May 別名除去

滝本宗宏[†] 原田賢一[††]

コード最適化やプログラム解析は，変数の定義と使用の関係（def-use relation，定義–使用関係）を基にして行われるものが多い．与えられたプログラムにおいて，別名変数が用いられていると，変数の定義–使用関係が曖昧になるので，別名変数を見つけ出すこと（alias analysis，別名解析）は，コード最適化の効果や解析の精度に大きく貢献する．別名解析によって得られる別名は，一般に Must 別名と May 別名の 2 つに分類される．Must 別名は，つねに同一のメモリ領域を表す別名であり，プログラム中の Must 別名は，統一的な名前の付替えによって定義–使用関係を正確に示すことができる．これに対して May 別名はどのメモリ領域を表すのかが特定できないので，May 別名が用いられている場合には，定義–使用関係を正確に示すことができない．本研究では，May 別名を，制御フローに依存して参照先が変わる部分 Must 別名と，その他の May 別名とに区別する．制御フロー上のあるプログラム点における部分 Must 別名は，特定の実行パスからそのプログラム点に少なくとも 1 つの Must 別名が到達するという性質をもつ．本稿では，このような部分 Must 別名をコード巻上げの手法によって Must 別名に変換し，従来法よりもさらに正確な定義–使用関係を明らかにする手法を提案する．本手法は，別名解析が終了していることを前提として，部分冗長除去に用いられるデータフロー解析と類似の解析法によって，まず部分 Must 別名を見つけ出し，次にそれらが Must 別名となるプログラム点を求める．これらの解析は，ビットベクタ表現を用いた単方向データフロー解析によって実現でき，プログラムサイズを n とした計算量は，$O(n^2)$ で抑えられる．

Eliminating May-aliases

Munehiro Takimoto[†] and Kenichi Harada[††]

Collecting aliasing information is useful to practical code optimizations and sophisticate static program analysis, because aliases make relations between definitions and uses of program variables ambiguous. The aliases introduced by pointer dereferences, is classified into must-aliases which always refer to the same memory location and may-aliases which may refer to different memory locations. Especially, the must-alias information can be effectively used to deal with their dereferences as referenced variables and to expose more relations between definitions and uses. We propose a new technique for replacing may-aliases with must-aliases. This approach pays attention to may-aliases of a specific property, which represents references to the same memory location on a certain execution path. Such alias, called as partial must-alias can be hoisted to the program points where they turn to be must-alias. This eliminating partial must-aliases will lead to an effective result as a pre-transformation that proceeds to application of various techniques based on static analysis. Since our approach can be implemented by a simple data flow analysis using bit-vector representation similar with partial redundancy elimination, its complexity is shown by $O(n^2)$ pessimistically.

1. はじめに

現在用いられているプログラミング言語の中には，同じメモリ領域への参照を異なる式で表現できるものが多い．この同じメモリ領域を表す複数の式は，互いに**別名**（aliases）[1] であると呼ばれる．

例：C 言語で記述した次のプログラムにおいて，関数 `printf` の実引数に使われているポインタの間接参照 `*p` は，変数 `n` の別名である．

```
main ()
{
    int *p, n;
    p = &n;
    n = 4;
    printf("%d",*p);
}
```

† 東京理科大学理工学部
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science
†† 慶應義塾大学理工学部
Department of Information and Computer Science, Faculty of Science and Technology, Keio University

配列要素や構造体のフィールドへの参照においても，別名が生じる可能性がある．

例：次のプログラムにおいて，関数 `printf` の実引数として使われる配列要素への参照 `a[i]` は，定数を添字とする `a[3]` の別名である．

```
main ()
{
    int a[10], i;
    i = 3;
    a[3] = 5;
    printf("%d",a[i]);
}
```

ポインタの間接参照による別名は，C や C++など，ポインタを直接操作できるプログラミング言語ばかりでなく，手続きやメソッドの呼出しにおいて引数を参照渡しすることができる言語にも現れる．別名の存在は，変数の定義に対する使用，あるいは使用に対する定義の特定を困難にする．この別名によって生ずる定義–使用関係の曖昧さは，コード最適化を含めた多くの静的解析の効果を低減させるおそれがある．

例：図 1 (a) のフローグラフにおいて，節 3 の右辺 `*p+2` は，`*p` が `x` の別名であることが分かれば，3 に畳み込むことができる．しかし，`*p` の定義は明示されていないので，その畳込みを単純に行うことはできない．また，図 1 (b) において，節 3 の右辺 `*p+z` と節 4 の右辺 `x+z` は同じ値を生成する式である．しかし，この場合でも，別名を考慮しなければ，その等値関係を単純に発見することはできない．

コード最適化やプログラム解析におけるこの別名の問題は，目的の解析を行う前に，別名関係にある変数を収集する**別名解析**（alias analysis）[1),2),17)] を行い，その解析結果を利用することによって改善される．

例：図 1 (a) と図 1 (b) に示すプログラムについて別名解析行うと，`*p` と `x` は別名であることが分かるので，図 1 (a) の節 3 の右辺は定数へ畳み込むことができる．また，図 1 (b) の節 3 と節 4 の式は共通部分式であることが発見できる．

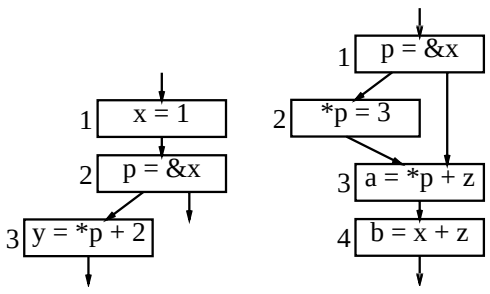
上の例に示すように，あるプログラム点においてつねに同一のメモリ領域を表す別名を **Must 別名**（must aliases）と呼ぶ．これに対して，別名解析をしても，別名がどのメモリ領域を指すのかが一意に特定できない場合がある．そのような別名を **May 別名**（may aliases）と呼ぶ．May 別名は，変数の統一的な置換えができないので，後のプログラム解析やコード最適化に対しては，保守的な取扱いを厳守する立場から，それらの効果を抑制する働きしかしない．

本研究では，従来提案されている May 別名をさらに次の 2 種類に分類し，より詳細なプログラム解析やより効果的なコード最適化を可能にするために，部分 Must 別名を有効に利用する手法を提案する．

