値番号に基づく部分冗長性除去

大 平 怜[†] 平 木 敬[†]

近年，実行時コンパイラが広く用いられてきている．実行時コンパイラは解析時間を短くすると同時に，部分冗長性除去と大域的値番号付けのような冗長性除去を行うと効果的である．しかし，従来の部分冗長性除去は字面で等価な式の冗長性しか扱えず，大域的値番号付けはすべての実行パスにおいて同じ値を計算する式の冗長性しか扱うことができない．したがって，一般のプログラム中に存在する，字面は異なるがある実行パスにおいてのみ同じ値を計算する式の冗長性を除去するためには，アルゴリズム全体を何回も繰り返すオーバヘッドの大きい手法を用いる必要がある．本論文では，部分冗長性除去と大域的値番号付けを組み合わせた効率的な冗長性除去の手法である PVNRE（Partial Value Number Redundancy Elimination）を提案する．本手法では，式の値番号をデータフロー解析の対象とすることで，見た目は異なるが同じ値を計算する式の冗長性を扱うことができる．また，静的単一代入形式の ϕ 関数を用い，実行パスの合流点で値番号の変換を行うことで，ある実行パスにおいてのみ同じ値を計算する式を除去することができる．さらに，値番号でデータ依存関係を表すことで，全体を何回も繰り返す必要はない．我々は本手法を Java の実行時コンパイラ上に実装し，解析時間と削減することができた命令の数に関して SPECjvm98 と Java Grande Benchmark を用いて従来手法との比較を行った．本手法は従来手法と同等かそれ以上の命令削減能力を示し，本手法と同等の削減能力を持つ従来手法と比べて解析時間は平均で 33%高速であった．

Partial Value Number Redundancy Elimination

Rei Odaira[†] and Kei Hiraki[†]

The runtime optimizing compiler has, in recent times, gained widespread use. A runtime optimizing compiler needs to keep analysis time short, and at the same time, to perform redundancy elimination such as Partial Redundancy Elimination (PRE) and Global Value Numbering (GVN). However, PRE can only deal with redundancy in lexically identical expressions, and GVN can merely remove redundancy in expressions which compute the same value on all execution paths. Therefore, we need to use the method that iterates its whole algorithm several times, in order to eliminate redundancy in lexically different expressions that compute the same value on not all execution paths. In this paper, we propose an efficient method for redundancy elimination called Partial Value Number Redundancy Elimination (PVNRE). Using value numbers in data flow analysises, PVNRE can deal with redundancy in expressions which are lexically different but compute the same value. It can also remove expressions which compute the same value on not all execution paths, by converting value numbers at join nodes through ϕ functions of the Static Single Assignment (SSA) form. Moreover, PVNRE need not iterate the whole algorithm because it uses value numbers as the representation of data dependency. We implemented PVNRE on Java Just-In-Time compiler, and compared its analysis time and the number of eliminated redundancy with those of existing techniques, using SPECjvm98 and Java Grande Benchmark. PVNRE shows equal or better ability to reduce instructions compared with existing algorithms, and on an average is faster in analysis time than an algorithm with the same ability by 33%.

1. はじめに

近年，Java 環境をはじめとして実行時コンパイラが広く用いらてきている．実行時コンパイラは解析時間をなるべく短くする必要があるが，同時に，プログラム全体の実行時間を大きく短縮することができる最適化手法である冗長性除去を行うと効果的である．

プログラムを実行中に，ある式の計算結果が過去に同じ式，もしくは別の式で計算した結果と同一となる場合が多数存在する．この冗長性を静的解析で取り除く最適化手法が冗長性除去であり，具体的なアルゴリズムとして部分冗長性除去（Partial Redundancy

† 東京大学情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

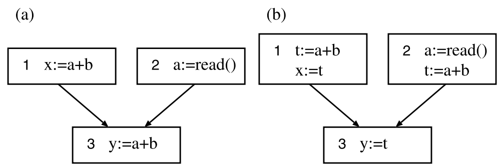


図 1 PRE による最適化の例
Fig. 1 Example of optimization by PRE.

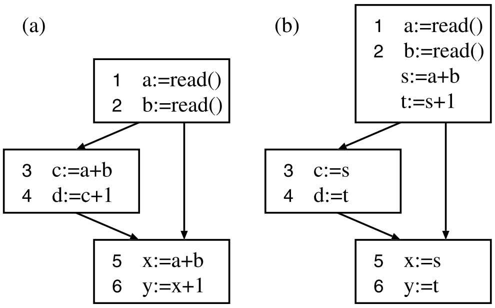


図 2 GVN による最適化の例
Fig. 2 Example of optimization by GVN.

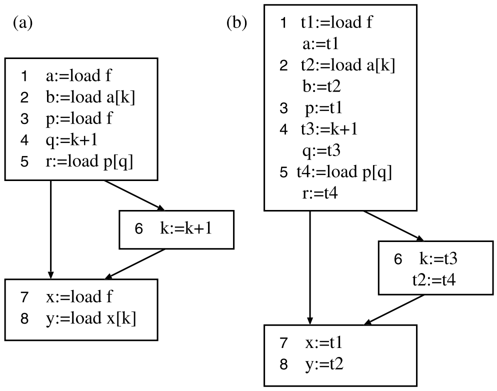


図 3 PRE と GVN で最適化困難な例
Fig. 3 Example that is difficult to be optimized by PRE or GVN.

Elimination; PRE）[2)~4),13)~15)] と大域的値番号付け（Global Value Numbering; GVN）[1),5)~8)] が広く用いられている．

PRE はある式に到達する少なくとも 1 つの実行パス上に存在する冗長性（部分冗長性）を取り除くことができる．PRE では，2 つの式の字面が同一であり，その間の実行パス上にその式のオペランドへの代入が存在しない場合を冗長性として扱う．冗長性を検出するために，利用可能な式（availability; AVAIL）の計算など，データフロー方程式を 3 回解く必要がある．**図 1** (a) では実行パスが左から流れてきた場合にのみ式 3 が冗長となる．そこで図 1 (b) のように一時変数 t を用いて変形することで部分冗長な式を完全冗長にして削除する．

一方，GVN はいかなる実行パスにおいても同一の値を計算する式を，その字面に関係なく検出する．GVN で用いられる値番号付けは一般に，同一の値を計算すると静的に保証できる式に同一の値番号を割り振ることで冗長性を検出する手法である．GVN の中でもボトムアップ型の手法は各式に値番号を割り振るのにハッシュ表を用いる[5)]．まずプログラム全体を静的単一代入（Static Single Assignment; SSA）形式[9)] に変換し，各式のオペランドの値番号とオペレータを鍵としてハッシュ表を引く．ハッシュ表にすでにその鍵が登録されていれば，冗長な式を発見したことになる．鍵が登録されていなければ，新たな値番号を生成して鍵とともに登録する．**図 2** (a) では式 3 と 5，4 と 6 に同じ値番号が割り振られるため，図 2 (b) のように変形することで冗長性を除去できる．特に式 4 と 6 は字面が異なるため，PRE では除去できない冗長性である．一方，図 1 の式 1 と 3 は左から実行パスが来た場合にのみ同じ値を計算するため，GVN では式 3 を除去できない．

以上のように PRE と GVN はそれぞれ一長一短であるが，一般のプログラム中にはいずれの手法でも除去できない冗長性が存在する．例として，Java Grande Forum Benchmark Suite[10)] に含まれる heapsort プログラムの最内ループ内部のコードを簡略化して**図 3** (a) に示す．式 8 は実行パスが左から来た場合は式 2，右から来た場合は式 5 と同一の結果を返し，冗長である．しかし，式の字面が異なるために PRE では除去できず，左右の実行パスで式 8 の値が異なるために GVN でも除去できない．

一部の実行パスにおいてのみ同じ値を計算する，字面の異なる式どうしの冗長性（Lexically Different Partial Redundancy; LDPR）を既存の手法で除去するためには，コピー伝搬と PRE を繰り返し用いる必要がある[16),19)]．すなわち，最初の PRE の結果生じたコピー文を伝搬させることで式の字面を可能な限り同一とし，次の PRE で除去する手法である．しかし，この手法では繰返しによるオーバヘッドが実行時コンパイラの解析時間にとって無視できない問題となる．

本論文では，PRE と GVN を組み合わせることで，アルゴリズム全体を繰り返す必要なしに LDPR を除去することができる手法，Partial Value Number Redundancy Elimination（PVNRE）を提案する．

PVNRE は式の字面ではなく値番号をデータフロー解析の対象とし，実行パスの合流点で SSA 形式の ϕ 関数を利用して値番号の変換を行う．値番号を冗長性の検出に用いることで GVN の最適化能力を含む．さ

らに合流点における値番号の変換によって，一部の実行パスのおいてのみ同じ値を計算する式どうしを値番号の上で関連付ける．この関連付けによって PRE の最適化能力も含むことができ，LDPR を除去することができる．図 3 (a) を PVNRE を用いて最適化した結果を図 3 (b) に示す．PVNRE は式 8 の冗長性を除去している．

PVNRE は各式に値番号を割り振った後で PRE に類似したデータフロー方程式を 4 つ解き，必要な箇所に式を挿入して冗長性を除去する．PVNRE は値番号を割り振る際に番号の大小関係でデータ依存関係を表現する．この大小関係を用いることでデータフロー方程式を解く最中にデータ依存関係を取り扱うことができ，コピー伝搬と PRE の繰返しによるオーバヘッドを排除することが可能となる．

本論文では以下の 2 章で PVNRE と PRE，GVN の動作の比較を詳しく行う．3 章では入力プログラムに関する仮定とその表記法を示す．4 章では PVNRE の定式化のために必要な，値番号の透過性と合流可能性を定義し，5 章で PVNRE のデータフロー方程式と式の挿入点を示す．6 章で実験結果を述べ，7 章で関連研究との比較を行い，8 章でまとめる．なお，PVNRE の実装の特徴的な部分に関する疑似コードを付録 A.2 で示す．

2. PVNRE と PRE，GVN の動作例

PRE が字面の異なる式の冗長性を扱えないのは，利用可能な式（AVAIL）の計算などのデータフロー解析で伝搬される情報が式の字面を基準としているためである．**図 4** (a) では a への代入で式 a+b の AVAIL 情報が無効化（KILL）される．

一方，PVNRE では式の字面ではなく式の値番号をデータフロー情報として伝搬させることで，字面は異なるが同じ値を計算する式の冗長性を検出できる．図 4 (b) では式 1 の値番号 *1* の AVAIL 情報は無効化されないため，同じ値番号を持つ式 4 を削除することができる．

GVN が一部の実行パスにおいてのみ同じ値を計算する式の冗長性を扱えないのは，SSA 形式において ϕ 関数が挿入されるためである．ϕ 関数はある変数の 2 つ以上の定義を合流点で結合させる仮装関数である．**図 5** で式 6 は式 2，4 に対して冗長な計算である．しかし，式 6 のオペランドは ϕ 関数による定義式 5 を参照するため，式 2，4，6 にそれぞれ別の値番号 *2*，*4*，*6* が割り振られて冗長性を検出できない．

PVNRE では ϕ 関数を用いて合流点で値番号の変換を行う．図 5 で値番号 *2*，*4* の右オペランドは b で等しく，左オペランドの値番号 *1*，*3* は式 5 の ϕ 関数によって *5* に変換される．ここで，右オペランドは b，左オペランドは値番号 *5* を持つ値番号を *6* と定義する．結果として値番号 *2*，*4* は合流して，いずれも値番号 *6* に変換されて以降に伝搬し，同じ値番号 *6* を持つ式 6 を削除することができる．

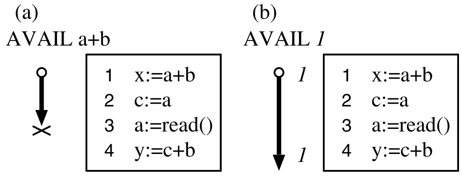


図 4 PVNRE による最適化の例（1）
Fig. 4 Example of optimization by PVNRE (1).

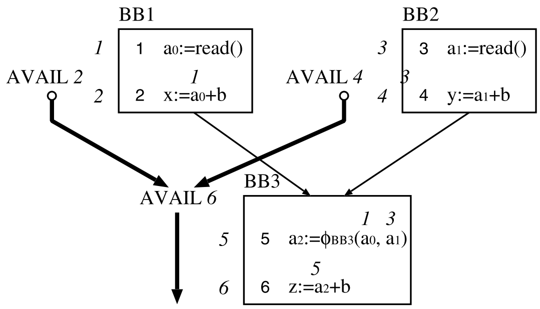


図 5 PVNRE による最適化の例 (2)
Fig. 5 Example of optimization by PVNRE (2).

