依存グラフの変形に基づく コード最適化の統一的枠組み

Integrated Framework of Code Optimization Based on Transforming Dependence Graphs

滝本 宗宏
Munehiro Takimoto

東京理科大学 理工学部 情報科学科
Department of Information Sciences
Faculty of Science and Technology
Tokyo University of Science

The document is submitted to Keio University
as doctoral dissertation.

目次

図目次

表目次

第 1 章

序章

コンパイラは，計算機上で実行可能なプログラムを自然言語に近い言葉で記述したいという欲求から，プログラミング言語を機械コードへ変換するソフトウェアとして 1950 年代に誕生した．コンパイラの研究と開発は，当初，その実現に努力が注がれ，プログラミング言語の定式化と標準化とともに，実用的なコンパイラの実現に向けて基礎理論およびプログラム解析・生成技術が整備されてきた．現在では，一部を除いてほとんどの部分を自動生成することが可能になり，その自動生成ツールも広く使用されるようになっている．

一方，コンパイラが生まれた当時から，より優れた機械コードの生成を目指す研究も続けられてきた．現在，その努力はコード最適化としてコンパイラ研究の中心課題となっている．コード最適化手法は，多くの研究者によって，多種多様なものが提案されてきたが，過去 10 年の間の進歩には著しいものがある．その多くは，プログラムに記述されている計算（式や代入文）の位置を移動させること，すなわち**コード移動**（code motion）に基づく新しい最適化手法，および従来法の拡張である．

本論文では，コード移動に基づく最適化のコストの高さに注目し，より効率の良いコード最適化を実現するための共通の枠組を提案する．

本章では，まず，コンパイラにおけるコード最適化の役割について述べ，コード移動に基づく最適化の効果と問題点について述べる．最後に，本論文で提案する手法について概説する．

1.1 コンパイラとコード最適化

プログラミング言語で書かれたプログラムを機械コードに変換するシステムを，**コンパイラ**（compiler）と呼ぶ．一般には，あるプログラミング言語で書かれたプログラムを他のプログラミング言語で書かれたプログラムに変換するものをコンパイラと呼ぶこともある．コンパイラは入力として特定のプログラミング言語で書かれた**原始プログラム**（source program）を受け取り，プログラミング言語の構文と意味に基づいて，対応する**目的プログラム**を生成する．原始プログラムから目的プログラムが生成される過程は，プログラムの解析

を行う**フロントエンド**（front end）とプログラムの合成を行う**バックエンド**（back end）に大別できる．

フロントエンドは，図 1.1に示すように，**字句解析部**（lexer または lexical analyzer），**構文解析部**（parser または syntax analyzer），**意味解析部**（semantic analyzer）からなる．字句解析部は，文字の並びである原始プログラムを，意味のある文字の列，すなわち**トークン**（token）へ変換する．次に構文解析部は，構文規則に基づいて，トークンの並びから構文のパターンを見付け，構文に対応する木表現である**構文木**（syntax tree）に変換する．構文解析部が構文木を作成する際には，**意味解析部**（semantic analyzer）と連携して，字句有効範囲に基づく変数宣言の解決や型チェックなどを行う．バックエンドの入力には，構文木を用いるコンパイラも存在するが，原始プログラムを記述しているプログラミング言語に依存する部分が少ない**中間表現**（intermediate representation）を用いるのが一般的である．中間表現を用いることによって，フロントエンドとバックエンドの独立性を高めることができ，各部分の再利用性を高めることができる．

バックエンドは**コード最適化部**（optimizer）と**コード生成部**（emitter または code generator）からなる（図 1.1）．コード最適化部は，入力として受け取った中間表現に対して特別なプログラム変換を適用することによって，高速に実行できるコードを生成したり，メモリサイズの小さいコードを生成したりする役割を果たす．このような目的プログラム改良のための変換を**コード最適化**（code optimization）あるいは単に**最適化**（optimization）と呼ぶ．コード最適化部は，複数のコード最適化ルーチンで構成されることが多く，中間表現によるプログラムは，各最適化ルーチンによって順に変換される．最終的に，中間表現をコード生成部に送り，目的プログラムが生成される．目的プログラムは機械コードで書かれる場合と，アセンブリコードで書かれる場合とがある．アセンブリコードの場合には，機械コードへの変換のために，さらに**アセンブラ**（assembler）と**リンカ**（linker）による処理が必要である．

コード最適化部に含まれる数多くのコード最適化は，さらに次の 3 レベルに分けることができる．

1. 原始プログラムの記述に用いられる言語に固有な情報を利用した最適化
2. 原始プログラムにも機械コードにも依存するところが少ない汎用性のある最適化
3. 目的機械の特徴を利用した最適化

1.2 コード移動に基づく最適化

コード最適化手法は，適用対象やプログラム変換の性質などによって，さまざまに分類できる．ここでは，コード移動を行うかどうかによって最適化手法を次の 2 つに分け，各手法

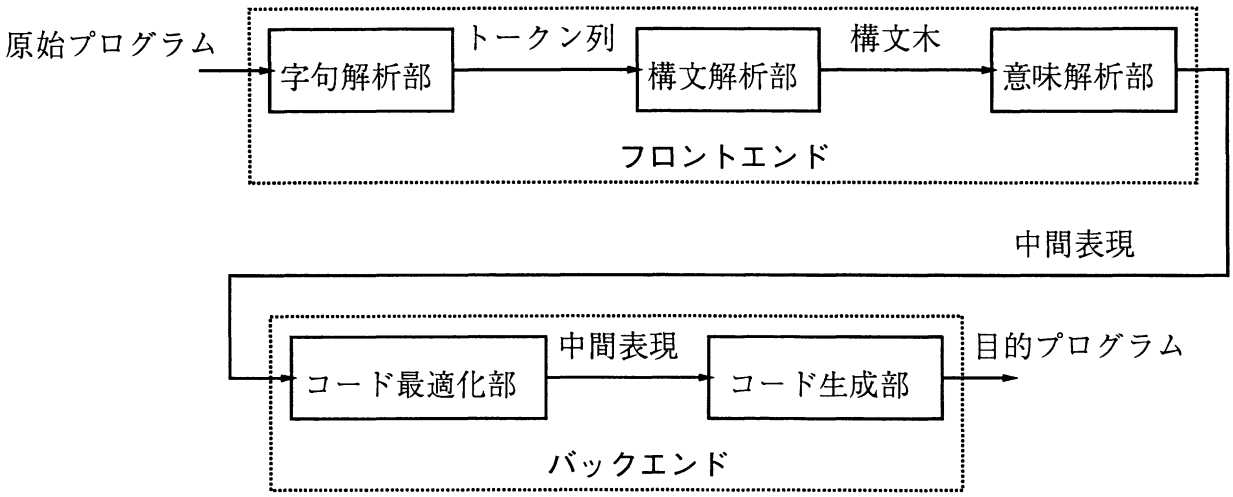


図 1.1 コンパイラの構成

```
1    x = a + b;          1
2    x = y;              2    x = y;
```

(a) 不要な計算の例　　(b) 最適化結果

図 1.2 基本最適化

の内容とその特徴を述べる.

1. コード移動を行わない基本最適化 ： 原始プログラムに記述されている式や代入文, すなわち元の計算の位置を変えることなく, 計算コストの軽減を図る最適化である. 従来, コンパイラの多くが, 基本最適化手法 [ASU86, Muc97, App98] として採用してきた手法である.

2. コード移動に基づく最適化 ： プログラムの意味を変えない範囲内でコード移動を行うことによって, 基本最適化では得られない情報を利用する最適化手法である.

近年, コード移動を行う手法の効果の高さが注目されている. コード移動に基づく最適化の多くは, 次の例に示すようにコードを移動させることによって, 移動を行わない最適化の適用範囲を広げたものと考えることができる.

例： 図 1.2(a) に示す C プログラムの断片において, 1 行目で変数 `x` に代入される `a + b` の値は, 2 行目における `x` への再代入によって以降使われることがない. このような計算

```
1     x = a + b;
2     if (p) x = y;
3     out (x);
```

図 1.3 基本最適化が効果を上げない例

```
1
2 if (p) {
3   x = a + b;
4   x = y;
5 } else x = a + b;
6 out (x);
```

(a) コード移動

```
1
2 if (p) {
3
4   x = y;
5 } else x = a + b;
6 out (x);
```

(b) 最適化結果

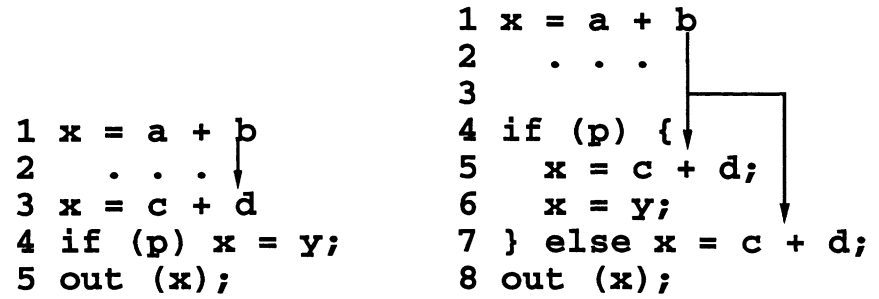
図 1.4 コード移動を用いた最適化

a + b は不要なので，図 1.2(b) に示すように変形することができる．一方，図 1.3の 1 行目の x = a + b は，条件式 p の値 が 0 でなければ，x への再代入によって不要になるが，p が 0 であれば，関数 out (x) で参照されるので，必ずしも不要ではない．したがって，x = a + b は，直接除去することはできない．

次に 図 1.3の x = a + b について，if 文へのコード移動を考える．結果として得られる図 1.4(a) のプログラムでは，3 行目の x = a + b が不要であることが分かる．コード移動は，最適化可能なプログラム点に対象を移動させることによって，より精密に最適化を適用できるようにする効果がある．結果として，図 1.4(b) のように，p が 0 以外の場合にだけ不要であった計算を除去できる． ∎

コード移動は，移動を伴わない基本最適化の適用範囲を拡張するという本来の性質に加えて，新しい最適化候補を生成するという性質をもつ．

例： 図 1.5(a) において，1 行目の式と 3 行目の式を入れ換えると，プログラムの意味が変わるので，1 行目の式は 2 行目よりも下方に移動させることはできない．しかし，図 1.5(b) のように，除去される候補として 3 行目の x = c + d が下方に移動されれば，x = a + b も移動可能になる．この例のように，最適化がさらに新たな最適化の機会を生じさせることを，最適化の**副次的効果** (second order effects) と呼ぶ．x = c + d の移動は，x = a + b を移動可能にする副次的効果を生み，最終的に x = a + b は，if 文の then 部で除去できる． ∎



(a) 副次的効果を生じる例　　(b) コード移動による副次的効果

図 1.5 コード移動を用いた最適化

コード移動に基づく最適化は，副次的効果をもたらすことによって効果を増大させる可能性をもつ．副次的効果を反映させる単純な方法は，コード移動に基づく最適化を繰返し適用することであるが，それには解析と変換コストが大きくなるという問題があった．

1.3 本研究の目的とアプローチ

本研究では，コード移動に基づくコード最適化を，効率的にかつ効果的に実現するための手法を提案する．具体的には，コード移動に基づく最適化の繰返し適用によって得られる副次的効果を直接的に反映できる枠組，すなわちそのための理論的基盤，データ構造，解析およびコード生成アルゴリズムを提案する．本枠組を用いることによって，繰返し適用を行わずにすべての副次的効果を反映したコードの生成が可能となる．

例： 図 1.5(a) の `x = a + b` の移動に関しては，その移動をブロックしている `x = c + d` に関する移動範囲の解析結果（図 1.6の点線矢印）を利用して，`x = a + b` を直接，移動後の `x = c + d` の直前に移動できるようにする（図 1.6の実線矢印）． ∎

本手法では，原始プログラムに記述されている各計算を節とし，保存しなければならない計算順序の関係を辺としたグラフ表現を用いて，移動範囲に関する解析結果を伝播させる．計算順序の関係は，計算の移動によって変化する場合があるので，グラフ表現も各プログラム点への移動後に対応させて，異なる構造が必要になる場合がある．そのために本研究では，同じ値の計算を表すグラフ表現を 1 つの節で表現する**拡張値グラフ**（extended value graph）と呼ぶ新しいグラフ表現を提案する．

従来，提案されてきた各種のコード移動に基づく最適化は，拡張値グラフ上の共通の枠組によって実現できるようになる．

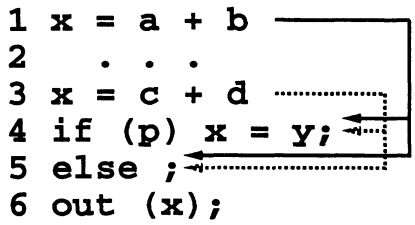


図 1.6 移動範囲の情報の利用

詳細は，第 2 章以降で述べるが，本研究の特色は，従来の研究に比べて次の 3 点に要約できる．

1. 従来，繰返し適用しなければ解析できなかったコード移動の可能な範囲を 1 回の解析で求めることができる．

2. その結果として，コード移動と基本最適化との組合わせによる効果を提案手法の 1 回の適用によって得ることができる．

3. 各プログラム点について，コード移動を行ったときに得られる最適化の効果を事前に予測できるので，従来法では達成できなかった最適化の効果を得ることができる．

1.4 本論文の構成

次章以降の本論文の構成は，次のとおりである．

第 2 章で，本提案手法を適用する場合の前提条件，および本論文で用いる用語や記法の説明をしたあと，関連研究を述べ，従来法との比較によって本研究の位置付けを明確にする．

第 3 章で，コード移動に基づく最適化手法の問題点を述べ，本提案手法について概説する．

第 4 章で，計算順序を保存するために必要な計算の依存構造を表すグラフに対して，等価変換の変形規則を導入することによって，各プログラム点において対応するグラフ構造を構築できることを示す．次に，依存構造を表すグラフとその変形規則を用いることによって，従来から行われてきた等価式の検出と定数畳込みが精密にかつ効率良く行えることを示す．

第 5 章で，変形によって構造が変化する依存構造を 1 つのグラフとして表現するための拡張値グラフを導入し，その性質を説明する．

第 6 章で，繰返しによる適用が効果を上げる例を示しながら，拡張値グラフを用いたコード移動に基づく最適化の実現法を述べる．

第 7 章で，本手法を実装したコンパイラを用いて行った評価を示し，本手法の有用性を示す．そして，最後に結論を述べる．

第 2 章

コード最適化と関連研究

本章では，本研究を説明するための準備として，まず以降で使用する前提や用語について説明する．次に，従来から頻繁に使用されている主要な基本最適化を紹介し，コード移動によって，基本最適化がどうのように拡張されるかを述べる．そのあと，本研究の位置付けを明確にするために，従来研究されてきた多くのコード移動に基づく最適化の効果と計算量の関係について，関連研究を総括して述べる．

2.1 中間表現と三アドレスコード

コード最適化部が入力および出力とする中間表現は，原始プログラムに近いものから機械コードに近いものへ順に示すと，一般に**高水準中間表現**（high-level intermediate representation，以降，HIR と呼ぶ），**中水準中間表現**（medium-level intermediate representation，以降，MIR と呼ぶ），**低水準中間表現**（low-level intermediate representation，以降，LIR と呼ぶ）に分けることができる．各中間表現に基づく最適化の特徴は，次のとおりである．

HIR レベルにおける最適化 ：原始プログラムのレベルにおいて，一連の操作を代数的に等価な高速の操作の列で置き換え，プログラムの実行を大幅に短縮する方法がある．例えば，データベース問合せ言語では，代数的変換の応用による実行時間の短縮が一般に行われている．このレベルの最適化は，プログラミング言語（以降，単に言語と呼ぶ）の性質に大きく依存することから，言語依存の最適化であるといえる [ASU86].

MIR における最適化 ：原始プログラムレベルや機械コードレベルの最適化に対して，コンパイラが内部で生成する原始プログラムの中間表現を対象とする最適化である．このレベルの最適化は，言語にも機械にも依存しない中間言語を選択することによって，移植性の高い最適化を実現することができる．

LIR レベルにおける最適化 ：コード生成部で生成された目的コードを対象として，命令列の再編成を行う最適化である．例えば，RISC プロセッサに対する最適化として，多

```
prod = 0;
i = 1;
do {
   prod = prod + a[i] * b[i];
   i = i + 1;
} while (i <= 20)
```

(a) 内積を求める C プログラム

```
1     prod := 0
2     i := 1
3     t1 := 4 * i
4     t2 := a + t1
5     t3 := * t2
6     t4 := 4 * i
7     t5 := b + t4
8     t6 := * t5
9     t7 := t3 * t6
10    t8 := prod + t7
11    prod := t8
12    t9 := i + 1
13    i := t9
14    if i <= 20 goto 3
```

(b) 三アドレスコード

図 2.1 MIR

くの実行サイクルを必要とする命令列を，実行サイクルの少ない命令列で置き換えることがよく行われる．このレベルの最適化は，目的コードが実行される機械のアーキテクチャに依存する．

本研究では，C や Pascal などに代表される命令型プログラミング言語を対象として，特定のプログラミング言語にも，また目的プログラムを記述する機械語にも依存しない MIR によるコード最適化を対象とする．MIR では，演算の種類を表す命令コードと，3 つのオペランドからなる仮想の機械コードによるプログラム表現が一般的である．その命令形式を一般には，**三アドレスコード**（three address code）[ASU86] と呼ぶ．以降，説明を容易にするために，三アドレスコードの 1 つの命令を**文**（statement），その右辺を**式**（expression）と呼ぶ．特に区別の必要がない場合は，単に**計算式**（computation）と呼ぶことにする．

例： 図 2.1(a) の C プログラム断片に対する MIR を図 2.1(b) に示す．図 2.1(b) 中の `t1`, `t2`, ... のように `t` で始まる名前の変数は，右辺の式の値を一時的に格納するためにコンパイラが生成したものである．この変数を**一時変数**（temporary）と呼ぶ．単項演算子の `*` は，オペランドが示すアドレスに対する間接参照を意味する． ∎

各文は，次の 3 種類に分類する．

代入文 ： 三アドレスコード $v := t$ の形で表現する．ここで，v は変数であり，t は演算子を高々 1 つ含む式とする．

空文 ： 操作を必要としない文．

重要文 ：メモリ操作を伴う文，関数呼出し，分岐文など，移動によってプログラムの意味が変わってしまう文．重要文（relevant statements）は元のプログラム点に固定のものすなわち，コード移動の対象にはならない文とする．また，重要文で用いられる変数は不要でないものとする．本稿では，重要文であることを明示するために，出力文の形で $out(t)$ という表現を用いる [KRS94b]．

2.2 制御フローグラフ

各原始プログラムの MIR に対しては，**制御フローグラフ**（control flow graph，以降 CFG と呼ぶ）が作成されているものとする．CFG は，次に示す節と辺を用いて，四つ組 $(N, E, \mathbf{s}, \mathbf{e})$ によって表すことができる．

N ： 途中に分岐や飛先ラベルをもたない文の並びを一般に基本ブロック[1]と呼ぶが，基本ブロックに対応する節の集合

E ： 実行の流れを表現するための有向辺の集合（$E \subset N \times N$）

$\mathbf{s}$ ： プログラムの実行開始点を表す特別な節である開始節（$\mathbf{s} \in N$）

$\mathbf{e}$ ： プログラムの実行終了点を表す特別な節である終了節（$\mathbf{e} \in N$）

例： 図 2.1(b) の CFG に対応する図を図 2.2に示す．図 2.2では，分岐命令や飛先ラベルを明示しているが，これらの情報は辺によって表現されるので，以降省略する． ∎

逐次実行される複数の文をまとめて 1 つの基本ブロックとして扱う代わりに，1 つの文を CFG の節として扱うことがある．以降，CFG と呼ぶ場合には，1 つの基本ブロックが節に対応するものとする．

CFG は，プログラム全体を解析する場合に，制御の流れ（以降，制御フローと呼ぶ）を明示する役割をもつ．制御フローには，関数呼出しによって生じるフローの関係も含まれるが，本研究では，CFG は各関数定義ごとに作成するものとし，関数どうしの関係は扱わない．$\mathbf{s}$ は，関数の入口を表し，$\mathbf{e}$ は関数の出口を表す．関数本体中に現れる関数呼出しは，通常の演算と同様に扱い，呼び出される側の関数本体での制御フローは考慮しないものとする．

