プログラミング 4−11
(1995. 12. 1)

副作用を含む関数型プログラムの部分評価に向けて

浅井 健一　　増原英彦*　　米澤明憲
E-mail: {asai, masuhara, yonezawa}@is.s.u-tokyo.ac.jp
東京大学 大学院理学系研究科 情報科学専攻
〒113 東京都文京区本郷 7-3-1

部分評価は、プログラムを既知の入力によって特化して効率化するプログラム変換の一種であり、理論、実際両面においてその有用性が注目されている。しかし、これまでの関数型言語の部分評価器は純粋な関数型言語を対象としており、副作用命令を扱うことはできない。一方、最近、C 言語用の部分評価器が提案されているが、offline の方式を取っているためわかりにくく、関数型言語との関連も見えにくい。本稿では、online の方式を用いて副作用命令を含む関数型言語の部分評価器を作成し、online の方式で何ができ、どのような問題が生じるのかを考察する。それを通して、より強力な部分評価器を作るにはどうしたらよいのかを考えていく。

Issues in Partial Evaluation of Functional Programs with Side-effects

Kenichi Asai　　Hidehiko Masuhara*　　Akinori Yonezawa
Department of Information Science, University of Tokyo

Partial evaluation is an effective program transformation technique which specializes a given program with respect to some known arguments and produces an efficient specialized version of it. It is widely used not only in theoretical fields but also in a low-level program optimization. However, conventional partial evaluators for functional programs deal with only pure-functional programs without side-effects. Although C-MIX, a partial evaluator for the C programming language, can deal with pointers and side-effects, it is not obvious if the technique developed there can be applied to functional languages. In this paper, we construct a partial evaluator for functional programs with side-effects based on online approach, examining its merits and limitations. We also consider how to combine the two approaches to overcome the limitations.

*教養学部情報・図形科学教室 (Department of Graphics and Computer Science, College of Arts and Sciences)

1 はじめに

部分評価とは、プログラムと引数の一部を受け取って、その既知の引数に特化したより効率の良いプログラムを出力するプログラム変換の一種である。プログラムによっては、既知の引数のみに依存する部分（staticな部分）と、未知の引数にも依存する部分（dynamicな部分）とに分けられるものがある。例えば、インタプリタはユーザの入力したプログラムの構文解析部と、そのプログラムへの入力を使っての計算部とに分けることができるが、このうち前者は、プログラムの形さえ既知であれば、そのプログラムへの入力は未知でも実行できる。このように既知の入力のみに依存した部分をあらかじめ実行し、その入力に特化したプログラムを作るのが部分評価である。部分評価の手法は、インタプリタに適用すればコンパイラとして機能することがわかっており、また、最近では各種の最適化技法のひとつとしても注目を集めている。

これまでの部分評価器の研究は、おもに副作用のない純粋な関数型言語を対象として行なわれてきた[6]。あらかじめプログラムを解析した上で部分評価する方式（offline の方式）としては Similix[4] や Schism[5]、解析は行なわずにプログラムを擬似実行し、実際に得られる情報をもとに部分評価する方式（online の方式）としては Fuse[9] などが良く知られている。offline の方式は、部分評価器本体を小さく抑えることができるため、コンパイラ生成の観点からは重要であるが、部分評価の能力としては online の方式のほうが強力である。また、部分評価器の構造も online の方式の方が簡単で理解しやすい。

一方、副作用命令の部分評価については、関数型言語に関してはほとんどなされておらず、いくつか手続き型言語を対象として行なわれているだけである。これまで、Pascal 用[8]、Fortran 用[3]、C 言語用[2] のものなどが発表されているが、前者ふたつは変数代入のみを扱った簡単なものである。C 言語用の部分評価器 C-MIX はポインタを扱うことのできる強力なものであるが、offline の方式を用いているため、その部分評価能力は解析次第であり、どこまで実際に部分評価できるのかがわかりにくい。また、完全な手続き型言語を対象としているため、関数型言語との関連も見えにくい。