3. 入力プログラム

以降では PVNRE のアルゴリズムの定式化を行うが，本章ではそのための前提事項を述べる．

3.1 入力プログラムの形式

入力プログラムは SSA 形式に変換されていることを前提とする．

制御フローグラフは可約であることを仮定する．不可約なグラフは基本ブロックをコピーすることで可約なグラフに変形することができる．

また，2 つ以上の後続ブロックを持つ基本ブロックから 2 つ以上の先行ブロックを持つ基本ブロックへのエッジ（クリティカルエッジ）は取り除かれているものとする．これはクリティカルエッジに式を挿入したい場合に対応するためである．

表記の簡単化のために，すべての基本ブロックの先行ブロックはたかだか 2 つであると仮定する．したがって ϕ 関数の引数も 2 つとなる．任意のプログラムの合流ブロックは分割することにより先行ブロックをたかだか 2 つとすることができる．

3.2 プログラムの表現

3.2.1 制御フロー

プログラム全体は有向グラフ

$G = \langle Nodes, Edges, start, end\rangle$

で表現される．右辺はそれぞれ命令，制御エッジ，プログラムの唯一の入口ノード，唯一の出口ノードを表す．

命令 m から n への制御エッジを $m \rightarrow n$ で表す．また，基本ブロックの集合を $BBNodes$ とし，基本ブロック M から N への制御エッジ $M \rightarrow N$ は M の末尾命令から N の先頭命令への制御エッジと同一視する．基本ブロック N の先行ブロック集合を $Pred(N)$，後続ブロック集合を $Succ(N)$ と表記する．命令 n を含む基本ブロックを $BB(n)$，逆に基本ブロック N の先頭命令を $Head(N)$ と表す．

命令 m から n へのすべての実行パスの集合を $P[m,n]$ で表す．ある実行パス $p \in P[m,n]$ の長さを $\lambda(p)$，i 番目 $(1 \leq i \leq \lambda(p))$ の命令を p_i で表す．p_i に対応する命令ノードを $Node(p_i)$ とし，$p_i \rightarrow p_j \overset{\text{def}}{\Leftrightarrow} Node(p_i) \rightarrow Node(p_j)$ とする．また，i 番目から j 番目の命令までの部分パスを $p[i,j]$ と表記する．p に沿って実行した場合に i 番目の命令が計算する値を $Value(p,i)$ とおく．

3.2.2 命令とデータフロー

命令のオペレータの集合を

$Operators = Normal \oplus Copy \oplus Phi \oplus Fixed$

で表す．右辺はそれぞれ，算術命令一般，コピー命令，ϕ 関数，その他の命令，である．その他の命令には関数の仮引数，関数呼び出し，メモリ操作命令などが含まれる．各命令からそのオペレータへの関数は $Op : Nodes \rightarrow Operators$ であり，$n \in Nodes$ に対して $n.Op$ と表記する．基本ブロック $N \in BBNodes$ に存在する ϕ 関数のオペレータは ϕ_N と表す．

Phi と同様に $Normal$ も一般性を失わずに 2 引数であると仮定する．$Copy$ に関しては左引数のみが存在する．各命令から左右の引数が参照する命令への関数は $Lt, Rt : Nodes \rightarrow Nodes$ であり，$n.Lt, n.Rt$ と表記する．ただし，一般に左右いずれの引数でも成り立つ場合は $n.X$，もう一方の引数は $n.Y$ と表記する．

また，実行パス上の命令に関して左右引数の参照先を

$$
\begin{aligned}
&p_i.X = p_j \overset{\text{def}}{\Leftrightarrow} \\
&\quad 1 \leq j < i \wedge Node(p_i).X = Node(p_j) \\
&\quad \wedge \forall j < k \leq i \ .\ Node(p_k) \neq Node(p_j)
\end{aligned}
$$

と定義する．

4. 値番号と冗長性

本章では，PVNRE の基盤となる値番号の透過性と合流可能性を定義する．次に，PVNRE が正しく動作するための値番号の正当性を定義し，正当性が満たされるならば同じ値番号を持つ 2 つの命令は実行時に冗長であることを示す．

以下では，検出する冗長性の対象を算術命令 $Normal$ に限定する．ただし，メモリに関する依存関係を明示的に表現することで，ロード命令に関する冗長性も同じ枠組みで検出することが可能である．

4.1 値番号

値番号付けは $Nodes$ から自然数 $\mathcal{N}$ の部分集合である値番号 $Nums$ への関数の集合として定義できる．

$Numberings = \{f \mid f : Nodes \rightarrow Nums \subset \mathcal{N}\}$

以降ではある任意の値番号付けを $Num \in Numberings$ とおく．GVN など，従来の値番号付けを用いたアルゴリズムでは Num の値域が $Nums$ と一致していたが，PVNRE では対応する命令のない値番号も許可する．これにより，ϕ 関数による変換で新たな値番号を生成することができる．実行パス上の命令 p_i についても，$Num(p_i) \overset{\text{def}}{=} Num(Node(p_i))$ と定義する．

また，$Nodes$ についてと同様に，$Nums$ について Op，Lt，Rt を

$Op : Nums \rightarrow Operators$

$Lt : Nums \rightarrow Nums$

$Rt : Nums \rightarrow Nums$

である任意の関数として定義する．ただし，PVNRE を正しく動作させるために，実際には我々は後述の 4.4 節で示す正当性条件を満たすよう設定する．

4.2 値番号の透過性

透過性（transparency）とは，データフロー方程式を解く際にある基本ブロック，もしくは制御エッジを越えて情報が有効か否かを表す条件である．PRE では基本ブロックにある式の引数への代入がある場合，その式の伝搬情報を無効化する．

一方，PVNRE では式の字面ではなく値番号を用いるために上記の意味での透過性条件は必要ないが，制御フローのバックエッジに沿って一部の値番号が伝播するのを防がなければならない．**図 6** では式 6 の値番号の AVAIL 情報をバックエッジに沿って伝播させた場合，その情報はループを 1 周して式 6 に到達するため，式 6 は部分冗長性を持つことになる．しかし，実際には式 6 はループ誘導変数であるためループの各イテレーションごとに値が異なり，部分冗長ではない．

そこで，PVNRE では値番号と制御エッジ，特に

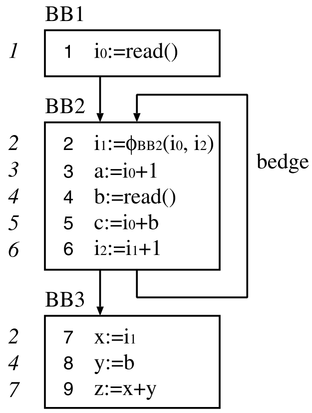


図 6 バックエッジと透過性の例
Fig. 6 Example of backedge and transparency.

バックエッジに関する透過性を定義する．あるループのイテレーションごとに異なる値を計算する式の値番号はそのループのバックエッジを伝搬させない．ある式がループのイテレーションごとに異なる値を計算する原因は，式の引数が *Fixed* 命令もしくは ϕ 関数に依存するためである．したがって，以下ではまず *Fixed* 命令と ϕ 関数について *DefBBNodes* を定義する．*DefBBNodes* はある値番号を生成する命令を含む基本ブロック集合である．

4.2.1 *Fixed* と *Phi* に関する *DefBBNodes*

定義 4.1 *Fixed* に関する *DefBBNodes*
$\forall\alpha \in \mathit{Nums}\ .\ \alpha.\mathit{Op} \in \mathit{Fixed}$ について，

$$\mathit{DefNodes}(\alpha) \stackrel{\mathrm{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op}\}$$

$$\mathit{DefBBNodes}(\alpha) \stackrel{\mathrm{def}}{=} \bigcup_{\forall n} \mathit{BB}(n) \quad \text{s.t. } n \in \mathit{DefNodes}(\alpha) \quad \diamondsuit$$

オペレータが *Fixed* である値番号 α の *DefNodes* に含まれる命令は，その値番号とオペレータが α に一致しなければならない．*DefBBNodes* は *DefNodes* を含む基本ブロックの集合である．

定義 4.2 *Phi* に関する *DefBBNodes*
$\forall\alpha \in \mathit{Nums}\ .\ \alpha.\mathit{Op} = \phi_N \in \mathit{Phi}$ について，

$$\mathit{DefBBNodes}(\alpha) \stackrel{\mathrm{def}}{=} \{N\} \quad \diamondsuit$$

図 6 の例では

$\mathit{DefBBNodes}(\mathit{2}) = \{\mathrm{BB2}\}$
$\mathit{DefBBNodes}(\mathit{4}) = \{\mathrm{BB2}\}$

となる．式 7 と 8 はコピー文なので，それぞれの値番号を生成する命令とは考えない．

4.2.2 バックエッジ

制御フローのバックエッジは支配関係 dom を用いて $\mathit{Bedges} \stackrel{\mathrm{def}}{=} \{m{\rightarrow}n \mid n \text{ dom } m\}$ と定義できる．ここでさらに，各命令と基本ブロックに関するバックエッジを以下のように定義する．

定義 4.3 命令/基本ブロックに関するバックエッジ
$\forall u \in \mathit{Nodes}$，$\forall N \in \mathit{BBNodes}$ について，

$$\mathit{Bedges}(u) \stackrel{\mathrm{def}}{=} \{e = m \rightarrow n \mid e \in \mathit{Bedges} \wedge \exists p \in P[u,m]\ \forall 1 \leq i \leq \lambda(p)\ .\ \mathit{Node}(p_i) \neq n\}$$

$$\mathit{Bedges}(N) \stackrel{\mathrm{def}}{=} \mathit{Bedges}(\mathit{Head}(N)) \quad \diamondsuit$$

すなわち，ある命令もしくは基本ブロックを囲む各ループのバックエッジすべてを表す．

図 6 では式 2〜6，BB2 の *Bedges* が { bedge } であり，それ以外の式の *Bedges* は空集合となる．

4.2.3 透 過 性

PVNRE における透過性を以下のように定義する．

定義 4.4 *Transp*
$\forall\alpha \in \mathit{Nums}$，$\forall e \in \mathit{Edges}$ について，

(1) if $\alpha.\mathit{Op} \in \mathit{Fixed} \vee \alpha.\mathit{Op} \in \mathit{Phi}$

$$\mathit{Transp}(\alpha, e) \stackrel{\mathrm{def}}{\Leftrightarrow} e \notin \bigcup_{\forall N} \mathit{Bedges}(N) \quad \text{s.t. } N \in \mathit{DefBBNodes}(\alpha)$$

(2) otherwise

$$\mathit{Transp}(\alpha, e) \stackrel{\mathrm{def}}{\Leftrightarrow} \mathit{Transp}(\alpha.\mathit{Lt}, e) \wedge \mathit{Transp}(\alpha.\mathit{Rt}, e) \quad \diamondsuit$$

表 1 bedge に関する *Transp*
Table 1 *Transp* for bedge.

値番号	*Transp*	説明
1	true	*DefBBNodes* がループの外にある．
2	false	*DefBBNodes* がループの中にある．
3	true	ループの中にあるが引数の依存先はループの外にあるのでループ不変量である．
4	false	*DefBBNodes* がループの中にある．
5	false	左引数はループ外であるが，右引数はループ内変数の値番号である．
6	false	ループ内変数の値番号に依存する．
7	false	ループ外にあるが，両引数はループ内変数の値番号である．

Fixed に関しては，メモリからロードした値や関数呼び出しの返り値は実行するたびに異なる値となることを仮定して，それを囲むバックエッジを伝搬させない．ϕ 関数に関しては，左右いずれの定義を参照するか実行するたびに異なることを仮定して，同様にそれを囲むバックエッジを伝搬させない．*Normal* については，左右いずれかの引数があるループのバックエッジを越えて有効でないならば，その値自身も有効でないとする．

図 6 で bedge に関する各値番号の $Transp$ の真偽値は**表 1** のようになる．

最後に，実行パス中のある区間内の全制御エッジについて $Transp$ が成り立つとき，$Transp^{\forall}$ と表記する．

定義 4.5 $Transp^{\forall}$

$$\forall\alpha \in Nums, \forall p \in P[start, end], \forall 1 \le i < j \le \lambda(p)\ .$$
$$Transp^{\forall}(\alpha, p[i,j]) \overset{\text{def}}{=} \bigwedge_{\forall i \le k < j} Transp(\alpha, p_k \rightarrow p_{k+1})\ \ \diamondsuit$$

4.3 値番号の合流

PVNRE は図 5 で示したように制御フローの合流点で一定の条件を満たす値番号どうしを合流させ，新たな値番号として以降に伝播させる．値番号の合流により，一部の実行パスにおいてのみ同じ値を計算する式どうしの冗長性を検出することができる．合流可能性の条件を以下のように定義する．

定義 4.6 $Join$，$Join'$

$\forall M, N \in BBNodes\ .\ M \in Pred(N)$,

$\forall \alpha_0, \alpha_1 \in Nums$ について，

$Join(N, \alpha_0, \alpha_1) \overset{\text{def}}{\Leftrightarrow}$ (1)〜(3) のいずれかが成り立つ

$Join'(N, \alpha_0, \alpha_1) \overset{\text{def}}{\Leftrightarrow}$ (2)，(3) のいずれかが成り立つ

(1) $\exists n \in Nodes\ .\ n.Op = \phi_N$
$\wedge Num(n).Op = \phi_N$
$\wedge \alpha_0 = Num(n.Lt)$
$\wedge \alpha_1 = Num(n.Rt)$

(2) $Join(N, \alpha_0.X, \alpha_1.X)$
$\wedge \alpha_0.Y = \alpha_1.Y \wedge Transp(\alpha_0.Y, M \rightarrow N)$
$\wedge \alpha_0.Op = \alpha_1.Op \in Normal$

(3) $Join(N, \alpha_0.Lt, \alpha_1.Lt)$
$\wedge Join(N, \alpha_0.Rt, \alpha_1.Rt)$
$\wedge \alpha_0.Op = \alpha_1.Op \in Normal$ $\diamondsuit$

(1) は ϕ 関数が合流性条件の基点となることを示しており，実際の合流性は $Join'$ で表される．(2) は $Normal$ である 2 つの値番号の一方のオペランドどうしが合流し，他方のオペランドどうしが等しい場合である．(3) は左右のオペランドどうしがいずれも合流する場合である．

合流により生成される新たな値番号を定義する．

定義 4.7 $Jtarget$，$Jtarget'$

$Jtarget(N, \alpha_0, \alpha_1) \overset{\text{def}}{=} Join$ の (1)〜(3) に対応してそれぞれ以下のとおり

$Jtarget'(N, \alpha_0, \alpha_1) \overset{\text{def}}{=} Join'$ の (2)，(3) に対応してそれぞれ以下のとおり

(1) $\{\alpha \in Nums \mid \alpha = Num(n)$
$\wedge n$ は $Join$の定義 4.6 (1) を満たす $\}$

(2) $\{\alpha_2 \in Nums \mid \alpha_2.Op\ = \alpha_0.Op$
$\wedge \alpha_2.X \in Jtarget(N, \alpha_0.X, \alpha_1.X)$
$\wedge \alpha_2.Y = \alpha_0.Y\}$

(3) $\{\alpha_2 \in Nums \mid \alpha_2.Op\ = \alpha_0.Op$
$\wedge \alpha_2.Lt \in Jtarget(N, \alpha_0.Lt, \alpha_1.Lt)$
$\wedge \alpha_2.Rt \in Jtarget(N, \alpha_0.Rt, \alpha_1.Rt)\}$ $\diamondsuit$

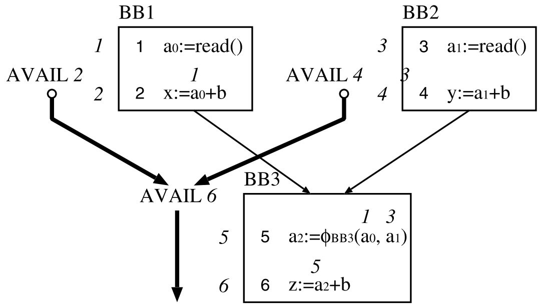


図 5 （再掲）PVNRE による最適化の例（2）
Fig. 5 (reshown) Example of optimization by PVNRE (2).

