなわち，(ac) → x

load x：x 番地の内容を累算器 ac にロードする．すなわち，(x) → ac

add x：x 番地の内容と累算器 ac の内容を加えて結果を ac にしまう．すなわち，(ac)＋(x) → ac

これらの命令を識別する述語およびオペランドを与えられたときに命令を作り出す関数を表1に示す．

表 1 命令を識別する述語と生成する関数

命　　令	述　　語	命令を生成する関数
li α	isli (s)	mkli (α)
sto x	issto (s)	mksto (x)
load x	isload (s)	mkload (x)
add x	isadd (s)	mkadd (x)

プログラムは命令の並びである．二つのプログラム p_1 と p_2 を並べることを $p_1 * p_2$ で表わす．

オブジェクトマシン（目的プログラムを走らせる機械）の状態ベクトルは機械の各レジスタ（累算器 ac もレジスタに含めて考える）にその値を与える．状態ベクトルに関連して二つの関数がある．

c (x, η)：状態ベクトル η におけるレジスタ x の内容を示す．

a (x, α, η)：状態ベクトル η においてレジスタ x の内容を α に変え，他のレジスタの内容はそのままにしておいて得られる状態ベクトルを示す．

機械の状態ベクトルが η のとき，プログラム p を実行して得られる状態ベクトルを outcome (p, η) で示す（outcome を形式的に定義すれば目的言語のセマンティクスは形式的に定義されるが，ここではトランスレータとインタープリタの大まかな相違を示すことが目的であるので，そこまで詳しくやる必要はない）．

このとき，数式のセマンティクスはつぎのトランスレータ compile で与えられる．

compile (e, t) ＝ <u>if</u> isconst (e) <u>then</u> mkli (val (e)) <u>else</u> <u>if</u> isvar (e) <u>then</u> mkload (loc (e, map)) else <u>if</u> issum (e) <u>then</u> compile (s1 (e), t) * mksto (t) * compile (s2 (e), t+1) * mkadd (t)

ただし，map は数式に含まれる変数，たとえば v，に割付けられた番地を与え，その番地を loc (v, map) で示すものとする．そしてつぎの関係がなりたっていると仮定する．

$$\text{c}\,(\text{loc}\,(v,\ map),\ \eta) = \text{c}\,(v,\ \xi)$$

これは，コンパイルされてできたプログラム（目的プログラム）が走る前の状態ベクトル η とソースプログラムの状態ベクトル ξ の初期値との対応を与える．compile の第2引数 t は，t 以上の番地をテンポラリストレッジとして使う（すなわち，すべての変数は t より小さい番地に割り付けられている）ことを示している．このとき，目的プログラムが走った結果，数式の値が累算器 ac に入り，t より小さい番地の内容は変えられない．この関係を

$$\text{outcome}\ (\text{compile}\ (e,\ t),\ \eta) \underset{t}{=} \text{a}\ (\text{ac},\ \text{value}\ (e,\ \xi),\ \eta)$$

で示す（この関係が成立することは文献 (6) に示されている）．

2.2 記述しやすさ

上例によってトランスレータとインタープリタの相違を，ある程度はっきりできたと思われるが，今度はどちらが書きやすそうかという問題を考えてみる．

トランスレータ compile においては，目的プログラムが走ることによって数式の正しい値が計算できるように翻訳をしなければならないので考え方が複雑になっている．このことは Feldman(7) がつぎのように指摘している．

「トランスレータを作る際の最もむずかしい概念の一つは，トランスレートタイムに行なわれることとランタイムに行なわれることの区別である．この区別が容易にできる者は，コンピュータ言語を理解するための素養があるといえる．ランタイムとコンパイルタイムの動作に関する区別は FSL（Feldman が設計したコンパイラ・コンパイラ：著者注）においては非常に重大であり，常にはっきりと示されなければならない」

一方，インタープリタ value においては，与えられた数式がどんな値を結果としてもたらすかということだけを考えればよいので，考え方が一面的であり，それだけ単純である．