ここでは、副作用を含む関数型言語の部分評価への試みとして、より直感的に理解しやすい online の方式を使った場合に、何ができ、どのような問題点が生じるのかを考察する。具体的には、我々の考案した履歴付記号値 [7, 1] を使って、副作用の使用に関して適当に制限を加えた言語に対する部分評価器を実際に構成する。さらに（本稿では紙面の都合で省略するが）履歴付記号値の性質を調べることにより、従来、不明確だった記号値からコードへの変換を明確に記述できることもわかる。一方、加えられた制限については、online の方式の限界を示すものと考えられるため、どのような情報があればそれが外せるかを考察し、その情報が offline の解析で求められることを述べる。

```
⟨exp⟩ ::= const | ⟨var⟩ | (known? ⟨var⟩)
        | (lambda (⟨var⟩ ...) ⟨filter⟩ ⟨exp⟩)
        | (if ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
        | (set! ⟨var⟩ ⟨exp⟩)
        | (letrec ((⟨var⟩ ⟨exp⟩) ...) ⟨exp⟩)
        | (begin ⟨exp⟩ ...)  | (⟨exp⟩ ...)
⟨filter⟩ ::= (filter ⟨exp⟩)
```

図 1: 部分評価器の対象とする言語

2 対象とする言語

本稿で扱う部分評価の対象言語は副作用命令を含む Scheme のサブセットである（図 1）。図には変数代入（`set!`）しか現われていないが、定数として入出力命令（`read`、`write` 等）と破壊的代入命令（`set-car!` と `set-cdr!`）を含むものとする。λ閉包にフィルターを書くようになっているが、これはλ閉包が適用されたときに、それを展開するか、コードとして残すかの判断を与えるためである。λ閉包を展開しても部分評価が止まるかどうかの判断は決定不能であるが、すでに良い発見的手法が考案されているので、それを将来用いることとし、ここではユーザによる注釈という形で与えている。

3 履歴付記号値

online 部分評価器は、基本的にはインタプリタのようにプログラムを解釈実行していき、未知の式に出会ったらコードを出力する。online 部分評価器では、通常の値とコードとを区別する必要があるため、タグを用いた記号値（Symbolic Value *Sval*）を使用する。ここでは、副作用命令を扱っているので、さらに履歴（preaction）が付与された履歴付記号値（preaction 付き記号値 *PSval*）を使う。

図 2 に記号値の定義を示す。このうち定数、ペア、λ閉包 1 が通常の値であり、他はコードを示している。λ閉包 2 は、コードとして残ったλ閉包を

$$
\begin{aligned}
Sval &= \mathbf{const}(con) \mid \mathbf{pair}(l_1, l_2, \sigma) \\
&\mid \mathbf{closure1}(params, body, \rho, \sigma) \\
&\mid \mathbf{var}(var) \mid \mathbf{closure2}(params, PSval) \\
&\mid \mathbf{if}(Sval, PSval, PSval) \mid \mathbf{apply}(Sval, Sval^*) \\
PSval &= \langle\!\langle Sval^* \rangle\!\rangle Sval
\end{aligned}
$$

図 2: 記号値と履歴付記号値

示す。履歴付記号値は、記号値の列（空でもよい）が前に履歴としてくっついた記号値である。例えば、$\langle\!\langle \mathbf{apply}(write,(3)) \rangle\!\rangle \mathbf{const}(4)$ は値としては 4 であるが、その前に 3 を出力することを意味しており、最終的には (begin (write 3) 4) のようなコードとなる。このように、履歴は副作用命令をうまくコードとして残す働きをする。ここで、履歴は記号値に付与されているだけで、既知の記号値を未知にしてしまうことはない。例えば上の履歴付記号値に 5 を加えると $\langle\!\langle \mathbf{apply}(write,(3)) \rangle\!\rangle \mathbf{const}(9)$ となる。

4 部分評価器

図3 に部分評価器本体を示す。この部分評価器 $\mathcal{PE}$ が従来の部分評価器と大きく異なる点は、副作用を扱うためにストア渡し形式で書かれていることである。従って、その構造はストア渡し形式のインタプリタと似たものとなっている。式と環境のほかにキャッシュも受け取っているが、これは再帰関数を無限に展開するのを防ぐために使われる。以下、順に各ルールを説明していく。定数、変数、λ 式を部分評価した結果はインタプリタと同じで、それぞれ定数自身、環境中の変数の値、λ 閉包を返す。