絶対 May 別名（absolute may-aliases）： プログラム点にどのような値が到達するのかを静的に知ることができないために，別名が表すメモリ領域を特定できない May 別名．

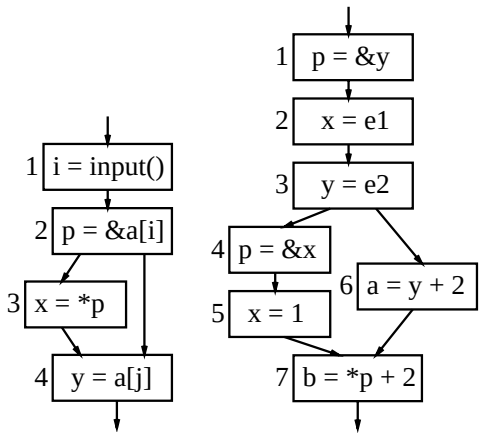
例：図 2 (a) における節 3 の `*p` は，値が分からない変数 `i` を添字とする `a[i]` の別名であり，どのメモリ領域を指すのかが特定できない．絶対 May 別名である `*p` は，節 4 の `a[j]` のような他の変数と別名関係にあるかどうかを特定することができない．

部分 Must 別名（partial must-aliases）： プログラム点に到達できる実行パスが複数存在する



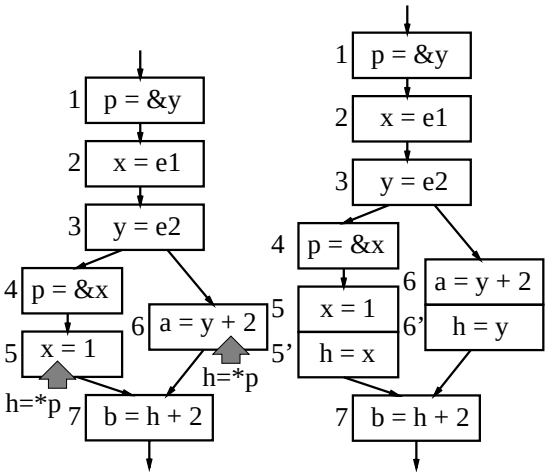
(a) 定数伝播の例 Example of constant propagation　(b) 等値式発見の例 Example of detecting equality

図 1 Must 別名を用いた最適化
Fig. 1 Effects of must-aliases.



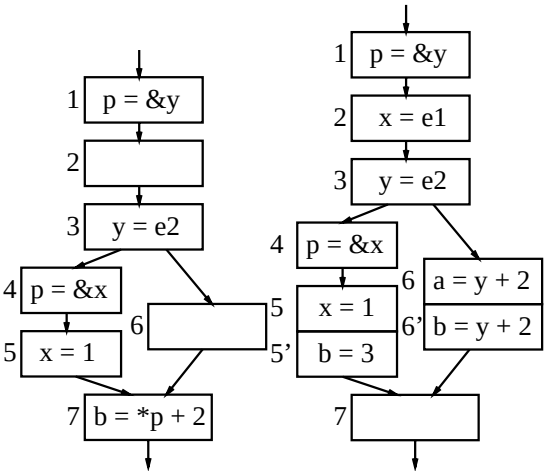
(a) 絶対 May 別名 Absolute may-aliases　(b) 部分 May 別名 Partial must-aliases

図 2 2 種類の May 別名
Fig. 2 Two kinds of may-alias.



(a) 部分 Must 別名の巻上げ Hoisting partial must-aliases (b) 間接参照除去 Indirection removal

図 3 May 別名除去
Fig. 3 Eliminating may-aliases.



(a) プログラムスライシング Program slicing (b) 定数伝播 Constant propagation

図 4 May 別名除去の利用 1
Fig. 4 Eliminating partial must-aliases.

場合でも，その中の少なくとも 1 つの実行パスについては，別名が指すメモリ領域を特定できる May 別名．

例：図 2 (b) における節 7 の `*p` は，節 5 を通って節 7 に到達する場合は，`x` の別名になり，節 6 を通って節 7 に到達する場合は，`y` の別名になる．

部分 Must 別名は，特定の実行パスでは，Must 別名であるので，その実行パスにだけ含まれる節に部分 Must 別名への参照を移動することによって，Must 別名に変換することができる．これを **May 別名除去**（may-alias elimination）[23] と呼ぶ．

例：図 2 (b) の `*p` は部分 Must 別名であり，**図 3** (a) のように，予備変数 `h` を導入して，`h=*p` としてそれぞれ節 5 と節 6 へ巻き上げ，元の `*p` を `h` で置き換えることができる．`*p` は，節 5 と節 6 において，本来 Must 別名であるので，**間接参照**（indirect references）をそれぞれ別名への**直接参照**（direct references）`x`，`y` で置き換えることによって図 3 (b) に示す結果が得られる．この変換は，**間接参照除去**（indirection removal）[20] として知られている．

図 3 (b) の結果からは，プログラム解析あるいはコード最適化において，それぞれ次に示す効果が期待できる．

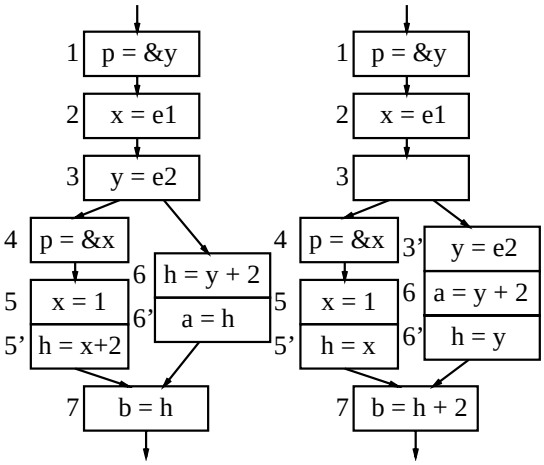
プログラムスライシング： 図 2 (b) の節 7 の文をスライス基準として，プログラムスライシング[3),25)] を適用した場合，別名情報を用いたとしても，図 2 (b) のプログラムスライスでは，節 6 の文以外のすべての文の集合が得られてしまう．これに対して，May 別名除去後のプログラムに適用した場合は，プログラムスライスから節 2 の文を取り除くことができる（**図 4** (a)）．