3. 部　分　評　価

つぎのような変換を変数 $c_1, \cdots, c_m$ に関する値 $c_1', \cdots, c_m'$ における計算過程，たとえば π，の部分評価と呼ぶ．

“$m+n$ 個の変数 $c_1, \cdots, c_m, r_1, \cdots, r_n$ を含む計算過程 π の変数 $c_1, \cdots, c_m$ に値 $c_1', \cdots, c_m'$ を与え，それらの値と π が含む常数だけで評価できる部分を評価する．このとき，残りの変数の値が与えられなければ

評価できない部分はそのままにしておくことによって，π を n 個の変数を持つ計算過程に変換する．こうして得られた計算過程に含まれる変数 $r_1, \cdots, r_n$ に値 $r_1', \cdots, r_n'$ を与えて評価すると，それは π に値 $c_1', \cdots, c_m', r_1', \cdots, r_n'$ を与えたときの評価と（ある種の等価性のもとで）一致する"．

このことを

$$\pi(c_1', \cdots, c_m', r_1', \cdots, r_n') = \alpha(\pi, c_1', \cdots, c_m')(r_1', \cdots, r_n') \quad (1)$$

と書き，α を部分評価アルゴリズム，$c_1, \cdots, c_m$ を部分評価変数，そして $r_1, \cdots, r_n$ を残りの変数と呼ぶ．部分評価に対して通常の評価を全評価と呼ぶこともある．

［例］ $f(x, y)=x*(x*x+x+y+1)+y*y$

なる関数で与えられる計算過程を，$x=1, y=1, 2, \cdots, l$ について繰り返し計算する場合を考える．

y の各値に対して $f(1, y)$ を評価する，すなわち

```
x:=1; for y:=1 step 1 until l do f[x, y]
    =x*(x*x+x+y+1)+y*y;
```

を実行するならば全部で $3l$ 回の掛算と $4l$ 回のたし算が行なわれる．たし算と掛算に要する時間をおのおの a と m で表わせば，上の計算では約 $(4a+3m)l$ 時間を要する．

$f(x, y)$ を $x=1$ について部分評価して，

$$\alpha(f, 1)(y)=1*(3+y)+y*y$$

を得たとすれば，部分評価に要する時間 k は $2a+m$ 以上である．すなわち，$k \geq 2a+m$（なぜならば部分評価は $x*x+x+1$ の計算を含むから）．

```
for y:=1 step 1 until l do f[1, y]:=1*(3+
    y)+y*y;
```

の実行に約 $(2a+2m)l$ 時間要する．

したがって，$k+(2a+2m)l<(4a+3m)l$ のとき，すなわち，

$$\frac{k}{2a+m}<l$$

ならば，部分評価による方が早く計算できる．

4. インタープリタよりコンパイラの生成

プログラミング言語のインタープリタは変数を含む計算過程であり，その変数の一つには言語に属する文（ソースプログラム）が値として与えられる．インタープリタ，たとえば int，に含まれる変数をつぎのように二つの類に分ける．ソースプログラムおよびその構造分析と意味分析（semantic analysis）に必要な情報を値として与えられるすべてのものを一まとめにして s とする．その他の変数を一まとめにして r で表わす．s に値 s' を与えて，インタープリタを s について部分評価することにより $\alpha(int, s')(r)$ が得られる．r の値を r' とすれば，式 (1) より

$$int(s', r')=\alpha(int, s')(r') \quad (2)$$

が成立する．部分評価の際に s' に関する計算が済まされていれば，$\alpha(int, s')$ はソースプログラム s' の構造分析と意味分析を含まない．しかもデータ r' を与えて評価すれば $int(s', r')$ と同じ結果をもたらす．したがって，$\alpha(int, s')$ は，s' がインタープリタを記述している semantic メタ言語に翻訳されたもの，すなわち s' の目的プログラムと考えられる．(2) の右辺において α を int について部分評価すれば，

$$\alpha(int, s')(r')=\alpha(\alpha, int)(s')(r')$$

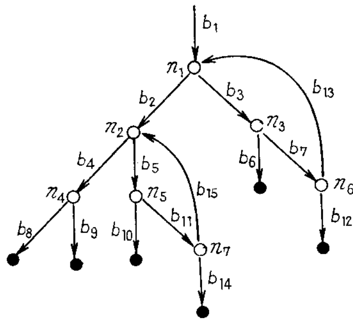
が得られる．$\alpha(\alpha, int)$ は s' に作用して目的プログラムを生成するので，コンパイラであると考えられる．