ストアの合成 部分評価器がインタプリタと異なる第一点は条件文である。条件文を部分評価するには、まずその条件部を部分評価する。その結果が真偽値になったときはインタプリタと同様で、それぞれ then 部分、else 部分を部分評価する。このとき、条件部の履歴が結果の履歴に加えられていることに注意されたい。条件部が未知のコードとなった時は、then 部分と else 部分両方を評価し、条件文のコードを結果として返している。これは、実行時にならないとどちらが選択されるかわからないためである。

ここまでは、従来の部分評価器と同様だが、副作用命令があると then 部分と else 部分でストアが異なる値に書き換えられてしまうことがある。これに対処するため、継続に渡すストアとしては、ふたつのストアの合成 $\sigma_t \oplus \sigma_e$ をとる。合成の取り方は、ストアを見比べ、異なる部分、例えば σ_t では a、σ_e では b となっている番地の内容を $\mathbf{if}(p, a, b)$ とする、というものである。例えば

```
(let ((a 1) (b 2))
  (if (= n 0) (set! a 3) (set! b 4))
  (cons a b)))
```

を部分評価すると

```
(let ((t (= n 0)))
  (cons (if t 3 1) (if t 2 4)))
```

が返ってくる[1]。Meyer[8] の Pascal 用の部分評価器では、値の異なっている番地に対して、条件文の各枝で代入文のコード[2]を挿入するとともにその番地を以後、未知とする、という方法が取られている。しかし、7 節で述べるようになるべく副作用命令はコードとして残さない方が良いので、ここではストアを合成するという方法を取っている。

代入文の扱い 次に、代入文 (set!) である。代入文も、なるべく副作用命令を残さない、との立場から、コードに残すことはせず、部分評価時に代入を実行し、ストアの値を書き換えてしまうというアプローチを取った。代入文は、データ構造を壊すことはないので、この方法でもかなりのことができる。この方法で扱えないものについては 7 節で述べる。

letrec および begin はインタプリタと同じである。ただし、履歴が結果に残っていることに注意されたい。関数適用は begin と同様にした後、関数を適用する補助関数 A を呼び出している。

履歴によるコード消滅の防止 A を図4 に示す。*cons* を適用した結果はペアであるが、そのとき、履歴に *cons* の引数が入っている。これは、後にペアの片方だけが使用されたときに、もう片方がなくなってしまうのを防ぐためである。例えば、(car (cons 1 (write 2))) を部分評価するときには、値としては 1 でよいが、その前に write 文を残す必要がある。offline の方式では、コードの消滅防止のために解析を行なっているが、ここでは履歴付記号値を使うことでそれを簡潔に実現している。なお、このとき先ほど同様同じ記号値は、共有されるのでコード複製の心配はない。

[1] このように共通部分式は除去されるので、条件部 p が複製されることはない。具体的には、非巡回グラフで表されている記号値で共有されている部分を let 文でくくり出している。これは、Fuse と同様である。

[2] これは explicator と呼ばれている。

$$
\begin{array}{rrcl}
 & \mathcal{PE} & : & Exp \to Env \to Cache \to Cont \to Store \to Answer \\
\rho \in & Env & : & Var \to Location \\
c \in & Cache & : & (Sval \times Sval^*) \to Sval \\
\kappa \in & Cont & : & PSval \to Store \to Answer \\
\sigma \in & Store & : & Location \to Sval \\
 & \mathcal{A} & : & (Sval \times Sval^*) \to Cache \to Cont \to Store \to Answer
\end{array}
$$