定数伝播： 図 2 (b) の節 7 の式 `*p+2` は，従来法では畳み込むことができない．これに対して，May 別名除去後のプログラムにおいては，図 3 (b) の節 7 の式をさらに巻き上げることができ，節 5' において，定数に畳み込むことができる（図 4 (b)）[18),26)]．

部分冗長除去： 図 2 (b) の節 7 の式 `*p+2` は，節 6 の式に対して**部分冗長**（partially redundant）[7),12),16),26)] であるが，従来法では節 7 の式を除去することができない．この場合でも，May 別名除去後のプログラムに対して，コピー代入によらない部分冗長除去法[8),19),21)] の適用によって，除去することができる（**図 5** (a)）．

部分不要コード除去： 図 2 (b) の節 3 の文は**部分不要コード**（partially dead code）[9),14),22),27)] であるが，節 7 の `*p` が，`x` と `y` の May 別名であるので，従来法では，部分不要コードと見なされない．これに対して，May 別名除去後のプログラムにおいては，変数 `y` の使用が節 6 にしかないことが分かるので，図 5 (b) のように除去することができる．また，`p=&x` や `p=&y` のようなアドレス値をポインタ変数へ代入する文は，節 7 以降の解析結果によっては不要になる可能性が出てくる．

本稿では，部分 Must 別名であるデータ参照を，Must 別名になるプログラム点まで巻き上げる手法



(a) 部分冗長除去 Partial redundancy elimination　(b) 部分不要コード除去 Partial dead code elimination

図 5 May 別名除去の利用 2
Fig. 5 Eliminating partial must-aliases.

を提案する．ただし，この巻上げを行う際，Must 別名となるプログラム点へ到達する前に，データ参照の安全な移動の点から，巻上げをブロックしなければならないことがある．その場合には，データ参照を無効な巻上げとして元の出現場所へ向けて降下させる操作を行う．この操作を以降，コードの**巻戻し**（delay）[12] と呼ぶ．この May 別名除去は，次の 3 段階で実現できる．

部分 Must 別名解析： 各データ参照について，部分 Must 別名となる範囲を求める．

可能な巻上げ解析： 各データ参照を，部分 Must 別名である範囲において，Must 別名になるプログラム点まで巻き上げる．

可能な巻戻し解析： 無効に巻き上げたデータ参照を巻き戻す．

本稿における以降の構成は，次のとおりである．2 章で，前提とするプログラム表現と解析について簡単に述べる．そして，3 章において，部分 Must 別名とその解析について述べる．次に，4 章において，データ参照の巻上げと無効な巻上げの巻戻しについて述べ，最終的なプログラム変換について述べる．5 章において，評価実験による本手法の効果を示すとともに，本手法全体の計算量について考察する．6 章で関連研究を示し，最後に結論を述べる．

2. プログラム表現

本手法の適用に際しては，原始プログラムに対応する**制御フローグラフ**（control flow graph，以降 CFG と呼ぶ）がすでに作成されているものとする．CFG は，単一の文からなる節の集合 N，辺の集合 $E \subset N \times N$，特別な節である開始節 $\mathbf{s}$ と終了節 $\mathbf{e}$ からなる 4 つ組 $(N, E, \mathbf{s}, \mathbf{e})$ である．CFG 節 n について，その先行節を $pred(n)$ で表し，後続節を $succ(n)$ で表す．

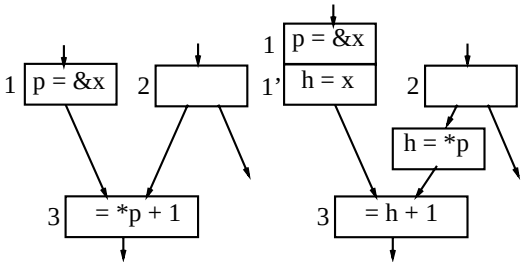
CFG に対して本手法を適用する以前に，別名解析は終了していて，別名情報は利用できるものとする．別名解析においては，解析の効率を考慮して，対象になる変数への参照をいくつかのパターンに限定し，それらのパターンをもつものだけについて，別名を発見する場合が多い[20]．以降，変数への参照は，それに対応するメモリ領域上のアドレスを表す式として表現されているものとする．この式を**参照式**と呼ぶ．参照式は，中間表現のレベルで定義され，入力プログラムにおける構文に左右されないものとする．

例：C 言語における基本データ型の変数 `x` への参照は，`x` に対応するメモリ領域のアドレスを `x_addr` とすると，参照式 `*x_addr` として表現できる．また，構造体 `s` のフィールド `f` への参照 `s.f` は，`s` の先頭アドレスを `s_addr`，`f` に対応する `s_addr` からのオフセットを `f_offset`（定数）とすると，参照式 `*(s_addr + f_addr)` として表現できる．配列要素 `a[i]` についても，`a` の先頭アドレスを `a_addr`，1 つの要素が占めるメモリ領域のバイト数を `a_unit`（定数）とすると，`*(a_addr + a_unit × (* i_addr))` として表現できる．

どの程度まで複雑な構文をもつ変数を別名として取り扱うかは，別名解析に依存する．本手法は，別名解析の結果を利用することを前提としているので，参照式としてより複雑なパターンを導入することも可能である．以下の図や説明においては，参照式の表現に通常の C 言語の記法を用いる．

別名解析は，一般に制御フローを考慮しない**フロー非依存別名解析**（flow-insensitive alias analysis）と，制御フローを考慮に入れた**フロー依存別名解析**（flow-sensitive alias analysis）[17] とに大別できる．本手法では，フロー依存別名解析が行われていることを仮定する．フロー依存別名解析は，データフロー解析を用いて，別名情報を各プログラム点に伝播させる方法が知られている．

本手法はコード巻上げに基づいているので，**図 6** (a) に示すように，2 つ以上の後続節をもつ節 2 から 2 つ以上の先行節をもつ節 3 への辺（**クリティカル辺**，critical edge）[12),13] があると，このままでは効果が制限されてしまう．このようなクリティカル辺は，図 6 (b) のように，合成節を挿入することによって取り除くこ



(a) クリティカル辺
Critical edges

(b) クリティカル辺の除去
Eliminating critical edges

図 6 クリティカル辺の扱い
Fig. 6 Critical edges and their elimination.