[1] ラベルは基本ブロックの先頭に現れ，分岐は基本ブロックの最後に現れる．

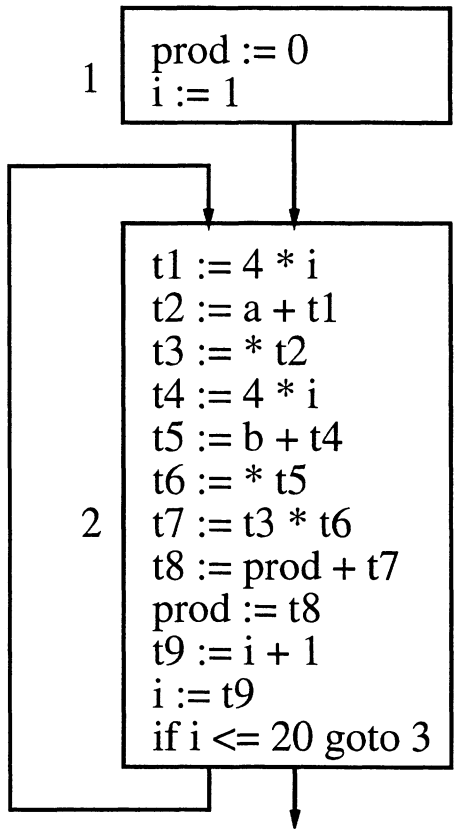


図 2.2 基本ブロックと制御フローグラフ

2.2.1 データフロー解析

プログラムにおける式や変数などの性質に関して，解析をする場合，その性質が各プログラム点ごとに独立に決定できる例は稀である．通常は，他のプログラム点からの影響を考慮して，プログラム全体から情報を集め，さらにその情報を各プログラム点へ伝播させる必要がある．この制御フローに従って式や変数などのデータに関する情報をプログラム全体にわたって収集するための方法として，一般に**データフロー解析**（dataflow analysis）と呼ばれる手法が用いられる．データフロー解析は，各プログラム点について収集した情報を，その情報が影響を及ぼすプログラム点へ伝播させることによって実現される．それぞれの情報は，各プログラム点において定義される**データフロー関数**（dataflow function）と呼ばれる関数によって求められ，そこで得られた情報はさらに次のプログラム点に伝播される．以降，この伝播される情報をフロー情報と呼ぶ．

データフロー関数の定義と伝播の仕方は，情報の集合を変数で表し，それらの間の関係を等式として表記した**データフロー方程式**（dataflow equation）によって記述するのが一般的である．データフロー方程式においては，各プログラム点での情報を，そこで生成される情報と隣接するプログラム点から得られる情報を用いて定義することによって，情報の伝播

の仕方が規定される．

データフロー解析は CFG 上で実現されることが多い．この場合，プログラム点として CFG 節を用い，フロー情報の伝播先に当たるプログラム点として，CFG 辺によって繋がれた節を用いる．CFG 節 n に対して，n の直前のプログラム点を表す CFG 節集合を**先行節**（predecessor）と呼び，$pred(n)$ で表す．また，直後のプログラム点を表す節集合を**後続節**（successor）と呼び，$succ(n)$ で表す．本稿では，プログラム点として，基本ブロック（複数の文）を用いる場合と，単一の文を用いる場合があるので，$pred(n)$ と $succ(n)$ も，基本ブロックの集合を表す場合と，文の集合を表す場合とがある．

一般に，各文を 1 つの CFG 節とするデータフロー解析は解析が容易であり，解析の効率に関しては，文よりも基本ブロックを CFG 節とするデータフロー解析の方がすぐれている．以降で，まず単一の文を CFG 節とする場合について，CFG 上でのデータフロー解析の定義と解法を述べ，その後，基本ブロックを CFG 節とするデータフロー解析に拡張する．

文をプログラム点としたときの解析

まず，1 つの文を CFG 節とするデータフロー解析の例を示す．

例： あるプログラム点において，それよりもあとの実行によって参照される変数は，その点において**生きている**と呼び，プログラム中の各プログラム点において生きている変数を求めることを**生存解析**（liveness analysis）と呼ぶ．文 n の直前において生きている変数の集合を $LIVE(n)$，n において実際に参照されている変数の集合を $USED(n)$，n における代入先の変数を集合 $DEF(n)$ として表したときの，生存解析のデータフロー方程式を図 2.3に示す．この方程式は，次のことを意味している．

1. プログラム点 n で生きている変数の集合 $LIVE(n)$ は，n に続く文の手前で生きてる変数の集合 $\bigcup_{s \in succ(n)} LIVE(s)$ から，n で代入される変数の集合 $DEF(n)$ を除いたもの，または

2. n で使用される変数の集合 $USED(n)$ を加えたものである．

$LIVE(n)$ は，上述の関係に基づいて，繰返しによって求められるために，その初期値は，$\emptyset$ とする．∎

上の例で示したデータフロー解析においては，そのデータフロー方程式が示すように，フロー情報を後続節から先行節へと伝播させる必要がある．このような制御フローと逆方向のフロー情報を扱うデータフロー解析を，**後向きデータフロー解析**と呼ぶ．また，制御フローと同じ方向のフロー情報を扱うデータフロー解析を**前向きデータフロー解析**と呼ぶ．

- すべての $LIVE(n)$ を $\emptyset$ に初期化しておく.

$$LIVE(n) \quad =_{def} \quad USED(n) \cup (\bigcup_{s \in succ(n)} LIVE(s) \setminus DEF(n)) \tag{2.1}$$

図 2.3 集合表現による変数の生存解析

- すべての $LIVE(n)$ を $false$ に初期化しておく.

$$LIVE(n) \quad =_{def} \quad USED(n) \vee (\sum_{s \in succ(n)} LIVE(s) \wedge \neg DEF(n)) \tag{2.2}$$

図 2.4 論理値表現による変数の生存解析

データフロー解析を実現する際には，データフロー方程式を満たす集合を求める代わりに，解析対象を 1 つに限って，それがデータフロー方程式に示された関係を満たすかどうかを決定する問題として取り扱うのが一般的である．この場合，個々の解析対象は，**データスロット**（data slot）と呼ばれる論理変数として表現でき，解析対象全体はスロットの列として表現できる．すなわちデータフロー解析の実現においては，全体をビット列として，各スロットはその列中のビット位置として表現できる．

例： 図 2.3に示した集合表現による生存解析に対して，論理式表現のデータフロー方程式を図 2.4に示す．ここで，$\sum$ は，その後に続く述語が 1 つでも $true$ であれば $true$，すべて $false$ であれば $false$ になる論理演算を表す．これ以降，述語がすべて $true$ であれば $true$ となり，1 つでも $false$ であれば $false$ となることを表すのに $\prod$ を用いる． ∎

以下では，ある解析対象に対してデータスロット s が割り当てられているものとして，後向きデータフロー解析に基づいた論理値表現によるデータフロー方程式の解法を示す.

1. ワークリストを用意して，各プログラム点 n とその点に対応するデータスロット s を組 (s,n) にして，ワークリストに加える.

2. 以下をワークリストが空（$\emptyset$）になるまで繰り返す.

   (a) ワークリストからデータスロットとプログラム点の組 (s,n) を 1 つ取り出し，プログラム点 n におけるデータスロット s の値をデータフロー方程式によって計算する.

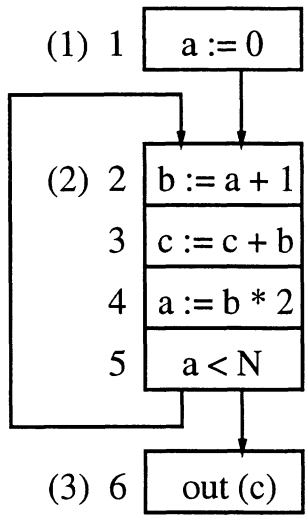


図 2.5 変数 a, b, c の生存解析

(b) その結果が，現在のデータスロットの値と異なった場合は，n におけるデータスロットの値を新しい値に変更し，データスロット s と，先行するプログラム点 $pred(n)$ （前向きデータフロー解析の場合は $succ(n)$）のそれぞれを組にしてワークリストに加える．

このデータフロー方程式の解法を，**スロットワイズ**（slot wize）法の解法と呼ぶ [DRZ92, KD94].

例： 図 2.4に示す方程式に基づいて，図 2.5のプログラムの生存解析を行うと，次の表に示す結果が得られる．このプログラムで使用されている変数は a, b, c であるので，各プログラム点での生存変数を表すために 3 つのスロット，すなわち 3 ビットのビット列を用意する必要がある．

文の番号	a のスロット	b のスロット	c のスロット
1	false	false	true
2	true	false	true
3	false	true	true
4	false	true	true
5	true	false	true
6	false	false	true

基本ブロックをプログラム点としたときの解析

データフロー解析は，各プログラム点のフロー情報に変化がなくなるまで，繰返し計算をする必要があるので，複数の文をひとまとめにした基本ブロックを 1 つの CFG 節とする

- すべての $N\text{-}LIVE(n)$ と $X\text{-}LIVE_{(n)}$ を $false$ に初期化しておく.

$$N\text{-}LIVE(n) =_{def} USED(n) \vee X\text{-}LIVE(n) \wedge \neg DEF(n) \quad (2.3)$$

$$X\text{-}LIVE(n) =_{def} \sum_{s \in succ(n)} N\text{-}LIVE(s) \quad (2.4)$$

図 2.6 基本ブロック単位での変数の生存解析

ことができれば，解析の効率を向上させることができる．基本ブロックを CFG 節とする場合，基本ブロックは一般に複数の文からなるので，その入口と出口におけるフロー情報が必要になる．

例：基本ブロック n の入口，出口において生きている変数の集合をそれぞれ $N\text{-}LIVE(n)$, $X\text{-}LIVE(n)$ として，図 2.4のデータフロー方程式を書き換えたものを図 2.6に示す．複数の文をひとまとめにしているので，その他の述語についても，意味を修正しなければならない．$USED(n)$ は，基本ブロック n において参照されていて，かつ n の入口からその参照までの間で代入が行われることがない変数の集合を意味する．$DEF(n)$ は，基本ブロック n において代入が行われる変数の集合を意味する． ∎

基本ブロックを CFG 節とする場合も，文を CFG 節とする場合と同様にスロットワイズ法によって解くことができる．基本ブロックを節とする場合は，文を節とする場合の解法に，入口と出口での扱いが加わわったものとなる．後向きデータフロー解析に基づく解法を次に示す．

1. ワークリストを用意しておき，各プログラム点 n について，n とその出口（前向きデータフロー解析の場合は入口）におけるデータスロット s を組 (s,n) にして，それをワークリストに加える．

2. 以下をワークリストが $\emptyset$ になるまで繰り返す．

   (a) ワークリストからデータスロットとプログラム点の組 (s,n) を 1 つ取り出し，プログラム点 n におけるデータスロット s の値から，データフロー方程式を用いて，入口（前向きのデータフロー解析の場合は出口）における s の値を計算する．

   (b) その結果が，現在のデータスロットの値と異なった場合は，データスロットの値を新しい値に変更し，先行するプログラム点 $pred(n)$（前向きデータフロー解析の場合は $succ(n)$）の各出口（前向きデータフロー解析の場合は入口）における

s を新しい値に変更した後，各 $pred(n)$（前向きデータフロー解析の場合は $succ(n)$）と s を組にしてワークリストに加える．

例： 図 2.6に示すデータフロー方程式に従って図 2.5のプログラムを解くと，次の結果が得られる．各基本ブロック n の $N\text{-}LIVE(n)$ は，n の先頭の文に対する $LIVE$ の結果と一致する．

基本ブロックの番号	a のスロット（入口，出口）	b のスロット（入口，出口）	c のスロット（入口，出口）
(1)	(false, true)	(false, false)	(true, true)
(2)	(true, true)	(false, false)	(true, true)
(3)	(false, false)	(false, false)	(true, false)

2.2.2 SSA 形式

以降では，計算式 c において，変数 x がオペランドとしてあるいは実引数として出現する場合，c は x を**使用する**といい，その c のことを x の**使用**（use）という．また，ある変数 x を代入先にもつ計算式 c において，c は x を**定義する**といい，その c のことを x の**定義**（definition）という．

本論文で対象とするプログラムの中間表現においては，各文は**静的単一代入形式**（static single assignment form，以降 **SSA** 形式と呼ぶ）に変換されていることを前提とする．SSA 形式とは，次の性質を満たすプログラムの表現形式である．

1. すべての変数の使用は，唯一の定義をもつ．すなわち，1 つの変数への代入は 1 箇所でしか行われることがない．

2. 元のプログラムで用いられているそれぞれの変数に関して，異なる辺から 2 つ以上の定義が使用に達する場合は，ϕ- **関数**と呼ぶ仮想関数を用いて，フローの合流点で定義の結合を明示する．そのような定義に関しては，フローが合流する CFG 節ごとに固有の ϕ- 関数を用いて各定義を区別する．以降，その集合を Φ で表す．

SSA 形式への変換には，変数の付替えと ϕ- 関数の挿入が必要になる．任意の原始プログラムに対する CFG から SSA 形式への変換については，効率的なアルゴリズム [CFRW91, SG95, App98] が知られている．

例： 図 2.7(a) に示す通常形式のプログラムに対する SSA 形式を図 2.7(b) に示す．同じ名前で表現されている変数は，定義ごとに変数名の後ろに数字を付けて区別する．定義が結合する箇所には，ϕ- 関数による代入を挿入し，定義の結合を明示的に表現している．例え

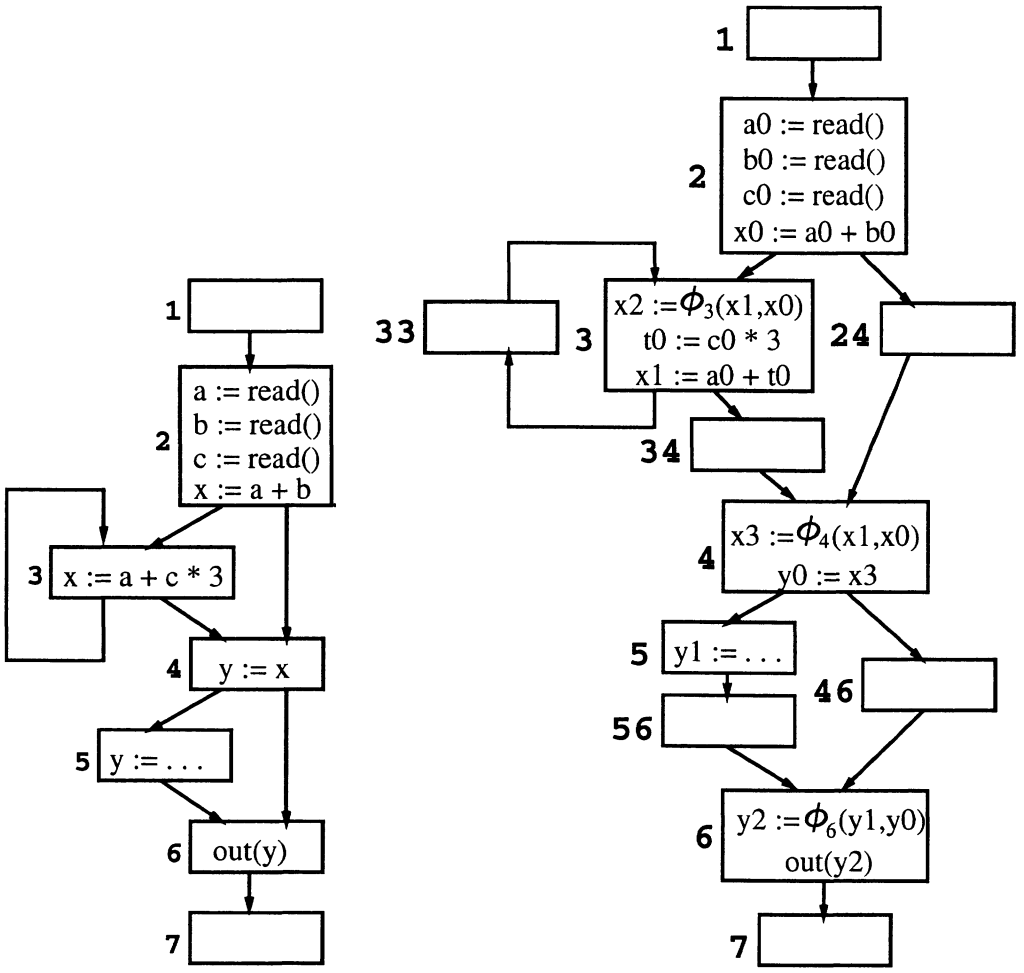
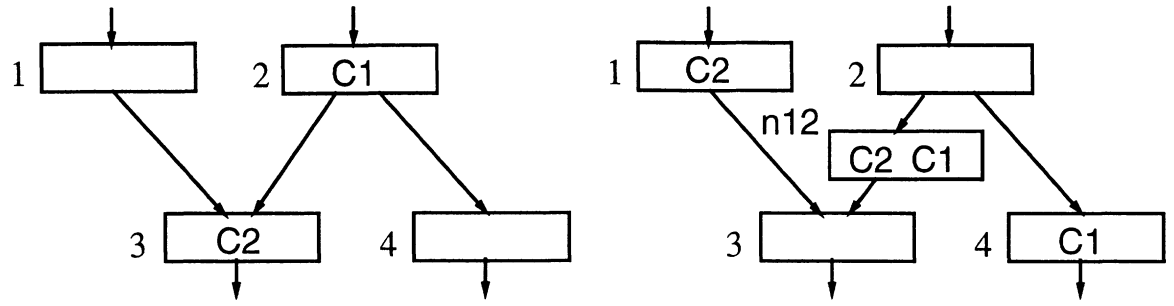


図 2.7 入力プログラムの表現

ば，節 3 の `x2 := φ3 (x1, x0)` は，節 2 の定義 `x0 := a0 + b0` と 節 3 の定義 `x1 := a0 + t0` が節 3 の入口で結合することを表している． ∎

以下，説明を簡単にするために，一般性を失うことなく ϕ- 関数の引数は 2 つとする．すなわち，CFG において，1 つの節に入ってくる辺は 2 つまでとする．

また，図 2.8(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[KRS94a, KRS92]）は取り除かれているものとする．理由は，有効な移動がクリティカル辺によってブロックされることがあるからである．例えば，図 2.8(a) において節 2 の計算 $C1$ を前向きに移動させようとすると，節 1 と節 3 を通る実行パス上に新しく $C1$ の計算を挿入してしまうことになるので，この移動は，ブロックされる．また，節 3 の計算 $C2$ を後向きに移動させようとするとき，同様に節 2 と



(a) クリティカル辺が存在する CFG (b) クリティカル辺を除去した CFG

図 2.8 クリティカル辺の除去

節 4 を通る実行パス上に新しく $C2$ の計算を挿入してしまうので，この移動もブロックされる．この場合には，図 2.8(b) のように，新しい CFG 節 n_{12} を挿入することによってクリティカル辺を除くことができ，さらに最適化の効果を上げることができる．

本稿では，2 つ以上の先行節をもつ節に入ってくる辺は，すべて新しく節を挿入することによって分割されているものとする．この変形によって，クリティカル辺は除去でき，後の解析を単純にすることができる[2][KRS92].