α が計算過程，たとえば π，を部分評価するときに，p_1）π に含まれる常数および部分評価変数に与えられた値だけを用いてできるなるべく多くの部分を評価し，しかも，p_2）残りの変数の値が与えられたとき実際に評価されない部分はなるべく評価しない，という性質を持つとする．このとき，いくつかの変数の値を変えて π を繰り返し評価するならば，一度部分評価した方が繰り返しの回数が多いほど計算時間について得をする．p_1 は部分評価によって得られた計算過程に残りの変数の値を与えて評価するさいの時間を早くするための性質である．p_2 は部分評価の時間を早くするための性質である．

α が性質 p_1 と p_2 をある程度持っているならば，ソースプログラムがループまたは再帰呼出しなどの繰り返しを含む時，または入力データを変えてソースプログラムを繰り返し実行するときには，直接インタープリートするよりコンパイルしてから実行するほうが得である．

一番単純な部分評価アルゴリズムは p_1 を無視したもの，すなわち部分評価変数に，それに与えられた値を代入するだけのものである．

性質 p_1 を考慮し，インタープリタの部分評価に適したアルゴリズム α_1 を以下に述べる．説明の都合上計算過程は図2のようなグラフであるとする．節（○印）は真偽の判定による分岐，枝（矢印）は判定による分岐を含まない計算過程および計算の流れ，そして



$n_1, \cdots, n_7$ は節につけられた名前，そして
$b_1, \cdots, b_{15}$ は枝につけられた名前である．

図 2　計算過程 π のグラフ表現

Fig. 2—Graph representation of computation process π.

葉（•印）は計算過程の終わりを示すものである．すべての節と枝におのおの n_i と b_j（異なるものには異なる添字がつく）という名前を付ける．特に入口（一つ以上あってもかまわないが，部分評価の際にはそのうちの一つだけが選ばれると仮定する）の枝は b_1 とし，枝の総数を m で表わす．

α_1 は部分評価の各段階においてつぎの (i), (ii) によって部分評価変数を定める．

（i）　部分評価変数または常数，またはそれらによって決まる値を代入された（または，実パラメータとして与えられた）変数（または，関数の仮パラメータ）は部分評価変数である．

（ii）　残りの変数またはそれに値が依存する表現の値を代入されている（または，実パラメータとして与えられている）変数（または，関数の仮パラメータ）は部分評価変数ではない．

α_1 のアルゴリズムは以下の (1)～(5) の操作で示される（説明中に整数変数 l, $j(1), \cdots, j(m)$ およびリスト変数 L を使用する）．

（1）　l および $j(1), j(2), \cdots, j(m)$ をすべて1に設定し，L を空にして（2）を行なう．

（2）　b_l を部分評価した結果が作り出される場所（計算機のメモリに作られるならばプログラムの先頭番地，紙上に書かれる場合は添付されるラベル）を割付け，それを記憶する．すなわち，b_l が $j(l)$ 回目に部分評価されたときの（b_l の入口における）部分評価変数とその値の対の集合を $S_l^{j(l)}$，部分評価の産物が作り出される場所の先頭を $a_l^{j(l)}$ とするならば，α_1 は3組 $(b_l, S_l^{j(l)}, a_l^{j(l)})$ をリスト L に登録する．(3) を行なう．

（3）　b_l の中で部分評価変数と常数だけで評価できる部分を評価し，その部分がすでに評価されたという印を付ける．ここで得られた新しい計算過程を $b_l^{j(l)}$ とする（$b_l^{j(l)}$ は b_l に基づいて作られた新しい計算過程であり，b_l はもとのまま残されている）．$j(l)$ の値を1ふやし，(4) を行なう．