とができる．

本稿では，2 つ以上の先行節をもつ節に入ってくる辺は，すべて空文にあたる節が挿入されているものとする．この変形によって，クリティカル辺の問題は解決でき，後のデータフロー解析を単純にすることができる☆[12)]．

3. 部分 Must 別名解析

May 別名除去を行うには，参照式の巻上げが可能な範囲を知る必要があり，そのためにまず，部分 Must 別名情報を収集する必要がある．この節では，部分 Must 別名を求める解析法について述べる．

部分 Must 別名は，ある参照式に関して，Must 別名となるプログラム点が先行して存在することを意味する．そこで，前述の仮定に従って，フロー依存別名解析によって Must 別名がすでに求められているとして，これを基に部分 Must 別名の求め方をデータフローフロー方程式の形で示す．

$MUST_{(x,n)}$ は，Must 別名の情報に基づいて，プログラム点 n における参照式 x が Must 別名をもつ場合に *true*，そうでなければ，*false* となる述語とする．**例：**図 2 (b) の `*p` に対する $MUST$ は，**図 7** (a) に示すとおりである．

部分 Must 別名解析の結果は，述語 $MUST$ を用いて，方程式 1 の最大解として得ることができる．

方程式 1（部分 Must 別名）

☆ 説明を簡単にするために，本稿の図や例では，この空の節の表示を省略している．

node	$MUST$
1	*true*
2	*true*
3	*true*
4	*true*
5	*true*
6	*true*
7	*false*

(a) Must 別名情報
Must-alias information

node	$PMUST$
1	*true*
2	*true*
3	*true*
4	*true*
5	*true*
6	*true*
7	*true*

(b) 部分 Must 別名情報
Partial must-alias information

図 7 Must 別名情報と部分 Must 別名解析の結果
Fig. 7 Given must-alias information and result of partial must-alias analysis.

$$PMUST_{(x,n)} =_{def} \begin{cases} false & if\ n\ is\ \mathbf{s} \\ MUST_{(x,n)} \vee (TRANSP_{(x,n)} \wedge \sum_{m \in pred(n)} PMUST_{(x,m)}) & otherwise \end{cases}$$

$PMUST_{(x,n)}$ は，参照式 x がプログラム点 n において部分 Must 別名であることを表す述語である．$TRANSP_{(x,n)}$ は，参照式 x やその部分式がプログラム点 n で変更されないことを表す述語である．$PMUST$ は，開始節 **s** については *false*，それ以外のすべての節については *true* に初期化し，上記の関係に基づいて $PMUST$ を前向きに伝播させる．この際，伝播を，$TRANSP = true$ となる節に限定することによって，部分 Must 別名情報の到達範囲は，データ参照の対象になるメモリ領域のアドレスが変わらない範囲となる．

部分 Must 別名解析は，プログラム中の各参照式 x について，$PMUST_{(x,n)} = true$ となるプログラム点 n の集合を求めることである．以降の説明においても，“解析” という用語を，述語の真偽を決定する問題として用いる．

例：図 2 (b) の `*p` に対する $PMUST$ は，図 7 (b) に示すとおりである．この結果は，図 2 (b) 中のすべてのプログラム点において，`*p` が部分 Must 別名であることを示している．

4. 巻上げ解析と変換

部分 Must 別名を Must 別名へ変換するためには，まず参照式の巻上げが可能なプログラム点を求める必要がある．参照式 x の巻上げは，元のデータ参照の出現を予備変数 h で置き換え，巻上げ先のプログラム点へ代入文 $h = x$ を挿入することによって実現する．このプログラム点の解析は，次の 2 段階からなる．

(1) 可能な巻上げ点の解析
(2) 無効な巻上げを巻き戻す点の解析

本章では，この 2 つの解析について述べた後，最終的な挿入点の求め方と Must 別名への変換方法を示す．

4.1 可能な巻上げの解析

部分 Must 別名をもつ参照式の可能な巻上げ点は，次に示す方程式 2 の最大解を求めることによって，得ることができる．

方程式 2（可能な巻上げ）

$$HOISTED_{(x,n)} =_{def} \begin{cases} false & if\ n\ is\ \mathbf{e} \\ COMP_{(x,n)} \vee TRANSP'_{(x,n)} \wedge \prod_{m \in succ(n)} (HOISTED_{(x,m)} \wedge PMUST_{(p,s)} \wedge \neg MUST_{(p,s)}) & otherwise \end{cases}$$

述語 $HOISTED_{(x,n)}$ は，参照式 x をプログラム点 n まで巻き上げることができることを表す．述語 $COMP_{(x,n)}$ は，解析対象になる CFG において，参照式 x がプログラム点 n に出現しているかどうかを表す述語である．$TRANSP'_{(x,n)}$ は，部分 Must 別名解析の場合と同様に，参照式 x やその部分式がプログラム点 n で変更されないことに加え，x が表すメモリ領域中の値も変更されないことを表す述語である．

$HOISTED$ は，終了節 **e** については $false$，それ以外のすべての節については $true$ に初期化し，後向きに伝播させる．この際，巻上げが有効であるかどうかは $PMUST$ によって示されるので，伝播の範囲は，$PMUST = true$ の範囲とする．$TRANSP' = false$ である節においては，データ参照の対象になるメモリ領域やそのメモリ領域中の値が変わる可能性があるので，部分 Must 別名解析と同様に，伝播をブロックしなければならない．また，$MUST = true$ のプログラム点は，部分 Must 別名を Must 別名に変換できるプログラム点なので，その点で伝播をブロックする必要がある．参照式 x の巻上げにおいては，はじめて Must 別名になるプログラム点が求められればよい．それ以上の巻上げは，後のレジスタ割付け[4] において，レジスタのスピルを引き起こす可能性を大きくするだけであるので，巻き上げる範囲は，$\neg MUST = true$ であるプログラム点に限定する．さらに，巻上げは，各実行パスに新しい参照を挿入しない範囲で行う必要がある．この性質は，一般に**安全**（safty）性と呼ばれ，対象とする計算がすべての隣接節（前向き移動の際は，すべての先行節，後向き移動の際は，すべての後続節）から，移動してくる場合に限って，さらなる移動を許すという条件で表現される．方程式 2 において，プロダクション $\prod$ は，この安全性の条件を表している．

例：図 8 の節 6 の `h=*p` を，節 1 や節 2 に巻き上げると，節 3 と節 4 を通る実行パスに新しい計算を導入することになる．したがって，図 8 の例の場合，安全に巻き上げることができる範囲は，節 5 と節 6 である．