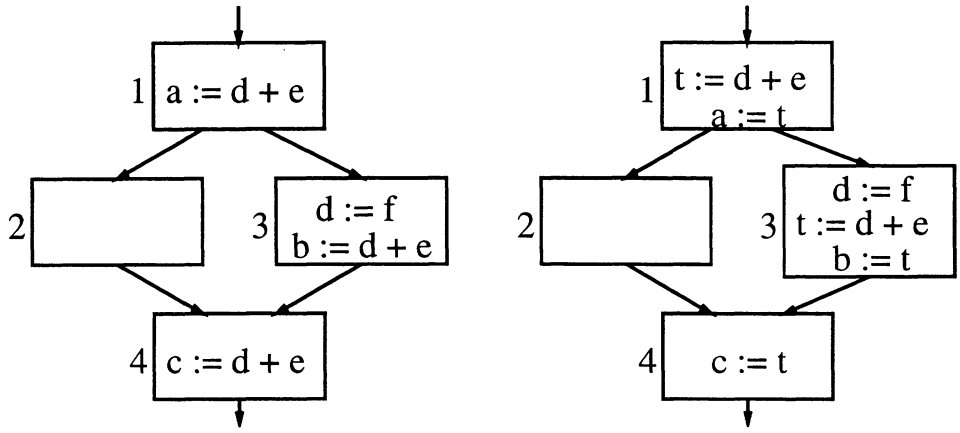
2.3 基本最適化

原始プログラムにおける計算位置を変えずに行う最適化は，実装が容易であり，最適化に要するコストも小さく抑えられるという特徴をもつ．その中でも，従来から広くコンパイラの最適化として採用されてきた代表的な手法に**共通部分式の除去**，**不要コード除去**，**定数畳込み**，**演算子の強さ軽減**がある．以下に，これらの手法を紹介する．

2.3.1 共通部分式の除去

原始プログラムにおいて，あるプログラム点 p に式 e が現われ，p に先行するプログラム点 p' にも同じ式 e が現われるとき，e で使用している変数の値が p' から p に至るまでの実行パス上で変更されることがなければ，式 e はその実行パス上で**冗長**（redundant），あるいは p の e は p' の e に対して冗長であるという．プログラムの開始点から，あるプログラム点における式に到達するすべての実行パス上で式 e が冗長であるとき，それを**共通部分式**（common subexpression）という．共通部分式は，先行する計算式の値を保存しておき，後でその値を利用することによって除去でき，元のプログラムの意味を変えずに実行時

[2]説明を簡単にするために，本稿の図において，不要な節は省略している．



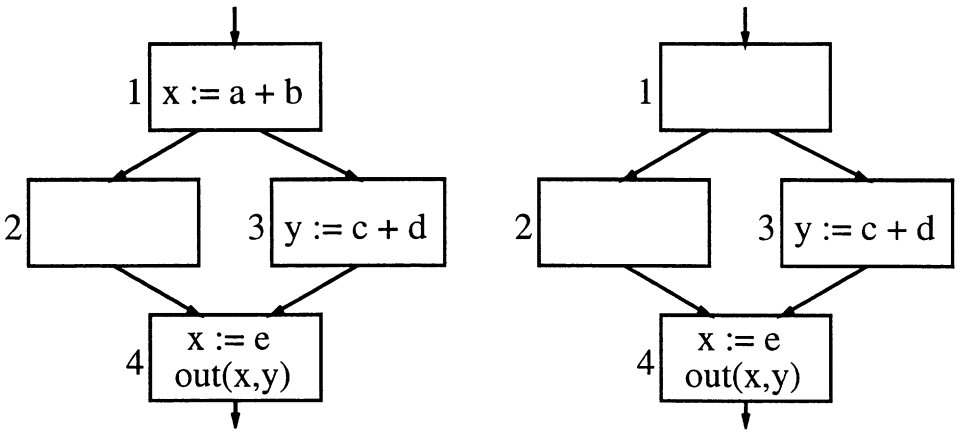
(a) 共通部分式　　(b) 共通部分式の除去

図 2.9 共通部分式の除去

の効率を上げることができる．これを共通部分式の除去（common subexpression elimination）という．

例： CFG で表現した図 2.9(a) において，節 1，3 の式 `d + e` と節 4 の式 `d + e` は共通部分式である．節 1，2，4 を通る実行パスを [1−2−4] と表記することにすると，節 4 の式 `d + e` は，[1−2−4] 上で，節 1 の式 `d + e` に対して冗長であり，実行パス [1−3−4] 上で，節 3 の式 `d + e` に対して冗長である．したがって，図 2.9(b) のように，一時変数 `t` を導入して，節 1，3 における `d + e` の各値を `t` に保持させ，節 4 の式の右辺を `t` で置き換えることによって，`d + e` の冗長性を取り除くことができる．このとき，実行パス [1 − 3 − 4] 上では，節 3 に `d := f` があるので，節 3 の `d + e` も節 4 の `d + e` も節 1 の `d + e` に関して冗長でないことに注意しなければならない． ∎

共通部分式の除去を行うと，一般に，図 2.9(b) の `c := t` のように，新しいコピー代入を導入することになる．このようなコピー代入は，変数 `c` のオペランドあるいは実引数としての出現を `t` で置き換えていく最適化，すなわち**コピー伝播**（copy propagation）[ASU86] や，変数に対してできるだけ多くのレジスタ割付けを行うアルゴリズム，すなわち**レジスタ彩色手法**（register coloring）[CH90] で用いられる**変数合体**（coalescing）によって取り除かれることが期待できる．



(a) 不要な代入文 (b) 不要な代入の除去

図 2.10 不要コードの除去

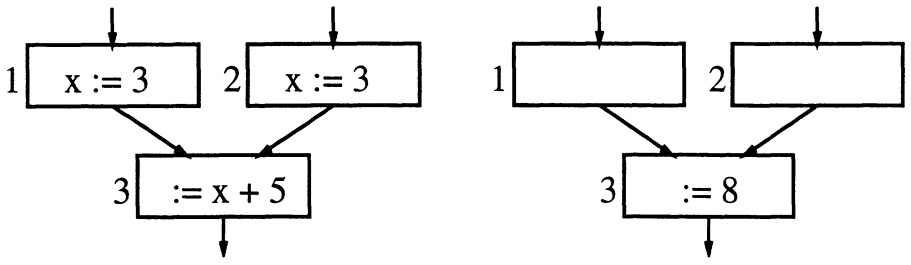
2.3.2 不要コード除去

プログラムのある点 p において，変数 x の値がその後の実行で使用されることがなければ，x は p で**死んでいる**（dead）という．変数の生死と同様の考えに従って，文やコードが，以後の実行において決して使用されることがない値を計算するのであれば，それらは死んでいる，あるいは**不要である**という．使用者が故意に不要なコードを書くことはないにしても，コンパイルの過程では，プログラムに記述されたコードが不要になることがある．この不要な計算を除去する最適化を不要コード除去（dead code elimination）という．

例： 図 2.10(a) において，節 1 の代入文 `x := a + b` は不要である．節 4 の `out` の引数として使用されている `x` の値は，その直前で実行される `x := e` によって上書きされるので，`out` に使用されるのは，`e` の値であり，節 1 の `x := a + b` による代入の結果が使われることはない．したがって，この代入文は除去することができ，図 2.10(b) に示すように変形することができる．一方，節 3 の `y := c + d` は `out` によって使用されているので，除去することはできない． ∎

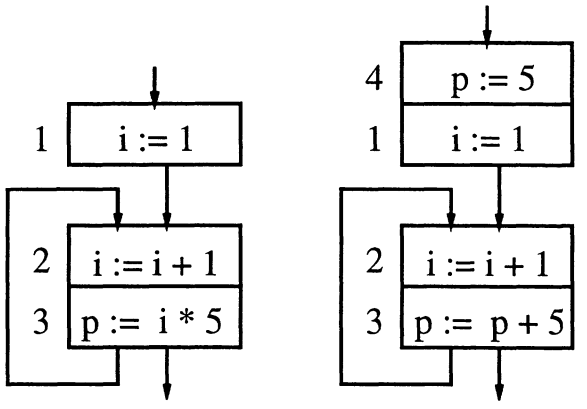
2.3.3 定数畳込み

計算式や関数の引数がすべて定数によって構成される場合には，その計算をコンパイル時に行うことによって，元の式や関数呼出しを定数で置き換えることができる．この最適化を定数畳込み（constant folding）という．定数畳込みは，定数伝播と組み合わせることによって，さらに定数畳込み可能な式を生成す可能性がある．



(a) 定数に畳み込めない例　　(b) コピー伝播後の定数畳込み

図 2.11 定数込畳みとコピー伝播



(a) 演算子の強さ軽減が適用可能な例　　(b) 演算子の強さ軽減の効果

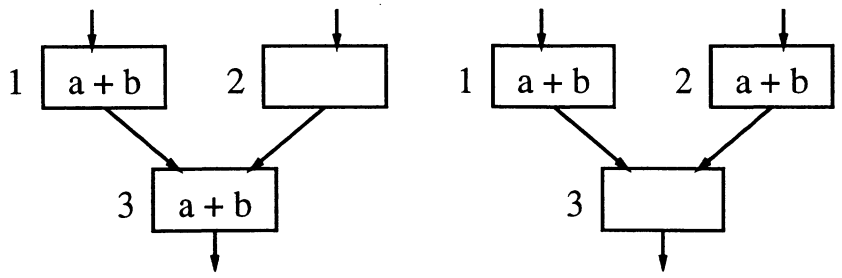
図 2.12 演算子の強さ軽減

例： 図 2.11(a) において，節 1 と節 2 の `x := 3` を節 3 に伝播させると，`x + 5` は `3 + 5` となる．これに定数畳込みを適用すると，図 2.11(b) に示すように `3 + 5` は定数 `8` によって置き換えることができる． ∎

2.3.4 演算子の強さ軽減

式の代数変形を利用して，演算子をよりコストの低いものに置き換える手法を演算子の強さ軽減（strength reduction）と呼ぶ．

例： 図 2.12(a) において，節 3 のオペランド `i` は，節 2 の `i + 1` の計算によって求められた値であることが分かる．節 2 で代入が行われる前の `i` を `i'` で表すと，節 3 の `p = i * 5` は，`p = (i' + 1) * 5` に書き換えることができる．式 `(i' + 1) * 5` は，`i' * 5 + 1`



(a) 部分冗長　　(b) 部分冗長除去

図 2.13 部分冗長と冗長への変形

`* 5` で置き換えることができるので，節 3 で代入される前の `p` の値が `i' * 5` であることを考慮すると，`p := p + 5` に変形することができる．節 4 に `p` の初期化 `p := 5` を挿入して，最終的に図 2.12(b) に示す結果を得ることができる． ∎

演算子の強さ軽減は，実行コストの低減ばかりでなく，ループ内の最適化を行う際の解析を容易にする効果ももつ．

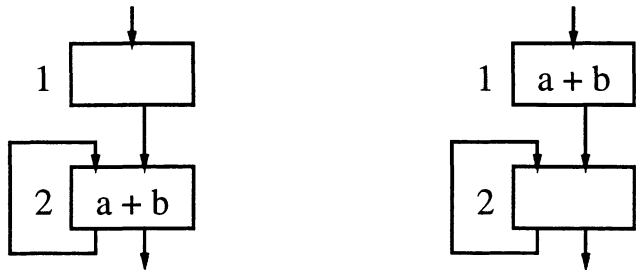
2.4 コード移動に基づく最適化

コード最適化の中には，計算式を原始プログラムの計算位置から移動させることによって効果を得る手法がある．コード移動が直接的に最適化効果を上げるものとしては，**ループ不変コード移動** (loop-invariant code motion) が代表的である．この最適化は，ループ内で値が変化しない式をループの外に移動することによって，式の計算回数を低減させるものであり，コード移動自体が最適化となる性質のものである．

一方，基本的なコード最適化の中には，コード移動との組合わせによって，最適化の効果を高めることができるものがある．コード移動には，組み合わせる最適化手法の性質によって，制御フローと同じ方向の**前向き** (forward) と制御フローと逆方向の**後向き** (backward) の 2 方向がある．後向きのコード移動と組み合わせた代表例として，冗長除去を拡張した**部分冗長除去**があり，前向きのコード移動と組み合わせた代表例として，不要コード除去を拡張した**部分不要コード除去**がある．以下，これらの方法について詳しく述べる．

2.4.1 部分冗長除去

プログラムの開始点から，ある特定の式に到達するすべての実行パス上に同じ計算式が存在する場合には，共通部分式の除去によって冗長な計算を除去できる．このような冗長性を特に**全冗長** (totally redundant) という．これに対して， 実行パスによって部分的に計算



(a) ループ不変式　　(b) 部分冗長除去による効果

図 2.14 ループ不変コード移動

が冗長になる場合もある.

例： 図 2.13(a) において，左上の基本ブロックからの実行パス上では，`a + b` が 2 回実行される．しかし，下側の式を除去してしまうと，右上からの実行パス上に `a + b` の式がなくなってしまうので，下側の `a + b` は冗長ではない. ∎

この例のようにすべての実行パス上で冗長ではないが，幾つかの実行パス上で冗長である式は**部分冗長**（partially redundant）であるという．部分冗長な式は，先行して同じ値を計算する式が存在しない実行パス上に式を挿入することによって，全冗長なものに変えることができる.

例： 図 2.13(a) において，右上の基本ブロックに `a + b` を挿入することによって，下側の `a + b` は全冗長になり，除去可能になる（図 2.13(b)）. ∎

部分冗長性を取り除くための挿入点と冗長な式の決定は，部分冗長の考えを基にしたデータフロー解析によって実現することができる．データフロー解析に基づいて部分冗長性を取り除く手法を部分冗長除去（partial redundancy elimination，以降 **PRE** と呼ぶ）という [DP93, MR79, KRS94a, KRS92]. PRE は，共通部分式の除去に加えてループ不変コード移動をも統一的に扱うことができる.

例： 図 2.14(a) において，ループの直前の節に `a + b` を挿入することによって，ループ内部の `a + b` は全冗長となる．すなわち，ループ内部の `a + b` は部分冗長であり，PRE によってループ不変コード移動が実現できる. ∎

式を上方に挿入して冗長な式を除去するという操作は，**コード巻上げ**（code hoisting）を行うことに等しい．以降，挿入とそれに伴う除去をコード巻上げ，または単に**巻上げ**と呼

ぶ.

PRE では，式を巻き上げた後の挿入点の決定にデータフロー解析が用いられる．ここでは，プログラム中のある式 e を解析対象とし，その適切な巻上げを行うことができるプログラム点の決定を考える．この解析に用いられるデータフロー方程式として，[KRS92] によって提案されたものを例に図 2.15に示す．図 2.15では，前述したようにデータフロー方程式を簡略化するために，1 つの文を CFG 節と仮定している．このとき，最適な巻上げ点は，*Last-INSERT(m)* = *true* となるプログラム点 m として求められる．PRE は，データフロー解析を行う前に，式 e について，各節 n における次の 2 種類の局所述語を求めておく必要がある．各局所述語の意味は，次のとおりである．

Used(n) ：式 e が n に存在していれば *true* となる．

Transp(n) ：式 e のオペランドの値が n で変更されことがなければ *true* となる．

これらの述語は，各節における情報だけで決定することができる．[KRS92] におけるデータフロー方程式の解を求めることは，次の 2 種類のコード移動を意味する．

多忙コード移動（busy code motion） ：プログラムの意味を変えずに，式 e を移動できるプログラム点のうちで最も開始節に近いプログラム点へ移動

遅延コード移動（lazy code motion） ：多忙コード移動によって除去できる式 e の数を減らさずに，巻き上げた式を巻き戻すことが可能なプログラム点のうちで最も終了節に近いプログラム点へ移動

PRE における冗長性の除去は，多忙コード移動を行うだけでも実現できる．ただし，その場合には，式の値を保持しておくために一時変数の導入が必要となるので，一般には，変数の生存期間が長くなり，本来ならレジスタに割り付けられるはずの変数がメモリに置かなければならない可能性が出てくる．この理由から，多忙コード移動による効果を維持したままで，不必要な移動を避けるために，その結果を初期値（局所述語の値）として遅延コード移動を行う．多忙コード移動によって冗長性の除去がまったく行われなかった式については，遅延コード移動によって，多忙コード移動を適用する前のプログラム点に巻き戻されることになる．図 2.15に示す各述語の意味は次のとおりである．

D-SAFE(n) ： どの実行パスにも新しく式を挿入することなく，後向きに e を節 n に移動することができれば，*true* となる．

EARLIEST(n) ： 開始節から，データフロー解析後の *D-SAFE(n′)* が *true* となった節 n' までの範囲に節 n が含まれていれば，*true* となる．

$$D\text{-}SAFE(n) = \begin{cases} false & if\ n = \mathbf{e} \\ Used(n) \vee & \\ \quad Transp(n) \wedge \prod_{m \in succ(n)} D\text{-}SAFE(m) & otherwise \end{cases} \quad (2.5)$$

$$EARLIEST(n) = \begin{cases} true & if\ n = \mathbf{s} \\ \sum_{m \in pred(n)} (\neg Transp(m) \vee & \\ \quad \neg D\text{-}SAFE(m) \wedge EARLIEST(m)) & otherwise \end{cases} \quad (2.6)$$

$$Ealiest\text{-}INSERT(n) = D\text{-}SAFE(n) \wedge EARLIEST(n) \quad (2.7)$$

$$DELAY(n) = D\text{-}SAFE(n) \wedge EARLIEST(n) \vee \begin{cases} false & if\ n = \mathbf{s} \\ \prod_{m \in pred(n)} \neg Used(m) \wedge DELAY(m) & otherwise \end{cases} \quad (2.8)$$

$$Latest\text{-}INSERT(n) = DELAY(n) \wedge (Used(n) \vee \neg \prod_{m \in succ(n)} DELAY(m)) \quad (2.9)$$

図 2.15 部分冗長除去のデータフロー方程式

Ealiest-INSERT(n) ： *D-SAFE(n)* が *true* となり，かつ *EARLIEST(n)* が *true* であれば *true* となる．このとき，節 n は，プログラムの意味を変えずに巻き上げられる最も開始節に近いプログラム点であり，多忙コード移動の移動点を表す．

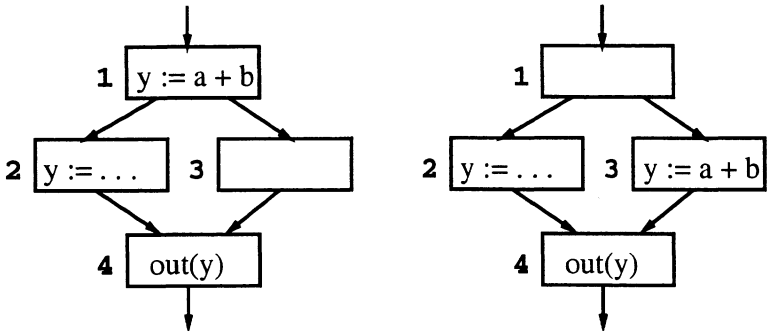
DELAY(n) ：多忙コード移動によって移動された計算式を，プログラムの意味を変えることなく，前向きに移動できる範囲に節 n が存在すれば *true* となる．

Last-INSERT(n) ：データフロー解析後の *DELAY(n')* が *true* である範囲のうちで，節 n が最も終了節に近い節であるか，あるいは式 e の出現場所，すなわち *USED(n)* が *true* となるとき *true* となる．このとき，節 n は，巻上げ前の式の出現場所に辿り着くまでの範囲で，最も終了節に近いプログラム点であり，遅延コード移動の移動点を表す．

実際のプログラム変形は，データフロー解析の結果を利用して式の除去と挿入によって次のように行う．

1. 各式 e に対して，その値を保持するための一時変数 t を用意し，すべての代入文 $x := e$ の右辺を t で置き換えること（$x := t$）によって，まず e を除去する．ここで，x は任意の変数を表す．

2. *Latest-INSERT(n)=true* であれば，節 n の入口に代入文 $t := e$ を挿入する．



(a) 除去前　　(b) 除去後

図 2.16 部分不要代入の除去

結果として得られるプログラムは，多くのコピー代入を含むことになるが，それらのコピーは，レジスタ割付けの過程で行われる変数合体によって，後に除去されることが期待できる．

2.4.2 部分不要コード除去

不要コード除去の効果を上げるために，コード移動と組み合わせた手法として，部分不要コード除去（partial dead code elimination）がある．

通常，不要コード除去は，変数への不要な代入を行う文を対象としている．実行パスによらずに常に不要である**全不要**（totally dead）に対して，図 2.16(a) の節 1 のように，分岐の左側のパスを通ったときは不要であっても，別のパス（右側）を通ったときは有効である代入文は**部分不要**（partially dead）[KRS94b] であるという．この場合には，代入文 `y := a + b` を節 1 から節 2 と節 3 の入口に**降下**（sink）させることによって，節 1 の代入文は節 2 において全不要になり，図 2.16(b) のように除去することができる．PRE と同様に，ループ不変コードのループ外移動も，部分不要コード除去の一種として扱うことができる．ただし，ループの外へ出る方向は，PRE と逆である．