(1) は ϕ 関数の値番号そのもの，(2)，(3) は ϕ 関数の値番号を基点とした再帰的データ依存関係で生成される値番号である．

図 5 では値番号 1，3 は定義 4.6 (1)，値番号 2，4 は (2) を満たすため，

$Join(\mathrm{BB3}, 1, 3)$,
$Join'(\mathrm{BB3}, 2, 4)$

の関係にあり，それぞれ

$Jtarget(\mathrm{BB3}, 1, 3) = \{5\}$,
$Jtarget'(\mathrm{BB3}, 2, 4) = \{6\}$

となる．

また，左右いずれかの方向から伝播してくる値番号が変換される先として，$Jlink$，$Jlink'$ を定義する．

定義 4.8 $Jlink$，$Jlink'$

$\forall M, N \in BBNodes\ .\ M \in Pred(N)$,

$\forall \alpha \in Nums$ について

$Jlink(N, \alpha, M \rightarrow N) \overset{\text{def}}{=}$

(1) if M が N の左側先行ブロック
$$\bigcup_{\forall\beta} Jtarget(N, \alpha, \beta) \text{ s.t. } Join(N, \alpha, \beta)$$

(2) if M が N の右側先行ブロック
$$\bigcup_{\forall\beta} Jtarget(N, \beta, \alpha) \text{ s.t. } Join(N, \beta, \alpha)$$

$Jlink'$ は $Jtarget'$ に対して同様に定義される． $\diamondsuit$

図 5 では

$Jlink(\mathrm{BB3}, 1, \mathrm{BB1}{\rightarrow}\mathrm{BB3}) = \{5\}$,
$Jlink(\mathrm{BB3}, 3, \mathrm{BB2}{\rightarrow}\mathrm{BB3}) = \{5\}$,
$Jlink'(\mathrm{BB3}, 2, \mathrm{BB1}{\rightarrow}\mathrm{BB3}) = \{6\}$,
$Jlink'(\mathrm{BB3}, 4, \mathrm{BB2}{\rightarrow}\mathrm{BB3}) = \{6\}$

となる．

(1) if $\alpha.Op \in \mathit{Normal}$

$$\mathit{DefNormalNodes}(\alpha) \overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op}\}$$

$$\mathit{DefPhiNodes}(\alpha) \overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} \in \mathit{Phi} \wedge \mathit{Join}'(\mathit{BB}(n), \mathit{Num}(n.\mathit{Lt}), \mathit{Num}(n.\mathit{Rt})) \wedge \mathit{Num}(n.X).\mathit{Op} = \alpha.\mathit{Op}\}$$

$$\mathit{DefNodes}(\alpha) \overset{\text{def}}{=} \mathit{DefNormalNodes}(\alpha) \cup \mathit{DefPhiNodes}(\alpha)$$

(2) if $\alpha.Op \in \mathit{Phi}$

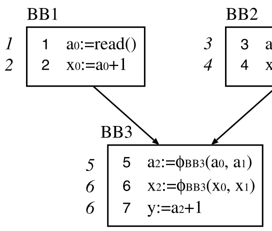
$$\mathit{DefNodes}(\alpha) \overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op} \wedge \mathit{Num}(n.X) \neq \alpha \wedge \neg \mathit{Join}'(\mathit{BB}(n), \mathit{Num}(n.\mathit{Lt}), \mathit{Num}(n.\mathit{Rt}))\}$$

(3) if $\alpha.Op \in \mathit{Copy}$

$$\mathit{DefNodes}(\alpha) \overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op} \wedge \mathit{Num}(n.\mathit{Lt}) \neq \alpha\}$$

図 8　定義 4.9 *DefNodes*

Fig. 8　Definition 4.9 *DefNodes*.



	Op	*Lt*	*Rt*	*DefNodes*
1	read()	--	--	1
2	+	*1*	const1	2
3	read()	--	--	3
4	+	*2*	const1	4
5	ϕ_{BB3}	*1*	*3*	5
6	+	*5*	const1	6,7

図 7　正当性条件を満たした値番号付けの例

Fig. 7　Example of correct value numbering.

4.4 正　当　性

ここまでの定義 4.1〜4.8 では，値番号付け *Num* と値番号に関する *Op*，*Lt*，*Rt* 関数は，4.1 節で示した任意の関数であるとしてきた．しかし，PVNRE による最適化が意味的に正しいものであるために，我々は以下で定義する正当性条件を満たすようにそれぞれの関数を設定する．正当性条件を満たした値番号付けと *Op*，*Lt*，*Rt* の例を**図 7** に示す．以下の本節ではこの例を用いて説明する．

4.4.1 値番号に関する *Op*，*Lt*，*Rt* の正当性

まず，*Normal*，*Phi*，*Copy* について *DefNodes* を定義する．*Fixed* に関してはすでに 4.2.1 項で定義済みである．*DefNodes* は一般に，プログラム中である値番号が生成される原因となる命令の集合を表す．

定義 4.9　*DefNodes*

図 8 を参照．　◇

DefNodes を用いて，各関数の正当性を定義する．

定義 4.10　値番号に関する *Op* 関数の正当性

値番号に関する *Op* 関数は正当である $\overset{\text{def}}{\Leftrightarrow}$

$\forall \alpha \in \mathit{Nums}$ について，

$\exists n \in \mathit{Nodes}\ .\ \mathit{Num}(n) = \alpha$

$\Rightarrow \mathit{DefNodes}(\alpha) \neq \emptyset$　◇

たとえば，ある値番号 α について

$\alpha.\mathit{Op} = \mathrm{read}() \in \mathit{Fixed}$

と定義したならば，そのプログラム中で値番号 α が割り振られた命令の中に少なくとも 1 つはオペレータとして read() を持つ命令がなければならない．図 7 では値番号 *1*，*3* がそれにあたる．

定義 4.11　値番号に関する *Lt*，*Rt* 関数の正当性

値番号に関する *Lt* 関数は正当である $\overset{\text{def}}{\Leftrightarrow}$

$\forall \alpha, \beta \in \mathit{Nums}$ について，

$\alpha.\mathit{Lt} = \beta \wedge \exists n \in \mathit{Nodes}\ .\ \mathit{Num}(n) = \alpha \Rightarrow$

(1) if $\alpha.\mathit{Op} \in \mathit{Normal}$

$\exists n \in \mathit{DefNormalNodes}(\alpha)\ .\ \mathit{Num}(n.\mathit{Lt}) = \beta$

$\vee \exists n \in \mathit{DefPhiNodes}(\alpha)\ .$

$(\mathit{Jtarget}'(\mathit{BB}(n), \mathit{Num}(n.\mathit{Lt}).\mathit{Lt}, \mathit{Num}(n.\mathit{Rt}).\mathit{Lt}) \ni \beta$

$\vee \mathit{Num}(n.\mathit{Lt}).\mathit{Lt} = \mathit{Num}(n.\mathit{Rt}).\mathit{Lt} = \beta)$

(2) if $\alpha.\mathit{Op} \in \mathit{Phi}, \mathit{Copy}$

$\exists n \in \mathit{DefNodes}(\alpha)\ .\ \mathit{Num}(n.\mathit{Lt}) = \beta$

Rt 関数の正当性も同様に定義される．　◇

図 7 では，値番号 *2* について $\mathit{2}.\mathit{Lt} = \mathit{1}$ であるが，*DefNormalNodes*(*2*) (= *DefNodes*(*2*)) の要素である式 2 の左引数は式 1 を参照しており，その値番号は *1* であるため，正当性を満たしている．

4.4.2 値番号付けの正当性

値番号付けの正当性条件は，ある 2 つの命令が同じ値番号を持つ場合に成り立たなければならない条件を定めたものである．

定義 4.12　値番号付けの正当性

値番号付け *Num* は正当である $\overset{\text{def}}{\Leftrightarrow}$

$\forall m, n \in \mathit{Nodes}\ .\ m \neq n \wedge \mathit{Num}(m) = \mathit{Num}(n)$

$\Rightarrow$ (1)〜(5) のいずれかが成り立つ

(1) $m.\mathit{Op} \in \mathit{Copy} \wedge \mathit{Num}(m.\mathit{Lt}) = \mathit{Num}(n)$

(2) $m.\mathit{Op} \in \mathit{Phi}$

$\wedge Num(m.Lt) = Num(m.Rt) = Num(n)$

(3) $m.Op = n.Op \in Normal \cup Phi$

$\wedge Num(m.X) = Num(n.X)$

(4) $m.Op \in Phi \wedge n.Op \in Normal$

$\wedge Join'(BB(m), Num(m.Lt), Num(m.Rt))$

$\wedge Jtarget'(BB(m), Num(m.Lt),$

$Num(m.Rt)) \ni Num(n)$

(5) (1)〜(4) で m と n を入れ替えたもの ◇

以下で各条件について説明する．

(1) コピー文はその引数の参照先命令と同じ値番号を持つことを許す．なぜならばコピー文はつねに引数と同じ値を返すからである．

(2) 左右の引数が同じ値番号を持つ ϕ 関数はその引数と同じ値番号を持つことを許す．これは，ある変数の 2 つの定義が同じ値であるならば，それらが合流したとしてもつねに同じ値となるからである．

(3) 算術命令と ϕ 関数については，オペレータと左右引数の値番号が同じならば，同じ値番号を持つことを許す．この条件は，GVN など値番号付けを用いた従来のアルゴリズムで，ハッシュ表を引く操作に相当する．

(4) GVN とは異なり PVNRE が ϕ 関数を介した冗長性を検出できるのは，条件 (4) が存在するからである．条件 (4) に関しては次の 4.4.3 項で例を用いて説明する．

(1)〜(5) の条件より，2 つの異なる $Fixed$ 命令が同じ値番号を持つと，いずれの条件も満たすことができず正当性に反する．ただし，メモリ依存関係の解析や関数間解析により条件を緩和することはできる．

また，いずれの条件も逆が成り立つことは要求されない．この場合アルゴリズムの正当性は保たれるが，検出することができる冗長性の数は減る．

4.4.3 例

図 7 で式 6 のオペレータは ϕ であるが，その値番号 6 の Op は+となっている．これは定義 4.10 の値番号に関する Op 関数の正当性を満たす．なぜならば $Join(\mathrm{BB3}, 1, 3)$ より $Join'(\mathrm{BB3}, 2, 4)$ であり，

$$DefNodes(6) \supset DefPhiNodes(6) = \{5\}$$

となるからである．

同様に，$Jtarget(\mathrm{BB3}, 1, 3) \ni 5$ であるから，定義 4.11 (1) より，値番号に関する Lt，Rt 関数の正当性を満たす．

また，式 6 と 7 には同じ値番号 6 が割り振られているが，この値番号付けは定義 4.12 (4) の正当性条件を満たしている．なぜならば $Jtarget'(\mathrm{BB3}, 2, 4) \ni 6$ が成り立つからである．式 6 と 7 に同じ値番号を割り振ることが可能であることから，我々は 2 つの式の間の冗長性を検出したことになる．すなわち，式 7 を $\mathrm{y} = \mathrm{x}_2$ と書き換えて，加算命令を 1 つ削除することができる．

4.5 値番号の冗長性

次の定理は PVNRE による最適化の正当性の基礎となる定理である．

定理 4.13 値番号に関する冗長性定理

値番号付け Num，および値番号に関する Op，Lt，Rt が正当性を満たすならば，以下の命題が成り立つ：

$\forall p \in P[start, end], \forall 1 \le i < j \le \lambda(p)$ について，

$$Num(p_i) = Num(p_j)$$
$$\wedge Transp^{\forall}(Num(p_i), p[i,j])$$
$$\Rightarrow Value(p,i) = Value(p,j)$$

証明の概略に関しては付録 A.1.1 を参照.