最終的に，参照式は，その出現場所（$COMP = true$）から，Must 別名（$MUST = true$）となるプログラム点までのうちで安全な範囲で巻き上げられる．

例：図 2 (b) の `*p` に対する $HOISTED$ は，**図 9** (a) に示すとおりである☆．

可能な巻上げの解析によって得られた，$HOISTED = true$ であるプログラム点のうち，開始節に最も近いプログラム点を参照式の挿入点として得ることがで

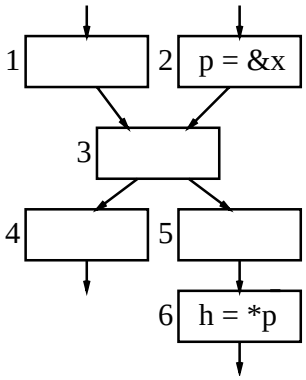


図 8 安全な巻上げ
Fig. 8 Safe hoisting.

node	$HOISTED$
1	$false$
2	$false$
3	$false$
4	$false$
5	$true$
6	$true$
7	$true$

(a) 可能な巻上げの解析結果
Result of possible hoisting

node	$PREINSERT$
1	$false$
2	$false$
3	$false$
4	$false$
5	$true$
6	$true$
7	$false$

(b) 仮の挿入点
Pre-insertion points

図 9 可能な巻上げの解析結果と挿入点
Fig. 9 Result of possible hoisting and insertion.

☆ 図 2 (b) 上で，$HOISTED$ を計算すると，節 5 は $false$ になる．実際は，2 章で述べた入力プログラムの仮定から，2 つ以上の先行節をもつ節 7 に入ってくる辺上には，空の節 n が存在し，この n において，$HOISTED = true$ となる．本稿では，n を省略し，節 5 において $HOISTED = true$ とすることで簡単化している．

きる．この挿入点を，最終的に得る挿入点と区別して，**仮の挿入点**（pre-insertion points）と呼ぶことにする．参照式 x に対する仮の挿入点 n は，上述の性質を満たすプログラム点として，方程式 3 に示す $PREINSERT_{(x,n)}$ によって表現することができる．

方程式 3（仮の挿入点）

$$PREINSERT_{(x,n)} =_{def} HOISTED_{(x,n)} \wedge \neg \prod_{m \in pred(n)} HOISTED_{(x,m)}$$

例：図 2 (b) の `*p` に対する $PREINSERT$ は，図 9 (b) に示すとおりである．

4.2 可能な巻戻しの解析

Must 別名となるプログラム点へ到達した部分 Must 別名に関しては，各参照式を仮の挿入点へ巻き上げることによって，Must 別名へ変換されることができる．しかし，部分 Must 別名の中には，巻上げの結果，Must 別名となるプログラム点に達しなかったものが存在することがある．

例：図 8 における節 6 の `h=*p` は，節 5 が仮の挿入点となるが，そこへの巻上げを行っても，Must 別名にはならない．

このような無効な巻上げは，後のレジスタ割付けにおいて，レジスタからのスピルを増大させる可能性があるので，元の出現場所に向けて巻き戻しておく必要がある．

例：図 8 の場合，このような巻戻しを行うことによって，節 6 の参照式の最終的な挿入点は節 6 自身となる．

可能な巻戻しによって計算される節の集合は，方程式 4 の最大解を求めることによって得ることができる．

方程式 4（可能な巻戻し）

$$DELAY_{(x,n)} =_{def} \begin{cases} false & if\ n\ is\ \mathbf{s} \\ PREINSERT_{(x,n)} \vee (HOISTED_{(x,n)} \wedge \\ \quad \prod_{m \in pred(n)} (\neg COMP_{(x,m)} \wedge DELAY_{(x,m)} \wedge \\ \quad \neg MUST_{(p,m)})) & otherwise \end{cases}$$

述語 $DELAY_{(x,n)}$ は，参照式 x がプログラム点 n において巻戻し可能であることを表す．$DELAY$ は，開始節 **s** については $false$，それ以外のすべての節については $true$ に初期化し，前向きに伝播させる．$DELAY$ 情報の伝播は，安全な移動を保証しなければならないので，先行節における $DELAY$ のプロダクション $\prod$ で表現する．参照式の巻戻しは，Must 別名になる場所あるいは元の出現場所でブロックしなければならないので，$\neg MUST = true$ かつ $\neg COMP = true$ の場合にだけ後続節に伝播させる．

例：図 2 (b) の `*p` に対する $DELAY$ は，**図 10** (a) に示すとおりである．

最後に，巻戻し解析の結果から，出口に最も近いプログラム点が，実際に参照式を挿入する点として得られる．すなわち，参照式 x に対する挿入点 n は，方程式 5 を満たす $INSERT_{(x,n)}$ によって与えられる．

node	$DELAY$
1	$false$
2	$false$
3	$false$
4	$false$
5	$true$
6	$true$
7	$false$

(a) 可能な巻戻し解析の結果 Result of possible delaying

node	$INSERT$
1	$false$
2	$false$
3	$false$
4	$false$
5	$true$
6	$true$
7	$false$

(b) 挿入点 Final insertion points

図 10 可能な巻戻し解析の結果と挿入点
Fig. 10 Result of possible delaying and final insertion.

方程式 5（挿入点）

$$INSERT_{(x,n)} =_{def} DELAY_{(x,n)} \wedge \neg \prod_{m \in succ(n)} DELAY_{(x,m)}$$

例：図 2 (b) の `*p` に対する $INSERT$ は，図 10 (b) に示すとおりである．

4.3 Must 別名への変換

部分 Must 別名から Must 別名への変換を元のプログラムに反映させるためには，参照式ごとに固有の予備変数を生成して，元の式をその変数で置き換え，元の式の値を予備変数へ代入する文を，$INSERT = true$ であるプログラム点へ挿入する．予備変数への置換えと，挿入点へのデータ参照の挿入は，次の 3 つの場合に分けて処理をする．

(1) 挿入点と元の出現場所が同じ場合：予備変数への置換えも挿入も行わない．
(2) 参照式 x が挿入点において Must 別名でない場合：予備変数 h によって元の出現を置き換え，挿入点に $h = x$ を挿入する．
(3) 参照式 x が挿入点において Must 別名である場合：予備変数 h によって元の出現を置き換え，x との Must 別名関係にある参照式の中から，より単純な参照式 x' を選んで，$h = x'$ を挿入点に挿入する．

