Vol. 35 No. 3 情報処理学会論文誌 Mar. 1994

完全遅延評価に適した関数プログラムの共有解析

金子敬一† 尾上能之† 武市正人†

遅延評価系で完全遅延性を実現するためのプログラム変換算法がいくつか提案されている．これらの算法は，極大自由式の同一性は検出するが，共通部分式であっても束縛されたものや，極大自由式の一部となったものは検出していない．本稿では，極大自由式だけでなくラムダ式や局所定義式など，一般の関数プログラムに適用することができる共有解析算法を提示する．さらにこの算法では共有解析中に極大自由式を検出することができる点を利用して，先の変換算法の埋め込みが可能であることを示す．また，共有解析した関数プログラムを完全遅延評価することにより，プログラマが意図しなかったような点においても実行効率の向上をもたらす場合があることも示す．

Sharing Analysis of Functional Programs for Fully Lazy Evaluation

Keiichi Kaneko,† Yoshiyuki Onoue† and Masato Takeichi†

There are several transformation algorithms of functional programs to implement full laziness by using a lazy evaluator. Though these algorithms detect identity of maximal free expressions, common subexpressions which are bound or parts of maximal free expressions are not found. In this paper, we provide an algorithm of sharing analysis which is applicable to general functional programs including lambda expressions and expressions with local definitions. Then we show that the former transformation algorithms are easily embedded in our sharing analysis algorithm taking advantage of the fact that maximal free expressions are detectable in the course of sharing analysis. Besides, we illustrate that fully lazy evaluation of resultant programs of sharing analysis may bring improvement of execution performance in points that programmers do not intend.

1. はじめに

プログラマには，正しくて，同時に効率のよいプログラムを書くことが要求される．関数プログラムは記述力が強いので，プログラムを小さくまとめることが可能であり，プログラムの正しさの検証も容易であるので，正しいプログラムを書くことには効果的である．また効率についても，翻訳技法の改良や計算機の能力の向上によって，実用的なレベルにまで到達している．さらに関数プログラムは，副作用を許さないので，参照透過性 (referential transparency) を持ち，これを利用してプログラム変換によって積極的に効率を改善することもできる．共有解析 (sharing analysis) もこのプログラム変換の一つであり，プログラム中の共通部分式 (common subexpression) を検出して，プログラムの実行時に重複計算を行わないように，一つにまとめることが目的である．同様の処理は手続き型言語のコンパイラの最適化手法にも共通部分式の削除[2] として見られるが，本稿では関数プログラムの性質を利用した有効な手法を提示する．

必須呼び出し (call-by-need) に基づく評価系を遅延評価系 (lazy evaluator) と呼ぶ．一方，完全遅延評価 (fully lazy evaluation) とは，すべての部分式がその中の変数の値が束縛された後は，高々一度しか評価されないような評価方式である．一般に遅延評価系を構成するのは容易であるが，完全遅延評価系を構成するのは困難である．このため，通常はプログラム変換によって入力プログラムを変形して，これを遅延評価することで完全遅延評価を実現している．今までに完全遅延評価のために，いくつかのプログラム変換算法が提案されている．以下では単に変換算法といえば，完全遅延評価用のプログラム変換算法のことを指すことにする．本稿では，代表的な変換算法であるラムダ巻き上げ (lambda hoisting)[9]，完全遅延ラムダ持ち

† 東京大学工学部計数工学科
Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo

上げ (fully lazy lambda lifting)[8]，超結合子 (super-combinator)[5] を取り上げる．これらの変換算法も，極大自由式 (maximal free expression) の同一性を検出しているが，これでは共通部分式であっても束縛されたものや，極大自由式の一部となったものは検出できない．本稿で提示する共有解析算法は，極大自由式だけでなく，ラムダ式や局所定義式など一般の関数プログラムに適用でき，さらにこの算法では共有解析中に極大自由式を検出することができる点を利用して，先の変換算法の埋め込みが可能であることを示す．

本稿の以下の構成は次の通りである．まず第2節で対象とする関数プログラムと完全遅延評価について説明し，第3節では名前の付け替えと局所定義の変換，レベルの割り当て，自由変数の割り当て，依存度解析，冗長な局所定義の除去，といった予備算法を定義する．第4節では，一般の関数プログラムに適用可能な共有解析算法を示す．この時点ではまだ完全遅延性については考慮していないが，第5節において，先に述べた各変換算法を簡単に埋め込むことが可能であることを示す．第6節では，この共有解析算法の有効性に関して例証する．

2. 関数プログラムと完全遅延評価

2.1 関数型言語

本稿では，**図1**に示すような関数型言語のプログラムを対象とする．式は，定数，変数，複合式，ラムダ式，局所定義式のいずれかに分類される．局所定義式は，さらに再帰的な **letrec** 式と非再帰的な **let** 式に分かれる．入力プログラムにおいて，複数の局所変数を定義する **let** 式は3.1節で示す規則を用いて，単一の局所変数を定義する **let** 式へ展開され，その後5.3節以外では現れない．

ここで簡単に用語の解説をしておく．自由変数とは注目しているラムダ抽象の中で束縛されていない変数のことをいう．自由式とは注目しているラムダ抽象の中で，束縛変数を含まない部分式のことをいう．極大自由式とは，その式を含む自由式が存在しない自由式のことをいう．

我々の提示する共有解析算法は，一般の関数プログラムを対象としている．これに対して，あらかじめ局所定義式を等価な関数適用に変換して，共有解析の対象から局所定義式を除いて算法を単純にするアプローチが考えられる．しかしながら，この方式では無駄な関数適用を導入することがある．たとえば，式

$\times(+\ x\ 1)(\mathbf{let}\ y=(+\ x\ 1)\ \mathbf{in}(+\ y\ y))$

に対して，これを関数適用に変換して

$\times(+\ x\ 1)((\backslash y.(+\ y\ y))(+\ x\ 1))$

という式を得てから，共有解析を実行すると次式を得る．

$\mathbf{let}\ \xi=(+\ x\ 1)\ \mathbf{in}(\times\ \xi((\backslash y.(+\ y\ y))\xi))$

一方，関数適用への変換を行わずに共有解析を行えば

$\mathbf{let}\ \xi=(+\ x\ 1)\ \mathbf{in}(\times\ \xi(+\ \xi\ \xi))$

という式を得ることができる．これは，関数適用への変換を行ってから変換した式に比べて，無駄な関数適

Syntactic Domains

$b \in \mathbf{Bas}$	basic values
$x \in \mathbf{Ide}$	identifiers
$e \in \mathbf{Exp}$	expressions

Abstract Syntax

$e ::= b \mid x \mid e\ e \mid \backslash x.e \mid \mathtt{let}\ x{=}e;\ldots;x{=}e\ \mathtt{in}\ e$
$\mid \mathtt{letrec}\ x{=}e;\ldots;x{=}e\ \mathtt{in}\ e$

Semantic Domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\mathbf{B} + \mathbf{F}]_+$	expressible values
$\mathbf{F} = \mathbf{D} \to \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \to \mathbf{D}_+$	environments

Environment for Denotational Specification

$\rho \in \mathbf{U}$

Semantic Functions

$B: \mathbf{Bas} \to \mathbf{B}$ (unspecified)
$E: \mathbf{Exp} \to \mathbf{U} \to \mathbf{E}$

$E[b]\rho = B[b]$
$E[x]\rho = \rho x$
$E[e_0 e_1]\rho = (E[e_0]\rho)(E[e_1]\rho)$
$E[\backslash x.e_0]\rho = \lambda\delta.E[e_0](\rho + \langle x \to \delta\rangle)$
$E[\mathtt{let}\ x_1{=}e_1;\ldots;x_n{=}e_n\ \mathtt{in}\ e_0]\rho = E[e_0]\rho'$
where $\rho' = \rho + \langle x_1 \to E[e_1]\rho\rangle + \ldots + \langle x_n \to E[e_n]\rho\rangle$
$E[\mathtt{letrec}\ x_1{=}e_1;\ldots;x_n{=}e_n\ \mathtt{in}\ e_0]\rho = E[e_0]\rho'$
where $\rho' = \rho + \langle x_1 \to E[e_1]\rho'\rangle + \ldots + \langle x_n \to E[e_n]\rho'\rangle$

Notations

Domain construction operator + stands for the disjoint sum.
For any domain $\mathbf{X}$, $\mathbf{X}_+ = \mathbf{X} + \{\mathbf{err}\}$.
For an environment ρ, $\rho + \langle x \to \delta\rangle$ denotes
$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ \delta\ \mathbf{else}\ \rho y$

Initial Environment

The initial environment ρ_ϕ satisfies $\rho_\phi x \neq \mathbf{err}$ for predefined identifiers x.

図1 関数型言語
Fig. 1 Functional language.