（4）　b_l のつぎ（すなわち，b_l のやじりの先）が終了の印（•）ならば部分評価を止める．判定 $n_{k(i)}$ ならば，

(4.1)　$n_{k(i)}$ が部分評価変数と常数だけで判定できるならば判定により一方の枝を選ぶ．その枝が b_p ならば l の値を p にし，(5) を行なう．

(4.2)　残りの変数の値が与えられないと $n_{k(i)}$ の判定ができないならば，判定を行なわずそのままにしておく．それに続く二つの枝が b_p, b_q とすると，l の値を p にして (5) を行ない，つぎに l の値を q にして (5) を行なう．

（5）　初めの2項が b_l および $S_l^{j(l)}$ と一致する3組があるかどうかリスト L を調べる．

(5.1)　もし有れば，その3組の第3項で示される場所 a_l^x にプログラムの制御が移るようにし（紙に書く場合には a_l^x に矢印を引き），部分評価を止める．

(5.2)　もしなければ (2) を行なう．

［例 1］　図2における n_1, n_3, n_6 の判定を部分評価変数と常数だけで行なうことができ，おのおのの判定により b_3, b_7, b_{12} が選ばれるとすれば，π は α_1 により図3のように変換される．

［例 2］　判定 n_1, n_6 が部分評価変数と常数だけで

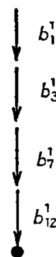


図 3　例 1

Fig. 3—Example 1.

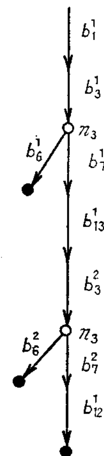
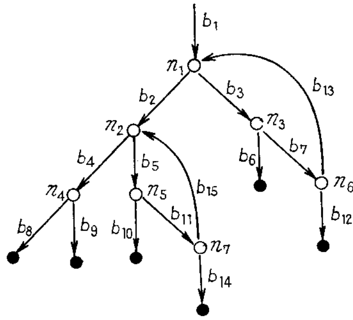


図 4　例 2

Fig. 2—Example 2.



$n_1, \cdots, n_7$ は節につけられた名前，そして $b_1, \cdots, b_{15}$ は枝につけられた名前である．

図 2 計算過程 π のグラフ表現

Fig. 2—Graph representation of computation process π.

葉（•印）は計算過程の終わりを示すものである．すべての節と枝におのおの n_i と b_j（異なるものには異なる添字がつく）という名前を付ける．特に入口（一つ以上あってもかまわないが，部分評価の際にはそのうちの一つだけが選ばれると仮定する）の枝は b_1 とし，枝の総数を m で表わす．

α_1 は部分評価の各段階においてつぎの (i), (ii) によって部分評価変数を定める．

（i） 部分評価変数または常数，またはそれらによって決まる値を代入された（または，実パラメータとして与えられた）変数（または，関数の仮パラメータ）は部分評価変数である．

（ii） 残りの変数またはそれに値が依存する表現の値を代入されている（または，実パラメータとして与えられている）変数（または，関数の仮パラメータ）は部分評価変数ではない．

α_1 のアルゴリズムは以下の (1)～(5) の操作で示される（説明中に整数変数 l, $j(1), \cdots, j(m)$ およびリスト変数 L を使用する）．

（1） l および $j(1), j(2), \cdots, j(m)$ をすべて1に設定し，L を空にして（2）を行なう．

（2） b_l を部分評価した結果が作り出される場所（計算機のメモリに作られるならばプログラムの先頭番地，紙上に書かれる場合は添付されるラベル）を割付け，それを記憶する．すなわち，b_l が $j(l)$ 回目に部分評価されたときの（b_l の入口における）部分評価変数とその値の対の集合を $S_l^{j(l)}$，部分評価の産物が作り出される場所の先頭を $a_l^{j(l)}$ とするならば，α_1 は3組 $(b_l, S_l^{j(l)}, a_l^{j(l)})$ をリスト L に登録する．(3) を行なう．