(1) から (3) に対応する変換を行うことによって，May 別名除去を実現することができる．(3) の挿入は，間接参照除去に対応する．

例：図 2 (b) の `*p` に対する変換結果は，図 3 (b) に示すとおりである．

5. 評価と計算量

この章では，本手法の効果を実験結果を用いて示し，本手法の実行コストについて，データフロー解析部を中心に考察する．

5.1 May 別名除去と別名解析

本手法では，従来 May 別名として扱われていた変数のうち，いくつかの実行パス上において Must 別名であるものを考慮するので，本手法の有効性は Must 別名解析の精度によって左右される．本実験では，次の BNF によって定義される参照式（*indir*）をもつデータ参照を別名解析の対象にした．

$$
\begin{aligned}
\mathit{base} &\rightarrow \texttt{const} \\
&\mid \texttt{var} \\
\\
\mathit{addr} &\rightarrow \mathit{base} \\
&\mid \mathit{base} + \texttt{const} \\
&\mid \mathit{base} + \texttt{const} \times \texttt{var} \\
\\
\mathit{indir} &\rightarrow * \mathit{addr} \\
&\mid ** \mathit{base}
\end{aligned}
$$

ここで，*base* の `const` は，変数や配列に対して静的に割り付けられるメモリ領域のアドレス，`var` は，ポインタ変数（`var`）を介して参照されるメモリ領域のアドレスを表す．*addr* の `const` は，*base* からのオフセット値，あるはバイト数を表し，*indir* は，メモリからの値の直接または間接参照を表す．別名解析にあたっては，その前に**定数伝播**（constant propagation）[1] の処理を行い，参照式中の定数オペランドをもつ部分式は，すべて畳込みを行うようにした．

例：図 11 の `a[i]` に対する別名解析の結果は，節 1 の `i=1` によって，節 1 において `a[1]` と `a[i]` が Must 別名になる．`a[i]` に対する参照式を `*( a_addr+4× i_addr)` とすると，節 1 において `i=1` によって畳み込まれる `*(a_addr+4)` を*(a_addr+4× *(i_addr))` の Must 別名として扱う．

5.2 評価実験の結果

本提案手法の効果を評価するために，これまで述べてきた方法をバックエンドに組み込んだ C コンパイラを実装し，それを用いて実験を行った．このコンパイラのフロントエンドは，既成のコンパイラ lcc [10] を用いて実現しており，lcc が内部にもつ低水準な中間表現をファイルに出力する．バックエンドは，このファイルから中間表現を入力して CFG を作成し，定数伝播と別名解析を行う．別名解析は，参考文献 17) の手法に基づく手続き内別名解析として実現した．最終的に得られる CFG に対して，本手法の適用を行った．

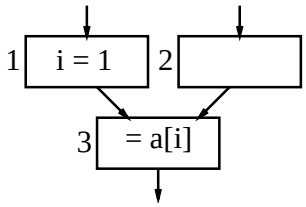


図 11 定数畳込みによる Must 別名
Fig. 11 Must-alias based on constant folding.

表 1 本手法の結果
Table 1 Experimental results.

programs	all-refs	p-musts	hoists	musts
sed	923	55	13	2
gzip	2,831	109	44	57
byacc	3,734	102	41	37
make	3,853	516	202	140
ng	10,702	493	206	221
lcc	8,005	546	218	570

評価には，Unix システムにおいて現実によく使用され，しかもサイズの異る次の 6 つのツールを取り上げ，それらのプログラムをサンプルとして用いた．

sed（3,207 行） gzip（9,344 行）
byacc（6,645 行） make（30,281 行）
ng（30,286 行） lcc（29,530 行）

表 1 に実行結果を示す．表の各列の意味は次のとおりである．

all-refs：プログラムに現れたすべての参照式の個数
p-musts：プログラムに現れたすべての部分 Must 別名の個数
hoists：本手法適用後に，巻き上げられた部分 Must 別名の個数
musts：本手法適用後に，部分 Must 別名から Must 別名へ変換された別名の個数

この評価実験の結果に基づいて，まず，プログラム中の参照式のうち，部分 Must 別名がどれだけの割合を占めていたかをグラフにして**図 12** に示す．グラフの示す値は，各プログラムごとに p-musts/all-refs を計算し，パーセント表示したものである．このグラフが示すように，各プログラムの部分 Must 別名の割合は 10%前後であり，全体の平均値は 6.22%であった．

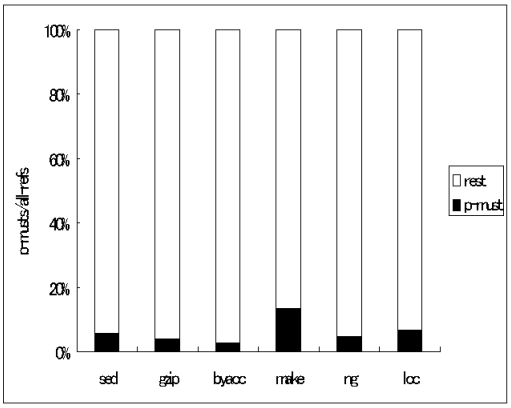


図 12 参照式中の部分 Must 別名の割合
Fig. 12 Rate of partial must-aliases.

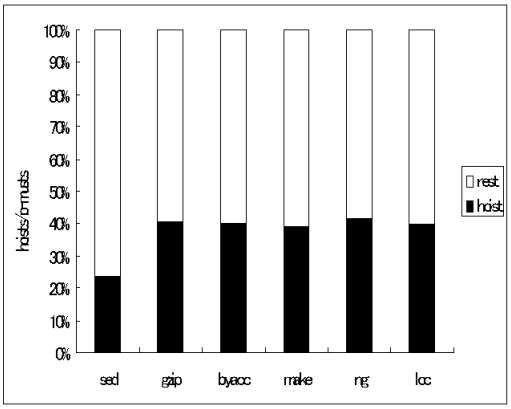


図 13 巻き上げられた部分 Must 別名の割合
Fig. 13 Rate of hoisted partial must-aliases.