$$
\begin{array}{l}
\mathcal{PE}[\![con]\!]\rho c\kappa = \kappa\ \langle\!\langle\rangle\!\rangle \mathbf{const}(con) \\
\mathcal{PE}[\![var]\!]\rho c\kappa\sigma = \kappa\ \langle\!\langle\rangle\!\rangle \sigma(\rho(var))\ \sigma \\
\mathcal{PE}[\![\texttt{(lambda } params\ body\texttt{)}]\!]\rho c\kappa\sigma = \kappa\ \langle\!\langle\rangle\!\rangle \mathbf{closure1}(params, body, \rho, \sigma)\ \sigma \\
\mathcal{PE}[\![\texttt{(if } p\ t\ e\texttt{)}]\!]\rho c\kappa = \mathcal{PE}[\![p]\!]\,\rho\,c\,\lambda p'.\,\mathbf{case}\ p'\ \mathbf{of} \\
\qquad \langle\!\langle P_p \rangle\!\rangle true \ : \ \mathcal{PE}[\![t]\!]\,\rho\,c\,(\lambda\langle\!\langle P_t \rangle\!\rangle t'.\,\kappa\ \langle\!\langle P_p \cdot P_t \rangle\!\rangle t') \\
\qquad \langle\!\langle P_p \rangle\!\rangle false \ : \ \mathcal{PE}[\![e]\!]\,\rho\,c\,(\lambda\langle\!\langle P_e \rangle\!\rangle e'.\,\kappa\ \langle\!\langle P_p \cdot P_e \rangle\!\rangle e') \\
\qquad \langle\!\langle P_p \rangle\!\rangle p \ : \ \lambda\sigma_p.\,\mathcal{PE}[\![t]\!]\,\rho\,c\,(\lambda\langle\!\langle P_t \rangle\!\rangle t'.\lambda\sigma_t.\,\mathcal{PE}[\![e]\!]\,\rho\,c \\
\qquad\qquad (\lambda\langle\!\langle P_e \rangle\!\rangle e'.\lambda\sigma_e.\,\kappa\ \langle\!\langle P_p \rangle\!\rangle \mathbf{if}(p', \langle\!\langle P_t \rangle\!\rangle t', \langle\!\langle P_e \rangle\!\rangle e')(\sigma_t \oplus \sigma_e))\,\sigma_p)\,\sigma_p \\
\mathcal{PE}[\![\texttt{(set! } v\ a\texttt{)}]\!]\rho c\kappa = \mathcal{PE}[\![a]\!]\,\rho\,c\,(\lambda a'.\,\lambda\sigma.\,\kappa\ \langle\!\langle\rangle\!\rangle \mathbf{const}(unspecified)\sigma[a'/\rho(v)]) \\
\mathcal{PE}[\![\texttt{(letrec ((}x_1\ f_1\texttt{) ... (}x_n\ f_n\texttt{)) } body\texttt{)}]\!]\rho c\kappa = \mathbf{let}\ \rho' = \rho[l_1/x_1, \ldots, l_n/x_n]\ \mathbf{in} \\
\qquad \mathcal{PE}[\![f_1]\!]\,\rho'\,c\,(\lambda\langle\!\langle F_1 \rangle\!\rangle f_1'.\mathcal{PE}[\![f_2]\!]\cdots\lambda\langle\!\langle F_{n-1} \rangle\!\rangle f_{n-1}'.\mathcal{PE}[\![f_n]\!]\,\rho'\,c\,(\lambda\langle\!\langle F_n \rangle\!\rangle f_n'. \\
\qquad\quad \lambda\sigma.\,\mathcal{PE}[\![body]\!]\,\rho'\,c\,(\lambda\langle\!\langle A \rangle\!\rangle a.\,\kappa\ \langle\!\langle F_1 \cdot f_1' \cdots F_n \cdot f_n' \cdot A \rangle\!\rangle a)\,\sigma[f_1'/l_1, \ldots, f_n'/l_n])\cdots) \\
\qquad\qquad \text{where } l_1, \ldots, l_n \text{ are fresh locations} \\
\mathcal{PE}[\![\texttt{(begin } a_1 \ldots a_n\texttt{)}]\!]\rho c\kappa = \mathcal{PE}[\![a_1]\!]\,\rho\,c\,(\lambda\langle\!\langle A_1 \rangle\!\rangle a_1'.\mathcal{PE}[\![a_2]\!]\cdots\lambda\langle\!\langle A_{n-1} \rangle\!\rangle a_{n-1}'.\mathcal{PE}[\![a_n]\!]\,\rho\,c \\
\qquad\qquad (\lambda\langle\!\langle A_n \rangle\!\rangle a_n'.\,\kappa\ \langle\!\langle A_1 \cdot a_1' \cdots A_{n-1} \cdot a_{n-1}' \cdot A_n \rangle\!\rangle a_n')\cdots) \\
\mathcal{PE}[\![\texttt{(}f\ a_1 \ldots a_n\texttt{)}]\!]\rho c\kappa = \mathcal{PE}[\![f]\!]\,\rho\,c\,(\lambda\langle\!\langle F \rangle\!\rangle f'.\mathcal{PE}[\![a_1]\!]\,\rho\,c\,(\lambda\langle\!\langle A_1 \rangle\!\rangle a_1'.\mathcal{PE}[\![a_2]\!]\cdots\lambda\langle\!\langle A_{n-1} \rangle\!\rangle a_{n-1}'.\mathcal{PE}[\![a_n]\!]\,\rho\,c \\
\qquad\qquad (\lambda\langle\!\langle A_n \rangle\!\rangle a_n'.\mathcal{A}(f', (a_1', \ldots, a_n'))\,c\,(\lambda\langle\!\langle P \rangle\!\rangle v.\,\kappa\langle\!\langle F \cdot A_1 \cdots A_n \cdot P \rangle\!\rangle v))\cdots))
\end{array}
$$