（3） b_l の中で部分評価変数と常数だけで評価できる部分を評価し，その部分がすでに評価されたという印を付ける．ここで得られた新しい計算過程を $b_l^{j(l)}$ とする（$b_l^{j(l)}$ は b_l に基づいて作られた新しい計算過程であり，b_l はもとのまま残されている）．$j(l)$ の値を1ふやし，(4) を行なう．

（4） b_l のつぎ（すなわち，b_l のやじりの先）が終了の印（•）ならば部分評価を止める．判定 $n_{k(i)}$ ならば，

(4.1) $n_{k(i)}$ が部分評価変数と常数だけで判定できるならば判定により一方の枝を選ぶ．その枝が b_p ならば l の値を p にし，(5) を行なう．

(4.2) 残りの変数の値が与えられないと $n_{k(i)}$ の判定ができないならば，判定を行なわずそのままにしておく．それに続く二つの枝が b_p, b_q とすると，l の値を p にして (5) を行ない，つぎに l の値を q にして (5) を行なう．

（5） 初めの2項が b_l および $S_l^{j(l)}$ と一致する3組があるかどうかリスト L を調べる．

(5.1) もし有れば，その3組の第3項で示される場所 a_l^x にプログラムの制御が移るようにし（紙に書く場合には a_l^x に矢印を引き），部分評価を止める．

(5.2) もしなければ (2) を行なう．

［例 1］ 図2における n_1, n_3, n_6 の判定を部分評価変数と常数だけで行なうことができ，おのおのの判定により b_3, b_7, b_{12} が選ばれるとすれば，π は α_1 により図3のように変換される．

［例 2］ 判定 n_1, n_6 が部分評価変数と常数だけで

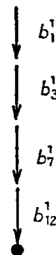


図 3 例 1

Fig. 3—Example 1.

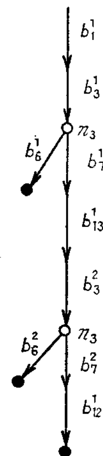


図 4 例 2

Fig. 2—Example 2.

行なえ，n_3 は残りの変数の値に依存する場合について考える．n_1 は常に b_3 を選び，n_6 は最初に b_{13} を選びつぎに b_{12} を選ぶものとする．このとき π は α_1 によって図4のように変換される．

［例 3］ 例2において n_6 が常に b_{13} を選ぶならば，一般には α_1 は停止せず図5のような無限のグラフを生成する．しかし，n_6 が常に b_{13} を選ぶ原因が，n_6 における部分評価変数が巡回することであるならば，操作 (5) によって α_1 は停止し，図6（部分評価変数が一定の場合）のような産物を得る．ループや再帰呼出しを含むソースプログラムについてインタープリタを部分評価する場合に上の後者の場合が生ずるので，操作 (2) と (5) は本論文で述べるコンパイラ・コンパイラの方法にとって本質的に重要である．

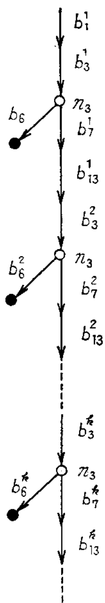


図5 例3（止まらない場合）
Fig. 5—Example 3 (non-terminating case).

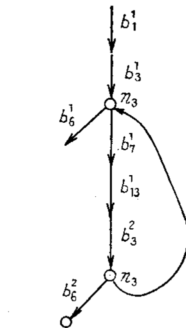


図6 例3（止まる場合）
Fig. 6—Example 3 (terminating case).

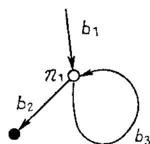


図7 例4
Fig. 7—Example 4.