ここで，降下とは，代入文の**前向き**（forward）の移動を意味する．このように，部分不要代入を適切なプログラム点に降下させることによって，全不要代入に置き換えて除去する手法を**部分不要コード除去**（partial dead code elimination，以降 PDE と呼ぶ）という．

文を降下させるための挿入点は，PRE と同様にデータフロー解析によって求めることができる．PDE で用いるデータフロー方程式を，[KRS94b] で提案されたものを例に図 2.17 に示す．これらの方程式は，プログラム中のある代入文 s に関して，s が降下可能なプログラム点として満たさなければならない条件と s が不要代入となる条件を表している．この解

$$
\begin{aligned}
N\text{-}DEAD(i) &= \neg USED(i) \wedge (X\text{-}DEAD(i) \vee MOD(i)) && (2.10)\\
X\text{-}DEAD(i) &= \prod_{i' \in succ(i)} N\text{-}DEAD(i') && (2.11)\\
N\text{-}DELAYED(n) &= \begin{cases} false & if\ n = \mathbf{s} \\ \prod_{m \in pred(n)} X\text{-}DELAYED_m & otherwise \end{cases} && (2.12)\\
X\text{-}DELAYED(n) &= LOCDELAYED(n) \vee && (2.13)\\
&\quad N\text{-}DELAYED(n) \wedge \neg LOCBLOCKED(n) &&\\
N\text{-}INSERT(n) &= N\text{-}DELAYED(n) \wedge LOCBLOCKED(n) && (2.14)\\
X\text{-}INSERT(n) &= X\text{-}DELAYED(n) \wedge \sum_{m \in succ(n)} \neg N\text{-}DELAYED(m) && (2.15)
\end{aligned}
$$

図 2.17 部分不要コード除去のデータフロー方程式

析結果を利用すると，降下によって新たに不要になる代入文を含めて，不要代入を除去することができる．

不要代入の解析は，PRE と同様に，文を 1 つの節として行う．

データフロー解析を行う前に計算しておく必要のある局所述語は次のとおりである．

USED(i) ： 代入文 s の左辺の変数が，節 i で使用されていれば *true* となる．

MOD(i) ：代入文 s の左辺あるいは右辺で用いられている変数の値が，節 i の文によって変更されていれば *true* となる．

LOCDELAYED(n) ：節 n に代入文 s が存在していれば *true* となる．

LOCBLOCKED(n) ：代入文 s の降下が，節 n でブロックされなければ， *true* となる．

PDE のデータフロー解析の解は，次の 2 種類のデータフロー解析を順に適用することによって得られる．

コード降下 ： 代入文 s について，プログラムの意味を変えずに前向きに移動できる範囲を求める．

不要代入除去 ： 後に続く実行において使用されることのない変数を左辺にもつ代入文を求める．

コード降下では，文 s が節 n の入口まで降下できることを表す *N-DELAYED(n)* と，出口まで降下できることを表す *X-DELAYED(n)* を用いて，前向きデータフロー解析によっ

て，文 s がブロックされずに最も降下できるプログラム点を計算する．この結果，挿入点は，入口への挿入 *N-INSERT(n)* あるいは出口への挿入 *X-INSERT(n)* が *true* となる節 n として求まる．降下によって挿入される文の多くは不要となるので，節 i の入口で不要であることを表す *N-DEAD(i)* と出口で不要であることを表す *X-DEAD(i)* を用いた後向きデータフロー解析によって，不要な代入文を決定し，最終的には，不要でない代入文だけを挿入する．

2.4.3 コード移動に基づく新しい最適化

コード移動に基づく最適化のほとんどは，従来法をコード移動を用いて拡張したものである．一方，近年既存の最適化の拡張でない手法も提案されるようになった．本節では，**別名情報**とコード移動を用いた滝本，原田の手法 [滝本 02a, 滝本 02b] を説明する．

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）[ASU86] であると呼ばれる．

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．

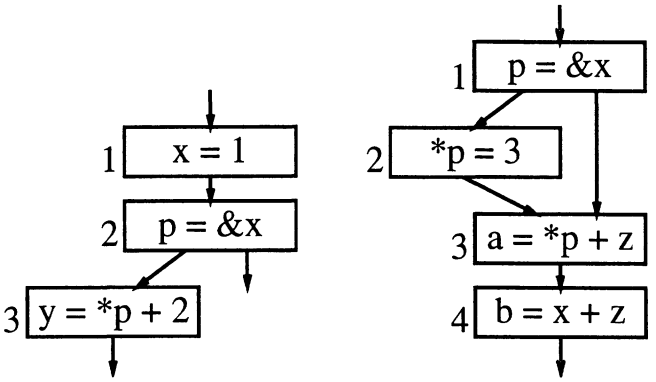
```
1    int *p, n;
2    p = &n;
3    n = 4;
4    printf("%d",*p);
```

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．別名は，変数への代入とその使用という関係を曖昧にし，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．

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）[App98, ASU86, Muc97] 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 2.18(a) と図 2.18(b) に示すプログラムについて別名解析を行うと，`*p` と `x` は別名であることが分かる．この結果を利用すると，図 2.18(a) の節 3 の右辺は定数へ畳み込むことができる．また，図 2.18(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常に同一のメモリ領域を表す別名のこ



(a) 定数伝播の例　　(b) 等値式発見の例

図 2.18 Must 別名を用いた最適化

とを **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．

May 別名は，次の 2 種類に分類することができる．

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

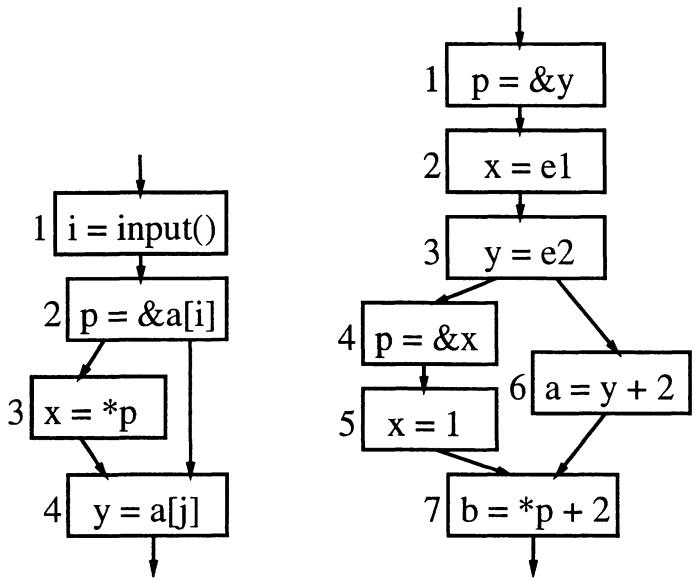
例： 図 2.19(a) における節 3 の`*p` は，値が分からない変数 `i` を添字とする `a[i]` の別名であり，どのメモリ領域を指すのかが特定できない．絶対 May 別名である`*p` は，節 4 の `a[j]` のような他の変数と別名関係にあるかどうかを特定することができない．

∎

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 2.19(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` の別名になるので，部分 Must 別名である．

∎



(a) 絶対 May 別名　　(b) 部分 Must 別名

図 2.19 2 種類の May 別名

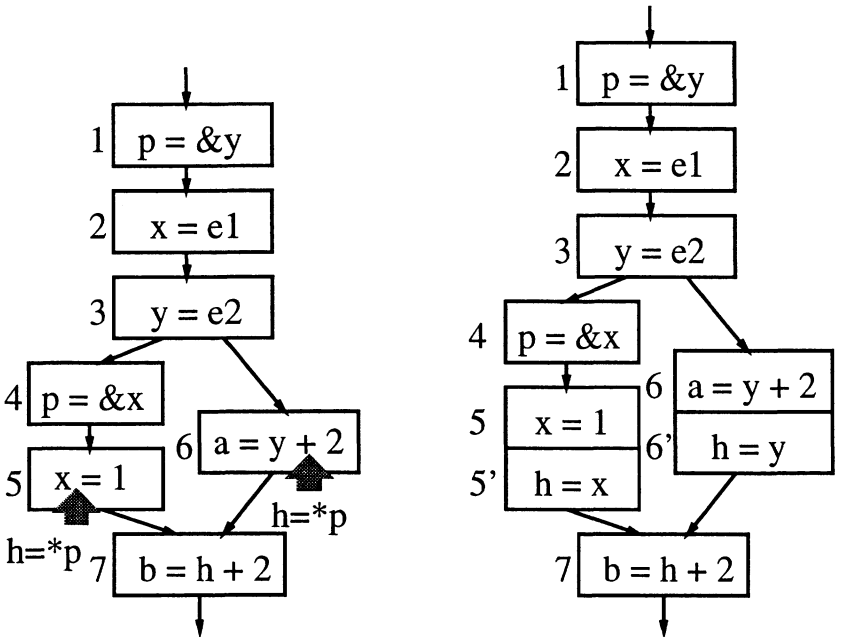
部分 Must 別名は，特定の実行パスでは，Must 別名であるので，PRE と同様の考えに基づいてその実行パスにだけ含まれる節に部分 Must 別名への参照を移動することによって，Must 別名に変換することができる．これを **May 別名除去**（may-alias elimination）[TH01, 滝本 02a] と呼ぶ．

例： 図 2.19(b) の `*p` は部分 Must 別名であり，図 2.20(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 reference）をそれぞれ別名への**直接参照**（direct reference）`x`，`y` で置き換えることによって図 2.20(b) に示す結果が得られる．この変換は，**間接参照除去**（indirection removal）[SCL+96] として知られている． ∎

May 別名除去の結果からは，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 2.19(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 [BG97, Wei84] を適用した場合，別名情報を用いたとしても，図 2.19(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



(a) 部分 Must 別名の巻上げ　　(b) 間接参照除去

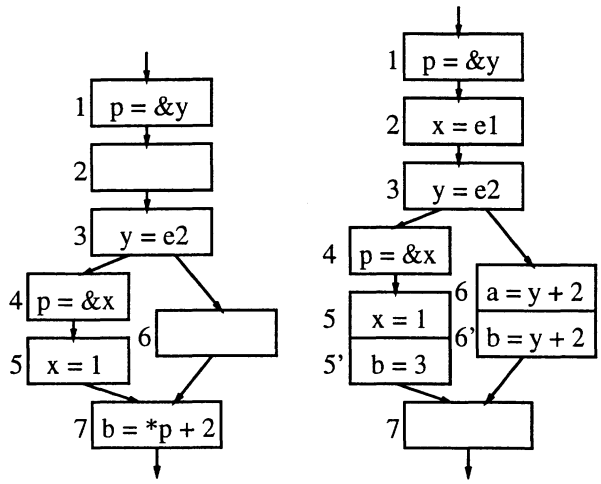
図 2.20 May 別名除去

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 2.21(a)）．

定数伝播： 図 2.19(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 2.20(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，`h + 2` を定数 `3` に畳み込むことができる（図 2.21(b)）[OKS99, 滝本 97].

部分冗長除去： 図 2.19(b) の節 7 の式 `*p + 2` は，節 6 の式に対して**部分冗長**（partially redundant）[DP93, MR79, KRS92, 滝本 97] 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 [RWZ88, SKO90, FCKLL97] の適用によって，除去することができる（図 2.22(a)）．

部分不要コード除去： 図 2.19(b) に示す節 3 の文は**部分不要コード**（partially dead code）[FKCX94, TH99, KRS94b, 滝本 00] 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，部分不要コードとみなされない．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，



(a) プログラムスライシング

(b) 定数伝播

図 2.21 May 別名除去の利用，その 1

図 2.22(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．

その他，May 別名除去と PRE を用いて冗長なロード命令の除去と間接参照除去を同時に行う手法も提案されている [滝本 02b]．

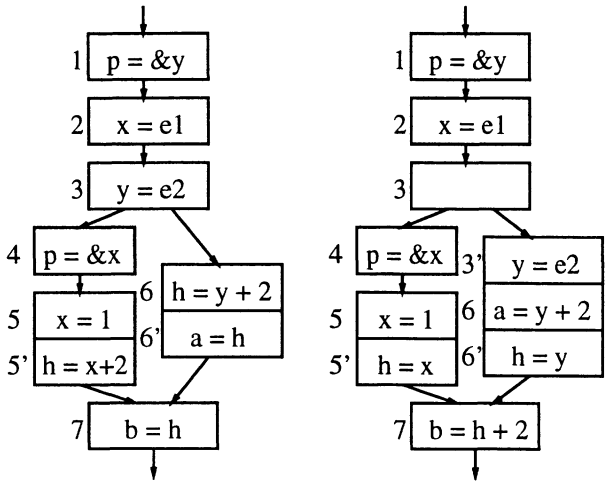
2.5 関連研究

本章のしめくくりとして，本稿で扱うコード移動に基づく最適化として代表的な PRE と PDE のそれぞれに関連して，冗長除去と不要コード除去の関連研究を述べ，その両方に関連して，ループ不変コード移動の関連研究について述べる．また，本手法によって，拡張を行う定数畳込みについても関連研究を述べる．

2.5.1 ループ不変コード移動

ループ不変コード移動については，不変コードをループの直前に移すためのアルゴリズムが Aho, Sethi, Ullman によって示されている [ASU86]．これは，次の 2 つのステップからなる．

1. ループ不変コードを検出する．



(a) 部分冗長除去

(b) 部分不要コード除去

図 2.22 May 別名除去の利用，その 2

2. そのうちで，ループ中の実行パス上に新たな計算を挿入することのないコード（以降，**安全なループ不変コード**と呼ぶ）をループの外へ出す．

ループ不変コードは，式で使用している変数への代入がループ内にないことを確かめることによって検出できる．安全なループ不変コードであるかどうかは，ループ不変コードのプログラム点がループのすべての出口を**支配**（dominate）[App98, ASU86] していることを確かめることによって検出できる．ここで，プログラム点 p がプログラム点 p' を支配するとは，開始節から p' に達するすべての実行パス上に p が存在することをいう．安全なループ不変コードを検出できれば，それをループの外に出すことができる．

Aho らの手法では，1 つのループ不変コードが，ループのすべての出口を支配しなければならなかったが，この制約は，同じ値を計算するループ不変コードが，ループヘッダからループの出口へのすべてのパス上に存在しなければならないという制約に置き換えることができる．このような拡張を行ったものには，Rosen，Wegman，Zadeck の手法（以降，**RWZ 法**と呼ぶ）[RWZ88] や，RWZ 法を効率化した Click の手法 [Cli95] がある．

2.5.2 定数畳込み

定数畳込みは，静的に（コンパイル時に）計算できる部分を計算し，その部分を定数で置き換える最適化である．定数畳込みは，定数伝播を行いながら繰返し適用することによって，効果を上げることができる [ASU86]．定数畳込みは，通常の代入文に現れる式だけでなく，条件分岐の条件式も含めることができるので，場合によってはコンパイル時に条件

文の実行パスを特定でき，不要なコードを除去することにも役立つ．このような条件式の畳込みも含めて，大域的に定数を畳み込む手法として，Wegman と Zadeck の手法がある [WZ91]．これは，SSA 形式に基づいて，定義と参照を辺で結んだグラフ上に定数をフローさせる方法であり，効率的な定数畳込みを可能にしている．しかし，従来の手法では，異なった実行パス上の 1 つの変数への代入によって，異なった値が到達する場合の畳込みについての提案はなされていない．

2.5.3 冗長な式の発見

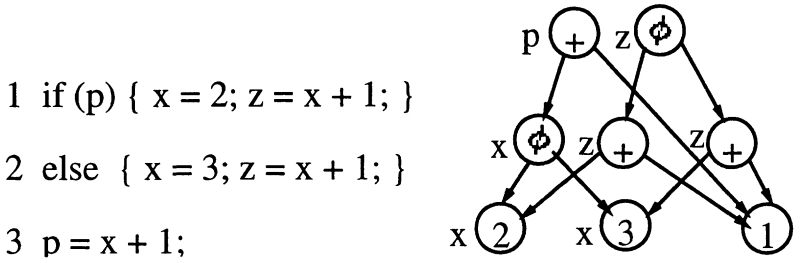
従来から，冗長な式の発見には，基本ブロックごとに等価な式を見付ける**値番号付け**（value numbering）[佐々 89] がよく使われてきた．値番号付けは，同じ値を保持する変数に対して同じ番号（以降，**値番号**と呼ぶ）を割り付けることによって，等価な式を発見する．変数が同じ値番号を保持するかどうかは，その変数への代入文の右辺について，対応する位置のオペランド変数が同じ値番号をもち，同じ演算子をもつかどうかによって判定する．

大域的な等価式発見には，データフロー解析を用いた Kildall の手法が有名である．この手法は，プログラムを前向きに走査していき，途中に現れる式を**列**（sequence）として記録する．式の列は，生成する値ごとに**区域**（partition）を設けて，等価な式をグループ分けしておく．式を区域に加えるときには，オペランドを，そのオペランドと同じ値を生成する区域内の他の式によって置き換えることによって，プログラム中に現われる字句形式の異なった式も同時に加える．この処理によって，走査点上の式の可能な字句形式が 1 つの区域にすべて含まれていることになり，最終的に，各プログラム点において静的に発見可能な等価式をすべて見付けることができる．この手法は，データフロー解析の際に式をフローさせることになるので，フローグラフの結合点において区域の併合が必要であり，各区域が同じ字句形式をもつかどうかの解析が必要になる．この解析コストは非常に大きく，Kildall の手法の計算量は，指数的に増加するという問題をもつ．

大域的な等価式発見において，発見できる式の数には制限があるが，計算量を小さく抑えて実践的にした手法として，Alpern，Wegman，Zadeck の大域的な**依存グラフ**（dependence graph）を用いた手法 [AWZ88] がある．ここで，依存グラフとは，変数の定義と使用の関係を明示するためにグラフ表現にしたものをいう．この手法は，式の等価関係を依存グラフ構造の合同関係に置き換えることによって，アルゴリズムを簡素化している．初期状態では，同じ演算子あるいは定数の節をすべて同じ値を生成するものとして同じ区域に入れておき，依存先が異なる節があると区域を分割する．等価式発見は，最終的に合同な構造を見付ける問題の最大解を得ることになるので，依存グラフ構造に循環をもつループ内の帰納変数（induction variables）についても等価な式を発見することができる．依存グラフの節の個数を N とすると，Alpern らの手法は，計算量が $O(N \log N)$ であり，効率が良い．しか

し，図 2.23(a) の 3 行目において，明らかに等しい値を保持する z と p は，複数の代入が到達することを示す ϕ を用いて依存グラフで表現すると，図 2.23(b) のように異なった構造になり，依存構造の相違から等価な式とみなすことができない．

同様に依存グラフを用いる手法に，Click の手法 [Cli95] がある．これは，依存グラフ上で，ラベルが等しく，依存先が同じ節をボトムアップで合体させていく方法であり，求められる解は最小解である．したがって，Alpern らの手法のように，帰納変数についての等価な式を見付けることはできないが，依存グラフの節数を N とすると，計算量が $O(N)$ となるので，効率が良い．



(a) プログラム片　　(b) 対応する依存グラフ

図 2.23 依存グラフ上で発見できない等価式

2.5.4 冗長除去

共通部分式の性質を利用した共通部分式の除去は [ASU86] に示されている．共通部分式 e を含む式 $e1$ は，$e1$ に到達するすべての実行パス上に e をもつ式が先行して存在する場合に限って，$e1$ における e の冗長性を除去することができる．

これに対して，PRE は，第 2.4.1 節で述べたように一部の実行パス上にだけ先行して式が存在する場合でも，部分冗長性を除去することができる．ループ不変コード移動もその効果の一部として含まれるという特徴をもつ．PRE は，Morel，Renvoise[MR79] によって初めて提案され，後に多くの改良がなされた．その中には，双方向のデータフロー解析を単方向の組合せにすることによって，データフロー方程式の双方向依存を無くし，ビットベクトルを用いたデータフロー解析の計算量を単方向と同等にする手法 [DP93] や，同様の方法で，無用な巻上げをまったく行わないことを保証する手法 [KRS92, KRS94a] などが提案されている．また，PRE が式を対象にしているのに対して，代入文自体を移動する手法も提案されている [Dha91, KRS95]．これらが除去の対象としているのは，字句形式が等しい計算式だけである．PRE とコピー伝播を組み合わせて複数回の適用を行うと，字句形式が異なる等価な式でも順次字句形式が等しいものに変わり，除去の対象になることがある．しか

し，その場合の計算量は，計算式の個数を C，基本ブロックの個数を N，式の依存の深さ（ランク）を R として，双方向データフロー解析によるものが $O(N^3R)$，単方向にしたものが $O(N^2R)$ である [RWZ88].