次に，プログラム中の部分 Must 別名のうち，本手法の適用によって巻き上げられた別名の割合 hoist/p-musts を，本手法の効果としてパーセント表示で**図 13** に示す．部分 Must 別名のうちで，40%程度の別名が May 別名除去による効果を得たことが分かる．平均は 37.50%であった．

最後に，巻き上げられた部分 Must 別名に関して，それらの個数と，4.3 節の場合 (3) によって変換された Must 別名の個数との比率（musts/hoists）を**図 14** に示す．ここで，ng と lcc の結果が，1 倍を超えているのが分かる．特に，lcc は 2.5 倍を上回っており，平均した値も 1.12 倍となっていることから，1 つの部分 Must 別名は，複数の実行パスにおいて Must 別名へ変換される可能性があることを示している．

5.3 解析コスト

本手法は，従来からよく用いられる単方向データフロー解析によって実現を行っているので，手法全体の計算量は，単方向データフロー解析の計算量と同じである．単方向のデータフロー解析は，プログラムサイズを表すパラメータ n を用いて，$O(n^2)$ で表されることが知られている．実際に，この計算量に達するのは，**既約な**（irreducible）CFG に対して適用した場合など，特異な場合だけに限られる．現実に存在するほとんどのプログラムが**可約な**（reducible）CFG であることを考慮すると，実際には，より小さいコストで解析が終了することが予想される．

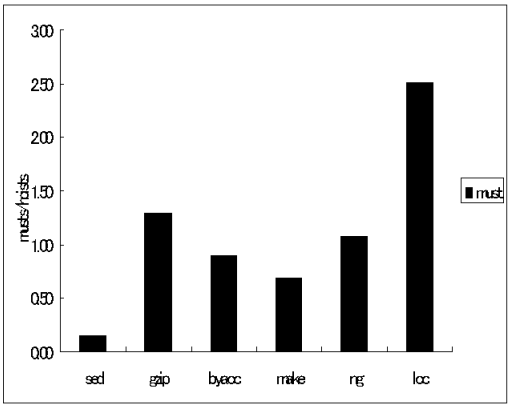


図 14 巻き上げられた部分 Must 別名のうち，Must 別名になった割合
Fig. 14 Rate of partial must-aliases translated into must-aliases.

本手法の実現においては，1 つのスロットを 1 ビットで表現し，ワード単位で計算を行うようにした．また，各方程式の解を得るために，データ伝播の候補になるワードだけをワークリストを用いて管理した．すなわち，計算によって変化が生じたワードは，ワークリストに加え，次の計算候補をそのワークリストから取り出すようにした．この方法は，ビットベクタの使用による並列計算の利点と，スロットごとの計算によって伝播の範囲を限定できる利点を兼ね備えた手法として知られている[11]．

実際にかかった計算量の目安として，データの伝播のためにアクセスしたワードの数を，部分 Must 別名解析，可能な巻上げの解析，可能な巻戻しの解析ごとに**図 15** に示す．X 軸は，プログラムサイズとして中間表現における命令数を示し，グラフ上の左からの各点は表 1 に示したプログラムの順序と対応している．このグラフから，どの解析も命令数に関して，ほぼ線形になっていることが確かめられた．

6. 関 連 研 究

本手法が前提にしている別名解析は，従来から多くの研究がなされてきた[1),2),17)]．別名解析は，データ

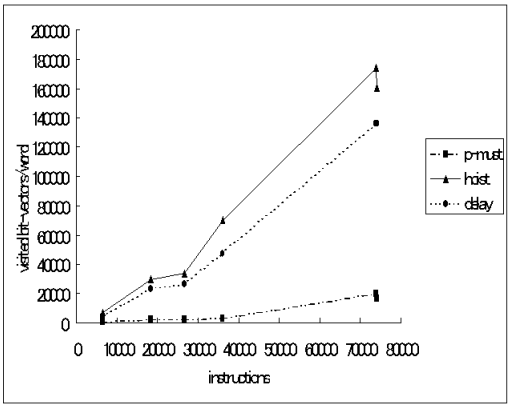


図 15 データフロー解析のコスト
Fig. 15 Cost for dataflow analyses.

フロー解析を用いて，各プログラム点における別名情報を集めるフロー依存の方法がよく知られている．別名解析をどれだけ詳細に行う必要があるかは，後の最適化の効果にかかわる問題であるが，別名解析のデータフロー解析では，データの評価を並列に行うビットベクタ法が利用できないので，詳細な解析には大きなコストがかかる．別名解析の効率向上のために，別名解析を**静的単一代入**（static single assignment，以降 SSA 形式と呼ぶ）[5] 形式上で高速に行う手法が提案されている[24]．

本手法は，データ参照を巻き上げることによって実現している．同様に，データ参照を巻き上げることによって，冗長なロード命令を除去する手法に，Lo らの手法[15] がある．本手法では，対象とする参照式への変更があると，そこで巻上げをブロックするので，効果が制限される．そこで，Lo らの手法とコピー伝播とを組み合わせれば，事前に多くの変数への代入を取り除くことができるので，本手法による効果の増大が期待できる．一方，Lo らの手法に本手法を適用することによって，ロードに要するコストをさらに減らすことも期待できる．つまり，Lo らの手法と本手法は，補完的に適用が可能である．

Lo らが冗長除去に採用している方法は，部分冗長除去であるので，同様の理由で，部分冗長除去は，本手法の効果を上げることが期待できる．

本手法は，部分冗長除去の 1 つである**遅延コード移動**（lazy code motion，以降 LCM と呼ぶ）[12] と同様に，可能な巻上げ範囲の解析と，可能な巻戻し範囲の解析に基づいている．可能な巻上げに関して，LCM は式が利用可能である場合，安全な巻上げでなくても巻上げを許すのに対して，本手法は安全な巻上げだけに制限している．この制限は，多くの最適化や静的解析の前処理として本手法を適用する際にプログラムの意味を保存するために必要である．また，可能な巻戻しに関して，LCM が元のプログラム点へ向けて巻戻し可能な式をすべて巻き戻すのに対して，本手法は，部分 Must 別名から Must 別名になった参照式を巻き戻さないようにして May 別名除去を実現している．