図 3: 部分評価器本体

その他　入出力命令の部分評価は、部分評価時に行ってしまうわけにはいかないので、コードとして残している。ここでも、履歴が使われている。

setcar 等については、部分評価時に実際に行ってしまうと同時に、コードに残すという方法を取っている。これは部分評価時になるべく未知の値が増えないようにするとともに、実行時にも正しいデータ構造が構築されるようにするためである。

上記以外の primitive 関数の適用は、引数の値がすべて既知なら実行し、そうでなければ、コードとして残すというごくごく自然なものとなっている。また、関数が未知のときには、その関数を適用するコードを残している。

最後は、λ閉包の適用である。λ閉包を適用するには（図4には示していないが）まず、フィルターを評価し、その値に従って処理をわけている。ちょっとわかりにくいが、ここは従来の部分評価器と同じなので、紙面の都合により省略する。ただ、λ閉包の中では引数全部を必ず使うとは限らないため *cons* のときと同様、コードの消滅を防ぐべく、引数をすべて履歴に残している。

5　記号値からコードへの変換

ここで示した部分評価器は、（primitive 関数の引数がすべて既知の場合を除けば）関数適用の際にその引数をすべて履歴に入れているため、コードの共有が起こる可能性のある記号値は、どこか前の方で必ず履歴に入れられていることがわかる。この性質を使うと、Fuse のように複雑なことをすることなく、記号値をその共有関係を保ったままコード化できることがわかるが、ここでは紙面の都合で省略する。