用のない点が優れているといえる．

2.2 完全遅延評価のための変換算法と共有解析

遅延評価系で完全遅延評価を実現するためのプログラム変換算法が，幾つか提案されている．ここでは，その中からラムダ巻き上げ，完全遅延ラムダ持ち上げ，超結合子を取り上げる．これらの変換算法の違いは，想定している実現方式の違いに由来する．基本的な操作は，入力式に対して，極大自由式を検出して，それが局所定義あるいは関数引数となるように式を変形すること，および，局所定義が自由である限り，ラムダ式の外に移動すること，からなる．この結果，その中の全変数が束縛されたとき直ちにその極大自由式も束縛される．したがって，変換後のプログラムを遅延評価すれば，もとのプログラムを完全遅延評価することになる．これらの変換算法では，極大自由式どうしの同一性の検出しかしないので，次のような場合は共通部分式を発見することはできない．

- 共通部分式が，極大自由式の一部である場合．たとえば，式 $\backslash g.(\backslash f.(f(g\ 1)(+(g\ 1)2)))$ において，内側のラムダ式の極大自由式は $(g\ 1)$ と $(+(g\ 1)2)$ であるが，第一の極大自由式は第二のものに含まれているので，いずれかの計算が冗長となる．一つの極大自由式中に同じ部分式が複数出現している場合も冗長な計算が残る．
- 共通部分式が束縛されている場合，すなわち，式 $\backslash f.(\backslash g.(f(g\ 1)(+(g\ 1)2)))$ において，二つの部分式 $(g\ 1)$ は，最も内側のラムダ変数によって束縛されているので共有するものとしては検出されない．

我々の提示する共有解析算法では，局所定義の浮上を行いつつ，すべての共通部分式を検出する．またごく一部を変更すれば，極大自由式も同時に検出することができるので，上述のような変換も簡単に実現することができる．変換算法中に共有解析を実行しようという従来の試みに対して，我々の方法の新しい点は，共有解析算法中に変換算法を埋め込んでいることである．

3. 準　　備

3.1 名前の付け替えと let 式の変換

関数プログラムのプログラマは，**let** 式を入れ子にするよりも，読みやすさを優先して，一つの **let** 式や **letrec** 式にまとめる傾向がある．このことは，3.5節での冗長な局所変数の検出などの解析の感度を鈍らせることになる．変数名を付け替えることによって，複数の局所変数を定義する **let** 式を単一の局所定義の **let** 式の入れ子に展開することができる．この規則を**図2**に示す．以下，構文上の要素を対象にするときは「·」で囲うことにする．また，自由式を元の位置からラムダ式の外側に移した時に，変数名が衝突することがあるが，この変数名の付け替え操作によってその可能性を除去することができる．

3.2 レベルの割り当て

部分式が自由であれば，ラムダ式の外側で共有されている可能性がある．部分式の共有性を，どの範囲まで調べなくてはならないかを決めるには，その部分式

Environment for Renaming

$\pi \in \mathbf{R} = [\mathbf{Ide} \to \mathbf{Ide}_+]$

Renaming Rules

$RN : \mathbf{Exp} \to \mathbf{R} \to \mathbf{Exp}$

$RN[b]\pi = \lceil b \rceil$

$RN[x]\pi = \pi x$

$RN[e_0 e_1]\pi = (RN[e_0]\pi)(RN[e_1]\pi)$

$RN[\backslash x.e_0]\pi = \lceil \backslash x'.RN[e_0](\pi + \langle x \to x' \rangle) \rceil$

where x' is a fresh identifier

$RN[\mathtt{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathtt{in}\ e_0]\pi$

$= \lceil \mathtt{let}\ x'_1 = RN[e_1]\pi\ \mathtt{in}$

$\cdots$

$\mathtt{let}\ x'_n = RN[e_n]\pi\ \mathtt{in}$

$RN[e_0](\pi + \langle x_1 \to x'_1 \rangle + \ldots + \langle x_n \to x'_n \rangle) \rceil$

where $x'_1, \ldots, x'_n$ are fresh identifiers

$RN[\mathtt{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathtt{in}\ e_0]\pi$

$= \lceil \mathtt{letrec}\ x'_1 = RN[e_1]\pi'; \ldots; x'_n = RN[e_n]\pi'\ \mathtt{in}$

$RN[e_0]\pi' \rceil$

where $x'_1, \ldots, x'_n$ are fresh identifiers

and $\pi' = \pi + \langle x_1 \to x'_1 \rangle + \ldots + \langle x_n \to x'_n \rangle$

Notation

For an environment for renaming π,

$\pi + \langle x \to x' \rangle$ denotes

$\lambda y.\mathtt{if}\ x = y\ \mathtt{then}\ x'\ \mathtt{else}\ \pi y$

Initial Environment

The initial environment π_ϕ satisfies $\pi_\phi x = x$ for predefined identifiers x.

図 2 名前の付け替えと **let** 式の展開規則
Fig. 2 Renaming rule of identifiers.

が依存しているラムダ変数を決定する必要がある．各ラムダ変数はレベル，すなわちその外側のラムダ抽象の入れ子の数によって識別することができる．したがって，すべての部分式に対して，その中のラムダ変数のレベルを計算することで，どのラムダ変数に依存しているかを決定することができる．ただし，定数のレベルを0，組み込みの変数のレベルを1とし，最も外側のラムダ変数には，常にレベル2を割り当てる．部分式に対するレベルの割り当て規則 L を**図3**に示す．規則 L は，その部分式が依存しているラムダ変数のレベルの集合を返す．また，ω は各変数に対して，そのレベルを保持する環境である．レベルの集合 $\{l_1, \cdots, l_n\}$ の最大値を $|\{l_1, \cdots, l_n\}|$ で表すことにする．

再帰的局所定義式に対する規則中には，再帰的方程式 $\omega' = \omega + \langle x_1 \rightarrow |L[e_1]\omega' l| \rangle + \cdots + \langle x_n \rightarrow |L[e_n]\omega' l| \rangle$ が存在する．これを反復法によって解く算法を次に示す．

Level Numbers

$l \in \mathbf{N}$

Environment for Level Numbers

$\omega \in \mathbf{L} = [\mathbf{Ide} \rightarrow \mathbf{N}_+]$

Level Assignment Rules

$L : \mathbf{Exp} \rightarrow \mathbf{L} \rightarrow \mathbf{N} \rightarrow \mathbf{2^N}$

$L[b]\omega l = \{0\}$

$L[x]\omega l = \{0\} \cup \{\omega x\}$

$L[e_0 e_1]\omega l = L[e_0]\omega l \cup L[e_1]\omega l$

$L[\backslash x.e_0]\omega l = L[e_0](\omega + (x \rightarrow l+1))(l+1) - \{l+1\}$

$L[\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0]\omega l = L[e_0](\omega + \langle x_1 \rightarrow |L[e_1]\omega l|\rangle) l$

$L[\mathbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e_0]\omega l = L[e_0]\omega' l$

where $\omega' = \omega + \langle x_1 \rightarrow l_1 \rangle + \ldots + \langle x_n \rightarrow l_n \rangle$

where $l_i = |L[e_i]\omega' l|$ for $i = 1, \ldots, n$

Notation

For an environment for assignment ω,

$\omega + (x \rightarrow l)$ denotes

λy **if** $x = y$ **then** l **else** ωy

Initial Environment

The initial environment ω_ϕ satisfies $\omega_\phi x = 1$ for predefined identifiers x.

図3 レベルの割り当て規則
Fig. 3 Assignment rules of levels.

for $i = 1, \cdots, n$ **do** $l'_i := 0$;
repeat
  for $i = 1, \cdots, n$ **do** $l_i := l'_i$;
  $\omega' := \omega + \langle x_1 \rightarrow l_1 \rangle + \cdots + \langle x_n \rightarrow l_n \rangle$;
  for $i = 1, \cdots, n$ **do** $l'_i := |L[e_i]\omega' l|$
until $\forall i, l_i = l'_i$

各変数 x_i に対するレベルの近似値 l_i は単調に増加し，$0 \leq l_i \leq l$ であることから，この算法は必ず停止する．

3.3 自由変数の割り当て

完全遅延ラムダ持ち上げや超結合子といった変換算法では，ラムダ式を再帰的超結合子 (recursive supercombinator) や超結合子に変換する．この変換においては，ラムダ式に含まれる自由変数を検出する必要がある．ただし，組み込み関数や既出の超結合子は定数として扱うことにしているので，これは除かなくてはならない．このための規則 FV を**図4**に示す．超結合子には，すべて組み込み関数と同じレベル1を割り当てる．そこで FV はレベル2以上の自由変数を検出するようにしている．

3.4 依存度解析

letrec 式 **letrec** $x_1 = e_1; \cdots; x_n = e_n$ **in** e_0 を，**let** 式と **letrec** 式の入れ子に展開する場合，まず局所定義変

Level Numbers

$l \in \mathbf{N}$

Environment for Level Numbers

$\omega \in \mathbf{L} = [\mathbf{Ide} \rightarrow \mathbf{N}_+]$

Free Variable Assignment Rules

$FV : \mathbf{Exp} \rightarrow \mathbf{L} \rightarrow \mathbf{N} \rightarrow \mathbf{2^{Ide}}$

$FV[b]\omega l = \{\}$

$FV[x]\omega l = v$

$v = \{x\}$, if $2 \leq \omega x$

$v = \{\}$, otherwise

$FV[e_0 e_1]\omega l = FV[e_0]\omega l \cup FV[e_1]\omega l$

$FV[\backslash x.e_0]\omega l = FV[e_0](\omega + (x \rightarrow l+1))(l+1) - \{x\}$

$FV[\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0]\omega l$

$= (FV[e_0]\omega l - \{x_1\}) \cup FV[e_1]\omega l$

$FV[\mathbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e_0]\omega l$

$= FV[e_0]\omega l \cup \ldots \cup FV[e_n]\omega l - \{x_1, \ldots, x_n\}$

図4 自由変数の割り当て規則
Fig. 4 Assignment rules of free variables.