すなわち，値番号が等しい 2 つの命令間の $Transp^{\forall}$ な実行パスに沿ってプログラムが実行された場合，その 2 つの命令が計算する値は等しくなる，したがって冗長であることを意味する．

冗長性定理に基づき，PVNRE はデータフロー方程式を解くことで $Transp^{\forall}$ な実行パスに沿った命令間の値番号の冗長性を検出する．検出した冗長性に従って必要な箇所に正当性条件を保つように命令を挿入し，命令を完全冗長にしてから除去する．次の章ではアルゴリズムを具体的に定式化する．

5. PVNRE のアルゴリズム

本章では PVNRE のデータフロー方程式を定式化し，必要な命令の挿入点を定義する．

5.1 値番号付け

ここまでは値番号に関して番号が等しいか異なるかのみを問題にしてきた．しかし，値番号の大小関係からデータ依存関係を限定できると便利であるため，我々は以下の条件を満たすように値番号を割り振る．

定義 5.1 値番号の大小関係に関する条件

$\forall \alpha \in Nums\ .\ \alpha.Op \in Normal \Rightarrow \alpha.X < \alpha$ ◇

つまり，算術命令に割り振られた値番号はその左右引数の値番号より大きくなければならない．

5.2 データフロー方程式

我々は命令間の冗長性を検出するために，値番号に関するデータフロー方程式を解く．ここで用いるのは，文献 3) で提案されている PRE のアルゴリズムを変形したものである．文献 3) では式の字面の情報を伝搬させるが，PVNRE では値番号を伝搬させ，さらに 4.3 節で述べた値番号の合流を考慮する．また，伝搬

$$AVAIL_{in}^{all}(\alpha,n) = \bigwedge_{\forall m \in Pred(n)} \Big((AVAIL_{out}^{all}(\alpha,m) \wedge Transp(\alpha, m \to n)) \quad (1)$$

$$\vee (AVAIL_{out}^{all}(\beta,m) \text{ s.t. } \alpha \in Jlink'(BB(n),\beta,m \to n)) \Big) \quad (2)$$

$$AVAIL_{out}^{all}(\alpha,n) = AVAIL_{in}^{all}(\alpha,n) \vee (n \in DefNodes(\alpha)) \quad (3)$$

$$AVAIL_{in}^{some}(\alpha,n) = \bigvee_{\forall m \in Pred(n)} \Big((AVAIL_{out}^{some}(\alpha,m) \wedge Transp(\alpha, m \to n)) \quad (4)$$

$$\vee (AVAIL_{out}^{some}(\beta,m) \text{ s.t. } \alpha \in Jlink'(BB(n),\beta,m \to n)) \Big) \quad (5)$$

$$AVAIL_{out}^{some}(\alpha,n) = AVAIL_{in}^{some}(\alpha,n) \vee (n \in DefNodes(\alpha)) \quad (6)$$

図 9 $AVAIL^{all}$，$AVAIL^{some}$ のデータフロー方程式
Fig. 9 Equation system of $AVAIL^{all}$ and $AVAIL^{some}$.

$$ANTIC_{out}^{all}(\alpha,m) = \bigwedge_{\forall n \in Succ(m)} \Big((ANTIC_{in}^{all}(\alpha,n) \wedge Transp(\alpha, m \to n)) \quad (7)$$

$$\vee \bigvee_{\forall \beta} (ANTIC_{in}^{all}(\beta,n) \text{ s.t. } \beta \in Jlink'(BB(n),\alpha,m \to n)) \Big) \quad (8)$$

$$ANTIC_{in}^{all}(\alpha,n) = ANTIC_{out}^{all}(\alpha,n) \vee (n \in DefNodes(\alpha)) \quad (9)$$

$$AVAIL_{in}^{wsome}(\alpha,n) = \bigvee_{\forall m \in Pred(n)} \Big((AVAIL_{out}^{wsome}(\alpha,m) \wedge Transp(\alpha, m \to n)) \quad (10)$$

$$\vee (AVAIL_{out}^{wsome}(\beta,m) \text{ s.t. } \alpha \in Jlink'(BB(n),\beta,m \to n)) \Big) \quad (11)$$

$$KILL(\alpha,n) = AVAIL_{in}^{all}(\alpha,n) \vee ANTIC_{in}^{all}(\alpha,n) \quad (12)$$

$$AVAIL_{out}^{wsome}(\alpha,n) = (AVAIL_{in}^{wsome}(\alpha,n) \wedge KILL(\alpha,n)) \vee (n \in DefNodes(\alpha)) \quad (13)$$

図 10 $ANTIC^{all}$，$AVAIL^{wsome}$ のデータフロー方程式
Fig. 10 Equation system of $ANTIC^{all}$ and $AVAIL^{wsome}$.

してきた情報が無効化されるのは式の引数への代入がある場所ではなく，*Transp* が false であるエッジにおいてである．

図 9 に $AVAIL^{all}$ と $AVAIL^{some}$ のデータフロー方程式を示す．文献 3) では $AVAIL^{all}$ のみ解いており，PVNRE でもアルゴリズムを定式化する際には $AVAIL^{all}$ のみが必要とされるが，我々は実装の都合上 $AVAIL^{some}$ も同時に解く．詳細は付録 A.2.3 で述べる．

$AVAIL_{in}^{all}$ の (1) は α がそのまま伝搬する場合，(2) は合流ブロックで β が α に変換されて伝搬する場合である．後者の場合は *Transp* の影響を受けないが，これはバックエッジを伝搬することができない値番号に関してはその値番号を引数にとる ϕ 関数がループのヘッダに必ず存在し，適切に変換されるからである．

$AVAIL^{some}$ は $AVAIL^{all}$ の AND 条件を OR 条件に置換したものである．

図 10 (7)–(9) に $ANTIC^{all}$ のデータフロー方程式を示す．$ANTIC_{out}^{all}$ の (7) は α がそのまま伝搬する場合である．(8) は制御フローを逆にたどると β が α に変換される場合を表す．ある α に対して $\alpha \in Jlink'(BB(n),\beta,m \to n)$ を満たす β は 1 つしか存在しないが，ある α に対して $\beta \in Jlink'(BB(n),\alpha,m \to n)$ を満たす β は複数個存在する可能性があるので，式 (2) と (8) は完全に対称的ではない．

最後に，図 10 (10)–(13) に $AVAIL^{wsome}$ のデータフロー方程式を示す．$AVAIL^{wsome}$ は $AVAIL^{some}$ と類似しているが，元のプログラム中である値の計算が存在しないパス上に新たに式が挿入されるのを防ぐための条件 *KILL* を追加する．

我々は上記のデータフロー方程式を付録 A.2 に示す繰返し（iterative）アルゴリズム[12] を用いて解く．定義 4.6，4.7 によれば *Join*，*Jtarget* による値番号の合流関係は，新たな番号を生成することで再帰的に無限に定義することができる．しかし我々の目的は，合流ブロックに到達した値番号が変換されてさらに先へと伝搬するか否かを調べることにある．したがって実装上は合流ブロックに到達した値番号のみに関して

$$InsertNormal(\alpha, m\rightarrow n) \stackrel{\text{def}}{\Leftrightarrow} (AVAIL_{in}(\alpha, n) = \mathsf{No} \wedge (n \in DefNormalNodes(\alpha))) \quad (14)$$
$$\vee(\neg \exists \beta \,.\, \alpha \in Jlink'(BB(n), \beta, m\rightarrow n)$$
$$\wedge AVAIL_{out}(\alpha, m) = \mathsf{No} \wedge AVAIL_{in}(\alpha, n) = \mathsf{May} \wedge ANTIC^{all}_{in}(\alpha, n)) \quad (15)$$
$$\vee(\exists \gamma \,.\, \gamma \in Jlink'(BB(n), \alpha, m\rightarrow n)$$
$$\wedge AVAIL_{out}(\alpha, m) = \mathsf{No} \wedge AVAIL_{in}(\gamma, n) = \mathsf{May} \wedge ANTIC^{all}_{in}(\gamma, n)) \quad (16)$$

$$InsertPhi(\alpha_2, N, \alpha_0, \alpha_1) \stackrel{\text{def}}{\Leftrightarrow} Join'(N, \alpha_0, \alpha_1) \wedge \alpha_2 \in Jtarget'(N, \alpha_0, \alpha_1) \quad (17)$$
$$\wedge(AVAIL_{in}(\alpha_2, Head(N)) = \mathsf{Must}$$
$$\vee(AVAIL_{in}(\alpha_2, Head(N)) = \mathsf{May} \wedge ANTIC^{all}_{in}(\alpha_2, Head(N)))) \quad (18)$$
$$\wedge \neg \exists n \in Nodes \,.$$
$$(n.Op = \phi_N \wedge Num(n) = \alpha_2$$
$$\wedge Num(n.Lt) = \alpha_0 \wedge Num(n.Rt) = \alpha_1) \quad (19)$$

図 11 算術命令の挿入点：*InsertNormal* と ϕ 関数の挿入点：*InsertPhi*
Fig. 11 Insertion points of arithmetic instructions: *InsertNormal* and ϕ functions: *InsertPhi*.

動的に合流可能性を判定して *Jlink* 情報を構築すれば効率的であり，そのために繰返しアルゴリズムの最中にも値番号付けを行う．

以下では，繰返しアルゴリズムの maximum fixed point（MFP）解とデータフロー方程式の meet-over-all-path（MOP）解を区別する場合に頭に *mfp*, *mop* を付加する．

定理 5.2 MFP 解と MOP 解
$AVAIL^{all}$，$AVAIL^{some}$，$AVAIL^{some}$ の繰返しアルゴリズムを用いた MFP 解はそれぞれの MOP 解に一致する．$ANTIC^{all}$ の MFP 解は MOP 解とは一致しないが，安全な解である．ただし，$AVAIL^{wsome}$ については $KILL = mfpAVAIL^{all}_{in}(\alpha, n) \vee mfpANTIC^{all}_{in}(\alpha, n)$ と定義する．
証明の概略は付録 A.1.2 を参照．

$ANTIC^{all}$ の MFP 解が安全な解であることからアルゴリズムの正当性は保たれるが，MOP 解に対して完全な解でないことから除去できない冗長性が存在する．しかし，現実のプログラムでそのような状況は稀である．以下では特に指定しない限りは MFP 解を扱う．

5.3 命令の挿入点

データフロー方程式の解に基づいて，部分冗長な命令を完全冗長にするためにプログラム中の必要箇所に算術命令と ϕ 関数を挿入する．以下では，

$$AVAIL = \mathsf{Must} \stackrel{\text{def}}{\Leftrightarrow} AVAIL^{all} \wedge AVAIL^{wsome}$$
$$AVAIL = \mathsf{May} \stackrel{\text{def}}{\Leftrightarrow} \neg AVAIL^{all} \wedge AVAIL^{wsome}$$
$$AVAIL = \mathsf{No} \stackrel{\text{def}}{\Leftrightarrow} \neg AVAIL^{all} \wedge \neg AVAIL^{wsome}$$

と表記する．

図 11 (14)–(16) に算術命令の挿入点 *InsertNormal* を示す．(14) は冗長でない命令の直前を表す．PRE や PVNRE では，冗長でない命令の値を一時変数に格納し，以降で同じ値を計算する冗長な命令はその一時変数を参照することで算術演算を削除する．(15)，(16) は，availability が No から May に変化する合流点のエッジに命令を挿入するための条件である．No→May エッジに命令を挿入することで，以降の部分冗長な命令を完全冗長とすることができる．(15) と (16) はそれぞれ，合流点で値番号が変換されない場合とされる場合を表す．

図 11 (17)–(19) に ϕ 関数の挿入点 *InsertPhi* を示す．ϕ 関数は値番号の変換をプログラム中で明示的に表すために挿入される．ただし，左右引数の参照先命令が存在しなければならないので，(18) の条件が必要である．また (19) の条件にあるとおり，元のプログラム中に同等の ϕ 関数が存在するならば新たに挿入する必要はない．

5.4 命令の挿入

InsertNormal と *InsertPhi* 条件により，命令を挿入すべき点と挿入すべき値番号が計算されるが，実際に命令を挿入するためには引数と書き込み先の変数名を決定しなければならない．

PVNRE はすべての値番号にそれぞれ一意な新たな変数名を割り当てる．割り当てられる変数名は元のプログラム中の変数名と一致してはならない．値番号 α に割り当てられた変数名を $Var(\alpha)$ と表記する．

エッジ $m\rightarrow n$ に挿入される算術命令の値番号集合を

$$InsertionNums(m\rightarrow n) \stackrel{\text{def}}{=} \{\alpha \in Nums \mid InsertNormal(\alpha, m\rightarrow n)\}$$

と定義すると，PVNRE は *InsertionNums* 中の値番

号の昇順に命令を挿入する．値番号 α に対して挿入される命令は $\mathit{Var}(\alpha) := \mathit{Var}(\alpha.\mathit{Lt})\ \alpha.\mathit{Op}\ \mathit{Var}(\alpha.\mathit{Rt})$ である．昇順に挿入する理由は，*InsertionNums* 中の値番号の間で真のデータ依存関係がある場合に，依存する命令をプログラム中で後ろに配置するためである．ここでは，値番号の大小関係に関する条件（定義 5.1）を利用している．

ϕ 関数については，$\mathit{InsertPhi}(\alpha_2, N, \alpha_0, \alpha_1)$ に対して命令 $\mathit{Var}(\alpha_2) := \phi_N(\mathit{Var}(\alpha_0), \mathit{Var}(\alpha_1))$ を基本ブロック N の先頭に挿入する．

さらに，$\forall n \in \mathit{Nodes}$ について，

$$\mathit{BaseNode}(n) \overset{\text{def}}{\Leftrightarrow} \\ n.\mathit{Op} \notin \mathit{Normal} \wedge n \in \mathit{DefNodes}(\mathit{Num}(n))$$

を満たす命令 n に関してその書き込み先変数を x とおくと，直後に命令 $\mathrm{x} := \mathit{Var}(\mathit{Num}(n))$ を挿入し，n の書き込み先を $\mathit{Var}(\mathit{Num}(n))$ へ変更する．

定理 5.3　変形の構文的正当性
5.4 節で述べた変形後もプログラムは構文のうえで正しい形をしている．すなわち，各命令の引数の参照先命令が少なくとも 1 つ存在する．
定理の証明の概略は付録 A.1.3 を参照.

定理 5.4　変形の意味的正当性
元のプログラム中の命令と，5.4 節で述べた変形後で対応する命令の計算する値は，すべての実行パスにおいて同一である．

証明　元のプログラム中の命令に対する変形は *BaseNode* に関する変形のみであるが，この操作によって値が変わらないことは自明である．□

5.5　冗長性の除去

前節の命令の挿入によって元のプログラム中にあった算術命令はすべて完全冗長となるため，すべての算術命令 n の右辺を $\mathit{Var}(\mathit{Num}(n))$ で置き換える．

定理 5.5　PVNRE の正当性
元のプログラム中の命令と，PVNRE による最適化の後で対応する命令の計算する値は，すべての実行パスにおいて同一である．
定理の証明の概略は付録 A.1.4 を参照.