基本ブロック単位でのハッシュ表を用いた値番号付け [ASU86] を大域的な範囲に拡張した Rosen, Wegman, Zadeck の手法（以降，RWZ 法と呼ぶ）[RWZ88] は，SSA 形式の利用によって意味等価な計算式も冗長除去の対象にしている．しかし，この手法は，ループの認識など入力プログラムの構造に大きく依存し，アルゴリズムに必要なデータ構造も複雑である．また，その計算量は $O(CN^2)$ である．

RWZ 法の効率を改良したものに，Click の手法 [Cli95] がある．この手法では，プログラムサイズに関してほぼ線形の計算量ですむ．しかし，計算式が存在しない実行パスに式を挿入することを許しているので，もとのプログラムよりも実行時間が長くなる可能性を含んでいる．また，巻上げの範囲が，複数の代入が到達するプログラム点を越えてさらに上方に及ぶことがないので，RWZ 法に比べると，制限されたものになっている．

Steffen, Knoop, Rüthing の手法 [SKO90] は本提案手法と同様に等価な式を始めに計算しておく手法である．彼らの手法では，計算式を**閉路なし有向グラフ**（directed acyclic graph, 以降，DAG と呼ぶ）で表現し，CFG に沿って伝播させることによって各計算点の計算式表現を作成する．この方法による計算量は $O(N^4)$ である[3].

本稿での提案は，以上の研究と比べると，計算量が $O(CN)$ と効率が良く，かつ簡単なデータ構造とアルゴリズムで実現できる．さらに，計算式の移動によって新たに可能になる定数畳込みを PRE に統合している点で，従来提案されている手法よりもさらに大きな効果が期待できる．

2.5.5 不要コード除去

コード移動を行わずに不要コードを除去する手法は，基本的なデータフロー解析で実現できることが [ASU86] に示されている．式を依存グラフで表現し，使用されない部分グラフを除去する効率的な手法は，[App98] や [CFRW91] に示されている．

PDE の重要性は，Feigen らによって指摘された [FKCX94]．Feigen らの手法では，変数 x への代入文 s が実行されるときのすべてのパスについて，x が頻繁に使用されるパス上のプログラム点に s を移動させ，文の移動がブロックされるときは，CFG の分岐構造を変形して，コード移動を行っている．分岐構造の変形は，一部の計算をコピーする可能性があるので，プログラムサイズを大きくする可能性がある．また，文の移動先が 1 つのプログラム点に制限されるので，除去できない部分不要コードが残る場合があり，ループ外コード移動

[3]この場合の N は，並行代入が可能な文だけを 1 つの基本ブロックとした場合の CFG の節数である．したがって，1 つの計算式に対して，ほぼ 1 つの基本ブロックが対応する．

の効果はもたない．これに対して，Knoop らの手法 [KRS94b] は，制御構造を変形しない範囲において，ループ外コード移動も含めて，Feigen らの手法を一般化している．Knoop らの手法は，データフロー解析を基に繰返し適用を必要とするので，最悪の場合には，プログラムサイズの 5 乗の計算量，合理的な仮定を用いても 3 乗の計算量と見積もられている．

制御構造を変更することを前提に，Feigen らの手法を一般化したものに，Bodik らの手法 [BG97] がある．これは，スライシングの技術を使って，効果的な PDE を実現している．しかし，その計算量はプログラムサイズに関して指数的である．

本手法は，Knoop らと同様に制御構造を変更しないことを前提にしている．計算量は，プログラムサイズの 2 乗のオーダであり，過去の提案手法よりも低いコストで PDE を実現できる．また，変数名の付替えによってコード移動がブロックされないようにする方法の提案は，本研究が初めてである．変数名の付替えは，他の手法と併用することも可能である．

以上に述べた以外にも，冗長なコードの除去に関しては，過剰なコード巻上げを抑えるためのコード降下を行う際に，副次効果として PDE を行う手法が提案されている [BC94, Cli95]．しかし，この PDE は複数の代入が到達するプログラム点まで適用されるだけで，前向きコード移動の範囲が制限されたものとなっている．

第 3 章

本提案手法の概要

本章では，本研究における提案を概説する．まず，コード移動が与える副次的効果の説明とその問題点を詳細に述べる．次に，副次的効果を直接反映させる本提案手法のアプローチの詳細について述べる．

3.1 コード移動に基づく最適化の副次的効果

PRE と PDE は，1 回のデータフロー解析とその結果に基づくプログラム変形を行うと，新たに最適化可能な候補を生成する可能性があるという副次的効果をもっている．すなわち，高い効果を得るためには，PRE や PDE を繰返し適用する必要がある．

コード移動に基づく最適化による副次的効果は，コード移動自身の性質による副次的効果と基本最適化の性質による副次的効果とに区別することができる．基本最適化の副次的効果とは，PRE に関しては共通部分式の除去の効果であり，PDE に関しては不要コード除去の効果である．

次に，これらの副次的効果について説明する．

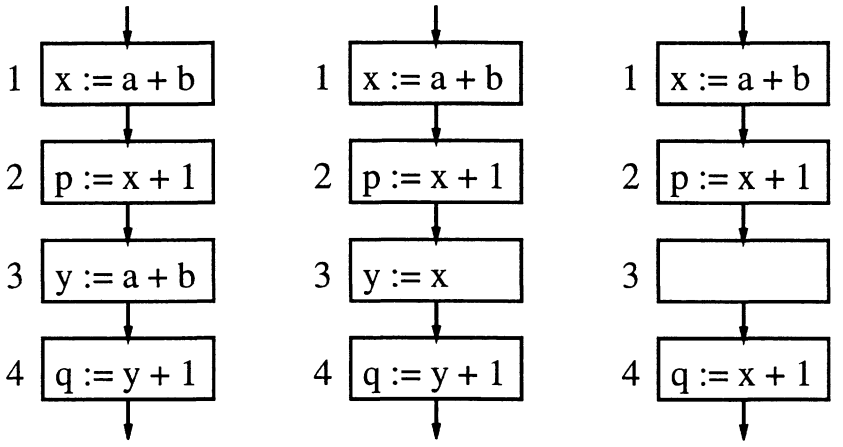
3.1.1 冗長除去の副次的効果

PRE は，字句形式が同一の式を対象に，冗長性を除去する手法であるので，字句形式が異なっていても，明らかに同じ値を計算する式との間の冗長性には対応することができない．

例： 図 3.1(a) において，式 1 と式 3 は字句形式が等しいので，冗長な式 3 を除去して，図 3.1(b) のように変換することができる．式 2 と式 4 は，明らかに同じ値を計算するが，字句形式が異なるので，冗長除去の対象にならない．式 2 と式 4 の冗長性が明らかになるのは，式 3 を除去した後に残るコピー代入がコピー伝播によって除去された後である（図 3.1(c)）．

∎

この例が示すように，冗長性の除去は新しい冗長な式を生成する可能性がある．この副次

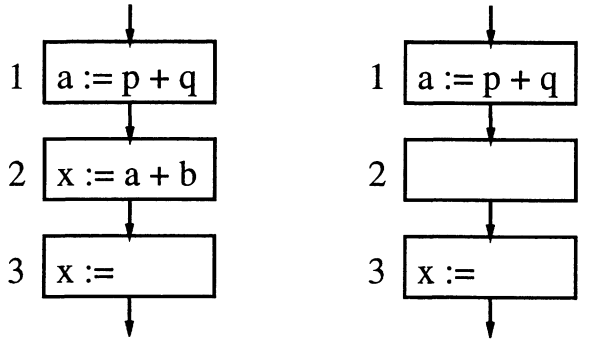


(a) 元のプログラム　(b) a + b の除去　(c) y := x のコピー伝播

図 3.1 PRE の副次的効果

的効果を**等価効果**と呼ぶ.

3.1.2 不要コード除去の副次的効果

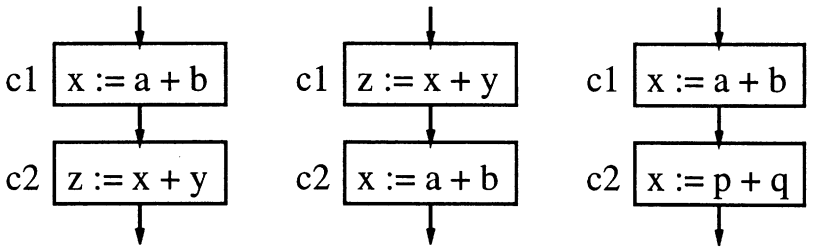


(a) 元のプログラム　(b) 不要コード除去後

図 3.2 PDE の副次的効果

PDE における不要コード除去は，確実に使用されることのない変数への代入文だけを対象にしている．しかし実際には，ある文の除去に伴って，そこで使用されていた変数が不要になり，その変数への代入文も不要になることがある．

例： 図 3.2(a) の文 2 は，文 3 の同じ変数 x への代入によって明らかに不要になっているので，文 2 を除去した結果は図 3.2(b) のようになる．このとき，文 2 の除去に伴って，そこ



(a) 真の依存　　(b) 逆依存　　(c) 出力依存

図 3.3 依存の種類

で使用していた a への代入文 1 が不要になる．文 1 が不要であると分かるのは，文 2 が除去された後である．∎

ある文が不要になることによって間接的に不要になる文は，**弱体**（faint）であるという．このように，不要代入文の除去は，弱体文を不要にするという副次的効果があることが分かる．この副次的効果を，**弱体コード効果**と呼ぶ．

3.1.3 コード移動の副次的効果

これまで述べてきたデータフロー解析に基づいて実現されるコード移動は，**依存**（dependency）関係をもつ計算の順序を保存するようにするために，依存先に当たる計算の元の出現位置でブロックされる（PRE の場合には，データフロー方程式の $TRANSP$ が $false$（図 2.15），PDE の場合は，$LOCBLOCKED$ が $true$（図 2.17）であるという条件によって示される）．2 つの計算を $c1$ と $c2$ とし，$c1$ が $c2$ より先行する場合，依存は，一般に次の 3 通りに分類される（図 3.3）．

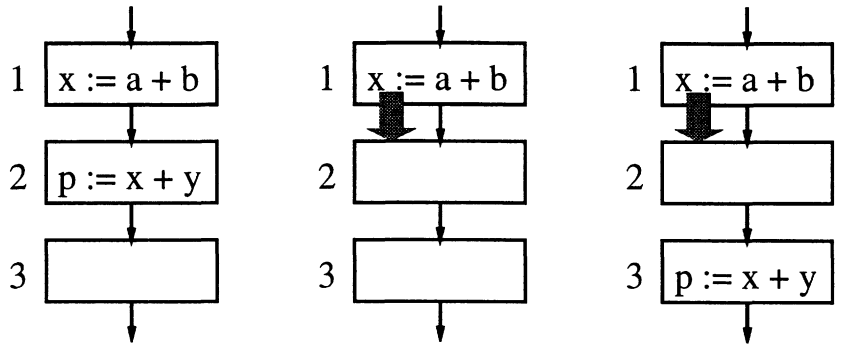
真の依存 ： $c1$ で定義した変数が，$c2$ で使用される．

逆依存 ： $c1$ で使用している変数が，$c2$ で定義される．

出力依存 ： $c1$ で定義した変数が，$c2$ で再定義される．

$c1$ と $c2$ の依存によって $c2$ の巻上げは $c1$ の位置でブロックされ，$c1$ の降下は $c2$ の位置でブロックされる．実際は，ブロックする側も移動あるいは，除去される場合があり，ブロックされた側はさらに移動できる可能性がある．

例： 図 3.4(a) の文 1 と文 2 の間には依存関係があるので，文 1 の降下は文 2 によってブロックされる．しかし，図 3.4(b) のような場合，ブロックしていた文 2 が除去されると，文 1 は



(a) 元のプログラム　(b) 除去 - 移動効果　(c) 移動 - 移動効果

図 3.4 コード移動の副次的効果

さらに降下が可能になる．また，図 3.4(c) のようにブロックしていた文 2 が降下しても，文 1 のさらなる降下が可能になる．∎

この例のように，ブロックしていた計算が除去されて移動可能になる効果を，**除去 - 移動効果**と呼ぶ．また，ブロックしていた計算が移動して移動可能になる効果を，**移動 - 移動効果**と呼ぶ．

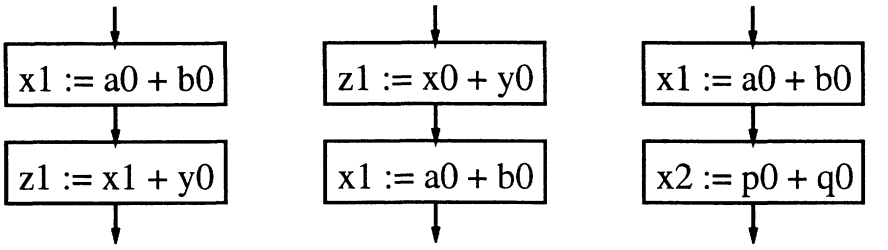
3.2 本研究の意義

本研究では，コード移動に基づくコード最適化を効率的にかつ効果的に実現するための手法を提案する．この節では，効率に関して問題となる副次的効果の直接的な反映を可能とする，実現のアプローチを述べ，本アプローチを用いて得られる新しい最適化の効果について述べる．

3.2.1 除去 - 移動効果と移動 - 移動効果の直接反映

前節で述べたように，計算の順序を保存する必要性から，コード移動は依存関係によって制約され，依存先の計算位置で移動がブロックされる．ブロックしていた計算が除去されることによって移動可能になる除去 - 移動効果，およびブロックしている計算が同時に移動することによって生じる移動 - 移動効果を有効にするためには，これまで述べてきた方法を繰返し適用する必要があり，そのために，大きなコストを必要とした．

本研究では，このコード移動の副次的効果を直接得る枠組を提案する．その基本的な考え方は，以下に述べるとおりである．いま，プログラム中における計算 $c1$ と $c2$ に関して，$c2$ は $c1$ に依存するものとする．コード移動を行うときには，$c1$ と $c2$ の計算順序を保存し



(a) SSA 形式の真の依存　(b) SSA 形式で存在しない逆依存　(c) SSA 形式で存在しない出力依存

図 3.5 SSA 形式における依存

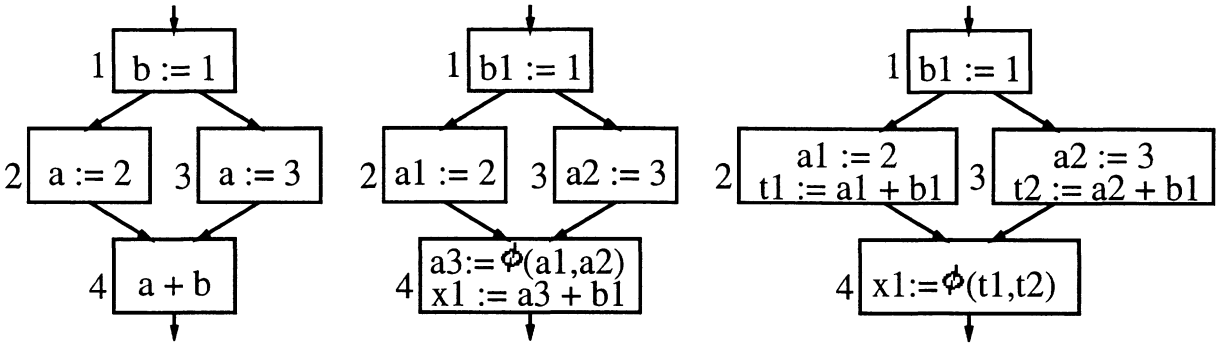
なければならないことを考えると，$c1$ の移動可能範囲は，$c2$ が移動できるプログラム点の直前であることが分かる．すなわち，$c1$ と $c2$ が互いに移動をブロックしないと仮定すると，$c2$ が移動できない範囲を $c1$ の移動可能範囲から取り除く，すなわちブロック側での移動できない範囲をブロックされる側の移動可能範囲から取り除く操作を順に行うことによって，計算順序を保存した状態で，可能な移動 - 移動効果をすべて直接反映させることができる．また，$c2$ が移動できない範囲，すなわち $c2$ のブロックによって $c1$ が移動できなかった範囲でも，$c2$ が除去できる場合には，$c2$ によるブロックが解除されるので，$c1$ の移動可能範囲からその部分を除かないようにすれば，除去 - 移動効果も直接反映させることができる．

このようにブロック側の移動や除去をブロックされる側に反映させるためには，各計算式について，ブロックされる側とブロックする側の関係を意味する依存関係が明らかになっていなければならない．

これまでの説明においては，図 3.6(a) に示すようなプログラム表現を用いてきたが，本提案手法では，図 3.6(b) に示すような SSA 形式のプログラムを対象とする．理由は，SSA 形式によるプログラム表現を用いると，使用する変数に対する定義が一意に決まるので，依存関係は，真の依存以外を考慮するだけですみ，依存関係の扱いを単純化できることによる．

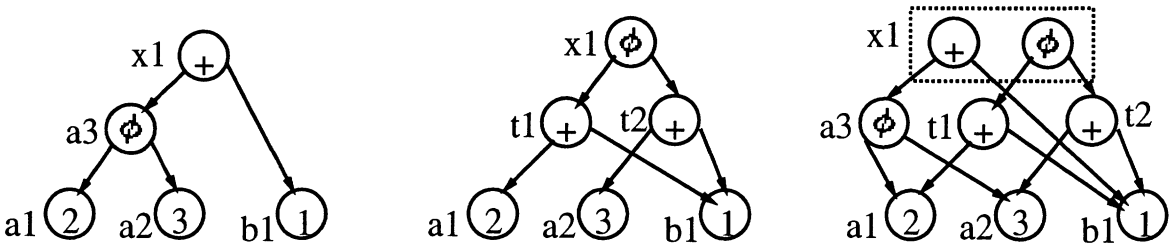
例： 図 3.3を SSA 形式に変換したものを，図 3.5に示す． ∎

真の依存関係の表現には，各定数と演算子を節とし，変数の定義と使用の関係を明示するために，それらに対応する節を辺で結んだ図 3.7(a) のような大域的に依存構造を表現するグラフを用いる．なお，値が不明な変数については，そのことを意味する特殊な定数（⊥）として扱う．以降，依存構造を表すために用いるグラフ表現を**依存グラフ**（dependence graph）



(a) 複数の定義が到達する例　(b) SSA 形式での表現　(c) 巻上げ結果

図 3.6 SSA 形式における巻上げ



(a) 図 3.6(b) の依存グラフ　(b) 図 3.6(c) の依存グラフ　(c) 同じ式を表す図 3.6(a), (b) の節

図 3.7 依存グラフの変形

と呼ぶ．依存グラフは，計算の元の位置での真の依存関係を表したものである．図 3.6(c) のように計算式 `a + b` を移動させると，定義と使用の関係が変わるので，それに伴って依存グラフの構造（依存グラフ構造）も図 3.7(b) のように変わる．そこで，計算式を各プログラム点へ移動させた場合の式の依存構造を維持するために，依存グラフを変形するという考えを導入する．さらに，依存グラフの変形前後の構造を統一的に扱うために，図 3.7(c) の点線で囲んだ節のように，1 つの値を表す依存グラフの節を組にして 1 つの節として扱う**拡張値グラフ**（extended value graph，以降 **EVG** と呼ぶ）を導入する．本研究では，この拡張値グラフによる依存構造に基づいて，コード移動の副次的効果を制御フローグラフ上に直接反映させるデータフロー解析の枠組を提案する．この共通の枠組の上で，実現したい最適化におけるデータのフローの方向に対応した変形を行い，データフロー方程式を与えるだけで，コード移動に基づく最適化を効率良く実現できるようになる．