数 $\{x_1, \cdots, x_n\}$ の依存関係を有向グラフ $G=(V, E)$ で表現する．ここで $V=\{x_1, \cdots, x_n\}$ であり，$e=(x_i, x_j) \in E \Leftrightarrow x_j \in (FV[e_i] \cap V)$ である．このグラフ G を，先の **letrec** 式の依存度グラフ (dependency graph) と呼ぶ．この有向グラフを強連結成分 (strongly-connected component) に分解し，さらにトポロジカルソート (topological sort) を行うことによって，依存関係に基づいて **letrec** 式を展開することができる．この操作を依存度解析 (dependency analysis) と呼ぶ．局所定義式に対して，この依存度解析を実現する規則を DA とする．例として，次式を考える．

letrec $x_1=+;\ x_2=x_1\ 2;\ x_3=\mathrm{cons}(x_2(\mathrm{hd}\ x_7))x_4;$
$x_4=\mathrm{cons}\ 4\ x_3;\ x_5=\mathrm{cons}(x_1\ 5(\mathrm{hd}\ x_6))x_7;$
$x_6=\mathrm{cons}\ 6\ x_5;\ x_7=\mathrm{cons}\ 7\ x_6$ **in** x_4

この局所定義変数 $\{x_1, \cdots, x_7\}$ の依存関係を有向グラフで表現したものが，**図 5** である．破線で囲まれた部分が，各強連結成分を示している．この式に DA を適用した結果，次式を得る．

let $x_1=+$ **in**
let $x_2=x_1\ 2$ **in**
letrec $x_5=\mathrm{cons}(x_1\ 5(\mathrm{hd}\ x_6))x_7;$
$x_6=\mathrm{cons}\ 6\ x_5;\ x_7=\mathrm{cons}\ 7\ x_6$ **in**
letrec $x_3=\mathrm{cons}(x_2(\mathrm{hd}\ x_7))x_4;$
$x_4=\mathrm{cons}\ 4\ x_3$ **in** x_4

3.5 冗長な局所定義の除去

以下の共有解析では冗長な局所定義が存在しないことを仮定している．これを実現するための規則 ER を**図 6** に示す．この規則 ER は，局所定義式に対して，そこで局所的に定義されている変数が本体中の自由変数に含まれていなければ，その局所定義は除去することができるという事実に基づいている．この事実によると，局所定義で定義された変数が一つでも本体中に含まれると，冗長な局所定義が混在していても除去できない．そこで，名前の付け替えや依存度解析を前もって行い，**let** 式や **letrec** 式を可能な限り小さな単位まで分割することによって，すべての冗長な局所定義を取り除くことができるようになる．式 e に対して $\langle e^*, v^* \rangle = ER[e]$ であるとき，v^* は以上で述べた意味で e において冗長でない変数の集合であり，また e^* は，e から冗長な局所定義を除去して得られる式である．

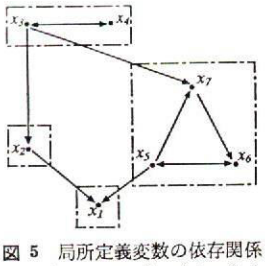

図 5 局所定義変数の依存関係
Fig. 5 Dependency graph of local definitions.

4. 単純な共有解析

図 1 の関数プログラムに対する単純な共有解析算法 SA を**図 7** に示す．SA は極大自由式の検出や処理は行わない．また SA を適用する式 e では，すべてのラムダ変数や局所変数は異なる変数名を持つこと，さらに本体に出現しない冗長な局所変数は存在しないことを仮定する．SA は，式 e の他に以下のものを引数とする．

Elimination Rules

$ER : \mathbf{Exp} \to [\mathbf{Exp} \times 2^{\mathbf{Ide}}]$

$ER[b] = \langle [b], \{\} \rangle$
$ER[x] = \langle [x], \{x\} \rangle$
$ER[e_0 e_1] = \langle [e_0' e_1'], v_0' \cup v_1' \rangle$
where $\langle e_0', v_0' \rangle = ER[e_0]$ and $\langle e_1', v_1' \rangle = ER[e_1]$
$ER[\backslash x.e_0] = \langle [\backslash x.e_0'], v_0' - \{x\} \rangle$
where $\langle e_0', v_0' \rangle = ER[e_0]$
$ER[\mathbf{let}\ x_1=e_1\ \mathbf{in}\ e_0] = \langle e^*, v^* \rangle$
where if $x_1 \in v_0'$
$e^* = [\mathbf{let}\ x_1=e_1'\ \mathbf{in}\ e_0']$
and $v^* = (v_0' - \{x_1\}) \cup v_1'$
where $\langle e_1', v_1' \rangle = ER[e_1]$
otherwise $e^* = e_0'$ and $v^* = v_0'$
where $\langle e_0', v_0' \rangle = ER[e_0]$
$ER[\mathbf{letrec}\ x_1=e_1; \ldots; x_n=e_n\ \mathbf{in}\ e_0] = \langle e^*, v^* \rangle$
where if $\{x_1, \ldots, x_n\} \cap v_0' \neq \phi$
$e^* = [\mathbf{letrec}\ x_1=e_1'; \ldots; x_n=e_n'\ \mathbf{in}\ e_0']$
and $v^* = v_0' \cup \ldots \cup v_n' - \{x_1, \ldots, x_n\}$
where $\langle e_i', v_i' \rangle = ER[e_i]$
(for $i = 1, \ldots, n$)
otherwise $e^* = e_0'$ and $v^* = v_0'$
where $\langle e_0', v_0' \rangle = ER[e_0]$

Notation

Tuples in $[\mathbf{Exp} \times 2^{\mathbf{Ide}}]$ are written as $\langle e, v \rangle$.

図 6 冗長な局所定義の除去規則
Fig. 6 Elimination rules of redundant local definitions.

- μ：各レベル k に対し，k に属する局所定義の集合を保持する宣言．
- ω：各変数に対し，そのレベルを保持する環境．
- l：現在のレベル．
- o：各変数に対し，その出現回数を保持する環境．
- σ：各変数に対し，それに対応する真の変数あるいは定数を保持する環境．

このとき SA は，式 e^* と宣言 μ^*，環境 ω^*，o^* の新たな四つ組 $\langle e^*, \mu^*, \omega^*, o^* \rangle$ を返す．引数のうち特に σ について説明する．たとえば局所定義式 **let** $x_1 = e_1$ **in** e_0 を処理するときは，まず e_1 の解析を実行して，それから e_0 の解析に移る．このとき e_1 の解析結果として変数あるいは定数を得る．これを x_1 に対する真の変数あるいは定数と呼び，その対応を保持する必要がある．このための環境が σ である．

式 e に対して共有解析を実行するには，まず，それぞれの引数の初期値を使い

Level Numbers
$l \in \mathrm{N}$

Environment for Level Numbers
$\omega \in \mathrm{L}$

Declarations for Local Definitions
$\mu \in \mathrm{M} = [\mathrm{N} \to 2^{\mathrm{Dec}}]$
$d = \mathrm{Dec}$ local definitions
$d ::= x = e$

Environment for Occurrence Counts
$o \in \mathrm{O} = [\mathrm{Ide} \to \mathrm{N}]$

Environment for Real Names
$\sigma \in \mathrm{S} = [\mathrm{Ide} \to \mathrm{Exp}]$