5.6　アルゴリズムの計算量

定義 5.6　値番号付けの最小性
値番号付け *Num* は最小性を満たす $\overset{\text{def}}{\Leftrightarrow}$
$\forall m, n \in \mathit{Nodes}\ .\ m.\mathit{Op}, n.\mathit{Op} \in \mathit{Normal}$ について

$$m.\mathit{Op} = n.\mathit{Op} \wedge \mathit{Num}(m.X) = \mathit{Num}(n.X) \\ \Rightarrow \mathit{Num}(m) = \mathit{Num}(n) \quad \diamondsuit$$

この条件は正当性の定義 4.12 (3) の逆を意味する．

元のプログラム中の ϕ 関数の数を p，それ以外の命令の数を n とおく．値番号付け *Num* が最小性を満たすならば，アルゴリズムの計算量は最悪の場合 $O(n^2 + np^{2^n})$ となる．ただし，現実のほとんどのプログラムでは $O((1+p)n^2)$ となる．アルゴリズムの計算量に関しては実装の説明とともに付録 A.2.7 で詳しく述べる．

6. 実　　験

我々は PVNRE を，我々が開発中の Java 用の実行時コンパイラ RJJ 上に実装した．RJJ は Java バーチャルマシンのフリーな実装である kaffe-1.0.7 [11] から呼び出される形で動作する．今回の実験では PVNRE の解析時間と除去した冗長性の数を計測するために RJJ を用い，実行コードは kaffe 付属の実行時コンパイラ jit3 を用いて生成した．

我々は比較対象として，1 章で述べた大域コピー伝搬と PRE を繰り返す手法（CPPRE）を RJJ に実装した．PRE としては Bodik らの提案した枠組み [3] を用いた．この枠組みは PVNRE が用いているものと同一である．以降では，コピー伝搬と PRE をそれぞれ 1 回だけ行う手法を CPPRE1，たかだか 2 回行う手法を CPPRE2，たかだか 3 回行う手法を CPPRE3 と呼ぶ．CPPRE2 と CPPRE3 は必ず 2 回，3 回行うのではなく，最低 1 回行った後は途中でプログラムに変化が生じなくなった時点で繰返しを打ち切る．

我々は算術命令だけでなく，オブジェクトフィールドと配列要素の読み込みの冗長性も除去の対象とする．我々はメモリを介した暗黙的なデータ依存関係を conservative な解析により仮想的なレジスタを用いて明示的にプログラム上で表現する．

Java では最適化により実行時に例外が起きる順序が変化してはならないが，本論文の実験では例外をいっさい考慮しない．これは制限を設けずに純粋に PVNRE の性能のみを計測するためである．また，順序を考慮せずに最適化しても，その後で補正コードを追加することで実行時の順序を保つことは可能である．

我々はベンチマークとして SPECjvm98 [17] からシングルスレッドプログラムを 7 つ，Java Grande Forum Benchmark Suite（JGFBS）[10] の section2 の 6 つ，section3 の 5 つを用いた．SPECjvm98 の問題サイズは 100，JGFBS は SizeA を指定したが，lu，heapsort，crypt，sor に関しては実行時間が短かすぎるので SizeB とした．各プログラムはいずれも個別実行した．

以後の計測はすべて，2 個の 2.20 GHz Xeon と 512 MB のメモリを搭載した Linux 2.4.18 マシン上

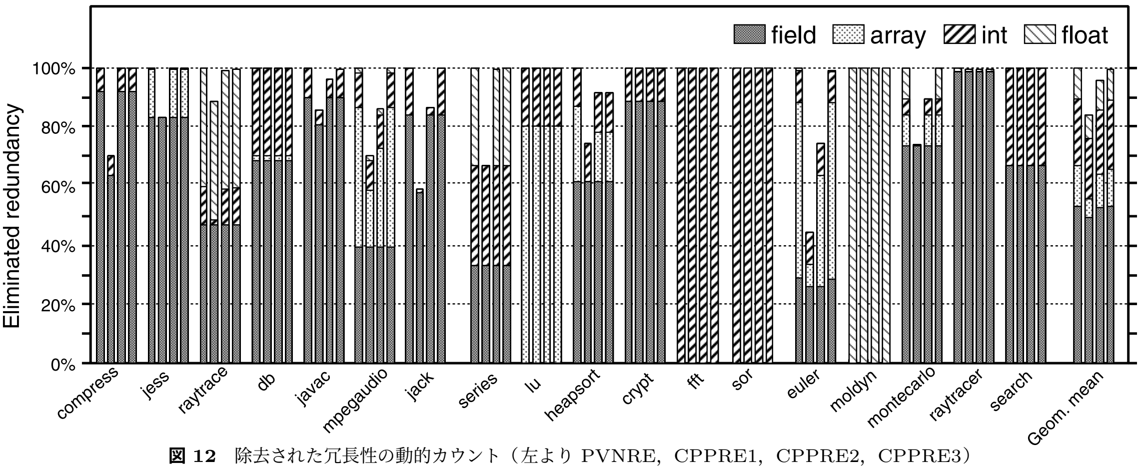


図 12 除去された冗長性の動的カウント（左より PVNRE，CPPRE1，CPPRE2，CPPRE3）
Fig. 12 Dynamic count of eliminated redundancy (from left to right: PVNRE, CPPRE1, CPPRE2 and CPPRE3).

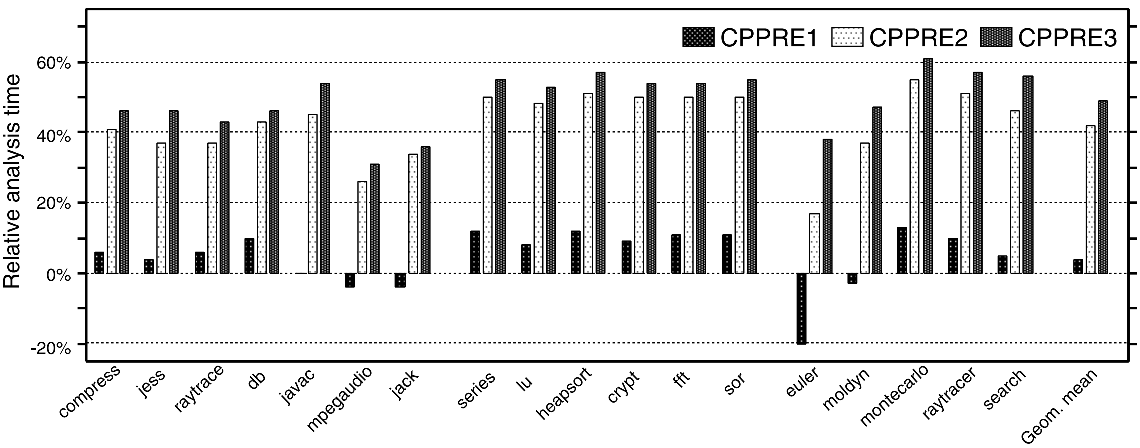


図 13 PVNRE に対する解析時間の比
Fig. 13 Ratios of analysis time to PVNRE.

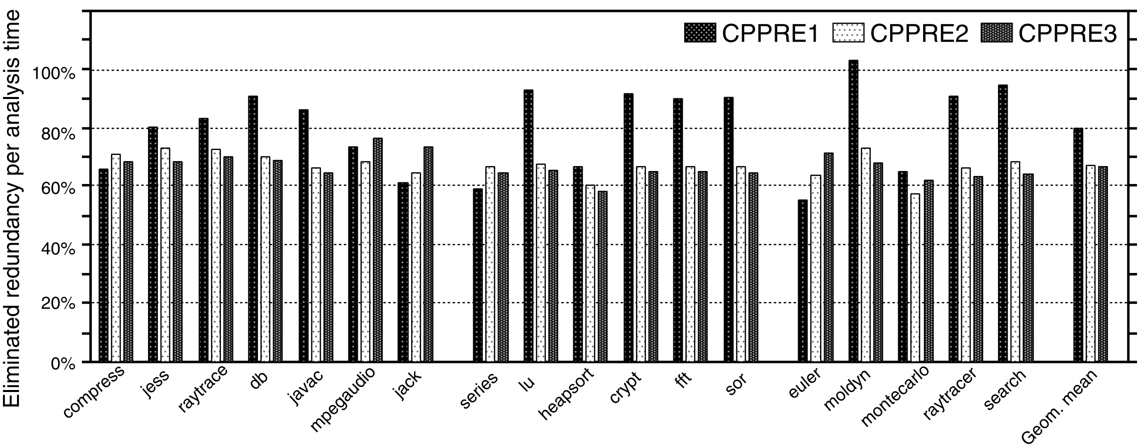


図 14 PVNRE に対する単位解析時間あたりの削減量
Fig. 14 Eliminated redundancy per analysis time compared to that of PVNRE.

で行った．kaffe はユーザレベルスレッドライブラリしか持たないため，実行には CPU1 個のみ用いられる．

6.1 冗長性除去の効果

我々は基本ブロックと制御エッジ単位で実行頻度のデータをとり，挿入された命令と除去された命令の差し引きを掛けることで，実行された全メソッドにわたる動的な命令数の削減量を求めた．結果を**図 12** に示す．グラフは各プログラムにつき左から順に PVNRE, CPPRE1, CPPRE2, CPPRE3 による削減量であり，PVNRE=100% として正規化している．Geom. mean は全プログラムの幾何平均を示す．各棒グラフの内部は命令種ごとの削減量を示しており，下から順に濃い灰色がフィールドアクセス，薄い灰色が配列アクセス，太い斜線が整数演算，細い斜線が浮動小数点演算を表す．

全体の平均では PVNRE に対して CPPRE1 は 84%，CPPRE2 は 96%，CPPRE3 は 99%の削減量である．

CPPRE1 は 18 プログラム中 10 プログラムで PVNRE の削減量を下回っている．最も削減量が少ないプログラムは euler であるが，これは最内ループ中に this.array[i][j] という形の式どうしの冗長性が含まれているからである．この式はフィールドアクセス除去 1 回と配列アクセス除去 2 回を経て完全に除去できるため，CPPRE2，CPPRE3 と繰返しを重ねるに従って配列アクセスの削減量が増えている．同じ傾向は mpegaudio でも見られる．同様に heapsort と montecarlo には this.array[i] という形の冗長性が存在する．

CPPRE2 は 6 個のプログラムで，CPPRE3 は heapsort で PVNRE より低い能力を示している．heapsort で除去できていない命令は，1 章で示した図 3 の式 8 である．式 8 は大域コピー伝搬を行っても式 2，5 と字面が同一にはならず，従来の PRE では除去できない．

全体の傾向として，フィールドアクセスの削減量が半分以上を占めている．これは Java のプログラミングスタイルとして this ポインタを介したフィールドアクセスが非常に多いためである．

以上の結果より，PVNRE は冗長な命令の削減においてコピー伝搬と PRE を繰り返す従来の手法と同等かそれ以上の能力を持つといえる．

6.2 解析時間

実行された全メソッドの合計解析時間の比較を**図 13** に示す．解析時間は 5 回実行した平均値を用いている．結果はすべてのプログラムで同じ傾向を示しており，PVNRE に比べて CPPRE1，CPPRE2，CPPRE3 の順に長くなっている．PVNRE とほぼ同等の命令削減能力を持つ CPPRE3 は PVNRE に対して最大 61%，平均で 49%遅い．すなわち，PVNRE は最大 38%，平均 33%の高速化を果たしている．

PVNRE と PRE 双方のデータフロー方程式を解く部分の実装は同一のデータ構造，アルゴリズムを使用している．また，PVNRE の前半の値番号付けでハッシュ表を用いる操作は，PRE の前半で式の字面情報を収集する際にハッシュ表を用いる操作とほぼ同一である．したがって解析時間に関して CPPRE と PVNRE の主な相違点は，

(1) CPPRE は大域コピー伝搬が必要
(2) PVNRE は方程式を 4 つ，PRE は 3 つ解く
(3) 方程式を解く際の収束までの時間
(4) PVNRE は最後に SSA 形式に復帰する操作が必要（付録 A.2.6 を参照）

となる．(1) はすべての式を対象に局所伝搬 2 回と大域伝搬 1 回を行うのに対し，(4) は一部の式を対象に支配木の探索を行うのみであり，(4) の方が軽い処理である．(2) に関して，$AVAIL^{some}$ は $AVAIL^{all}$ と同一のループ内で *Transp* や *Jlink* 情報を共用しながら計算可能であるので，解析時間は 4/3 倍よりも短い．(3) に関して，PVNRE の $O((1+p)n^2)$（5.6 節）に対して PRE は $O(n^2)$ であるが，可約な現実のプログラムサイズの範囲では両者に共通の定数項オーバヘッドが大半を占める．以上より，CPPRE1 は PVNRE より平均で 4%解析時間が長い．

CPPRE は繰返しの中のあるイテレーションで命令が削減できた場合，削減できる命令が新たに生まれた可能性があるので次のイテレーションを開始する．したがって解析時間は CPPRE1，CPPRE2，CPPRE3 と単調増加するが，実際にはもう削減できる命令が存在しない場合があるので，命令の削減量は必ずしも増加しない．しかし一方で，イテレーションを重ねることで初めて削減できる冗長性を持つメソッドが高い頻度で実行される場合には削減量の動的カウントへの寄与は大きくなる．このことは，**図 14** に示す単位解析時間あたりの命令削減量の比較において，コピー伝搬と PRE の繰返しによって削減できる命令数が多いプログラムほど CPPRE3 と PVNRE の性能が高いことからも裏付けられる．