6　部分評価の例

ここでは、副作用を使ったプログラムの部分評価例を示す。以下のプログラムは変数代入を使って書いた階乗を求めるプログラムである。letrec のか

$$
\begin{aligned}
\mathcal{A}(cons,(a_1,a_2))c\kappa\sigma &= \textbf{let } \sigma' = \sigma[a_1/l_1, a_2/l_2] \textbf{ in } \kappa\, {}^{\langle\!\langle a_1,a_2\rangle\!\rangle}\mathbf{pair}(l_1,l_2,\sigma')\,\sigma' \\
&\qquad \text{where } l_1 \text{ and } l_2 \text{ are fresh locations} \\
\mathcal{A}(write,(a))c\kappa &= \kappa\, {}^{\langle\!\langle a,\mathbf{apply}(write,(a))\rangle\!\rangle}\mathbf{const}(unspecified) \\
\mathcal{A}(setcar,(p,a))c\kappa\sigma &= \textbf{let } \mathbf{pair}(l_1,l_2,\sigma') = p \textbf{ in} \\
&\qquad \kappa\, {}^{\langle\!\langle p,a,\mathbf{apply}(setcar,(p,a))\rangle\!\rangle}\mathbf{const}(unspecified)\,\sigma[a/l_1] \\
\mathcal{A}(prim,(a_1,\ldots,a_n))c\kappa &= \begin{cases} \kappa\, {}^{\langle\!\langle\rangle\!\rangle}prim(a_1,\ldots,a_n) & \text{if all known} \\ \kappa\, {}^{\langle\!\langle a_1,\ldots,a_n\rangle\!\rangle}\mathbf{apply}(prim,(a_1,\ldots,a_n)) & \text{otherwise} \end{cases} \\
\mathcal{A}(f,(a_1,\ldots,a_n))c\kappa &= \kappa\, {}^{\langle\!\langle f,a_1,\ldots,a_n\rangle\!\rangle}\mathbf{apply}(f,(a_1,\ldots,a_n)) \qquad \text{if } f \text{ is unknown}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{A}(\mathbf{closure1}((x_1,\ldots,x_n),body,\rho,\sigma')(=f),(a_1,\ldots,a_n))c\kappa\sigma \\
&= \begin{cases}
\mathcal{PE}[\![body]\!]\rho[l_1/x_1,\ldots,l_n/x_n]\,c\,(\lambda^{\langle\!\langle P\rangle\!\rangle}v.\,\kappa\, {}^{\langle\!\langle f,a_1,\ldots,a_n,P\rangle\!\rangle}v)\,\sigma[a_1/l_1,\ldots,a_n/l_n] \\
\qquad\qquad \text{if filter evaluates to 'unfold} \\
\kappa\, {}^{\langle\!\langle f,a_1,\ldots,a_n\rangle\!\rangle}\mathbf{apply}(f',(a'_1,\ldots,a'_m))\,\sigma \qquad \text{if cache hits : } c(f,(b_1,\ldots,b_n)) = f' \\
\mathcal{PE}[\![body]\!]\rho[l_1/x_1,\ldots,l_n/x_n]\,c[\mathbf{var}(t)/(f,(b_1,\ldots,b_n))] \\
\qquad (\lambda^{\langle\!\langle P\rangle\!\rangle}v.\,\textbf{let } f'' = \mathbf{closure2}((x'_1,\ldots,x'_m),{}^{\langle\!\langle P\rangle\!\rangle}v) \textbf{ in} \\
\qquad\quad \kappa\, {}^{\langle\!\langle f,a_1,\ldots,a_n\rangle\!\rangle}\mathbf{apply}(f'',(a'_1,\ldots,a'_m))) \\
\qquad \sigma[b_1/l_1,\ldots,b_n/l_n] \qquad\qquad \text{otherwise}
\end{cases} \\
&\quad \text{where} \begin{cases}
\{a'_1,\ldots,a'_m\} = \{a_i \mid a_i \text{ is not propagated}\} \\
\{x'_1,\ldots,x'_m\} = \{x_i \mid a_i \text{ is not propagated}\} \\
b_i = \begin{cases} a_i & \text{if to be propagated} \\ x_i & \text{otherwise} \end{cases} \\
t = \text{a fresh variable}
\end{cases}
\end{aligned}
$$

図 4: 補助関数 $\mathcal{A}$

わりに define を使っているが、これは簡単に変換できる。

```
(define (power x n)
  (define ans 1)
  (define (loop)
    (if (= n 0)
        'end
        (begin (set! ans (* ans x))
               (set! n (- n 1))
               (loop))))
  (loop)
  ans)
```

このプログラムを (power y 3) として部分評価すると次のようになる。ここで y は未知の変数である。

```
(begin (define g0 (* 1 y))
       (define g1 (* g0 y))
       (* g1 y))
```

n が既知のため、関数を展開することができ、変数代入はすべてなくなって、直接 y を3回、掛けるコードとなっている。なお、本稿では述べていないが、後処理によって上のコードは容易に

```
(* (* (* 1 y) y) y)
```

に変換できる。

7 副作用の使用に対する制限とその解消法

ここで作成した部分評価器は、副作用の使用に関してふたつの制限がある。ここでは、それらを述べるとともに、それらがどこまで外せるかを考察する。

制限 1 `set-car!`、`set-cdr!` をされるセルは部分評価時に既知でなくてはならない。

書き換えられるセルが既知でないと、どこが書き換えられるかわからないため、ストアを全て未知とする必要がある。すると、その後の部分評価では変数参照などストアを参照する命令がまるっきり行なえなくなってしまい、ほとんど部分評価できずにコードとして残ってしまう。そのため 4 節で述べた部分評価器では、そういうことはないものと仮定した。