$SA[b]\mu\omega l o\sigma = \langle \lceil b \rceil, \mu, \omega, o \rangle$
$SA[x]\mu\omega l o\sigma = \langle e^*, \mu, \omega, o^* \rangle$
where $o^* = o + \langle e^* \to oe^* + 1 \rangle$
where $e^* = \sigma x$
$SA[e_0 e_1]\mu\omega l o\sigma = \langle e^*, \mu^*, \omega^*, o^* \rangle$
where if there exists x such that $\lceil x = e_0' e_1' \rceil \in \mu_l' k$
$e^* = \lceil x \rceil, \mu^* = \mu_l', \omega^* = \omega_l', o^* = o_l' + \langle x \to o_l' x + 1 \rangle + \langle e_0' \to o_l' e_0' - 1 \rangle + \langle e_1' \to o_l' e_1' - 1 \rangle$
otherwise
$e^* = \lceil x' \rceil, \mu^* = \mu_l' + (k \to \mu_l' k \cup \{\lceil x' = e_0' e_1' \rceil\}), \omega^* = \omega_l' + \langle x' \to k \rangle,$
$o^* = o_l' + \langle x' \to 1 \rangle$ (†)
where x' is a fresh identifier
where $k = |L[e_0' e_1']\omega_l' l|$
where $\langle e_1', \mu_l', \omega_l', o_l' \rangle = SA[e_1]\mu_0' \omega_0' l o_0' \sigma$
where $\langle e_0', \mu_0', \omega_0', o_0' \rangle = SA[e_0]\mu\omega l o\sigma$
$SA[\backslash x.e_0]\mu\omega l o\sigma = \langle e^*, \mu^*, \omega^*, o^* \rangle$
where $e^* = \lceil x' \rceil, \omega^* = \omega_0' + \langle x' \to k \rangle, o^* = o_0' + \langle x' \to 1 \rangle$
$\mu^* = \mu_0^{(n)} + (k \to \mu_0^{(n)} k \cup \{\lceil x' = \backslash x.(\mathrm{letrec}\ \mu_0^{(n)}(l+1)\ \mathrm{in}\ e_0^{(n)})\rceil\})$ (+)
where $k = |L[\backslash x.(\mathrm{letrec}\ \mu_0^{(n)}(l+1)\ \mathrm{in}\ e_0^{(n)})]\omega_0' l|$, x' is a fresh identifier
where $\mu_0^{(i)} = \mu_0^{(i-1)} + \langle l+1 \to \mu_0^{(i-1)}(l+1) \cup \{\lceil x_i = e_i^{(n)} \rceil\} \rangle$, if $o_0' x_i \geq 2$
$\mu_0^{(i)} = \mu_0^{(i-1)}$, otherwise
for $i = 1, \ldots, n$
where $\mu_0^{(0)} = \mu_0' + \langle l+1 \to \{\} \rangle$
where $e_j^{(i)} = e_j^{(i-1)}[x_i := e_i^{(i-1)}]$ $(j = 0, \ldots, n)$, if $o_0' x_i = 1$
$e_j^{(i)} = e_j^{(i-1)}$ $(j = 0, \ldots, n)$, otherwise
for $i = 1, \ldots, n$
where $e_0^{(0)} = e_0', e_i^{(0)} = e_i$
where $\mu_0'(l+1)$ is $\{\lceil x_1 = e_1 \rceil, \ldots, \lceil x_n = e_n \rceil\}$
where $\langle e_0', \mu_0', \omega_0', o_0' \rangle = SA[e_0](\mu + \langle l+1 \to \{\} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)\sigma\sigma$ (*)
$SA[\mathrm{let}\ x_k = e_1\ \mathrm{in}\ e_0]\mu\omega l o\sigma = SA[e_0]\mu^*\omega^* l o^* \sigma^*$
where $\mu^* = \mu_1, \omega^* = \omega_1, o^* = o_1 + \langle \xi_1 \to o_1 \xi_1 - 1 \rangle, \sigma^* = \sigma + \langle x_1 \to \xi_1 \rangle$
where $\langle \xi_1, \mu_1, \omega_1, o_1 \rangle = SA[e_1]\mu\omega l o\sigma$
$SA[\mathrm{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathrm{in}\ e_0]\mu\omega l o\sigma = SA[e_0]\mu^*\omega^* l o^* \sigma^*$
where $\mu^* = \mu_n, \omega^* = \omega_n, o^* = o_n + \langle \xi_1 \to o_n \xi_1 + o_n x_1 - 1 \rangle + \cdots + \langle \xi_n \to o_n \xi_n + o_n x_n - 1 \rangle, \sigma^* = \sigma_n$
where $\langle \xi_i, \mu_i', \omega_i, o_i \rangle = SA[e_i]\mu_{i-1}\omega_{i-1} l o_{i-1} \sigma_{i-1}$
μ_i' is obtained by replacing all occurrences of x_i in μ_i' by ξ_i
σ_{i-1}' is obtained by replacing all occurrences of x_i in σ_{i-1} by ξ_i
$\sigma_i = \sigma_{i-1}' + \langle x_i \to \xi_i \rangle$
for $i = 1, \ldots, n$
where $\mu_0 = \mu, \omega_0 = \omega', o_0 = o + \langle x_1 \to 0 \rangle + \cdots + \langle x_n \to 0 \rangle, \sigma_0 = \sigma$
where $\omega' = \omega + \langle x_1 \to l_1 \rangle + \cdots + \langle x_n \to l_n \rangle, l_i = |L[e_i]\omega' l|$ for $i = 1, \ldots, n$

Notations
Tuples in $[\mathrm{Exp} \times \mathrm{M} \times \mathrm{L} \times \mathrm{O}]$ are written as $\langle e, \mu, \omega, o \rangle$.
For a declaration set μ, $\mu + \langle k \to \nu \rangle$ denotes
$\lambda l.$if $k = l$ then ν else μl

Initial Set of Declarations
The initial set of declarations μ_ϕ satisfies $\mu_\phi l = \{\}$ for any $l \in \mathrm{N}$.

Initial Environment of Occurrence Counts
The initial environment of occurrence counts o_ϕ satisfies $o_\phi x = 0$ for all identifiers x.

Initial Environment of Real Names
The initial environment of real names σ_ϕ satisfies $\sigma_\phi x = x$ for all identifiers x.

図 7 単純な共有解析規則
Fig. 7 Rules for simple sharing analysis.

$\langle e^*, \mu^*, \omega^*, o^*\rangle = SA[e]\mu_\phi\omega_\phi 1 o_\phi$

によって，四つ組を得る．この時 μ^*0 および μ^*1 には，定数や組み込み関数を含む部分式が宣言されて残っている．そこで，o^* で出現回数を調べて書き戻すものと残すものとに分ける．残したもの $x_1=e_1, \cdots, x_n=e_n$ は，e^* を本体とする局所定義式 **letrec** $x_1=e_1; \cdots; x_n=e_n$ **in** e^* にする．以下，式 e に対する場合分けによって SA を説明する．

4.1 定数または変数の処理

式 e が定数 b の場合は，何も変更せずにそのまま四つ組を返す．e が変数 x の場合は，その x に対応する真の変数または定数 σx を返し，この出現回数を1増やす．

4.2 複合式の処理

式 e が複合式 e_0e_1 の場合は，e_0 と e_1 を順に処理し，その結果の e_0' と e_1' からなる複合式を，最新の局所定義の環境 μ_1' が保持しているか否か調べる．保持していれば対応する変数 x を返す．この時，変数 x の出現回数を1増やし，e_0'，e_1' の出現回数を1ずつ減らす．もし，保持していなければ新しい変数 x' を返す．x' に応じて宣言と各環境を更新する．

4.3 ラムダ式の処理

式 e がラムダ式 $\backslash x.e_0$ の場合は，まず e_0 に対して共有解析を行う．その結果，x に依存する式が $\mu_0'(l+1)$ で得られるので，この各々について出現回数を調べ，1であればその定義を書き戻し，2以上であれば，局所定義として宣言する．最後にラムダ式全体を新しい変数として返し，宣言と各環境も更新する．

4.4 非再帰的な局所定義式の処理

式 e が非再帰的な局所定義式 **let** $x_1=e_1$ **in** e_0 の場合，まず e_1 を処理し，結果として ξ_1（変数または定数）を得る．以下の処理では x_1 を ξ_1 で置換する必要があるので環境 σ を更新する．また e_1 は局所定義であり，式中に出現している訳ではないので ξ_1 の出現回数を1減らす．この宣言および環境で e_0 を解析すればよい．

4.5 再帰的な局所定義式の処理

式 e が再帰的な局所定義式 **letrec** $x_1=e_1; \cdots; x_n=e_n$ **in** e_0 の場合，まず各局所定義のレベルを計算し，ω を更新する．その後 $x_1, \cdots, x_n$ の出現回数を0に初期化し，e_i $(i=1, \cdots, n)$ を逐次的に処理する．e_i を処理した結果として ξ_i（変数または定数）を得る．以下の処理では x_i を ξ_i に対する真の変数または定数で置換する必要があるので，μ 中に既に現れた x_i を置き換え，σ を再帰的定義に対応して更新する．最後に ξ_i の出現回数に（x_i の出現回数）-1 を加え，その宣言および環境で e_0 を解析する．

5. 変換算法を埋め込んだ共有解析

5.1 ラムダ巻き上げ

ある自由式 e が極大自由式となるのは，複合式 ee' または $e'e$ において，e' が束縛されているときか，ラムダ式 $\backslash x.e$ の形で e が出現するときである．極大自由式を検出した場合はその出現回数を無限大にして，書き戻される可能性を除去する．ラムダ巻き上げは SECD マシン[4]のような環境型の遅延評価系を仮定した算法なので，極大自由式が変数からなる場合は巻き上げない．また複合式 e_0e_1 において，e_0，e_1 の両方が自由であってもレベルが異なれば，レベルの低いほうを極大自由式と同等に扱う．

図7の単純な共有解析算法に，ラムダ巻き上げ変換算法を埋め込むには（†）部を

if e_i (either $i=0$ or 1) is compound and
$|L[e_i']\omega_i'l| < |L[e_j']\omega_i'l|$ (either $j=1$ or 0)
$o^* = o_1' + \langle e_i' \to \infty\rangle + \langle x' \to 1\rangle$ (either $i=0$ or 1)
otherwise
$o^* = o_1' + \langle x' \to 1\rangle$

とし，（＊）部を以下のようにすればよい．

where if e_0 is compound and $|L[e_0']\omega_0'(l+1)| \le l$
$o_0' = o_0'' + \langle e_0' \to \infty\rangle$
otherwise
$o_0' = o_0''$
where $\langle e_0', \mu_0', \omega_0', o_0''\rangle$
$= SA[e_0]\mu_0(\omega + \langle x \to l+1\rangle)(l+1)o\sigma$
where $\mu_0 = \mu + \langle l+1 \to \{\}\rangle$

5.2 完全遅延ラムダ持ち上げ

完全遅延ラムダ持ち上げでは，すべてのラムダ式を再帰的超結合子に変換する．再帰的超結合子は局所定義を許す超結合子である．この変換算法は G マシン[7]のようなグラフ簡約型の遅延評価系を仮定している．そこでは局所定義はグラフとして実現されており，これらのグラフは本体の評価に先だって生成される．したがって，使用されない可能性を持つ局所変数を定義する式においては，冗長なグラフが生成される可能性がある．たとえば

$\backslash y.$(**letrec** $x_1=$(cons(+ y 1)x_1);
$x_2=$(cons y x_2) **in**
if (even y)x_1 x_2)