以上の結果より PVNRE は解析時間に関して，コピー伝搬と PRE を繰り返すことで PVNRE と同等の命令削減能力を示す従来手法より，つねに高速に解析を行うといえる．

7. 関連研究

本章では，一部の実行パスにおいてのみ同じ値を計算する，字面の異なる式どうしの冗長性（LDPR）の削除を対象とした関連研究を本研究と比較し，続いて PRE と GVN に関する過去の研究を紹介する．

7.1 LDPR の除去

Rosen らは ϕ 関数を用いて式の字面を変換することで LDPR を除去する手法を提案した[16]．Rosen らの手法では，まず各式にデータ依存関係の深さを表す “rank” と呼ばれる整数を割り振り，同じ rank が割り振られた式の集合ごとに SSA 形式上でのコピー伝搬と式の上方への移動，削除を行う．

Rosen らの手法は，SSA 形式を用いながらも結局は各 rank ごとにコピー伝搬と冗長性除去を繰り返しているため，従来のコピー伝搬と PRE を繰り返す手法と本質的な差はない．一方，PVNRE は値番号付けを冗長性除去に用いつつ，データ依存関係を rank ではなく値番号の大小で表現しているために，コピー伝搬と冗長性除去の繰返しは必要ない．

Steffen らは式の値の冗長性を求めた後で PRE を行う，2 段階のアルゴリズムを提案した[18]．彼らの手法では「Herbrand 等価」と呼ばれる値の冗長性を，プログラムの疑似実行を収束するまで行うことで求める．求めた冗長性を「値フローグラフ」と呼ばれるグラフ構造で字面の形で明示的に表現し，値フローグラフの上で PRE を行う．

Steffen らの手法は 2 段階の手法という点で PVNRE と類似しているが，Herbrand 等価を求める処理は値番号付けよりも遅く，値番号付けで発見できる冗長性を発見できない場合がある．また，PVNRE は Steffen らのように PRE の前に冗長性を明示的にグラフの形で表現しないためにフロー関数の分配則が成り立たない．しかし，そのために失われる最適化能力は実際のプログラムでは問題とならず，プログラム表現を変換するオーバヘッドがかからない利点が生まれる．

滝本らは「拡張値グラフ」というプログラム表現を用いる手法を提案した[19]．滝本らの手法は Steffen らと同様に値フローグラフを使用するが，前半部分で拡張値グラフを用いて式を上方へ移動した後にハッシュ表を用いて冗長性を検出する．

滝本らの手法は Steffen らの手法よりもグラフを変換するオーバヘッドが PVNRE に比べさらに増加している．また，図 3 であげた例は PVNRE では除去できるが拡張値グラフの上では除去できない．

7.2 PRE

PRE は Morel ら[15] をはじめとして様々な手法が提案されてきたが，近年広く用いられている枠組みは Knoop らによって提案された Lazy Code Motion[13),14)] である．Lazy Code Motion は $AVAIL^{all}$ と $ANTIC^{all}$ に加えもう 1 つ方程式を解くことでレジスタの生存区間を最も短くする挿入点を見つける．彼らの手法は式の字面の冗長性を対象としており，1 回あたりの解析時間は PVNRE より短いが，除去可能な式の数は少ない．

Bodik らは $AVAIL^{all}$，$ANTIC^{all}$，$AVAIL^{wsome}$ の 3 つのデータフロー方程式を解くことで，より分かりやすい形で Lazy Code Motion と同等の効果が得られることを示した[3]．PVNRE では Bodik らが提案した枠組みを利用している．

Briggs らは式の結合則と分配則を利用して，PRE の枠組みでより多くの冗長性を検出する手法を提案した[4]．この手法では途中で値番号付けを行うが，その後に行われる PRE のために字面を整えることのみを目的としており，値番号付けの結果を冗長性除去に直接は利用していない．したがって 1 回の解析で LDPR を除去することはできない．

7.3 GVN

Alpern らはパーティショニングの手法を用いた GVN を提案した[1]．ハッシュ表を用いた GVN と異なり，彼らの手法ではデータ依存関係にループが存在する場合にも同一の値番号を割り振ることができる．一方，本論文で我々は実装していないが，ハッシュ表を用いると交換則などを利用して検出可能な冗長性の数を増やすことができる．PVNRE ではデータフロー解析を解く最中にも新たに値番号を割り振る必要があるので，パーティショニングを用いることはできない．

Click は GVN と積極的なコード移動を組み合わせた手法を提案した[6]．Click の手法ではハッシュ表を用いた GVN を行った後で，SSA 形式の値グラフと支配関係の情報を用いてコードを積極的にループの外に出す．安全でないパスにも命令を挿入するために除去可能な冗長性の数は Alpern らの手法より多いが，GVN の枠組みの中を出ていないので LDPR を除去することはできない．

Cooper らはハッシュ表を用いた GVN を行った後で値番号をデータフロー解析の対象とした PRE を行う手法を提案した[8]．値番号を伝搬させるという意味で上記のいずれの手法よりも PVNRE に類似しているが，ϕ 関数を介した変換を行っていないので，除去可能な冗長性の数は Click の手法と同等かそれ以下で

ある．

8. まとめ

本論文では，一部の実行パスにおいてのみ同じ値を計算する，字面の異なる式どうしの冗長性を効率的に除去することができる新しい部分冗長性除去の手法，PVNRE を提案した．PVNRE は値番号をデータフロー解析の対象とすることで，字面は異なるが同じ値を計算する式の冗長性を扱うことができる．また，合流点で ϕ 関数を用いて値番号を変換することで，一部の実行パスにおいてのみ同じ値を計算する式の冗長性を検出できる．PVNRE は値番号の大小関係でデータ依存関係を表すことで，依存関係で上流から順にコピー伝搬と冗長性除去を繰り返す必要がない．

我々は PVNRE を Java の実行時コンパイラとして実装し，SPECjvm98 と Java Grande Benchmark を用いて実験を行った．PVNRE は冗長性の削減量において従来手法と同等かそれ以上の能力を持つことを確認した．また，PVNRE は同等の削減能力を持つ従来手法に対して最大で 38%，平均で 33%高速に解析を行うことを示した．このことから，PVNRE は解析時間が問題となる実行時コンパイラ向けの優れた最適化手法であるといえる．

今後の研究課題として，Java の例外処理に関する最適化との統合があげられる．本論文では算術命令とメモリからのロード命令のみを冗長性除去の対象としたが，ヌルポインタチェックや配列の範囲チェックの冗長性除去へ拡張することが考えられる．

参考文献

1) Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting Equality of Variables in Programs, *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, pp.1–11 (1988).
2) Bodik, R. and Anik, S.: Path-Sensitive Value-Flow Analysis, *Symposium on Principles of Programming Languages*, pp.237–251 (1998).
3) Bodik, R., Gupta, R. and Soffa, M.L.: Complete Removal of Redundant Computations, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.1–14 (1998).
4) Briggs, P. and Cooper, K.D.: Effective Partial Redundancy Elimination, *ACM SIGPLAN Notices*, Vol.29, No.6, pp.159–170 (1994).
5) Briggs, P., Cooper, K.D. and Simpson, L.T.: Value Numbering, *Software Practice and Experience*, Vol.27, No.6, pp.701–724 (1997).
6) Click, C.: Global code motion: global value numbering, *ACM SIGPLAN Notices*, Vol.30, No.6, pp.246–257 (1995).
7) Cooper, K. and Simpson, T.: SCC-Based Value Numbering, Technical report, CRPC-TR95636-S, Rice University (1995).
8) Cooper, K. and Simpson, T.: Value-Driven Code Motion, Technical report, CRPC-TR95637-S, Rice University (1995).
9) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
10) Java Grande Benchmarking Project: Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/
11) Kaffe.org: Kaffe Open VM. http://www.kaffe.org/
12) Kildall, G.A.: A unified approach to global program optimization, *Proc. 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp.194–206, ACM Press (1973).
13) Knoop, J., Rüthing, O. and Steffen, B.: Lazy code motion, *ACM SIGPLAN Notices*, Vol.27, No.7, pp.224–234 (1992).
14) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: theory and practice, *ACM Trans. Prog. Lang. and Syst.* (*TOPLAS*), Vol.16, No.4, pp.1117–1155 (1994).
15) Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
16) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global value numbers and redundant computations, *Proc. 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.12–27, ACM Press (1988).
17) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/
18) Steffen, B., Knoop, J. and Ruthing, O.: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *European Symposium on Programming*, pp.389–405 (1990).
19) 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).

付　　録

A.1　定理の証明の概略

A.1.1　定理 4.13 値番号に関する冗長性定理

$\forall u \in \mathit{Nodes}$ について，

$$\mathit{DomBedges}(u) \stackrel{\mathrm{def}}{=} \{e = m \rightarrow n \mid e \in \mathit{Bedges} \wedge n \text{ dom } u\}$$

と定義する．グラフが可約であることから，以下の 2 つの補題が成り立つ．

補題 A.1.1

$$\begin{aligned}&\forall p \in P[\mathit{start}, \mathit{end}], \forall 1 \leq j < i \leq \lambda(p)\ .\\&\quad p_i.\mathit{Op} \in \mathit{Normal} \cup \mathit{Copy} \wedge p_j = p_i.X\\&\quad \Rightarrow \forall j \leq k < i\ .\\&\qquad p_k \rightarrow p_{k+1} \notin \mathit{DomBedges}(\mathit{Node}(p_j))\end{aligned}$$

補題 A.1.2

$$\begin{aligned}&\forall e \in \mathit{Edges}, \forall n \in \mathit{Nodes}\ .\\&\quad e \notin \mathit{DomBedges}(n) \Rightarrow \mathit{Transp}(\mathit{Num}(n), e)\end{aligned}$$

次の定理 A.1.3 は，実行パス上で命令とその引数の間は引数の値番号に関して Transp であることを表す．

定理 A.1.3　命令の引数に関する透過性

$$\begin{aligned}&\forall p \in P[\mathit{start}, \mathit{end}], \forall 1 \leq j < i \leq \lambda(p)\ .\\&\quad p_i.\mathit{Op} \in \mathit{Normal} \cup \mathit{Copy} \wedge p_j = p_i.X\\&\quad \Rightarrow \mathit{Transp}^{\forall}(\mathit{Num}(p_j), p[j, i])\end{aligned}$$

証明の概略　補題 A.1.1，A.1.2 および定義 4.4 (2) より成立．□

また，$\mathit{Jtarget}$ はループを一周して元の位置まで伝搬することはできない．

補題 A.1.4

$$\begin{aligned}&\forall \alpha_0, \alpha_1, \alpha_2 \in \mathit{Nums}, \forall N \in \mathit{BBNodes},\\&\forall p \in P[\mathit{start}, \mathit{end}], \forall 1 \leq i < j \leq \lambda(p)\ .\\&\quad \mathit{Join}(N, \alpha_0, \alpha_1) \wedge \alpha_2 \in \mathit{Jtarget}(N, \alpha_0, \alpha_1)\\&\quad \wedge \mathit{Node}(p_i) = \mathit{Node}(p_j) = \mathit{Head}(N)\\&\quad \Rightarrow \neg \mathit{Transp}^{\forall}(\alpha_2, p[i, j])\end{aligned}$$

定理 A.1.3，補題 A.1.4 より定理 4.13 が証明される．

証明の概略　実行パス上で Fixed を基点とするデータ依存関係の深さに関する数学的帰納法で証明する．$\mathit{Num}(p_i) = \mathit{Num}(p_j)$ であることから，値番号付けの正当性（定義 4.12）の (1)〜(5) それぞれについて場合分けを行う．定理 A.1.3 により引数との間が Transp であることを利用して帰納法の仮定を満たす．(4) に関しては補題 A.1.4 を用いる．□

A.1.2　MFP 解と MOP 解

データフロー方程式のすべてのフロー関数が単調かつ分配則が成立するならば，繰返しアルゴリズムの MFP 解が MOP 解に一致する[12]．図 9，図 10 の Transp，$(n \in \mathit{DefNodes}(\alpha))$，$\mathit{KILL}$ は単調かつ分配則が成立するが，Jlink' に関しては単調であるが $\wedge$ について分配則が成立しない．Jlink' のフロー関数を一般化して表すと $1 \leq \alpha, \beta \leq \mathit{max}(\mathit{Nums})$ について $f(\langle x_\alpha \rangle) = \langle x_\alpha \vee x_\beta \rangle$ となる．

補題 A.1.5

Jlink' のフロー関数は単調である．

証明

$\langle x_\alpha \rangle \leq \langle y_\alpha \rangle \Rightarrow x_\alpha \leq y_\alpha \Rightarrow x_\alpha \vee x_\beta \leq y_\alpha \vee y_\beta \Rightarrow \langle x_\alpha \vee x_\beta \rangle \leq \langle y_\alpha \vee y_\beta \rangle \Rightarrow f(\langle x_\alpha \rangle) \leq f(\langle y_\alpha \rangle)$ □

補題 A.1.6

Jlink' のフロー関数は $\wedge$ について分配則は成り立たず，$\vee$ について成り立つ．

証明　$\wedge$ について，

$$\begin{aligned}&f(\langle x_\alpha \rangle \wedge \langle y_\alpha \rangle) = \langle (x_\alpha \wedge y_\alpha) \vee (x_\beta \wedge y_\beta) \rangle \neq\\&\langle (x_\alpha \vee x_\beta) \wedge (y_\alpha \vee y_\beta) \rangle = f(\langle x_\alpha \rangle) \wedge f(\langle y_\alpha \rangle)\end{aligned}$$

より．$\vee$ についても同様の計算．□

MFP 解と MOP 解に関する定理 5.2 の証明の概略は以下のとおり．

証明の概略　補題 A.1.5，A.1.6 より，$\mathit{AVAIL}^{\mathit{some}}$，$\mathit{AVAIL}^{\mathit{wsome}}$ の MFP 解は MOP 解に一致する．$\mathit{ANTIC}^{\mathit{all}}$ の MFP 解は Kildall の証明[12] により安全な解であるが，MOP 解には一致しない．

$\mathit{AVAIL}^{\mathit{all}}$ のフロー関数について実際には分配則が成り立つ．なぜならば補題 A.1.4 より，図 9 (1)，(2) において

$$\begin{aligned}&\forall \alpha \exists \beta\ .\ \alpha \in \mathit{Jlink}'(\mathit{BB}(n), \beta, m \rightarrow n)\\&\Rightarrow \neg \mathit{AVAIL}^{\mathit{all}}_{\mathit{out}}(\alpha, m)\end{aligned}$$

が成立し，フロー関数は実際には $f(\langle x_\alpha \rangle) = \langle x_\beta \rangle$ となるからである．したがって $\mathit{AVAIL}^{\mathit{all}}$ の MFP 解は MOP 解に一致する．□

A.1.3　定理 5.3 変形の構文的正当性

命令 n の前方に必ず $\mathit{Var}(\alpha)$ への代入が存在することを表す Cover を以下のように定義する．

定義 A.1.7　*Cover*

$\forall \alpha \in \mathit{Nums}, \forall n \in \mathit{Nodes}$ について，

$\mathit{Cover}(\alpha, n) \stackrel{\mathrm{def}}{\Leftrightarrow} \forall p \in P[\mathit{start}, n], \exists 1 \leq i < \lambda(p)$ s.t.