［例 4］ 図7において，n_1 は残りの変数に依存するものとする．このとき b_3 の過程を繰り返し部分評価しても同じ S_3^x が作られなければ無限のグラフが生成される．しかし，全評価をすれば b_3 がなん度か計算されたあとで n_1 が b_2 を選び計算は終了するかもしれない．もし b_3 が残りの変数を含まず，しかも無限のループを含んでいるときに，n_1 が全評価によって必ず b_2 を選ぶならば，それは全評価が止まって部分評価が止まらない自明な例となる．

最後の場合のように，全評価の際に評価されない部分を評価することを避ける（すなわち p_2 を考慮に入れる）ためには，つぎのようにすればよい．評価されるかどうか分からない部分，すなわち残りの変数を含む判定に続く部分（ただし，判定は例外），が残りの変数に依存しない場合にも評価せずに，部分評価変数に対応する値を代入するだけにする．これは単に無駄を排除するという意味ばかりでなく，インタープリタに含まれるエラーメッセージの印字，その他の入出力操作など，評価されては困る部分を保護するためにも必要である．判定を例外としたのは，判定できるものはしてしまい部分評価の結果に含まれる判定と枝の数を減らすためである．残りの変数を含む判定に続く部分が残りの変数を含んでいれば，その部分には α_1 を適用する．それは残りの変数を含む計算過程はインタープリタの再帰呼出しを含んでいる可能性が強く，少々の危険をおかしても部分評価する価値があるという考えに基づく．したがって，部分評価の最中に評価されてはまずい関数，手続および擬似変数には，印をつけて特別あつかいしなければならない．

しかし，例4の最後の場合におけるような無駄なループは，インタープリタを真面目に書けば除外できる．したがって，エラーの処理，入出力およびその他の副作用を持つ操作のうちで部分評価の際に実行されてはこまるもの以外はすべて実行するように α_1 を変更すれば，希望どおりのアルゴリズムが得られる．

以上で部分評価アルゴリズムの概要を述べたわけであるが，詳細については触れていない．詳細は，計算過程を記述するプログラミング言語に従って大差がある．

［例 5］ LISP[8] 関数 appnd $[x; y]$ の部分評価

$$\text{append}[x;y]=[\text{null}[x]\to y;T\to \text{cons}[\text{car}[x];\ \text{append}[\text{cdr}[x];y]]]$$

であるから，

$$\alpha_1[\text{append};(A,B)][y]=\text{cons}[A;\ \text{cons}[B;y]]$$

$$\alpha_1[\text{append};(A,B)][x]=[\text{null}[x]\to(A,B);\ T\to\text{cons}[\text{car}[x];\alpha_1[\text{append};(A,B)][\text{cdr}[x]]]$$

［例 6］ ALGOL プログラムの部分評価

a, b ともに整数のリストを表わす <u>integer</u> <u>array</u> で，$a[0]$, $b[0]$ にリストの長さが入り，$a[1], \cdots, a[a[0]]$ および $b[1], \cdots, b[b[0]]$ にリストの要素が入るものとする．このとき，a と b をつないで一つのリストを作るプログラム（ただし，bigm はアレイ

a の上限を示すものとする)[9].

```
  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
```

を b について, $b[0]=2$, $b[1]=10$, $b[2]=20$ において部分評価すると,

```
  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
```

となる.

5. 今後の課題

(a) インタープリタからコンパイラが作れる, すなわちインタープリタの部分評価が停止する, ための基準(自明でない十分条件)はなにか.

(b) インタープリタをソースプログラムについて部分評価して得られた目的プログラムは, ソースプログラムよりどの部分がどれだけ(定量的に)よいか. また目的プログラムはどんな特性を持ちそれをオプティマイズする(時間とメモリについて)にはどんな手法が必要か.

(c) インタープリタは, 翻訳過程よりどれだけ(定量的に)記述しやすいか. また, インタープリタから部分評価によって生成されるコンパイラの能率を, 翻訳過程から生成されたものに近づけることは可能か.

(d) どんな言語でインタープリタを記述したらうまく部分評価できるか.

現在のところ, 著者は, それらの疑問にはっきりと答えることはできない. そうできるためにはつぎの研究開発が必要であろう.