のような式においては，x_1 や x_2 に対応して生成されたグラフのうち，いずれか一方は必ず冗長となる．不必要なグラフの生成を防ぐためには，完全遅延性を保ちつつ局所定義を式の内側に沈めてやる必要がある．このための算法 SI を**図 8** に示す．式 e に対して $\langle e^*, \nu^* \rangle = SI[e]\nu_\phi$ であるとき，ν^* は e において，内側に沈めることのできない最外の局所定義の集まりであり，また e^* は，e において局所定義を可能な限り沈み込ませたものから，最外の局所定義を除いたものである．複合式 ee' の関数部分 e が局所定義式であると，G マシン用の出力コードの有効な最適化を妨げる[8]ので，このような場合，SI は複合式全体に局所定義がかかるようにしている．すなわち (**letrec** $x_2 = e_2$ **in** $e_0)e_1$ のような式は，**letrec** $x_2 = e_2$ **in** $e_0 e_1$ という式に変換される．持ち上げられた局所定義や極大自由式に対する局所定義を含めて依存度解析を実行して，SI の能力を向上させている．

Tuples of Local Definitions

$t \in \mathbf{T} = \mathbf{2^{Dec}}$ tuples of local definitions
$d \in \mathbf{Dec}$ local definitions
$t ::= \vec{x} = \vec{e}$
$d ::= x = e$

Set of Tuples of Local Definitions

$\nu \in \mathbf{2^T}$

Sink-in Rules

$SI : \mathbf{Exp} \to \mathbf{2^T} \to [\mathbf{Exp} \times \mathbf{2^T}]$

$SI[b]\nu = (\lceil b \rceil, \{\})$
$SI[x]\nu = (\lceil x \rceil, \nu)$
$SI[e_0 e_1]\{\vec{x}_1 = \vec{e}_1, \ldots, \vec{x}_n = \vec{e}_n\} = \langle e^*, \nu^* \rangle$
where $\nu^* = \nu_0^* \cup \nu^n$ and $e^* = e_0^*(\mathbf{letrec}\ \nu_1^*\ \mathbf{in}\ e_1^*)$
where $\langle e_0^*, \nu_0^* \rangle = SI[e_0]\nu_0^0$ and $\langle e_1^*, \nu_1^* \rangle = SI[e_1]\nu_1^n$
where if $\vec{x}_i \cap FV[e_0] \neq \phi$ and $\vec{x}_i \cap FV[e_1] = \phi$
$\nu_0^i = \nu_0^{i-1} \cup \{[\vec{x}_i = \vec{e}_i]\}, \nu_1^i = \nu_1^{i-1}, \nu^i = \nu^{i-1}$
else if $\vec{x}_i \cap FV[e_0] = \phi$ and $\vec{x}_i \cap FV[e_1] \neq \phi$
$\nu_0^i = \nu_0^{i-1}, \nu_1^i = \nu_1^{i-1} \cup \{[\vec{x}_i = \vec{e}_i]\}, \nu^i = \nu^{i-1}$
otherwise $\nu_0^i = \nu_0^{i-1}, \nu_1^i = \nu_1^{i-1}, \nu^i = \nu^{i-1} \cup \{[\vec{x}_i = \vec{e}_i]\}$
for $i = 1, \ldots, n$
where $\nu_0^0 = \nu_1^0 = \nu^0 = \{\}$
$SI[\backslash x.e_0]\nu = (\lceil \backslash x.\mathbf{letrec}\ \nu_0^*\ \mathbf{in}\ e_0^* \rceil, \nu)$ where $\langle e_0^*, \nu_0^* \rangle = SI[e_0]\{\}$
$SI[\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0]\{\vec{y}_1 = \vec{e}_1, \ldots, \vec{y}_m = \vec{e}_m\} = \langle e_0^*, \nu_0^* \cup \nu^m \rangle$
where $\langle e_0^*, \nu_0^* \rangle = SI[e_0](\nu_0^m \cup \{([x_1 = \mathbf{letrec}\ \nu_1^*\ \mathbf{in}\ e_1^*])\})$
where $\langle e_1^*, \nu_1^* \rangle = SI[e_1]\nu_1^m$
where if $\vec{y}_i \cap FV[e_0] \neq \phi$ and $\vec{y}_i \cap FV[e_1] = \phi$
$\nu_0^i = \nu_0^{i-1} \cup \{\vec{y}_i = \vec{e}_i\}, \nu_1^i = \nu_1^{i-1}, \nu^i = \nu^{i-1}$
else if $\vec{y}_i \cap FV[e_0] = \phi$ and $\vec{y}_i \cap FV[e_1] \neq \phi$
$\nu_0^i = \nu_0^{i-1}, \nu_1^i = \nu_1^{i-1} \cup \{\vec{y}_i = \vec{e}_i\}, \nu^i = \nu^{i-1}$
otherwise $\nu_0^i = \nu_0^{i-1}, \nu_1^i = \nu_1^{i-1}, \nu^i = \nu^{i-1} \cup \{\vec{y}_i = \vec{e}_i\}$
for $i = 1, \ldots, m$
where $\nu_0^0 = \nu_1^0 = \nu^0 = \{\}$
$SI[\mathbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e_0]\{\vec{y}_1 = \vec{e}_1, \ldots, \vec{y}_m = \vec{e}_m\} = (e_0^*, \nu_0^* \cup \nu^m)$
where $\langle e_0^*, \nu_0^* \rangle = SI[e_0](\nu_0^m \cup \{([x_1 = \mathbf{letrec}\ \nu_1^*\ \mathbf{in}\ e_1^*]), \ldots, ([x_n = \mathbf{letrec}\ \nu_n^*\ \mathbf{in}\ e_n^*])\})$
where $\langle e_i^*, \nu_i^* \rangle = SI[e_i]\nu_i^m$ for $i = 1, \ldots, n$
where if $\exists j$ such that "$\vec{y}_i \cap FV[e_j] \neq \phi$ and $\vec{y}_i \cap FV[e_k] = \phi\ (\forall k \neq j)$"
$\nu_j^i = \nu_j^{i-1} \cup \{\vec{y}_i = \vec{e}_i\}, \nu_k^i = \nu_k^{i-1} (\forall k \neq j), \nu^i = \nu^{i-1}$
otherwise $\nu_j^i = \nu_j^{i-1}$ (for $j = 0, \ldots, n$), $\nu^i = \nu^{i-1} \cup \{\vec{y}_i = \vec{e}_i\}$
for $i = 1, \ldots, m$
where $\nu_0^0 = \ldots = \nu_n^0 = \nu^0 = \{\}$

Notation

If $\nu = \{\vec{x}_1 = \vec{e}_1, \ldots, \vec{x}_n = \vec{e}_n\}$ and $\vec{x}_i = \vec{e}_i$ is $\{x_{i1} = e_{i1}, \ldots, x_{ik_i} = e_{ik_i}\}$, $[\mathbf{letrec}\ \nu\ \mathbf{in}\ e_0]$ denotes $[\mathbf{letrec}\ x_{11} = e_{11}; \ldots; x_{nk_n} = e_{nk_n}\ \mathbf{in}\ e_0]$

Initial Set of Tuples of Declarations

The initial set of tuples of declarations ν_ϕ satisfies $\nu_\phi = \{\}$.

図 8 局所定義の沈下規則
Fig. 8 Rules for sinking-in local definitions.

ラムダ式と各環境を引数とし，これを再帰的超結合子に変換し，各環境を更新する算法として RSC を定義する（**図 9**）．RSC は，実際には Hughes[5),6)] の手法を用いて，再帰的超結合子への変換時には，仮引数をレベルによってソートして，η 簡約を実行することで冗長な再帰的超結合子の生成を抑制している．図 7 の単純な共有解析算法に，完全遅延ラムダ持ち上げ変換算法を埋め込むには，ラムダ式に対する処理の（＋）部を以下のように置換すればよい．

where$\langle e^*, \mu^*, \omega^*, o^* \rangle$
$= RSC[\backslash x.(\mathbf{letrec}\ \nu'\ \mathbf{in}\ e')]\mu_0'^{(n)}\omega_0' l o_0'$
where$\langle e', \nu' \rangle$
$= SI[DA[\mathbf{letrec}\ \mu_0^{(n)}(l+1)\ \mathbf{in}\ e_0^{[n]}]]\nu_\phi$

5.3 超結合子

超結合子への変換は完全遅延ラムダ持ち上げに類似している．しかし，超結合子は局所定義を許さないので，これらを以下のように関数適用に変換する．

- **let** $x_1 = e_1$ **in** e_0
  $\Rightarrow (\backslash x_1.\ e_0)e_1$
- **let** $x_1 = e_1; \cdots; x_n = e_n$ **in** e_0
  $\Rightarrow (\backslash x_1 \cdots \backslash x_n.\ e_0)e_1 \cdots e_n$
- **letrec** $x_1 = e_1$ **in** e_0
  $\Rightarrow (\backslash x_1.\ e_0)(Y(\backslash x_1.\ e_1))$
  ただし，Y は $Y\ f = f(Y\ f)$ で定義される不動点結合子である．
- **letrec** $x_1 = e_1; \cdots; x_n = e_n$ **in** e_0
  $\Rightarrow U_n(\backslash x_1 \cdots \backslash x_n.\ e_0)$
  $(Y(U_n(\backslash x_1 \cdots \backslash x_n.(P_n\ e_1 \cdots e_n))))$
  ただし，P_k は k 個の引数からなるタプルを構成する結合子であり，U_k は $SEL_{ki}(P_k\ e_1 \cdots e_k) = e_i$ で定義される結合子 SEL_{ki} を使って，$U_k\ f\ p = f(SEL_{k1}\ P) \cdots (SEL_{kk}\ p)$ と定義される．