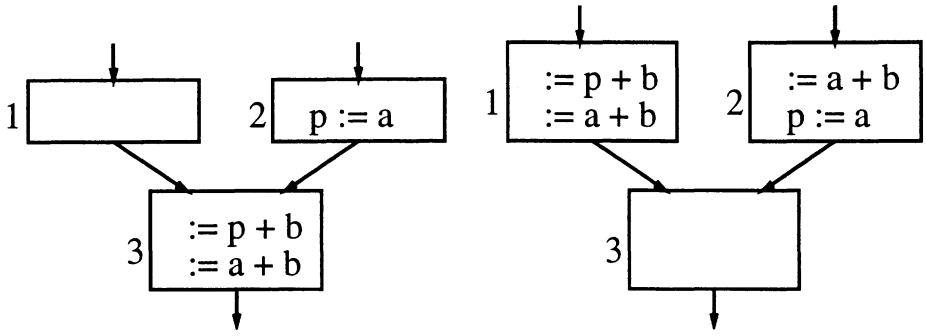
3.2.2 等価効果の直接反映

PRE は，後向きコード移動に基づいた最適化手法であることから，移動 - 移動効果をもつが，それ以外に，3.1.1節で述べたように，PRE の 1 回の適用が等価な式を新たに生じさせる等価効果をもつ．

本研究では，除去 - 移動効果や移動 - 移動効果を直接得るために導入した EVG を用いて，すべての等価効果を 1 回の解法によって得る手法も提案する．EVG は，式の依存グラフ構造を表現しているので，その構造等価な部分を見付けることによって，コード移動とは独立に，すべての等価な式を発見することができる．従来の依存構造を用いた等価式発見の手法は，原始プログラム上の計算位置における構造表現だけを用いていたので，図 3.7(a) と (b) のように，同じ値を表現する式であっても，依存グラフの構造が異なる場合には，それらが等価であることを発見することができなかった．これに対して，EVG は，各プログラム点における依存グラフ構造をすべて保持している．すなわち，図 3.7(c) の点線で表した節のように，同じ値を表す式であれば，プログラム点によって異なる依存構造をもつ場合でも，それらの構造を組にして 1 つの節で表現できるので，組になった節のうち 1 つでも等しくなる構造をもつ EVG 節があれば合体させていくことによって，すべての等価な式を発見しておくことができる．この後のコード移動においては，等価情報を利用して式の冗長性を決定することによって，繰返し適用なしにすべての冗長な式を取り除くことができる．

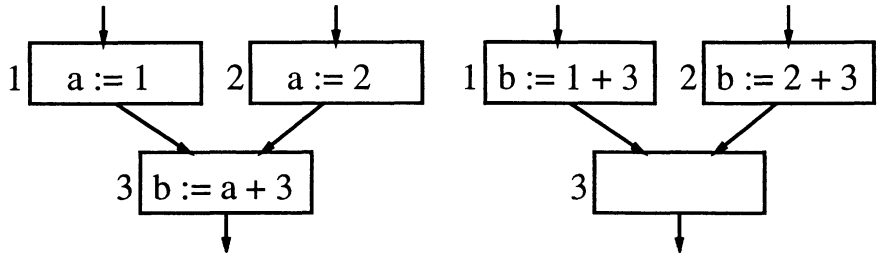
EVG 上で，等価式発見とコード移動を独立に行う手法を用いると，従来法では発見できなかった冗長性を除去することもできる．PRE では，変数の生存期間を短くするために，無用なコード巻上げを行わないようにしている．この方法では，図 3.8(a) に示すような，巻き上げた後に変数を置き換えることによって冗長性が見付かるような場合を扱うことができない．節 3 の 2 つの式を節 1，2 へ巻き上げることによって，節 2 では，`p + b` と `a + b` が同じ値を計算する式となるので，図 3.8(b) のように部分冗長な式を除去することができる．この効果は，巻き上げを行ってみて初めて発見できるものであり，従来の PRE においては，無用なコード巻上げとみなされる．これに対して，巻上げ後の構造も合わせもつ EVG 上での等価式発見は，図 3.8(a) のような場合も巻上げ後の効果を見付けることができ，後のコード移動で除去することができる．

また，PRE をコード移動と等価式発見の組合わせで実現することによって，等価式発見の前に，EVG 上で定数畳込みを行うことができる．これは，1 つの節から出るすべての辺の先が定数のラベルをもつ節である場合，その節のラベルによって示される演算を行い，計算結果をラベルとする新たな節を生成して，元の節と置き換えることによって実現できる．この定数畳込みも，EVG 上で行うことによって，従来法では得られなかった畳込みを実現することができる．



(a) 元のプログラム (b) 有効な PRE

図 3.8 従来法では行うことができなかった PRE



(a) 元のプログラム (b) 有効な定数畳込み

図 3.9 従来法では行うことができない定数畳込み

例： 図 3.9(a) の節 3 では， a + 3 を定数に畳み込むことはできないが，この式を巻き上げることによって，図 3.9(b) に示すように畳込みが可能になる．図 3.9(a) の式 a+3 は， 従来の PRE では冗長な式として扱うことができないので，コード巻上げの対象にはならず，畳込みを行うことができなかった．これに対して， EVG は各プログラム点の依存構造をもっていることから，図 3.9(a) のような場合であっても，コード巻上げによる畳込みを行うことが可能である． ∎

EVG 上での畳込み可能な部分を事前に計算しておくことによって，どのプログラム点に移動させれば，実際に畳込みを行うことができるかを明らかにすることができ，この定数畳込みの効果を含むように PRE のデータフロー解析を拡張することができる．

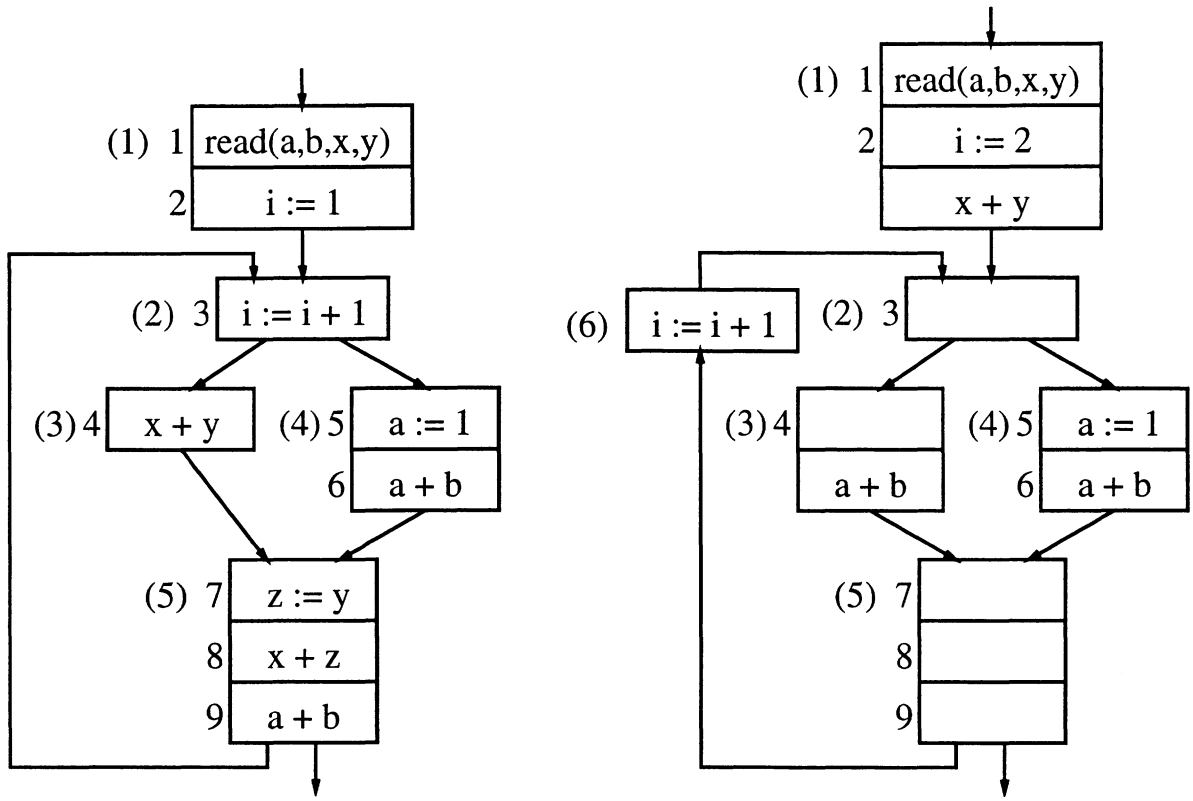
図 3.10(a) のプログラムに対して，本手法を適用して得られる結果を図 3.10(b) に示す．

3.2.3 弱体コード効果の直接反映

PDE には，ある計算 $c1$ における代入先変数を計算 $c2$ で使用している場合，$c2$ が不要コードとして除去されると，$c1$ も不要コードとなる弱体コード効果がある．この効果は，移動 - 移動効果と同様に，$c2$ が不要であるという情報を EVG 上にフローさせることによって，直接的に反映することができる．この場合，$c1$ と $c2$ は，真の依存の関係にあるので，EVG 上の辺として表現される．そして，$c2$ が不要になったプログラム点においては，$c1$ も不要になることを考えると，EVG 辺上を辿って，$c2$ から $c1$ へ不要であるという情報をフローさせれば，$c1$ が弱体コードとなるプログラム点を求めることができ，そのプログラム点に $c1$ をコード降下させることによって，可能な弱体コード効果をすべて反映することができる．

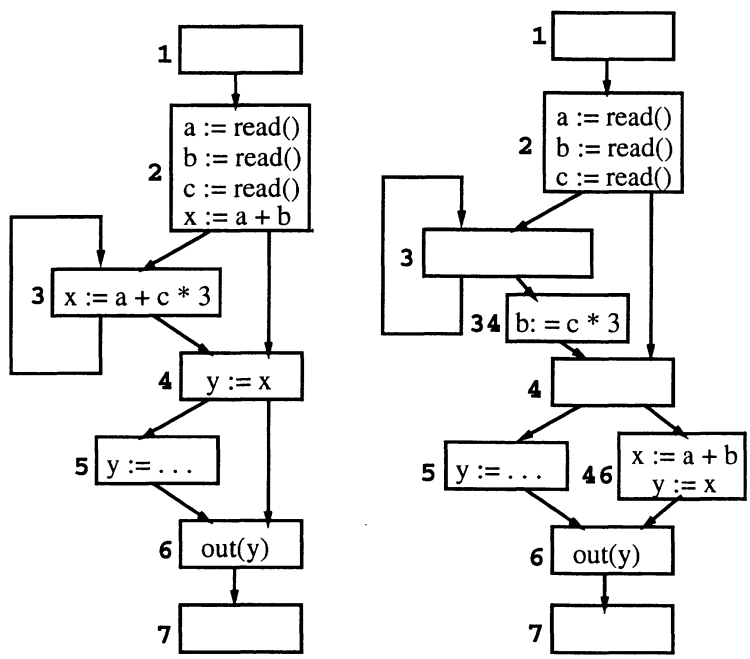
さらに，EVG がもつ各プログラム点の依存構造を利用すると，代入文の降下できる範囲をさらに大きくすることができる．降下範囲が大きくなることは，部分不要代入が除去できる可能性を大きくすることになるので，より多くの部分不要代入を除去できる可能性がある．

例：図 3.11(a) において，節 4 の代入文 `y := x` を節 5 に降下させることができれば，その文は，節 5 の入口で全不要になり，除去できる．その結果，`y := x` で使用している `x` も不要になる．その場合，`x` を代入先変数とする文を節 5 の入口に降下させることができれば，さらにその文も不要になる．このような降下が可能になるためには，**結合点** (join point) である節 4 において，左側から降下してくる文と，右側から降下してくる文とで字句形式が一致しなければならない．図 3.11(a) の場合，節 2 の `x := a + b` と，節 3 の `x := a + c * 3` とは計算式の字句形式が異なるので，このままでは，節 4 への降下はブロックされる．しかし，部分式 `c * 3` の値を `b` に代入することができる場合には，両者の代入文は同じパターンになり，図 3.11(b) に示すように `x := a + b` を節 46 へ降下させることができる．このような変数の付替えは，EVG を生成する際の変形過程で，変形パターンに合うように新しい節を付加することに等しい．したがって，EVG の作成段階ですべての可能な変数付替えを実現できることになる．



(a) 元のプログラム (b) 本手法による PRE

図 3.10 本手法による PRE の効果



(a) 元のプログラム (b) 本手法のよる PDE

図 3.11 本手法による PDE の効果

第 4 章

値グラフとその変形

本研究は，プログラム中の各点における計算式の依存関係を明らかにすることによって，依存関係をもつ計算式が元の位置でブロックされることのないデータフロー解析の実現を目的としている．

従来，各プログラム点における依存関係を得る方法として，各式の出現場所における局所的な依存関係を木あるいはDAGによって表現し，それらを伝播させる方法が用いられた．しかし，この方法は，データフロー解析におけるフロー情報として，木やDAGを用いることに等しいので，計算コストが高く，消費するメモリの量も大きかった．そこで，本研究では，依存グラフの一種である**値グラフ**を用いて各プログラム点における依存性を凝縮して表現する方法を考える．

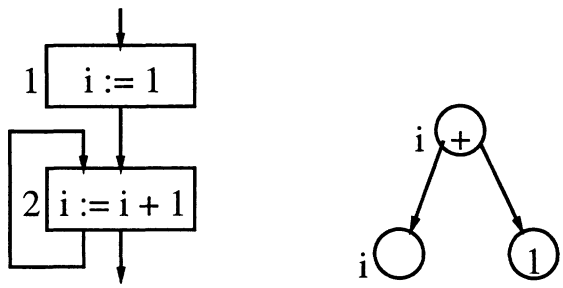
この章では，値グラフの定義を述べ，各プログラム点における依存グラフ構造を得るために，値グラフの変形が必要であることを述べる．値グラフの変形は，同じ値を表現するグラフへの等価変形であり，可能な変形パターンを示すことによって定義できる．また，変形の効果を用いた直接的な応用についても述べる．

4.1 値グラフ

計算の依存関係を，字句形式に制限されることなく扱うために，変数の定義と使用を辺で結んだグラフ表現がよく使用される．CFG上の基本ブロック内では，そこに含まれる式を表現するために木やDAG表現が使われる．木やDAGは，計算の局所的な依存関係を表現するだけであり，複数の基本ブロックに跨る依存関係を表現することはできない．

例： 図4.1(a)における `i` の値は，`i := i + 1`が繰返し実行されることを考えると，節2における`i`の使用は，節1における`i`の定義と節2の定義に依存する．しかし，節2だけについて，この式を木構造で依存を表現したとすると，図4.1(b)のようになり，依存の関係を部分的にしか表現することができない． ∎

一方，プログラム全体での依存関係を表現するためには，依存関係をグラフで表現した依



(a) プログラム例　　(b) 木による依存表現

図 4.1 木を用いた複数の基本ブロックに跨る依存表現

存グラフが必要になる．定義と使用を結んだ依存グラフである**定義 - 使用チェイン**（def-use chain）や**使用 - 定義チェイン**（use-def chain）[ASU86] は，計算の依存関係全体をグラフで表現することができる．しかし，ある変数の使用に対して複数の定義が到達すること，すなわち定義の結合を明示する方法をもたないので，その変数で使用される値が制御フローによって異る場合には，そのことを明示的に表現することができない．

これに対して，SSA 形式のプログラムにおいて，演算子と定数ラベルをもつ節と，定義と使用関係にある節の間を結ぶ辺で構成された依存グラフである値グラフ（value graph，以降 **VG** と呼ぶ）[AWZ88] や **SSA グラフ**（static single assignment graph）[Cli95] は，定義の結合も合わせて，プログラム全体での依存性を表現できるので，制御フローと独立に計算構造を取り扱うことができる．これらのグラフにおいて，定義の結合は，SSA 形式のところ（2.2.2節）で述べた ϕ- 関数を用いて，ϕ をラベルとする節によって表現される．VG では，使用から定義へ向かう辺によって計算の依存関係を表し，SSA グラフでは定義から使用へ向かう辺によって値の流れを表現する．本研究では，解析の際に必要となる依存関係を VG によって表現する．

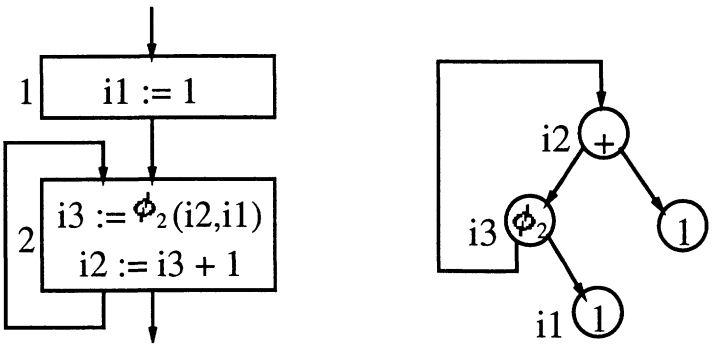
次に VG の定義を与える．以下では，定数の集合を **C**，演算子の集合（代入は演算に含めない）を **Op**，ϕ- 関数の集合を **Φ**，静的に知ることができない値を $\perp$ で表し，$OP = \mathbf{C} \cup \mathbf{Op} \cup \mathbf{\Phi} \cup \perp$ とする．

定義 1 (値グラフ) *VG* は次の 3 つの集合からなる三つ組 (N_{VG}, E_{VG}, L_{VG}) である．

N_{VG} ： 節の集合

E_{VG} ： 辺の集合 $E_{VG} \subseteq N_{VG} \times N_{VG}$ を表す．

L_{VG} ： ラベル付け関数 $N_{VG} \rightarrow OP$．ある節が与えられたときに，その節のラベルを返す．



(a) SSA 形式　　(b) 対応する VG

図 4.2 VG を用いた依存表現

VG の節 v は，内部節である場合には，ラベル $L_{VG}(v)$ によって示される演算を行う式の構造を表す．関数の仮引数の参照，添字の値が特定できない配列参照，別名が指すメモリ領域を特定できない変数の参照など，静的に変数の定義が分からない場合には，それらの値を表すためにラベルとして $\perp$ をもつ節を用いる．

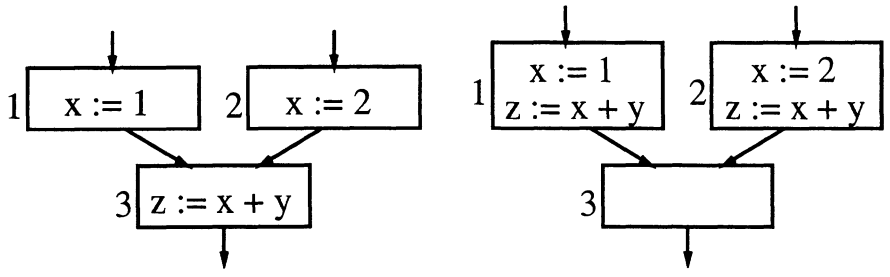
例： 図 4.1(a) のプログラム例を SSA 形式に変換したものを図 4.2(a) に示し，そこから作成した VG を図 4.2(b) に示す．SSA 形式における変数名への対応を節の左に示してある．

∎

SSA 形式のプログラムに ϕ- 関数が現れる場合，VG ではそれを演算子とみなして，ϕ- 関数に相当する節（以降，**ϕ 節**と呼ぶ）によって表す．ϕ- 関数は，使用に到達する定義が複数ある場合に，それらの定義を，最初に合流する CFG 節（結合点）において結合する働きをもつので，各 ϕ- 関数にはその CFG 節を明示しておく必要がある．そこで，CFG 節 i における ϕ- 関数は，その帰属を表すために，i を添字として付加した "ϕ_i" という表記を用いる（図 4.2(a)）．なお，ϕ_i は節 i からは移動することがないものとする．VG は，基本ブロックごとの DAG を結び合わせた構造と異なり，ϕ 節が対応する結合点（以降，**ϕ 節固有の結合点**と呼ぶ）で，先行する CFG 節上にある計算式への依存を束ねる働きをする．これによって，依存グラフに必要最小限の制御情報を加え，プログラム全体にわたる依存を正確に表現することができる．

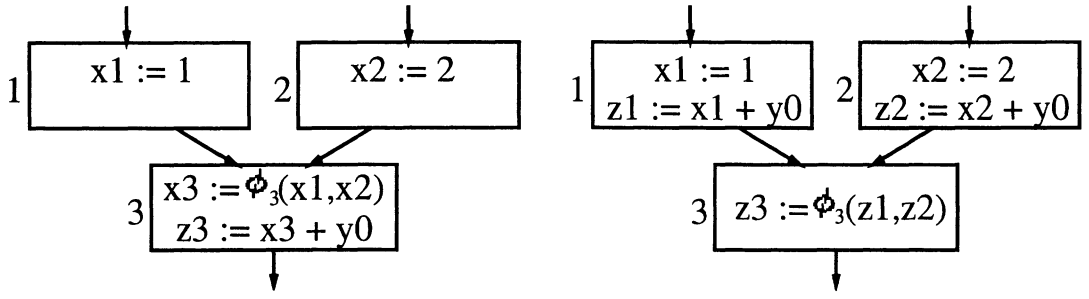
4.2 値グラフの変形

ϕ 節が，固有の結合点 i において，複数の計算式への依存を束ねる役割をするという性質は，CFG 節 i とその先行節の間で，ϕ_i を含む計算式を移動させる際に，VG の構造上の変



(a) 1 つの定義が到達する例　　(b) 複数の定義が到達する例

図 4.3 計算位置によって依存構造が異なる通常のプログラム形式



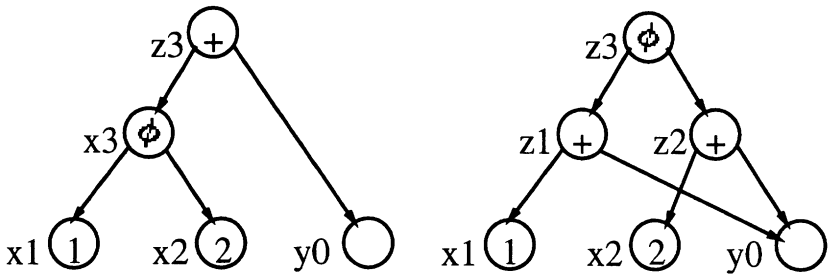
(a) 図 4.3(a) の SSA 形式　　(b) 図 4.3(b) の SSA 形式

図 4.4 計算位置によって依存構造が異なる SSA 形式

更が必要になることを意味している.