SSA 形式における別名情報の表現方法については，Cytron らの手法[6] や Chow らの手法[20] がある．SSA 形式では，変数への代入を明示しなければならないので，代入によって生じる別名への潜在的な代入を明示しなければならない．この潜在的代入の明示化は，SSA 形式によるプログラム表現のサイズを大きくする．この問題に対して，Cytron らの手法は，適用する最適化において，必要となる別名情報だけを SSA 形式に反映させる方法を提案している．Cytron らの手法では，不要な別名情報が現れることがないので，プログラム表現のサイズを小さくする効果があるが，必要に応じて，影響を及ぼす別名情報をプログラムに反映させなければならないので，コストがかかる．本手法はこのコストの低減に役立つ．また，Chow らは，別名表現を加えた SSA 形式のプログラムに等値式発見を適用することによって，プログラムサイズを小さくしたプログラム表現である，HSSA（Hashed SSA）形式を提案している．Chow らの手法では，May 別名をすべて単一の変数名に置き換えることによって，さらにプログラムサイズを小さくするようにしている．本手法は，絶対 May 別名と部分 Must 別名を区別して扱い，部分 Must 別名の一部は Must 別名へ変換することができるので，HSSA の表現をより詳細化することができる．

7. 結　　論

本稿では，May 別名を除去するための効果的な新しいアルゴリズムを提案した．

従来，May 別名として扱われていた別名を，絶対 May 別名と部分 Must 別名の 2 種類に分類することを提案し，部分 Must 別名が巻上げによって Must 別名に変換できることを示した．

本手法の結果は，定義–使用あるいは使用–定義関係に基づく多くのプログラム解析やコード最適化をより効果的なものにすることが期待できる．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).

2) Appel, A.W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
3) Bodik, R. and Gupta, R.: Partial Dead Code Elimination using Slicing Transformations, *Proc. Programing Language Design and Implementation* (*PLDI '97*), pp.159–170, ACM (1997).
4) Chow, F.C. and Hennessy, J.L.: The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.4, pp.501–536 (1990).
5) Cytron, R., Ferrante, J., Rosen, B.K. and Wegman, M.N.: Efficiently Computing Static Single Assignment Form and Control Dependence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
6) Cytron, R. and Gershbein, R.: Efficient Accommodation of May-Alias Information in SSA Form, *Proc. Programing Language Design and Implementation* (*PLDI '93*), pp.36–45, ACM (1993).
7) Dhamdhere, D.M. and Patil, H.: An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.2, pp.321–336 (1993).
8) Chow, F.C., Chan, S., Kennedy, R., Liu, S.M. and Lo, R.: A New Algorithm for Partial Reundancy Elimination based on SSA Form, *Proc. Programing Language Design and Implementation* (*PLDI '97*), pp.273–286, ACM (1997).
9) Feigen, L., Klappholz, D., Casazza, R. and Xue, X.: The Revival Transformation, *Proc. Principles of Programming Languages* (*POPL '94*), pp.421–434, ACM (1994).
10) Fraser, C. and Hanson, D.: *A Retargetable C Compiler: Design and Implementation*, Addison Wesley (1995).
11) Khedker, U.P. and Dhamdhere, D.M.: A Generalized Theory of Bit Vector Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.5, pp.1472–1511 (1994).
12) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. Programing Language Design and Implementation* (*PLDI '92*), pp.224–234, ACM (1992).
13) Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).
14) Knoop, J., Rüthing, O. and Steffen, B.: Partial Dead Code Elmination, *Proc. Programing Language Design and Implementation* (*PLDI '94*), pp.147–158, ACM (1994).
15) Lo, R., Chow, F., Kennedy, R., Liu, S. and Tu, P.: Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, *Proc. Programing Language Design and Implementation* (*PLDI '98*), pp.26–37, ACM (1998).
16) Morel, E. and Renvoise, C.: Global Optimization by Supperssion of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
17) Muchnick, S.S.: *Advanced Compiler Design Implementation*, Morgan Kaufmann (1997).
18) Rüthing, O., Knoop, J. and Steffen, B.: Detecting Equalities of Variables: Combining Efficiency with Precision, *Proc. Int. Static Analysis Symposium* (*SAS '99*), Vol.1694 of LNCS, Venice, pp.232–247, Springer-Verlag (1999).
19) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global Value Numbers and Redundant Computations, *Proc. Principles of Programming Languages* (*POPL '88*), pp.12–27, ACM (1988).
20) Chow, F.C., Chan, S., Liu, S., Lo, R. and Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA Form, *Proc. Int. Compiler Construction* (*CC '96*), LNCS, Berlin, Springer-Verlag (1996).
21) Steffen, B., Knoop, J. and Rüthing, O.: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *Proc. Int. European Symposium on Programming* (*ESOP '90*), Copenhagen, Denmark, pp.389–405, Springer-Verlag (1990).
22) Takimoto, M. and Harada, K.: Partial Dead Code Elimination Using Extended Value Graph, *Proc. Int. Static Analysis Symposium* (*SAS '99*), Vol.1694 of LNCS, Venice, Springer-Verlag (1999).
23) Takimoto, M. and Harada, K.: Eliminating May-aliases, *Proc. Int. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet* (*SSGRR '01*), L'Aquila (2001).
http://www.ssgrr.it/en/ssgrr2001/papers.htm
24) Wegman, M.N. and Zadeck, F.K.: Constant Propagation with Conditional Branches, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.2, pp.181–210 (1991).
25) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol.SE-10, No.4, pp.352–357 (1984).
26) 滝本宗宏，原田賢一：拡張値グラフに基づく効果的な部分冗長除去法，情報処理学会論文誌，Vol.38, No.11, pp.2237–2250 (1997).
27) 滝本宗宏，原田賢一：拡張値グラフを用いた部分

無効コード除去法，情報処理学会論文誌，Vol.41, No.1, pp.46–58 (2000).

(平成 14 年 1 月 7 日受付)
(平成 14 年 6 月 3 日採録)

滝本　宗宏（正会員）

1967 年生．1994 年慶応義塾大学大学院理工学研究科計算機科学専攻修士課程修了．現在，東京理科大学理工学部情報科学科助手，プログラミング言語およびその処理系に興味を持つ．ACM，ソフトウェア科学会会員．

原田　賢一（正会員）

1940 年生．1966 年慶応義塾大学大学院工学研究科管理工学専攻修士課程修了．1967 年同大学工学部助手．1970〜1989 年同大学情報科学研究所助手，専任講師，助教授，教授．1989 年 4 月より同大学理工学部計測工学科教授．1996 年 4 月より同学部情報工学科教授．この間，1973〜1975 年米国メリーランド大学訪問研究員．工学博士．ソフトウェア工学，プログラミング言語およびその処理系の研究に従事．ACM，IEEE，ソフトウェア科学会会員．