相互再帰的な局所定義を関数適用に変換すると，タプルを構成するために実行時に記憶領域を必要とし負荷が掛かる．そこで，各再帰的局所定義の依存度解析を行い，再帰的なものと非再帰的なものに分解し，前者の数を最小にする．さらに，依存関係のない非再帰的局所定義どうしをまとめて，一つの大域関数の引数とし，定義される大域関数の数を最小にして簡約段数を減らす．このように局所定義式の依存度解析を行ってできた局所定義の入れ子に対して，そのうちの **let** 式をまとめて単一の **let** 式にして，入れ子の数を最小にする算法を CL と定義する．依存関係のない非再帰的局所定義をまとめるためには，**図 10** に示すようなラベル付け算法を使う．再帰的局所定義から作られる依存度グラフ $G=(V, E)$ に対して，強連結成分分解を行った結果，判明した各成分を点，成分間の依存関係を有向枝とする補助グラフ $G'=(V', E')$ を作成する．この補助グラフ G' の点 V' に対して，対応する強連結成分が閉路 (circuit) を含むなら赤，含まないなら白，と色を付ける．強連結成分が一つの点からなる場合でも，自己閉路 (loop) が存在すれば赤を付ける．このようにしてできた補助グラフ $G'=(V', E')$，および V' への赤白の色付けに対して，このラベル付け算法を適用する．赤い点は再帰的局所定義に対応し，白い点は非再帰的局所定義に対応する．必要なのは白い点の間の依存度から構成される白い点のラベル付けなので，赤い点はあたかも存在しないように処理する．このため，赤い点を始点とする有向枝に対しては，その終点に始点と同じラベルを付けている．その結果，各点に対してラベルが割り当てられるので，同じラベルを持つ白い点に対応する強連結成分を同一の局所定義とすることができる．

CL を使う方法では，実行時に必要とする記憶量を最小に抑えることができる．一方，再帰的局所定義をそのまま関数適用に変換する場合にも，記憶領域の割り当てが一回で済むという利点がある．したがっ

Level Numbers

$l \in \mathbf{N}$

Environment for Level Numbers

$\omega \in \mathbf{L}$

Local Definitions

$\mu \in \mathbf{M} = [\mathbf{N} \to 2^{\mathbf{Dec}}]$

$d = \mathbf{Dec}$ local definitions

$d ::= x = e$

Environment for Occurrence Counts

$o \in \mathbf{O} = [\mathbf{Ide} \to \mathbf{N}]$

Translation Rules for Recursive Super-Combinators

$RSC : \mathbf{Exp} \to \mathbf{M} \to \mathbf{L} \to \mathbf{N} \to \mathbf{O} \to [\mathbf{Exp} \times \mathbf{M} \times \mathbf{L} \times \mathbf{O}]$

$RSC[\backslash x.e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^* \rangle$

where $e^* = \lceil x' \rceil$,

$\mu^* = \mu' + \langle k \to \mu k \cup \{\lceil x' = \Phi\ v_1 \ldots v_n \rceil\} \rangle$,

$\omega^* = \omega' + \langle x' \to k \rangle, o^* = o' + \langle x' \to 1 \rangle$,

where $k = \lfloor L[e']\omega' l \rfloor$

where $e' = \lceil \Phi\ v_1 \ldots v_n \rceil$,

$\mu' = \mu + \langle 1 \to \mu 1 \cup \{\lceil \Phi = \backslash v_1 . \ldots . \backslash v_n . \backslash x . e \rceil\} \rangle$,

$\omega' = \omega + \langle \Phi \to 1 \rangle, o' = o + \langle \Phi \to 1 \rangle$,

where $\{v_1, \ldots, v_n\} = FV[\backslash x.e_0]\omega l$

and x', Φ are fresh identifiers

図 9 再帰的超結合子への変換規則

Fig. 9 Translation rules into recursive super-combinators.

```
for all x ∈ V' do label(x) := −1;
Start := all in-degree 0 nodes;
for every node x ∈ Start do
  begin
    label(x) := 0;
    if the color of x is white then
      update-label(x, 1)
    else
      update-label(x, 0)
    endif
  end;

procedure update-label(x, l)
begin
  Next := children of node x;
  for every node y ∈ Next do
    if label(y) < l then label(y) := l;
    if the color of y is white then
      update-label(y, l + 1)
    else
      update-label(y, l)
    endif
end;
```

図 10 強連結成分のラベル付け算法

Fig. 10 Algorithm for labeling of strongly connected components.

て，処理系やプログラムの性質に応じて，中間のアプローチも考えられる．たとえば，再帰的局所定義式に依存度解析を行い，その結果を CL にかけて

$$\begin{array}{l}\textbf{let } x_1=e_1;\cdots;x_{n_1}=e_{n_1}\textbf{ in}\\ \quad\textbf{letrec } x_{n_1+1}=e_{n_1+1};\cdots;\\ \qquad\qquad x_{n_2}=e_{n_2}\textbf{ in}\\ \quad\textbf{let } x_{n_1+1}=e_{n_2+1}\textbf{ in}\\ \quad\textbf{letrec } x_{n_2+2}=e_{n_2+2};\cdots;\\ \qquad\qquad x_{n_3}=e_{n_3}\textbf{ in}\\ \quad\textbf{let } x_{n_3+1}=e_{n_3+1};\cdots;\\ \qquad\qquad x_{n_4}=e_{n_4}\textbf{ in } e_0\end{array}$$

という式を得たとする．このとき，二つの再帰的局所定義をまとめて

$$\begin{array}{l}\textbf{let } x_1=e_1;\cdots;x_{n_1}=e_{n_1}\textbf{ in}\\ \quad\textbf{letrec } x_{n_1+1}=e_{n_1+1};\cdots;\\ \qquad\qquad x_{n_3}=e_{n_3}\textbf{ in}\\ \quad\textbf{let } x_{n_3+1}=e_{n_3+1};\cdots;\\ \qquad\qquad x_{n_4}=e_{n_4}\textbf{ in } e_0\end{array}$$

と変形したほうが，P_k や U_k によるタプルの構成や分解に，その大きさによらず大きな時間がかかる場合は，良い実行結果をもたらす可能性もある．

また，変換する式と各環境を引数にとり，これを自由変数を引数とするような超結合子に変換し各環境を更新する算法を SC と定義する（**図11**）．SC でも RSC と同様に η 簡約を実行して，冗長な超結合子の生成を抑制している．図7の単純な共有解析算法に超結合子変換算法を埋め込むには，ラムダ式に対する処理の(+)部を，この SC を使って以下のように置換すればよい．

where $\langle e^*, \mu^*, \omega^*, o^*\rangle = SC[\backslash x.e']\mu_0'^{(n)}\omega_0' l o_0'$
  where $e' = CL[DA[\textbf{letrec } \mu_0'^{(n)}(l+1)\textbf{ in } e_0'^{(n)}]]$

Level Numbers
$l \in \mathbf{N}$

Environment for Level Numbers
$\omega \in \mathbf{L}$

Local Definitions
$\mu \in \mathbf{M} = [\mathbf{N} \to \mathbf{2^{Dec}}]$
$d = \mathbf{Dec}$ ・・・ local definitions
$d ::= x = e$

Environment for Occurrence Counts
$o \in \mathbf{O} = [\mathbf{Ide} \to \mathbf{N}]$

Translation Rules for Super-Combinators
$SC : \mathbf{Exp} \to \mathbf{M} \to \mathbf{L} \to \mathbf{N} \to \mathbf{O} \to [\mathbf{Exp} \times \mathbf{M} \times \mathbf{L} \times \mathbf{O}]$