例： 図 4.3(a) の節 3 上の計算式 `z := x + y` と図 4.3(b) の節 1, 2 上の計算式 `z := x + y` は, 節 3 における変数 `z` の値を考えると, 同じ値を生成する. すなわち, 図 4.3(b) は, 図 4.3(a) の `z := x + y` をすべての先行節に後向きに移動させたプログラムであり, 一方, 図 4.3(a) は, 図 4.3(b) の `z := x + y` を後続節に前向きに移動させたプログラムであると言える.

これらの各プログラムに対する SSA 形式のプログラムは, 図 4.3(a), (b) のそれぞれに対応して, 図 4.4(a), (b) のようになる. 図 4.4(a) において, 節 3 に到達する定義は `x1 := 1` と `x2 := 2` であるので, 節 3 では, それらを結合する ϕ- 関数 ϕ_3 が必要である. 一方, 図 4.4(b) において, 節 3 に到達する定義は `z1 := x1 + y0` と `z2 := x2 + y0` であり, 節 3 以降の計算によって `z` が使用されることを仮定すると, 節 3 上に `z1` と `z2` を結合する ϕ- 関数が必要である. ∎



(a) 図 4.4(a) に対する VG　　(b) 図 4.4(b) に対する VG

図 4.5 計算位置で異なる VG の構造

以上の例から，図 4.4(a) のプログラムが計算式 `z := x + y` の後向き移動によって，図 4.4(b) のプログラムに変形したと考えると，意味的に等価な変形の手順を次のようにまとめることができる．

1. 計算式 `z3 := x3 + y0` における変数 `x3` の使用に関して，`x3` は ϕ- 関数 ϕ_3(`x1`, `x2`) によって定義されるので，まず `x3` の使用を節 3 の各先行節 1，2 に対応する引数 `x1`, `x2` でそれぞれ置き換えた計算式を作る．

2. 上のステップで作成した計算式をそれぞれの先行節 1，2 へ移動させる．

3. 移動した計算式を単一代入形式にするために新たに別の変数名 `z1`，`z2` を生成し，元の代入先 `z3` をそれぞれの名前で置き換える．

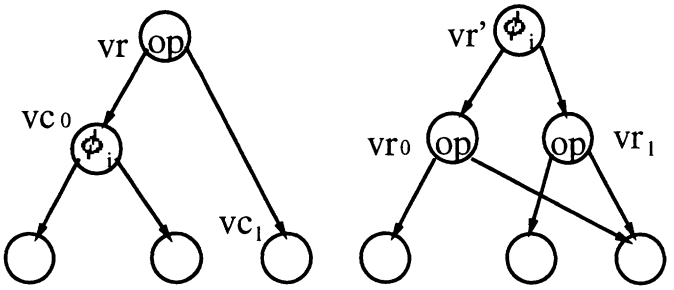
4. 単一代入となった計算式の代入先 `z1`，`z2` を結合するために，`z3 :=` ϕ_3 `(z1, z2)` を生成して，CFG 節 3 に挿入する．

また，図 4.4(b) のプログラムを計算式 `z := x + y` の前向き移動によって，図 4.4(a) に変形したと考えると，次の手順で等価な変形が可能であることが分かる．

1. 移動対象となる計算式 `z1 := x1 + y0`，`z2 := x2 + y0` の代入先 `z1`，`z2` を結合する ϕ- 関数 `z3 :=` ϕ_3 `(z1, z2)` が後続節に存在するので，まず代入先 `z1`，`z2` をそれぞれ ϕ- 関数の代入先 `z3` で置き換える．

2. それらの計算式の中では，オペランドとして異なる変数名 `x1`，`x2` が使用されているので，それらを結合する ϕ- 関数 `x3 :=` ϕ_3 `(x1, x2)` を後続節に作成する．そして，計算式中の `x1`，`x2` をそれぞれ代入先 `x3` で置き換える．

3. これらの計算式を 1 つにして，後続節 3 に移動させる．



(a) 元の VG　　(b) 変形後の VG

図 4.6 左オペランドが複数の節に依存している場合の後向き変形

図 4.4(a) と図 4.4(b) の SSA 形式プログラムから作成した VG を図 4.5(a) と図 4.5(b) に示す．コード移動を行う場合には，SSA 形式において変形操作が必要になったのと同様に，VG に関しても，後向き移動では，図 4.5(a) から図 4.5(b) への変形が必要であり，前向き移動では，図 4.5(b) から図 4.5(a) への変形が必要となる．これは，SSA 形式における変形と対応して，後向き移動においては，移動対象になる計算式の VG が ϕ 節を含んでいる場合には，変形を行う必要があり，一方前向き移動においては，計算式を表す VG の副グラフの根に対して依存している ϕ 節があると，変形が必要になるということである．VG 上での変形において重要なことは，プログラム中での計算式の位置とは独立に，特定の位置に ϕ 節を含む変形パターンを定義でき，そのパターンに一致した VG の副グラフに対して変形を適用できるという性質である．

4.2.1 変形パターン

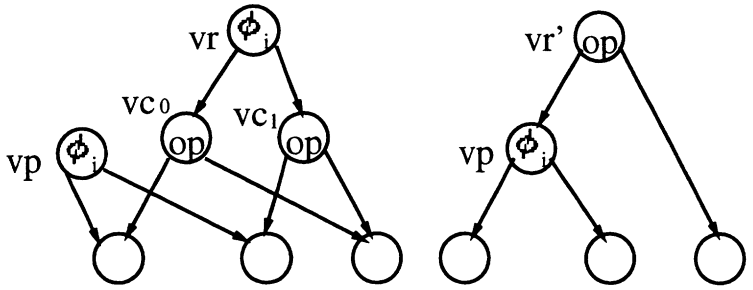
依存関係を適切に表現するための変形は，VG 上での ϕ 節の位置関係によって，次の 3 種類が存在する．

以下，変形の適用対象になる VG の副グラフの根を vr，その依存先を vc_0，vc_1 で表す．このほか，各 VG 節 v の依存先は，$i = 0, 1$ として，節 $child_i(v)$ で表す．

1. **左オペランドが複数の定義をもつ場合**

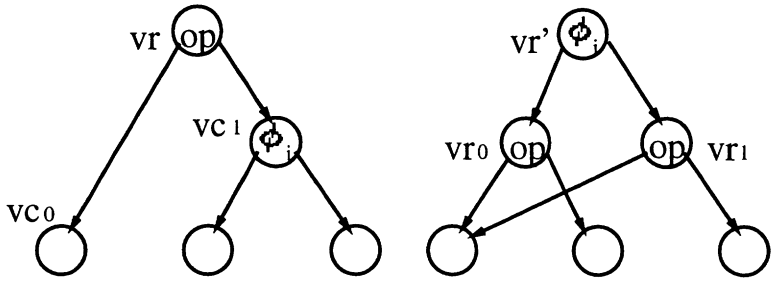
後向き移動 ：

- 適用条件：$L_{VG}(vr) \in OP \wedge L_{VG}(vc_0) \in \Phi \wedge L_{VG}(vc_0) \neq L_{VG}(vc_1)$.
- 変形（図 4.6(a) から図 4.6(b)）：
  (a) vc_0 と同じラベルをもつ節 vr'（$L_{VG}(vr') = L_{VG}(vc_0)$）を生成し，$vr$ に依存している辺を vr' への辺に付け替える．



(a) 元の VG　　(b) 変形後の VG

図 4.7 左オペランドが複数の節に依存している場合の前向き変形



(a) 元の VG　　(b) 変形後の VG

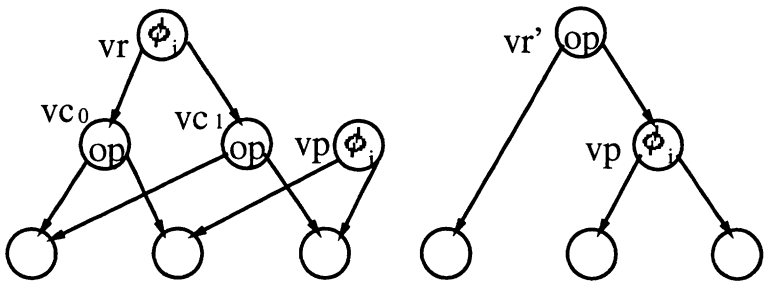
図 4.8 右オペランドが複数の節に依存している場合の後向き変形

(b) vr を 2 つの節 vr_0 と vr_1 に分離し，$child_0(vr') = vr_0$，$child_1(vr') = vr_1$ となる辺を生成する．

(c) vr_0，vr_1 の辺を，$child_0(vr_0) = child_0(vc_0)$，$child_0(vr_1) = child_1(vc_0)$ となるように付け替える．

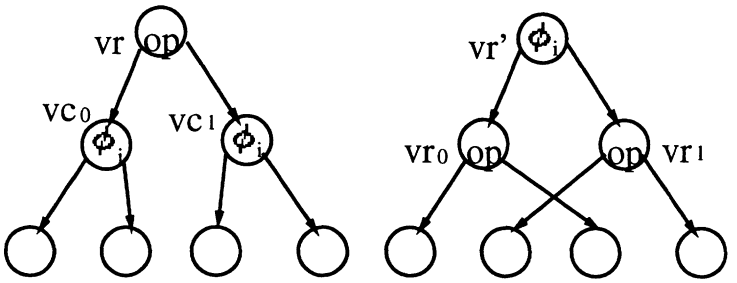
前向き移動 ：

- 適用条件：$L_{VG}(vr) \in \Phi \wedge L_{VG}(vc_0), L_{VG}(vc_1) \in OP \wedge L_{VG}(vc_0) = L_{VG}(vc_1) \wedge child_1(vc_0) = child_1(vc_1)$，かつ計算式のオペランドが一致している必要性から，次の条件を満たす VG 節 vp が存在する．
  $L_{VG}(vr) = L_{VG}(vp) \wedge child_0(vp) = child_0(vc_0) \wedge child_1(vp) = child_0(vc_1)$
- 変形（図 4.7(a) から図 4.7(b)）：
  (a) $L_{VG}(vr') = L_{VG}(vc_0)$ である節 vr' を生成し，vr に依存している辺を vr' に付け替える．



(a) 元の VG　　(b) 変形後の VG

図 4.9 右オペランドが複数の節に依存している場合の前向き変形



(a) 元の VG　　(b) 変形後の VG

図 4.10 左右両方のオペランドが複数の節に依存している場合の後向き変形

(b) $child_0(vr') = vp$, $child_1(vr') = child_1(vc_0)$ となる辺を生成する.

2. **右オペランドが複数の定義をもつ場合**

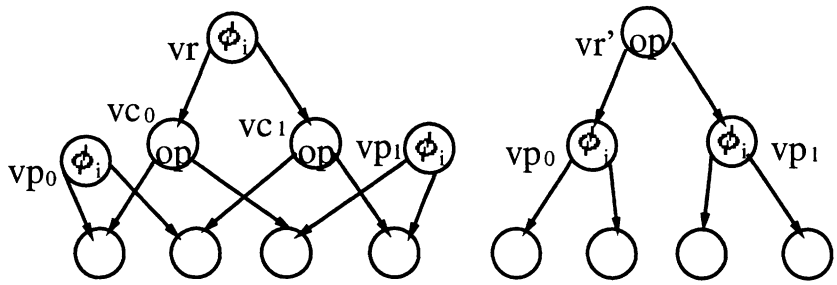
後向き移動 ：

- 適用条件：$L_{VG}(vr) \in OP \wedge L_{VG}(vc_1) \in \Phi \wedge L_{VG}(vc_0) \neq L_{VG}(vc_1)$.
- 変形（図 4.8(a) から図 4.8(b)）：

  (a) vc_1 と同じラベルをもつ節 vr' ($L_{VG}(vr') = L_{VG}(vc_1)$) を生成し, vr に依存している辺を vr' への辺に付け替える.

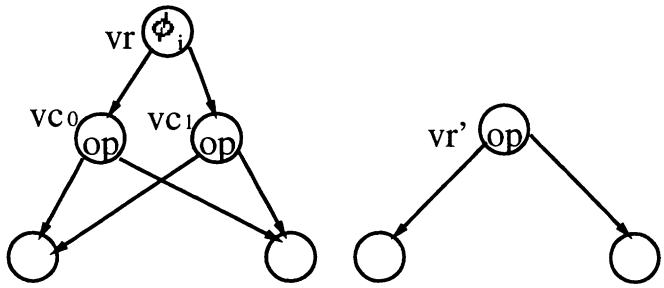
  (b) vr を 2 つの節 vr_0 と vr_1 に分離し, $child_0(vr') = vr_0$, $child_1(vr') = vr_1$ となる辺を生成する.

  (c) vr_0, vr_1 の辺を, $child_1(vr_0) = child_0(vc_1)$, $child_1(vr_1) = child_1(vc_1)$ となるように付け替える.



(a) 元の VG　　(b) 変形後の VG

図 4.11 左右両方のオペランドが複数の節に依存している場合の前向き変形



(a) 元の VG　　(b) 変形後の VG

図 4.12 左右両方のオペランドが同じ節に依存している場合の前向き変形

前向き移動 ：

- 適用条件： $L_{VG}(vr) \in \Phi \wedge L_{VG}(vc_0), L_{VG}(vc_1) \in OP \wedge L_{VG}(vc_0) = L_{VG}(vc_1) \wedge child_0(vc_0) = child_0(vc_1)$ かつ次の条件を満たす VG 節 vp が存在する．
  $L_{VG}(vr) = L_{VG}(vp) \wedge child_0(vp) = child_1(vc_0) \wedge child_1(vp) = child_1(vc_1)$
- 変形（図 4.9(a) から図 4.9(b)）：
  (a) $L_{VG}(vr') = L_{VG}(vc_0)$ である節 vr' を生成し，vr に依存している辺を vr' への辺に付け替える．
  (b) $child_1(vr') = vp$, $child_0(vr') = child_0(vc_0)$ となる辺を生成する．

3. **左右両方のオペランドが複数の定義をもつ場合**

後向き移動 ：

- 適用条件： $L_{VG}(vr) \in OP \wedge L_{VG}(vc_0) = L_{VG}(vc_1) \in \Phi$.

- 変形（図 4.10(a) から図 4.10(b)）：
  (a) vc_0 または vc_1 と同じラベルをもつ節 vr' （$L_{VG}(vr') = L_{VG}(vc_0)$）を生成し，$vr$ に依存している辺を vr' への辺に付け替える．
  (b) vr を 2 つの節 vr_0 と vr_1 に分離し，$child_0(vr') = vr_0$，$child_1(vr') = vr_1$ となる辺を生成する．
  (c) vr_0，vr_1 の辺を，$child_0(vr_0) = child_0(vc_0)$，$child_0(vr_1) = child_1(vc_0)$，$child_1(vr_0) = child_0(vc_1)$，$child_1(vr_1) = child_1(vc_1)$ となるように付け替える．

前向き移動 ：

- 適用条件：$L_{VG}(vr) \in \Phi \wedge L_{VG}(vc_0), L_{VG}(vc_1) \in OP \wedge L_{VG}(vc_0) = L_{VG}(vc_1)$，かつ次の条件を満たす VG 節 vp_0 と vp_1 がともに存在する．
  $L_{VG}(vr) = L_{VG}(vp_0) \wedge child_0(vp_0) = child_0(vc_0) \wedge child_1(vp_0) = child_0(vc_1)$
  $L_{VG}(vr) = L_{VG}(vp_1) \wedge child_0(vp_1) = child_1(vc_0) \wedge child_1(vp_1) = child_1(vc_1)$
- 変形（図 4.11(a) から図 4.11(b)）：
  (a) $L_{VG}(vr') = L_{VG}(vc_0)$ である節 vr' を生成し，vr に依存している辺を vr' への辺に付け替える．
  (b) $child_0(vr') = vp_0$，$child_1(vr') = vp_1$ となる辺を生成する．

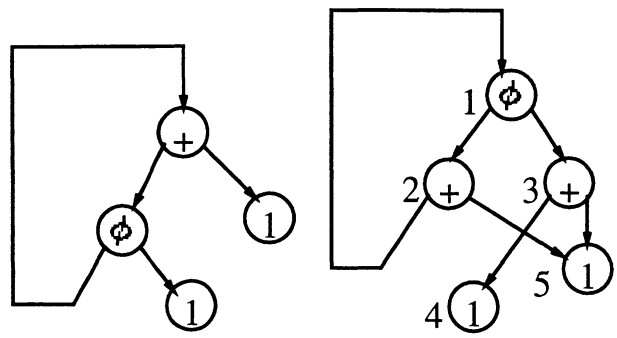
4. **左右両方のオペランドが同じ定義をもつ場合**
この変形が必要になるのは，前向き移動における図 4.12(a) から図 4.12(b) の場合だけに限られる．後向き移動では，ϕ 節がオペランドとして現われることがないので，図 4.12(b) から図 4.12(a) のような変形は起こらない．

前向き移動 ：

- 適用条件：$L_{VG}(vr) \in \Phi \wedge L_{VG}(vc_0) = L_{VG}(vc_1) \in OP \wedge$
  $child_0(vc_0) = child_0(vc_1) \wedge child_1(vc_0) = child_1(vc_1)$
- 変形（図 4.12(a) から図 4.12(b)）：
  (a) vc_1 と同じラベルをもつ節 vr' （$L_{VG}(vr') = L_{VG}(vc_0) = L_{VG}(vc_1)$）を生成し，$vr$ に依存している辺を vr' への辺に付け替える．
  (b) $child_0(vr') = child_0(vc_0)$ となる辺と $child_1(vr') = child_1(vc_0)$ となる辺を生成する．

4.2.2 変形アルゴリズムの停止性と効率

特定の計算式に関して，移動方向を定めれば，上に示した適用条件に合う VG の副グラフを変形することによって，各プログラム点における依存グラフ構造を求めることができ