(1) if $\alpha.\mathit{Op} \in \mathit{Normal}$

$$\begin{aligned}&\mathit{Transp}^{\forall}(\alpha, p[i, \lambda(p)])\\&\wedge(\mathit{Node}(p_i) \in \mathit{DefNodes}(\alpha)\\&\quad \vee \mathit{InsertNormal}(\alpha, p_i \rightarrow p_{i+1})\\&\quad \vee \exists \alpha_0, \alpha_1 \in \mathit{Nums}\ .\\&\qquad \mathit{InsertPhi}(\alpha, \mathit{BB}(\mathit{Node}(p_i)), \alpha_0, \alpha_1))\end{aligned}$$

(2) otherwise

$$Transp^{\forall}(\alpha, p[i, \lambda(p)])$$
$$\wedge Node(p_i) \in DefNodes(\alpha) \quad \diamondsuit$$

補題 A.1.8

$\forall\alpha \in Nums, \forall n \in Nodes$ について

$mfpAVAIL^{wsome}(\alpha, n) \Rightarrow Cover(\alpha, n)$

証明の概略 $mfpAVAIL^{wsome} \Rightarrow mopAVAIL^{wsome}$ と $InsertNormal$, $InsertPhi$ の定義より成立. □

補題 A.1.9

$\forall n \in Nodes$ について

$n.Op \in Normal \Rightarrow Cover(Num(n), n)$

証明の概略 $mfpAVAIL^{wsome}(Num(n), n)$ のとき補題 A.1.8 より成立. それ以外のとき, 図 11 (14) より, 直前に命令が挿入される. □

補題 A.1.10

$\forall\alpha \in Nums . \alpha.Op \in Normal \cup Copy, \forall n \in Nodes$

$$mfpAVAIL^{all}(\alpha, n) \Rightarrow mfpAVAIL^{all}(\alpha.X, n)$$

証明の概略 定義 4.4 より

$\forall e \in Edges\ .\ Transp(\alpha, e) \Rightarrow Transp(\alpha.X, e)$

であることと定理 A.1.3 を用いて, $Jlink'$ の連鎖の長さに関する数学的帰納法により

$$mopAVAIL^{all}(\alpha, n) \Rightarrow mopAVAIL^{all}(\alpha.X, n)$$

が成り立つ. MFP 解と MOP 解が一致することから補題は成立. □

補題 A.1.11

$\forall\alpha \in Nums . \alpha.Op \in Normal \cup Copy, \forall n \in Nodes$

$$mfpANTIC^{all}_{in}(\alpha, n) \Rightarrow$$
$$mfpAVAIL^{all}_{in}(\alpha.X, n) \vee mfpANTIC^{all}_{in}(\alpha.X, n)$$

証明の概略 $ANTIC^{all}$ に関する $Jlink'$ のフロー関数の性質より成立. □

補題 A.1.12

$\forall\alpha \in Nums . \alpha.Op \in Normal \cup Copy, \forall n \in Nodes$

$mfpAVAIL^{wsome}(\alpha, n)$

$\Rightarrow mfpAVAIL^{wsome}(\alpha.X, n)$

証明の概略 補題 A.1.10 と A.1.11 より,

$KILL(\alpha, n) \Rightarrow KILL(\alpha.X, n)$

であることから成立. □

以上より, 算術命令に関する挿入の正当性が言える.

補題 A.1.13

$\forall\alpha \in Nums . \alpha.Op \in Normal, \forall m, n \in Nodes$.

$InsertNormal(\alpha, m \rightarrow n) \Rightarrow$

$InsertNormal(\alpha.X, m \rightarrow n)$

$\vee Cover(\alpha.X, m)$

証明の概略 図 11 (14) のとき, $n.X$ に関して定理 A.1.3 と補題 A.1.9 を用いて成り立つ. 図 11 (15), (16) のとき, $mfpAVAIL^{wsome}_{out}(\alpha.X, m)$ ならば補題 A.1.8 より成立. それ以外のとき, 補題 A.1.12 より, $m \rightarrow n$ は $\alpha.X$ の挿入点となり, 成立. □

よって変形の構文的正当性に関する定理 5.3 が証明される.

証明の概略 $InsertNormal$ については補題 A.1.13 より正しい. $InsertPhi$ で挿入される ϕ 関数の引数については補題 A.1.8 と $InsertNormal$ の定義より参照先が存在する. □

A.1.4 定理 5.5 PVNRE の正当性

証明の概略 定理 5.3 変形の構文的正当性より, 元のプログラム中の命令だけでなく, 新たに挿入された命令についても, 割り振られた値番号とその値番号に関する Op, Lt, Rt 関数の正当性が成り立つ. よって変形後のプログラム上でも定理 4.13 値番号の冗長性定理が成り立つ. ここで, すべての算術命令 n は補題 A.1.9 より $Cover$ 条件が成立するため完全冗長となっており, かつ $Var(Num(n))$ へ書き込む命令との間で冗長性定理が成り立つ. 定理 5.4 変形の意味的正当性より n の値は変わっていないため, $Var(Num(n))$ へ書き込む命令の値は元のプログラム中の n の値と等しい. よって, n の右辺を $Var(Num(n))$ で置き換えても値は変わらない. □

A.2 PVNRE の実装と動作

PVNRE の実装は 5 章で示した流れに従って,

(1) プログラム中の各命令に値番号を割り振る,

(2) 4 つのデータフロー方程式を繰返しアルゴリズムで順に解く,

(3) 必要な命令を挿入し, 冗長性を除去する,

となるが, データフロー方程式を解く処理と値番号付けの一部を同時に行うため, 通常の繰返しアルゴリズム[12]とは異なる部分が存在する. 本節では PVNRE の実装の特徴的な部分を疑似コードで示す. 動作の例を図 18 のプログラムを用いて説明する.

A.2.1 データ構造

- $JT(N)$, N は合流ブロック ; $\langle trgt, \langle l, r\rangle\rangle$ を要素とする集合で $\langle trgt, \langle l, r\rangle\rangle \in JT(N) \overset{\text{def}}{\Leftrightarrow} trgt \in Jtarget(N, l, r)$ とする. JT' も $Jtarget'$ に対して同様に定義する. $lnk = \langle l, r\rangle$, N の左右の先行ブロックからの制御エッジを e_l, e_r とおくと, $lnk(e_l) = l$, $lnk(e_r) = r$ と表記する. 実装上は $trgt$ と $\langle l, r\rangle$ それぞれでアクセス可能な 2 つの可変長テーブルを用いると効率的である.

- $UnTransp(e), e \in Edges$；値番号を要素とする集合で $\alpha \in UnTransp(e) \overset{\mathrm{def}}{\Leftrightarrow} \neg Transp(\alpha, e)$ とする．バックエッジに関してのみ計算すればよい．
- $GEN(N), N \in BBNodes$；N で計算される値番号の集合．5 章のデータフロー方程式の定式化では $AVAIL^{all}$，$AVAIL^{some}$ などを命令単位で定義したが，実装上は実行効率のために基本ブロック単位で計算する．そのため GEN が必要となる．
- $edgeAVAIL^{some}(e), e = M \rightarrow N \in Edges$；制御エッジ e を最後にたどったときの $AVAIL^{some}_{out}(M)$ で，値番号の変換に用いる．
- $AVAIL^{all}$，$AVAIL^{some}$，$ANTIC^{all}$，$AVAIL^{wsome}$ や $UnTransp$，GEN，$edgeAVAIL^{some}$ はビットベクトルを用いた効率的な実装が可能である．

A.2.2 値番号付け

値番号付けのアルゴリズムを**図 15** に示す．$Bedges$ と $UnTransp$ の計算を除いてハッシュ表を用いた従来の値番号付けのアルゴリズムと同一である．ここでは算術命令についてのみハッシュ表を検索しているが，ϕ 関数についても同様の処理が可能である．

各値番号に $Bedges$ という属性を付加するが，$Fixed$ 命令と ϕ 関数に関しては $Transp$ の定義 4.4 (1) に従ってその定義命令を囲むバックエッジの集合を指す（16 行目）．算術命令に関しては定義 4.4 (2) に従って両引数の $Bedges$ の和集合について $UnTransp$ に設定する（39〜42 行目）．算術命令についてはデータフロー解析の対象となるので GEN に追加する（23 行目）．図 9 では定式化のためにすべての命令を対象としたが，実装上は算術命令以外の情報は伝搬させる必要はない．最後に，各 ϕ 関数を JT に登録する（28〜30 行目）．

図 18 (a) に値番号が割り振られた状態を示す．なお，$UnTransp$(BB6→BB5) $= \emptyset$ である．

A.2.3 $AVAIL^{all}$，$AVAIL^{some}$

$AVAIL^{all}$，$AVAIL^{some}$ の繰返しアルゴリズムの前半部分を**図 16** に示す．$ConvThroughPhi$（13 行目）で $JT(N)$，$JT'(N)$ に要素を追加し，$CompAVAILin$（14 行目）で $JT'(N)$ に基づいて $AVAIL^{all}_{in}$，$AVAIL^{some}_{in}$ を計算する．$CompAVAILin$ の 30〜42 行目が図 9 (2)，(5) に相当する．

アルゴリズムの後半部分を**図 17** に示す．$AVAIL^{all}$ を計算するためには，新たに JT に要素を追加するための情報として $AVAIL^{all}$ を用いれば十分であるが，我々は $AVAIL^{some}$ を用いる．$AVAIL^{all}$ を用いた場合，$AVAIL^{some}$ に比べて合流ブロックに到達する値番号の数が少ないため，この後の $ANTIC^{all}$ の計算でも JT に要素を追加する処理が必要となる．JT に対する処理は繰返しアルゴリズムの他の部分に比べて複雑であるため，実装を簡略化するために処理を 1 カ所にまとめて，$AVAIL^{some}$ を用いる．

```
1  for each N ∈ BBNodes do
2    for each n ∈ N do
3      Num(n) := 0
4    end
5  end
6  for each e ∈ Edges do
7    UnTransp(e) = ∅
8  end
9  VN := 0
10 for each N ∈ BBNodes in reverse post order do
11   GEN(N) = ∅
12   for each n ∈ N in pre order do
13     if n.Op ∈ Fixed, Phi then
14       VN := VN + 1
15       VN.Op := n.Op
16       VN.Bedges := Bedges(N)
17       Num(n) := VN
18     else if n.Op ∈ Copy then
19       Num(n) := Num(n.Lt)
20     else if n.Op ∈ Normal then
21       α := call
             Hash(n.Op, Num(n.Lt), Num(n.Rt))
22       Num(n) := α
23       GEN(N) := GEN(N) ∪ {α}
24     end
25   end
26 end
27 for each N ∈ BBNodes do
28   for each n ∈ N . n.Op ∈ Phi do
29     JT(N) := JT(N) ∪
           {⟨Num(n), ⟨Num(n.Lt), Num(n.Rt)⟩⟩}
30   end
31 end
32
33 Hash(op, l, r) {
34   ⟨op, l, r⟩ を鍵とする番号が登録されていればそれを返す．
35   登録されていなければ，
36   VN := VN + 1
37   ⟨op, l, r⟩ を鍵として VN を登録する．
38   VN.Op := op, VN.Lt := l, VN.Rt := r
39   VN.Bedges := l.Bedges ∪ r.Bedges
40   for each b ∈ VN.Bedges do
41     UnTransp(b) := UnTransp(b) ∪ {VN}
42   end
43   return VN
44 }
```

図 15 値番号付けのアルゴリズム
Fig. 15 Algorithm of value numbering.