$SC[b]\mu\omega lo = \langle\lceil b\rceil, \mu, \omega, o\rangle$
$SC[x]\mu\omega lo = \langle\lceil x\rceil, \mu, \omega, o\rangle$
$SC[e_0 e_1]\mu\omega lo = \langle\lceil e_0 e_1\rceil, \mu, \omega, o\rangle$
$SC[\backslash x.e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^*\rangle$
  where $e^* = \lceil x'\rceil, \mu^* = \mu_0'' + \langle k \to \mu_0'' k \cup \{\lceil x' = \Phi\ v_1 \ldots v_m\rceil\}\rangle,$
    $\omega^* = \omega_0'' + \langle x' \to k\rangle, o^* = o_0'' + \langle x' \to \infty\rangle,$
  where $k = |L[\Phi\ v_1 \ldots v_m]\omega_0'' l|$
    where $\mu_0'' = \mu_0' + \langle 1 \to \mu_0' 1 \cup \{\lceil \Phi = \backslash v_1. \ldots .\backslash v_m.\backslash x.e_0'\rceil\}\rangle,$
      $\omega_0'' = \omega_0' + \langle \Phi \to 1\rangle, o_0'' = o_0' + \langle \Phi \to \infty\rangle,$
    where x', Φ are fresh identifiers and $\{v_1, \ldots, v_m\} = FV[e_0']\omega_0'(l+1) - \{x\}$
      where $\langle e_0', \mu_0', \omega_0', o_0'\rangle = SC[e_0]\mu(\omega + \langle x \to l+1\rangle)(l+1)o$
$SC[\textbf{let } x_1=e_1 \textbf{ in } e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^*\rangle$
  where $e^* = \lceil \Phi\ v_1 \ldots v_m\ e_1\rceil, \mu^* = \mu_0' + \langle 1 \to \mu_0' 1 \cup \{\lceil \Phi = \backslash v_1. \ldots .\backslash v_m.\backslash x_1.e_0'\rceil\}\rangle,$
    $\omega^* = \omega_0' + \langle \Phi \to 1\rangle, o^* = o_0' + \langle \Phi \to \infty\rangle$
  where Φ is a fresh identifier and $\{v_1, \ldots, v_m\} = FV[e_0']\omega_0' l - \{x_1\}$
    where $\langle e_0', \mu_0', \omega_0', o_0'\rangle = SC[e_0]\mu\omega lo$
$SC[\textbf{let } x_1=e_1; \ldots; x_n=e_n \textbf{ in } e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^*\rangle$
  where $e^* = \lceil \Phi\ v_1 \ldots v_m\ e_1 \ldots e_n\rceil, \mu^* = \mu_0' + \langle 1 \to \mu_0' 1 \cup \{\lceil \Phi = \backslash v_1. \ldots .\backslash v_m.\backslash x_1. \ldots .\backslash x_n.e_0'\rceil\}\rangle,$
    $\omega^* = \omega_0' + \langle \Phi \to 1\rangle, o^* = o_0' + \langle \Phi \to \infty\rangle$
  where Φ is a fresh identifier and $\{v_1, \ldots, v_m\} = FV[e_0']\omega_0' l - \{x_1, \ldots, x_n\}$
    where $\langle e_0', \mu_0', \omega_0', o_0'\rangle = SC[e_0]\mu\omega lo$
$SC[\textbf{letrec } x_1=e_1 \textbf{ in } e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^*\rangle$
  where $e^* = \lceil(\Phi_1\ u_1 \ldots u_{m_1})(Y(\Phi_2\ v_1 \ldots v_{m_2}))\rceil,$
    $\mu^* = \mu_0' + \langle 1 \to \mu_0' 1 \cup \{\lceil \Phi_1 = \backslash u_1. \ldots .\backslash u_{m_1}.\backslash x_1.e_0'\rceil, \lceil \Phi_2 = \backslash v_1. \ldots .\backslash v_{m_2}.\backslash x_1.e_1\rceil\}\rangle,$
    $\omega^* = \omega_0' + \langle \Phi_1 \to 1\rangle + \langle \Phi_2 \to 1\rangle, o^* = o_0' + \langle \Phi_1 \to \infty\rangle + \langle \Phi_2 \to \infty\rangle$
  where Φ_1, Φ_2 are fresh identifiers and $\{u_1, \ldots, u_{m_1}\} = FV[e_0']\omega_0' l - \{x_1\},$
    $\{v_1, \ldots, v_{m_2}\} = FV[e_1]\omega_0' l - \{x_1\}$
    where $\langle e_0', \mu_0', \omega_0', o_0'\rangle = SC[e_0]\mu\omega lo$
$SC[\textbf{letrec } x_1=e_1; \ldots; x_n=e_n \text{ in } e_0]\mu\omega lo = \langle e^*, \mu^*, \omega^*, o^*\rangle$
  where $e^* = \lceil U_n(\Phi_1\ u_1 \ldots u_{m_1})(Y(U_n(\Phi_2\ v_1 \ldots v_{m_2})))\rceil,$
    $\mu^* = \mu_0' + \langle 1 \to \mu_0' 1 \cup \{\lceil \Phi_1 = \backslash u_1. \ldots .\backslash u_{m_1}.\backslash x_1. \ldots .\backslash x_n.e_0'\rceil\},$
      $\cup \{\lceil \Phi_2 = \backslash v_1. \ldots .\backslash v_{m_2}.\backslash x_1. \ldots .\backslash x_n.P_n\ e_1 \ldots e_n\rceil\}\rangle,$
    $\omega^* = \omega_0' + \langle \Phi_1 \to 1\rangle + \langle \Phi_2 \to 1\rangle, o^* = o_0' + \langle \Phi_1 \to \infty\rangle + \langle \Phi_2 \to \infty\rangle$
  where Φ_1, Φ_2 are fresh identifiers and $\{u_1, \ldots, u_{m_1}\} = FV[e_0']\omega_0' l - \{x_1, \ldots, x_n\},$
    $\{v_1, \ldots, v_{m_2}\} = FV[e_1]\omega_0' l \cup \ldots \cup FV[e_n]\omega_0' l - \{x_1, \ldots, x_n\}$
    where $\langle e_0', \mu_0', \omega_0', o_0'\rangle = SC[e_0]\mu\omega lo$

図11 超結合子への変換規則
Fig. 11 Translation rules into super-combinators.

6. 評　価

6.1 実 行 例

具体例として以下のような式に変換を試みる．

$\backslash x.(\backslash y.(\times(+\ x\ y)(+\ x\ x)))$

ラムダ巻き上げ版の共有解析では次式を得る．

$\backslash x.(\textbf{letrec } a=(+x); b=(a\ x)\textbf{in}(\backslash y.(\times(a\ y)b)))$

一方，完全遅延ラムダ持ち上げ版では次の式を得る．

$$\begin{array}{l}\textbf{let } \Phi_1=\backslash a.(\backslash b.(\backslash y.(\times(a\ y)b)))\textbf{ in}\\ \quad\textbf{let } \Phi_2=\backslash x.(\textbf{let } a=(+x)\textbf{ in } \Phi_1\ a(a\ x))\textbf{ in } \Phi_2\end{array}$$

さらに超結合子版では次式を得る．

$$\begin{array}{l}\textbf{let } \Phi_1=\backslash a.(\backslash b.(\backslash y.(\times(a\ y)b)))\textbf{ in}\\ \quad\textbf{let } \Phi_2=\backslash x.(\backslash a.(\Phi_1\ a(a\ x)))\textbf{ in}\\ \qquad\textbf{let } \Phi_3=\backslash x.(\Phi_2\ x(+\ x))\textbf{ in } \Phi_3\end{array}$$

いずれの場合にも，式 $(+\ x)$ の計算は一度だけであり，式 $(+\ x\ x)$ は y の値が決まらなくても計算することができる所に浮上しているのがわかる．

また実行速度や記憶消費量を実測するために，冪乗関数 p を使った以下の例で実験した．

$$\begin{array}{l}\textbf{letrec}\\ \quad p=\backslash n.(\backslash x.(\textbf{if}(=\ n\ 0)1\\ \qquad\qquad(\textbf{if}(=\ n\ 1)x\end{array}$$

```
(if(even n)
  (p(/ n 2)(× x x))
  (×(p(/ n 2)(× x x))x))))) in
+(p10 2)(p10 3)
```

ラムダ巻き上げと超結合子についての処理系は，LISPを使って通訳系を構成した．完全遅延ラムダ巻き上げの処理系には，Gマシン用のコードを生成して，機械命令に展開する翻訳系を使用した．すべての実験は，サンマイクロシステムズ社のSPARCstation 1で行った．各々の処理系で共有解析をしたものとしないものについて，その実行時間と記憶消費量を測定した結果を**表1**に示す．これから，先の例ではいずれの場合でも共有解析を行ったほうが，実行時間と記憶消費量の両面において良い結果となることがわかる．

6.2 部分計算への応用

関数プログラムを完全遅延評価することによって，ある種の部分計算(partial computation)[3]を実現することができる．この節では，我々が開発した対話型の遅延評価系を用いて，共有解析によって関数プログラムの実行効率が改善される例を見る．この遅延評価系では，式を大域変数に束縛し，評価結果をそれ以後の計算に利用できるように保持することができる．

異なる条件分岐中の共通部分式をまとめることは無意味に見える．しかし，評価結果の再利用を前提とする部分計算では，いずれか一方の分岐の評価で他の部分式も評価することができるので有効な変換となる．たとえば，リスト l，自然数 n，述語 p をとり，l の要素のうち p を満たすものを，先頭から n 個集めてできるリストを返す関数 f を考えよう．

```
f=\l.(\n.(\p.(if(or(= n 0)(null l))[]
                (if(p(hd l))
                  (cons(hd l)(f(tl l)(- n 1)p))
                  (f(tl l)n p)))))
```

この関数 f に，ラムダ巻き上げを埋め込んだ共有解析を適用すると以下の式を得る．

```
f=\l.letrec ξ0=null l; ξ1=hd l; ξ2=cons ξ1;
            ξ3=f(tl l)in
  \n.letrec ξ4=if(or(= n 0)ξ0)[];
            ξ5=ξ3(- n 1);
            ξ6=ξ3 n in
    \p.(ξ4(if(p ξ1)(ξ2(ξ5 p))(ξ6 p)))
```

ここで重要なのは，異なる条件分岐にある共通部分式 $f(\mathrm{tl}\ l)$ が，一つにまとめられていることである．このために，

$F=f[1,2,3]$

という定義によって，新たに関数 F を作りだし，式

$F1(\backslash x.\mathrm{false})$

を評価することによって，ξ_3 の評価結果を ξ_5, ξ_6 の両方で享受することができるようになる．この結果，関数 f をリスト [1, 2, 3] に特化した関数 F を得ることができる．このように，共有解析を完全遅延評価と組み合わせたとき，プログラマが意図しなかったような点においても実行効率の向上をもたらす場合がある．