この制限を取り除くためには、現在、行なおうとしている代入文によって破壊される可能性のあるセルを特定できれば良い。それがわかれば、それらのセルだけを未知とすることで、その後もまだ十分、部分評価できるであろう。しかし、この情報は擬似実行の枠組には現れてこないため、解析なしに on-line で求めるのは困難と思われる。この情報は、C-MIX 等で行なわれている points-to 解析 (alias

解析）を行なうことで求めることができるので、これらの手法を組み合わせるのが良いであろう。

制限 2 コードとして残るλ閉包には（入出力命令以外の）副作用命令が含まれてはいけない。

この制限は、いつ適用されるかわからないλ閉包には副作用命令が入ってはいけない、ということである。この制限があるため 4 節では、なるべく副作用命令を残さないようにした。次のようなプログラムを考えてみる。

```
(let ((f (if p (lambda (x) (set! a x))
               (lambda (x) (set! b x)))))
  (f 3))
```

p が未知であれば、ふたつのλ閉包はコードとして残る。すると、(f 3) を行なった後は、a あるいは b が書き換わることになる。すなわち、コードとして残るλ閉包に副作用命令が含まれていると、それを適用した時に、ストアが書き換わってしまうのである。4 節で述べた部分評価器では、未知の関数を適用するところでストアが書き換わっていないが、これは上のような仮定をおいていることを示している。

この制限を取り除くためには (1) このλ閉包はどこを書き換え得るか、そして (2) このλ閉包はどこで適用され得るか、のふたつを解析し、その上で適用される可能性のある場所の直後で、書き換えられるかも知れない番地を未知とする必要がある。前者はλ閉包中で評価した副作用命令を保持しておくことで online の枠組でも求められると思われるが、後者は offline 解析で行なわれている closure 解析を行なう必要がある。

このように見てくると、制限を外すにはかなりの offline の解析が必要である。しかし、offline 部分評価器では特化するべきプログラムすべてを解析する必要があるのに対し、我々の場合はコード化される部分に対してのみ解析を行なえばよい。展開されているうちはインタプリタと同様に進んでいくので問題はなく、コード化するときになって初めてそれがどのような影響を及ぼすかを考えれば良い。従って、offline 部分評価器と比べれば、解析はかなり簡単になると思われる。

8 おわりに

本稿では、副作用命令を含む関数型言語の部分評価器を作成し、その際に設けられた制限について考察した。履歴付記号値を使った online の部分評価器で、副作用をかなりうまく扱えることを示すとともに、そこで出てくる制限が offline の方式で使われている解析に密接な関係があることを示した。その意味で、本研究は online 部分評価器と offline 部分評価器の橋渡し的存在でもある。今後は、この枠組みに基づき実際に online の手法と offline の手法を組み合わせてより強力な部分評価器を作成していく予定である。

謝辞 日頃からたくさんのアドバイス、議論をいただいている松岡聡氏、細谷晴夫氏に感謝いたします。

参考文献

[1] Asai, K., H. Masuhara, S. Matsuoka, and A. Yonezawa "Partial evaluation as a compiler for reflective languages," Technical report of Department of Information Science, University of Tokyo, to appear (1995).

[2] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, University of Copenhagen (May 1994).

[3] Baier, R., R. Gluck, and R. Zochling "Partial Evaluation of Numerical Programs in Fortran," In *PEPM'94*, pp. 119–132, (June 1994).

[4] Bondorf, A. *Similix 5.0 Manual*, DIKU, University of Copenhagen, (May 1993).

[5] Consel, C. "A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages," In *PEPM'93*, (1993).

[6] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[7] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," In *OOPSLA'95*, pp. 300-315, (October 1995).

[8] Meyer, U. "Techniques for partial evaluation of imperative languages," In *PEPM'91*, pp. 94–105, (1991).

[9] Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.