(1) プログラミング言語のインタープリタの構造の理解.

(2) プログラミング言語の abstract syntax, 抽象機械の状態(スタック, テーブル, リストなど)とその遷移および数値計算とリスト処理, の記述が容易であり, しかも能率よくコンパイル(メタコンパイル)できるようなセマンティクス用メタ言語の開発.

(3) 特定のセマンティクス用メタ言語に対する完全な部分評価アルゴリズムの開発, 実験および改良.

(4) 計算機プログラムの部分評価に関する理論的研究.

(5) セマンティクス用メタ言語のオプティミゼイションの研究.

著者は(3)について多少行なったにすぎない. LISP プログラムの部分評価アルゴリズム(α_1 とほとんど等価なもの)を LISP で記述し, プログラム機能(program feature)[8] のインタープリタからコンパイラを生成した. そのコンパイラは, ALGOL 形プログラムを等価な方程式のシステムに翻訳する.

たとえば,

```
prog [[u; v];
          u:=n;
L1 [null[u]→return[v]];
    v:=cons[car[u]; v];
    u:=cdr[u];
    go[L1]]
```

を,

```
g1[a]=g2[ppair[(U V); a]]
g2[a]=prog2[rplacd[assoc[U; a]; eval[N;
  a]]; g3[a•]]
g3[a]=g4[a];
g4[a]=g5[a];
g5[a]=[eval[(NULL U); a]→eval [V; a];
  T→g6[a]]
g6[a]=g7[a]
g7[a]=prog2[rplacd[assoc[V; a]; eval
  [(CONS(CAR U) V); a]]; g8[a]]
g8[a]=prog2[rplacd[assoc[U; a]; eval
  [(CDR U); a]]; g9[a]]
g9[a]=g4[a]
```

に翻訳する. ただし, g1~g9 はコンパイラによって生成された関数名であり, g1[a] が目的プログラムである. 方程式のシステム中に g3[a]=g4[a], g4[a]=g5[a] などの無駄な定義が含まれているが, これはセマンティクス用メタ言語(この場合は LISP)のオプティミゼイションをやれば排除できる.

6. 結　言

本文で述べたコンパイラ生成方法はまだ着想の段階を出ておらず, 近い将来実用化できるかどうかは未検討である. しかし, 理論的に追求されているプログラミング言語の形式的記述方法と現実のコンパイラとの関係が本方法によりある程度明らかになり, コンパイラ・コンパイラの研究に何らかの助けになることがで

きれば幸いである．

謝辞 本研究の初期の段階からしばしば議論をして下さった日立中研 吉村一馬 主任研究員および本研究に興味を持たれ有益な助言をして下さった日立中研 嶋田正三 主管研究員と 田上 嵩 第9部部長に対し，つつしんで感謝の意を表わす．

文　献

(1) P. Naur : "Revised report on the algorithmic language Algol 60", Comm. of ACM, **6**, p. 1 (Jan. 1963).

(2) K. Walk, et al. : "Abstract syntax and interpretation of PL/I", TR 25.082, IBM Laboratory Vienna, 28 (Jun. 1968).

(3) P. Lauer : "Formal definition of ALGOL 60, TR 25.088", IBM Laboratory Vienna, 13 (Dec. 1968).

(4) R. M. Burstall and R. J. Popplestone : "POP-2 reference manual. Machine Intelligence 2", p. 205 (1968).

(5) K. Bandat : "On the formal definition of PL/I", Proc. of SJCC 68, p. 363.

(6) J. McCarthy and J. Painter : "Correctness of a compiler for arithmetic expressions", TR No. CS 38 (April 1966), Computer Science Department, Stanford University.

(7) J.A. Feldman : "A formal semantics for computer-oriented languages", Comput. Ctr., Carnegie Institute of Technology (1964).

(8) J. McCarthy, et al. : "LISP 1.5 Programmer's Manual", M.I.T. Press (1962).

(9) H. Rutishauser : "Description of ALGOL 60", Springer (1967).

(昭和 45 年 11 月 17 日受付)