一方，共有解析をまったく実行せずにラムダ巻き上げだけを実行した場合，f の定義は

```
f=\l.letrec ξ0=null l; ξ1=hd l; ξ2=cons(hd l);
            ξ3=f(tl l); ξ3'=f(tl l) in
  \n.letrec ξ4=if(or(= n 0)ξ0)[];
            ξ5=ξ3(- n 1);
            ξ6=ξ3' n in
    \p.(ξ4(if(p ξ1)(ξ2(ξ5 p))(ξ6 p)))
```

となる．ここでは関数 f の定義において，共通部分式 $f(\mathrm{tl}\ l)$ は，ξ_3, ξ_3' として別々に出現しているので，いずれも評価するには，リスト l の第一要素の1に対して異なる結果を与える二つの述語を使う必要がある．リスト l の第二，第三要素についても同様なので，関数 F において l に依存する部分すべてを評価するには，2^3 個の異なる述語を与えて特化しなくてはならない．

6.3 局所定義の合併

超結合子への変換に用いる算法 *CL* では，実行時のタプルの構成による記憶消費量を最小に抑えることができる．一方，再帰的局所定義をそのまま関数作用に変換する場合にも，記憶領域の割り当てが高々一回で済むという利点がある．したがって，処理系やプログラムの性質に応じて，中間のアプローチも考えられる．

たとえば，再帰的局所定義式に依存度解析を行い，

表1 共有解析の有無による実行時間と記憶消費量
Table 1 Execution time and memory consumption with/without sharing analysis.

	共有解析あり		共有解析なし	
	実行時間[s]	記憶量[ワード]	実行時間[s]	記憶量[ワード]
ラムダ巻き上げ	9.00×10^{-3}	980	1.67×10^{-2}	1118
完全遅延ラムダ持ち上げ	1.64×10^{-3}	645	2.16×10^{-3}	1040
超結合子	2.17×10^{-2}	336	4.63×10^{-2}	412

その結果を CL にかけて

letrec $x_1=e_1;\cdots;x_i=e_i$ **in**

letrec $x_{i+1}=e_{i+1};\cdots;x_n=e_n$ **in** e_0

という式を得た場合，二つの再帰的局所定義をまとめて

letrec $x_1=e_1;\cdots;x_n=e_n$ **in** e_0

と変形する方法もある．この方法では，タプルの構成結合子 P_n による記憶領域の割り当ては，高々一回である．しかしながら，実行時に片方の **letrec** 式の局所変数のみが本体で参照される場合，参照されない局所定義に対して余分な記憶領域を割り当てることになる．一方，記憶領域の割り当て回数を減らす目的で

letrec $x_1=e_1;\cdots;x_{i-1}=e_{i-1}$ **in**

let $x_i=e_i$ **in**

letrec $x_{i+1}=e_{i+1};\cdots;x_n=e_n$ **in** e_0

という式を

letrec $x_1=e_1;\cdots;x_n=e_n$ **in** e_0

のように，非再帰的な局所定義をも含めて合併してしまう方法も考えられる．この場合も，先の例と同様に余分な記憶領域を割り当てる可能性がある．

局所定義を合併して利点のある場合を特定するために，以下の仮定を設ける．

- $P_k\ e_1\cdots e_k$ を評価してタプルを構成するためには $p(k)=p_1+p_2k$ の時間と $m(k)=m_1+m_2k$ の記憶領域を必要とする．
- $U_k\ f\ p$ を評価して $f(SEL_{k1}\ p)\cdots(SEL_{kk}\ p)$ を得るには，$u(k)=u_1+u_2k$ の時間を必要とする．
- $Y\ f$ を評価して $f(Y\ f)$ を得るには y の時間を必要とする．

このとき，**letrec** $x_1=e_1;\cdots;x_n=e_n$ **in** e_0 の評価においてタプルの構成と分解に必要な時間は $\alpha p(n)+(\alpha+1)u(n)+\alpha y$，記憶消費量は $\alpha m(n)$ である．ただし，本体 e_0 の評価中に $x_1,\cdots,x_n$ のいずれかが参照される場合は $\alpha=1$ であり，それ以外は $\alpha=0$ である．

以上の仮定から，局所定義式

letrec $x_1=e_1;\cdots;x_{n_1}=e_{n_1}$ **in**

letrec $x_{n_1+1}=e_{n_1+1};\cdots;x_{n_1+n_2}=e_{n_1+n_2}$ **in** e_0

を，合併せずにこのまま評価すると $\alpha_1p(n_1)+(\alpha_1+1)u(n_1)+\alpha_1y+\alpha_2p(n_2)+(\alpha_2+1)u(n_2)+\alpha_2y$ の時間と $\alpha_1m(n_1)+\alpha_2m(n_2)$ の記憶領域が必要となる．一方，先の局所定義式を合併すると $\alpha_1p(n_1+n_2)+(\alpha_1+1)u(n_1+n_2)+\alpha_1y$ の時間と $\alpha_1m(n_1+n_2)$ の記憶領域が必要となる．ただし，一般性を失わずに $\alpha_1=\max(\alpha_1,\alpha_2)$ と仮定した．実行時間の差 $\alpha_2p_1+(\alpha_2+1)u_1+\alpha_2y-(\alpha_1-\alpha_2)(p_2n_2+u_2n_2)$ が正のとき，合併すると実行時間は小さくなる．これより，$\alpha_1=\alpha_2$ の場合は合併したほうが良く，$\alpha_1=1,\alpha_2=0$ の場合は，$u_1-(p_2n_2+u_2n_2)\geq 0$ ならば合併したほうが良い．記憶領域の差 $\alpha_2m_1-(\alpha_1-\alpha_2)m_2n_2$ が正のとき，合併することで記憶消費量は小さくなる．これより，$\alpha_1=\alpha_2$ の場合は合併したほうが良く，$\alpha_1=1,\alpha_2=0$ の場合は，合併すると記憶消費量は大きくなる．

7. 結　論

本稿において我々は一般の関数プログラムに適用可能な共有解析算法を提示した．さらに遅延評価系で完全遅延評価を実現するためのプログラム変換算法の性質に注目し，主要な変換算法を，共有解析算法中に埋め込めることを示した．この際，完全遅延ラムダ持ち上げと超結合子の両変換算法に対しては，元の算法より良いコードを出すような改良を試みた．また，共有解析算法を実際に適用した実験を行い，その有効性を確認した．

同一のラムダ式を式の複数の場所で使用することは稀であるため，本稿で提示する共有解析算法では，ラムダ式の共有性は考慮しなかった．もしラムダ式の共有性の検出を行うのであれば，ラムダ変数に依存しない方法であるべきで，我々は，ラムダ式を結合子表現 (combinator expression)[10] へ変換してから，同一性を検査する方法を検討している．

謝辞　超結合子のための翻訳系は，慶應義塾大学の米津光浩氏に提供していただいたものに手を加えて構成した．また査読者の方々には，本論文の改善に有効な助言をいただいた．ここに感謝する．なお本研究の一部は文部省科学研究費補助金（奨励研究（A）05780231：完全遅延評価による関数プログラムの部分計算）の援助による．

参 考 文 献

1) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: *The Design and Analysis of Computer Algorithms*, p. 470, Addison-Wesley, Reading (1974).
2) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers, Principles, Techniques, and Tools*, p. 796, Addison-Wesley, Reading (1986).
3) Futamura, Y.: Partial Computation of Programs, In *RIMS Symposia on Software Science and Engineering*, Goto, E. et al. (eds.), LNCS 147, Springer-Verlag, pp. 1-35 (1983).
4) Henderson, P.: *Functional Programming:*

Application and Implementation, p. 355, Prentice-Hall, London (1980).
5) Hughes, R. J. M.: Super-Combinators: A New Implementation Method for Applicative Languages, *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 11-20 (1982).
6) Hughes, R. J. M.: *The Design and Implementation of Programming Languages*, Ph. D. Thesis, p. 130, Oxford University, Oxford (1983).
7) Johnsson, T.: Efficient Compilation of Lazy Evaluation, *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, Vol. 19, No. 6, pp. 58-69 (1984).
8) Peyton Jones, S. L.: *The Implementation of Functional Programming Languages*, p. 445, Prentice-Hall, London (1987).
9) Takeichi, M.: Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs, *New Generation Computing*, Vol. 5, pp. 377-391 (1988).
10) Turner, D. A.: A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9, pp. 31-49 (1979).

(平成5年7月8日受付)
(平成5年12月9日採録)

金子 敬一（正会員）

1962年生．1985年東京大学工学部計数工学科卒業．1987年同大学大学院工学系研究科情報工学修士課程修了．現在，同大学計数工学科助手．関数プログラミング，並列プログラミング，部分計算等の研究に従事．日本ソフトウェア科学会，ACM各会員．

尾上 能之

1969年生．1992年東京大学工学部計数工学科卒業．現在，同大学大学院工学系研究科計数工学修士課程在学中．関数プログラミング，部分計算等の研究に従事．日本ソフトウェア科学会，ACM各会員．

武市 正人（正会員）

1948年生．1970年東京大学工学部計数工学科卒業．1972年同大学大学院工学系研究科計数工学修士課程修了．現在，同大学計数工学科教授．工学博士．関数プログラミング，並列プログラミング，構成的算法論等の研究に従事．日本ソフトウェア科学会，ACM各会員．