合流ブロックに到達した値番号を先行ブロックごとに調べていく（2 行目）．ただし，前回に同じ制御エッジをたどったときからの差分のみを調べればよい（3 行目，23 行目）．値番号を昇順に見ていくことで，データ依存関係で上流から順に調べる（4 行目）．α の左右引

```
1  for each N ∈ BBNodes do
2    AVAIL_out^all(N) = ⊤, AVAIL_out^some(N) = ∅
3  end
4  for each e ∈ Edges do
5    edgeAVAIL^some(e) = ∅
6  end
7
8  AVAIL_out^all(start) = ∅, AVAIL_out^some(start) = ∅
9  W := Succ(start)
10 while W ≠ ∅ do
11   N ∈ W
12   W := W \ N
13   call ConvThroughPhi(N)
14   call CompAVAILin(N)
15   newAVAIL_out^all := AVAIL_in^all(N) ∪ GEN(N)
16   newAVAIL_out^some := AVAIL_in^some(N) ∪ GEN(N)
17   if newAVAIL_out^all ≠ AVAIL_out^all(N)
       ∨ newAVAIL_out^some ≠ AVAIL_out^some(N) then
18     AVAIL_out^all(N) := newAVAIL_out^all
19     AVAIL_out^some(N) := newAVAIL_out^some
20     W := W ∪ Succ(N)
21   end
22 end
23
24 CompAVAILin(N) {
25   AVAIL_in^all(N) := ⊤, AVAIL_in^some(N) := ∅
26   for each M ∈ Pred(N) do
27     AVAIL_in^all(N) := AVAIL_in^all(N)
            ∩ (AVAIL_out^all(M) − UnTransp(M→N))
28     AVAIL_in^some(N) := AVAIL_in^some(N)
            ∪ (AVAIL_out^some(M) − UnTransp(M→N))
29   end
30   for each ⟨trgt, lnk⟩ ∈ JT′(N) do
31     ba = true, bs = false
32     for each M ∈ Pred(N) do
33       ba := ba ∧ (lnk(M→N) ∈ AVAIL_out^all(M))
34       bs := bs ∨ (lnk(M→N) ∈ AVAIL_out^some(M))
35     end
36     if ba then
37       AVAIL_in^all(N) := AVAIL_in^all(N) ∪ {trgt}
38     end
39     if bs then
40       AVAIL_in^some(N) := AVAIL_in^some(N) ∪ {trgt}
41     end
42   end
43 }
```

図 16 $AVAIL^{all}$, $AVAIL^{some}$ のアルゴリズム（前半）
Fig. 16 Algorithm for $AVAIL^{all}$, $AVAIL^{some}$ (the first half).

数に変換先が存在する場合（10 行目）が定義 4.7 (3)，どちらかの引数にのみ変換先が存在する場合（14 行目，19 行目）が定義 4.7 (2) に相当する．α の変換先 *trgt* を *Hash* を引くことで求める（28 行目）．*trgt* に合流するその他のエッジからの値番号を求め（30〜37 行目），JT, JT' に追加する（38，39 行目）．ここで JT に追加した要素は，データ依存関係で下流にある値番号を変換する際に用いることができる（4〜6 行目）．

```
1  ConvThroughPhi(N) {
2    for each M ∈ Pred(N) do
3      diff := AVAIL_out^some(M) − edgeAVAIL^some(M→N)
4      for each α ∈ diff in ascending order do
5        ljs := {⟨trgt, lnk⟩ ∈ JT(N) | lnk(M→N) = α.Lt}
6        rjs := {⟨trgt, lnk⟩ ∈ JT(N) | lnk(M→N) = α.Rt}
7        if ljs ≠ ∅ then
8          if rjs ≠ ∅ then
9            for each ⟨l, llnk⟩ ∈ ljs, ⟨r, rlnk⟩ ∈ rjs do
10             call DoConv(N, α, l, llnk, r, rlnk)
11           end
12         else if α.Rt ∉ UnTransp(M→N) then
13           for each ⟨l, llnk⟩ ∈ ljs do
14             call DoConv(N, α, l, llnk, α.Rt, nil)
15           end
16         end
17       else if rjs ≠ ∅
               ∧ α.Lt ∉ UnTransp(M→N) then
18         for each ⟨r, rlnk⟩ ∈ rjs do
19           call DoConv(N, α, α.Lt, nil, r, rlnk)
20         end
21       end
22     end
23     edgeAVAIL^some(M→N) := AVAIL_out^some(M)
24   end
25 }
26
27 DoConv(N, α, l, llnk, r, rlnk) {
28   trgt := call Hash(α.Op, l, r)
29   if ∃x . ⟨trgt, x⟩ ∈ JT(N) then return end
30   for each M ∈ Pred(N) do
31     if llnk = nil then lt := l
32     else lt := llnk(M→N) end
33     if rlnk = nil then rt := r
34     else rt := rlnk(M→N) end
35     β := call Hash(α.Op, lt, rt)
36     lnk(M→N) := β
37   end
38   JT(N) := JT(N) ∪ ⟨trgt, lnk⟩
39   JT′(N) := JT′(N) ∪ ⟨trgt, lnk⟩
40 }
```

図 17 $AVAIL^{all}$, $AVAIL^{some}$ のアルゴリズム（後半）
Fig. 17 Algorithm for $AVAIL^{all}$, $AVAIL^{some}$ (the second half).

図 18 (a) で $AVAIL^{all}$，$AVAIL^{some}$ を計算する途中で新たに生成される値番号を**表 2** に示す．また，BB4 と BB9 における JT，JT' を**表 3** に示す．もし $AVAIL^{all}$ のみを計算して $AVAIL^{some}$ を計算しない場合，JT，$JT'(BB9)$ に $\langle \mathit{9}, \langle \mathit{14}, \mathit{2} \rangle\rangle$ と $\langle \mathit{10}, \langle \mathit{15}, \mathit{3} \rangle\rangle$ は追加されない．これらは $AVAIL^{some}_{out}$(BB7) に *3*，*4* が伝搬してくることで JT，$JT'(BB9)$ に追加されるが，$AVAIL^{all}_{out}$(BB7) に *3*，*4* は伝搬してこないからである．もしこれらの値番号の変換情報が存在しないと，次に $ANTIC^{all}$ を計算するときに *9*，*10* の $ANTIC^{all}$ 情報が変換されて上へ伝搬しないため，冗

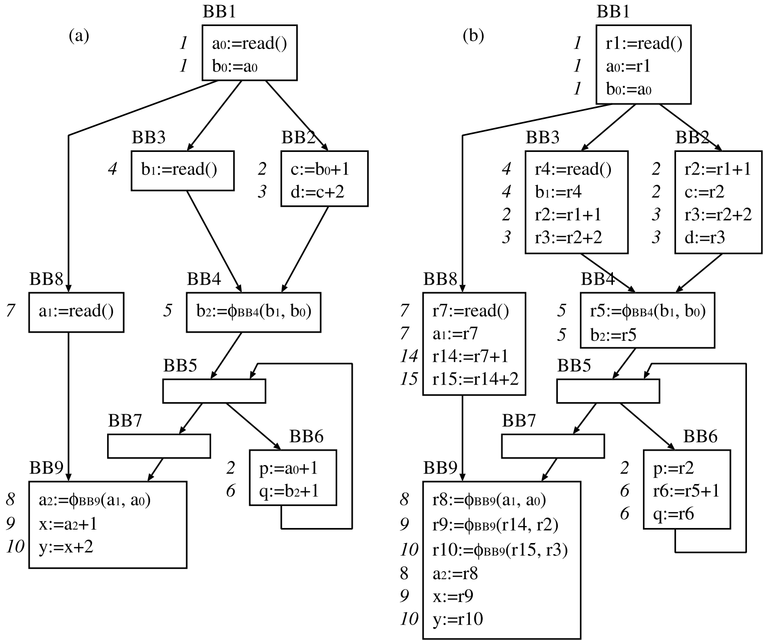


図 18 PVNRE による最適化の例
Fig. 18 Example of optimization by PVNRE.

表 2 新たに生成された値番号
Table 2 Newly created value numbers.

	Op	*Lt*	*Rt*
11	+	*4*	const 1
12	+	*6*	const 2
13	+	*11*	const 2
14	+	*7*	const 1
15	+	*14*	const 2

表 3 合流ノードにおける JT と JT'
Table 3 JT and JT' at join nodes.

BB4		B9	
JT	$\langle 5, \langle 4, 1\rangle\rangle$	JT	$\langle 8, \langle 7, 1\rangle\rangle$
JT, JT'	$\langle 6, \langle 11, 2\rangle\rangle$	JT, JT'	$\langle 9, \langle 14, 2\rangle\rangle$
JT, JT'	$\langle 12, \langle 13, 3\rangle\rangle$	JT, JT'	$\langle 10, \langle 15, 3\rangle\rangle$

長性を発見できない．$AVAIL^{all}$，$AVAIL^{some}$ の計算結果を**表 4** の左半分に示す．

A.2.4 $ANTIC^{all}$，$AVAIL^{wsome}$

$ANTIC^{all}$，$AVAIL^{wsome}$ の計算は，図 16 の *CompAVAILin* と同様に JT' を用いて値番号の変換を行う他は従来の繰返しアルゴリズムと同じ実装である．例における計算結果を表 4 の右半分に示す．

A.2.5 命令の挿入と冗長性の除去

PRE と同様に，全基本ブロックを巡回して命令を挿入し，算術命令の右辺を一時変数で置き換える．さらに合流ブロックの先頭には *InsertPhi* に従って ϕ 関数を挿入する．

図 18 (a) において

$$InsertionNums(\mathrm{BB2}{\rightarrow}\mathrm{BB4}) = \{2, 3\}$$
$$InsertionNums(\mathrm{BB8}{\rightarrow}\mathrm{BB9}) = \{14, 15\}$$
$$InsertPhi(9, \mathrm{BB9}, 14, 2)$$
$$InsertPhi(10, \mathrm{BB9}, 15, 3)$$

である．冗長性除去の結果を図 18 (b) に示す．値番号 α に対応する変数名を rα としている．

A.2.6 SSA 形式への復帰

A.2.5 の操作により，ある一時変数への代入が複数箇所に生じて SSA 形式の条件を満たさなくなる．よって我々は文献 9) の手法を用いて再度 SSA 形式へ変形し直す．図 18 (b) では BB4 に r2，r3 の ϕ 関数を挿入する必要がある．

A.2.7 アルゴリズムの計算量

元のプログラム中の ϕ 関数の数を p，それ以外の命令の数を n とおく．

図 17 の 28 行目と 35 行目で *Hash* を引く回数を考える．値番号 1 つについて合流性を調べて JT に追加する際に *Hash* を定数回引く．

ここで，データ依存関係で最も上流の算術命令 x と p 個の ϕ 関数について考える．x の左右引数について合流可能性を調べ，さらに変換されて新たに生じた値番号についても再帰的に合流可能性を調べる．しかし，値番号付けに関して最小性（定義 5.6）が満たさ

表 4 $AVAIL^{all}$, $AVAIL^{some}$, $ANTIC^{all}$, $AVAIL^{wsome}$
Table 4 $AVAIL^{all}$, $AVAIL^{some}$, $ANTIC^{all}$, $AVAIL^{wsome}$.

BB	$AVAIL_{in}^{all}$	$AVAIL_{out}^{all}$	$AVAIL_{in}^{some}$	$AVAIL_{out}^{some}$	$ANTIC_{in}^{all}$	$ANTIC_{out}^{all}$	$AVAIL_{in}^{wsome}$	$AVAIL_{out}^{wsome}$
1					*9,10*	*9,10*		
2		*2,3*		*2,3*	*2,3,9,10*	*2,3,9,10*		*2,3*
3					*2,3,9,10*	*2,3,9,10*		
4			*2,3,6,12*	*2,3,6,12*	*2,3,9,10*	*2,3,9,10*	*2,3,6,12*	*2,3*
5			*2,3,6,12*	*2,3,6,12*	*2,3,9,10*	*2,3,9,10*	*2,3,6*	*2,3*
6		*2,6*	*2,3,6,12*	*2,3,6,12*	*2,3,9,10*	*2,3,9,10*	*2,3*	*2,3,6*
7			*2,3,6,12*	*2,3,6,12*	*2,3,9,10*	*2,3,9,10*	*2,3*	*2,3*
8					*9,10,14,15*	*9,10,14,15*		
9		*9,10*	*2,3,6,9,10,12*	*2,3,6,9,10,12*	*9,10*		*2,3,9,10*	*2,3,9,10*

れているので，最悪の場合でも p 個の ϕ 関数から 2 つ（2 引数であるため）を選ぶすべての組合せ，すなわち $O(p^2)$ 回だけ合流可能性を調べることになる．結果として JT の要素数は $p+O(p^2)=O(p^2)$ となる．

この作業をデータ依存関係で下流の命令へ順に続けていくと，JT の要素数は最悪の場合 $O(p^2)$，$O(p^4)$，$O(p^8)$ と増えていくので，n 個の命令では $O(p^{2^n})$ となる．これが $Hash$ を引く回数の計算量となり，すなわち最終的に割り振られた値番号の最大値は元のプログラム中の式と合わせて $O(n+p^{2^n})$ である．

繰返しアルゴリズムの計算量は束の高さに命令数を掛けたものになる．PVNRE は値番号の部分集合の集合を束として用いるため，アルゴリズム全体の計算量は最悪の場合 $O(n^2+np^{2^n})$ となる．しかし，現実のプログラムでは 1 つの命令からたかだか p 個程度 JT に要素が追加されるので，値番号の最大は $O(n+pn)$ であり，アルゴリズム全体の計算量は $O((1+p)n^2)$ となる．

(平成 15 年 12 月 20 日受付)
(平成 16 年 2 月 24 日採録)

大平 怜
2000 年東京大学理学部情報科学科卒業．現在，同大学院情報理工学系研究科コンピュータ科学専攻博士課程在籍．最適化/実行時コンパイラに関する研究に従事．他に仮想マシン，スレッドシステム，オペレーティングシステム，計算機アーキテクチャに興味を持つ．

平木 敬（正会員）
1976 年東京大学理学部物理学科卒業．1982 年同大学院理学系研究科物理学専攻博士課程修了．理学博士．1982 年通商産業省工業技術院電子技術総合研究所入所．1988 年より 2 年間 IBM 社 T.J. Watson 研究センタ客員研究員．1990 年より東京大学理学部情報科学科（現在，大学院情報理工学系研究科コンピュータ科学専攻）に勤務．現在，超並列アーキテクチャ，超並列超分散計算，並列オペレーティングシステム，ネットワークアーキテクチャ等の高速計算システムの研究に従事．日本ソフトウェア科学会会員．