(a) 元の VG　　(b) 変形後の VG

図 4.13 後向き変形における循環

る．VG 全体について，すべての必要な変形を行うアルゴリズムとして，適用条件を満たす変形候補がなくなるまで変形を繰り返す単純な方法が考えられるが，これには停止性が問題となる．

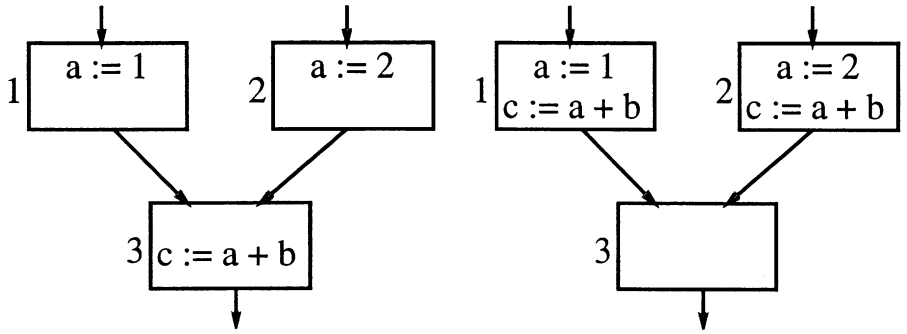
前向き移動に基づく変形（以降，**前向き変形**と呼ぶ）では，少なくとも 1 つの ϕ 節が減少することを考えると，最悪でも，VG 内にある ϕ 節の数だけの変形が行われて停止する．一方，後向き移動に基づく変形（以降，**後向き変形**と呼ぶ）では，停止しない状況が存在する．

例： 図 4.13(a) は図 4.2(a) の VG を表している．この VG に後向き変形を適用すると，図 4.13(b) に示すグラフが得られる（変形後の各 VG 節は，番号を付けて区別する）．ここで，VG 節 2 は，新たに後向き変形可能な候補となる． ∎

この例から分かるように，依存関係が閉路を構成する場合には，後向き変形の適用が停止しないことがある．VG の後向き変形は，計算式の巻上げに伴って生じるものなので，プログラムを静的に一巡する際に必要になる変形を行うだけでよい．これは，CFG を終了節から順に各節を 1 度ずつ訪れて，出会った ϕ- 関数に対応する ϕ 節を含む副グラフに変形を適用すれことによって実現できる．この変形のための計算量は，VG 節の数を V，CFG 節の数を N とすると，$O(V \cdot N)$ である．

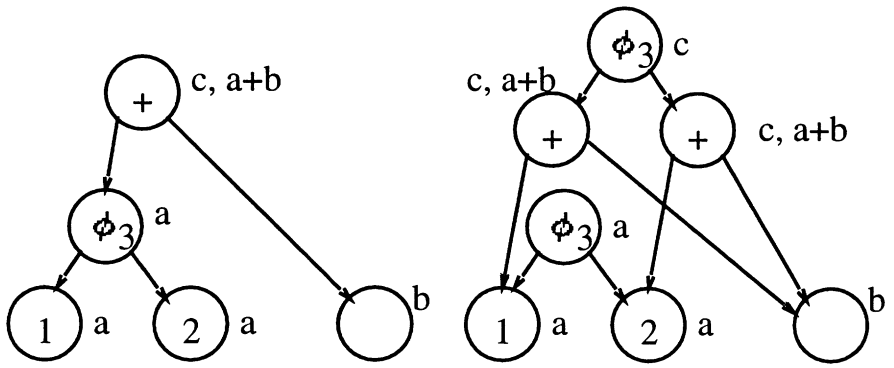
4.3 値グラフ変形の応用

前節では，各プログラム点における計算式を前方または後方に移動させたときの依存構造を求めるために VG の変形を導入した．ある値を表現する依存構造を，VG の変形によって同じ値を表す別のグラフ構造に変形できるということは，別の見方をすれば，プログラ



(a) 1 つの定義が到達する例　　(b) 複数の定義が到達する例

図 4.14 値グラフが異なる構造をもつ例



(a) 図 4.14(a) に対する VG　　(b) 図 4.14(b) に対する VG

図 4.15 同じ値を表現する異なった値グラフ

ム中の計算式の依存構造は，同じ値を表現しているからといって，必ずしも同一のグラフ構造をもつとは限らないことを意味する．同じ値を表現する大域的な依存構造の差異は，VG の変形を用ることによって吸収させることができる．この節では，VG の変形によって，グラフの構造上の差異を吸収できることを用いた応用例を示す．

4.3.1 値グラフと等価性

式の構造を VG で表現すると，式内に現れる変数名に依存することなく，等価な式を見付けることができる．しかし，VG が ϕ 節を含む場合には，等価な式であってもその ϕ 節の出現場所によって異なる構造をとる場合がある．

(a) 元の VG　　(b) 変形後の VG

図 4.16 不要な依存の除去

例： 図 4.14の (a), (b) に対する VG をそれぞれ図 4.15(a), (b) に示す．各 VG の根は，CFG 節 3 における変数 c の値を表してる． ∎

この例に示すような構造の違いは，異なる定義が式のオペランドに到達する結合点において，ϕ 節の出現場所の違いとなって現れる．この構造上の差異を取り除いて同じ値を表現していることを示すためには，図 4.15(a) に後向き変形（図 4.6）を適用し，その結果が図 4.15(b) と一致することを示せばよい．また，等価な式を表す節を 1 つの節に合体させていく過程では，図 4.16(a) のような構造が現われる可能性がある．2 つの辺が同じ依存先をもつ ϕ 節は，不要なので取り除き，図 4.16(b) のようにする．

変形後の根が依存する先の節は，変形の際に新たに生成するものとする．この新たに生成した節を**合成節**と呼ぶ．

4.3.2 後向き変形を用いた等価式発見アルゴリズム

等価な式は，式を構成する演算子とそのオペランドに当たる VG 節に付けた番号の組をキーとして，同じキーをもつ VG 節を，ハッシュ表から探索し，表に存在しなければ登録することを繰り返すことによって発見できる [Cli95, 滝本 97]．ある節 v について表を探索し，等価な式を表す節 v' を発見した場合は，グラフ中の v を v' で置き換え，v への依存は v' への依存に置き換える．この操作を節の**合体**と呼ぶ．この節の合体によって，部分式の等価情報を後の探索に反映させることができる．探索の際に，等価式が発見できない場合は，可能な後向き変形を適用し，その結果について再び等価式の発見を試みる．

発見した等価情報を効率良く反映させるためには，部分式の方からその部分式を含む式へという順序で探索を行う必要がある．したがって，グラフ上では，葉から根に向かって順に，ハッシュ表を探索して，同じ構造をもつ節があれば合体させ，なければ可能な変形を行うことを繰り返す．この合体と変形の繰返しは，ϕ 節が他の ϕ 節に依存している構造が存

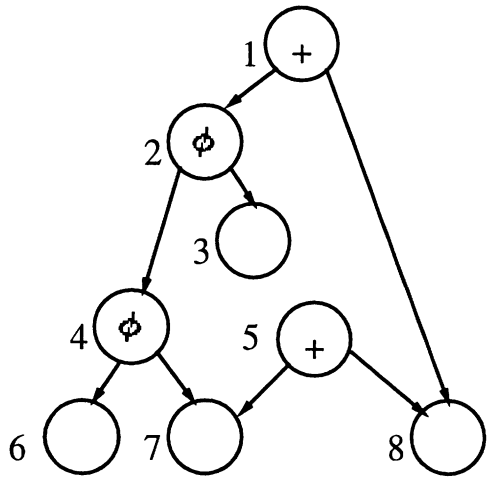


図 4.17 後向き変形がブロックされる例

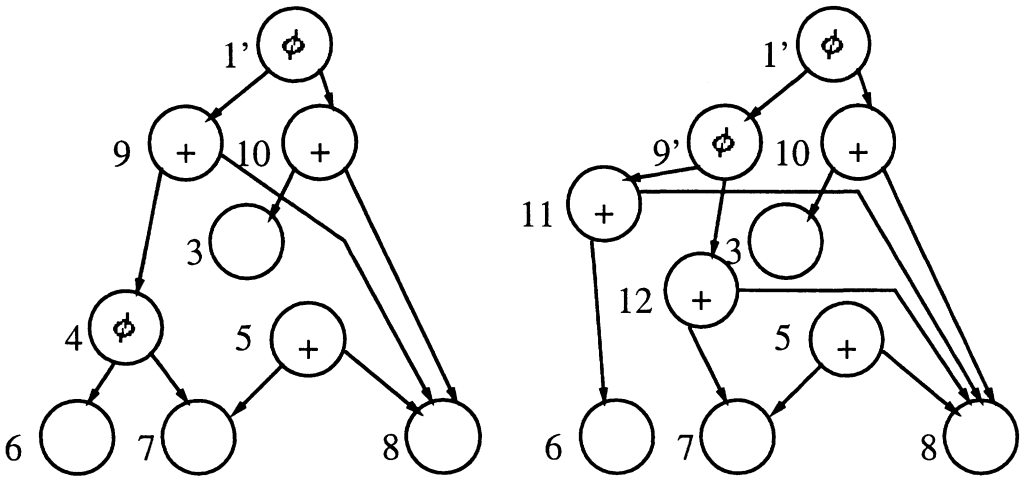
在するとブロックされる.

例： 図 4.17に示す例において，(1,7,8) と (5,7,8) は，等価な式である．この場合，葉から根に向かって等価式発見を適用すると，節 5 と合体できる節はなく，節 4 の ϕ 節に依存する節で変形パターンに当てはまるものもないので，等価式発見は失敗する．しかしながら，ϕ 節の節 2 に依存する節 1 から変形を適用すると，変形パターンを満たし，図 4.18(a) に示すように変形できる．さらに，節 4 に依存する節 9 が変形候補になるので，図 4.18(b) に示すように変形することができる．図 4.18(b) では，節 12 が節 5 と合体可能な節となっていることが分かる． ∎

この例から分かるように，ϕ 節が ϕ 節に依存するような構造をもつとき，変形の適用は，根に近いほうから，葉に向かって行う必要がある．したがって，等価式発見を葉から根に向かって行う場合，いま調べている節を $v \in V$ とすると，v から上方に ϕ 節だけをたどって到達できる節のうちで，ϕ 節でない節の集合 $\phi\text{-}Closure(v)$ が次の変形候補となる．ここで，$\phi\text{-}Closure(v)$ を次のように定義する.

$$\phi\text{-}Closure(v) =_{def} \{v' | v'\text{から}v\text{へのパス上の任意の節}\ v''\ \text{が}L_{VG}(v'') \in \Phi \wedge L_{VG}(v') \notin \Phi\}$$

$\phi\text{-}Closure$ を用いた等価式発見のアルゴリズムの詳細を示す．ここで，変形が循環するのを防ぐために，VG は**後退辺**（back edge） [ASU86] が取り除かれているものとする．また，各節とプログラム中の式との対応を示すために，VG 節を引数として，対応する式の集合を返す関数 Exp が定義されているものとする.



(a) 1 回目の変形後　　(b) 2 回目の変形後

図 4.18 ϕ-$Closure$ を用いた後向き変形

[ϕ-$Closure$ を用いた等価式発見アルゴリズム]

初期化 ：

1. 等価式の発見を行いたい VG 中の各節 v について，“(v のラベル, 左オペランド節への参照, 右オペランド節への参照)”，すなわち $(L_{VG}(v), child_0(v), child_1(v))$ をキーとして v を登録する．ただし，登録済みの節と同じキーをもつ節は除く．ここで，定数 C をラベルにもつ節については，$(C, 0, 0)$ をキーとする．
2. ハッシュ表に登録しなかった節をワークリストに加える．

繰返し ：ワークリストが空になるまで以下を繰り返す．

1. ワークリストから 1 つの節 v を取り出す．
2. v のキーでハッシュ表を探索する．

Case 1 ： 同じキーをもつ節 v' が存在する場合，

Case 1' ： v' に依存する節に ϕ 節が含まれない場合，

(a) ϕ-$Closure(v)$ をワークリストに加える．

(b) $Exp(v')$ に $Exp(v)$ を加え，グラフ上の v を v' で置き換える．

(c) ハッシュ表から v を取り除く．

Case 2' ： さもなければ，

(a) ϕ-$Closure(v')$ をワークリストに加える.

(b) $Exp(v)$ に $Exp(v')$ を加え，グラフ上の v' を v で置き換える.

(c) ハッシュ表から v と v' を取り除き，改めて v を登録する.

Case 2 ： 同じキーをもつ節が存在しない場合,

(a) 節 v を根とする副グラフが，図 4.6(a)，図 4.8(a)，図 4.10(a)，図 4.12(a) のパターンのいずれかと照合する.

Case 1' ： マッチした場合には,

i. それぞれのパターンに対応する変形を行う.

ii. 変形を行った場合には，生成された合成節をワークリストに加える.

Case 2' ： マッチしなかった場合には，ハッシュ表から v を取り除き，改めて v をハッシュ表に登録する.

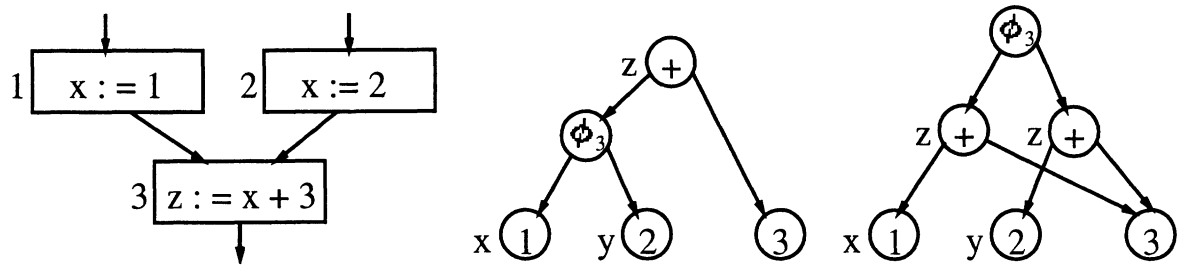
4.3.3 値グラフと定数畳込み

VG を用いると，定数伝播も含めた定数畳込みを容易に実現することができる．ある節を v とするとき，その v の依存先 $c0$，$c1$ がすべて定数であれば，ラベル $L_{VG}(v)$ によって示される演算，すなわち，$L_{VG}(c0)$ $L_{VG}(v)$ $L_{VG}(c1)$ の計算を行って，得られた結果（定数）をラベルとする節で v を置き換えることができる．この VG を用いた定数畳込みは，次に例を示すように後向き変形と組み合わせることによって拡張することができる.

例： 図 4.19(a) において，計算式 `z := x + 3` の `x` には，2 つの定数が到達可能である．しかし，この 2 つの定数は異なるので，従来法では，畳込みを行うことができない．この計算式を表現する VG （図 4.19(b)）は，後向き変形を適用して，図 4.19(c) のように変形することができる．変形後，ラベル `+` をもつ 2 つの節の依存先はすべて定数となるので，畳込みが可能となる. ∎

4.3.4 後向き変形を用いた定数畳込みアルゴリズム

後向き変形を利用することによって，後向き変形を用いた等価式発見と同じように，グラフの葉の方から，定数だけに依存する演算子を探しながら，計算結果をラベルとする節で置き換えていくことによって，従来の定数畳込みをさらに拡張することができる．この過程において，ϕ 節を依存先とする演算子ラベルの節に出会った場合には，図 4.6(a)，図 4.8(a)，図 4.10(a)，図 4.12(a) に示す変形を適用する．このアルゴリズムの詳細を次に示す.



(a) 元のプログラム　　(b) (a) に対する VG　　(c) 変形後の VG

図 4.19 定数畳込みの拡張

[後向き変形を用いた定数畳込みアルゴリズム]

初期化 ：

1. 定数ラベルの節を依存先にもつすべての節をワークリストに加える.

繰返し ：ワークリストが空になるまで以下を繰り返す.

1. ワークリストから 1 つの節 v を取り出す.
2. v の依存先 vc_0, vc_1 のラベルを調べる.

   Case 1 ： もし, vc_0, vc_1 がすべて定数ラベルをもっていれば, ラベル $L_{VG}(v)$ によって示される計算を行い, 結果をラベルとする節を生成して v と置き換える.

   Case 2 ： もし, vc_0, vc_1 の少なくとも一方のラベルが, 定数でも ϕ でもければ, 何も行わない.

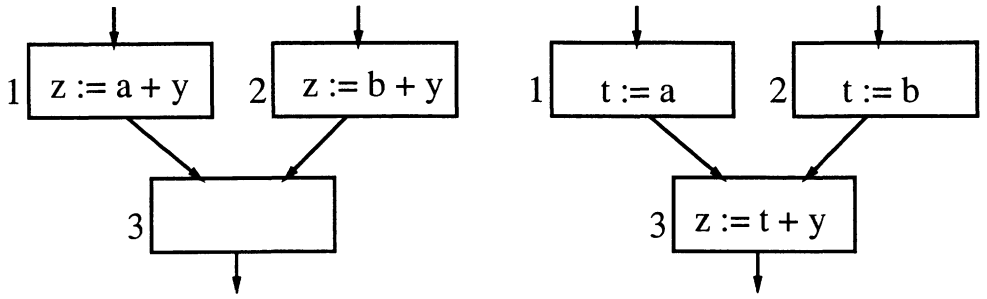
   Case 3 ： もし, vc_0, vc_1 のうち 1 つが ϕ 節で, 他が定数ラベルの節か, 両方とも ϕ- 節であれば,

   (a) 節 v を根とする副グラフが, 図 4.6(a) から図 4.10(a) までのパターンと合うかどうかを調べ, 可能な変形を行う.

   (b) 変形の際に, 生成される合成節をワークリストに加える.

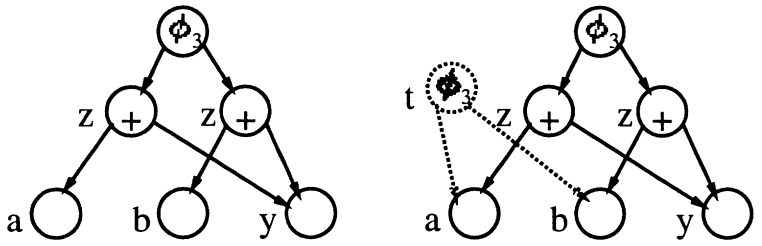
4.3.5 変数の付替え

部分不要コード除去のように代入文の降下を行う場合, 代入先が同じ変数名の代入文を, 複数の先行節から 1 つの CFG 節に降下させるためには, 右辺の字句形式が一致する必要がある. すなわち, 従来のコード降下では, 図 4.20(a) のような, 異なるオペランドをもつ代



(a) 元のプログラム　　(b) 変数付替えと降下

図 4.20 変数の付替えによって可能なコード降下



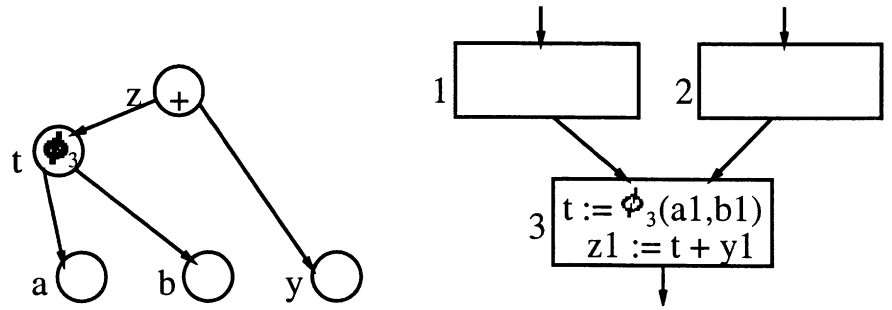
(a) 元の VG　　(b) 変形後の VG

図 4.21 値グラフ変形に基づいた変数の付替え

入文の降下は，演算子が一致していても結合点でブロッキングされる可能性があった．しかし，このオペランドの相違による降下のブロックは，図 4.20(b) のように異なる変数名を同じ名前に付け替えることができる場合には，解除することができる．この変数の付替えは，さらなるコード降下を可能にし，部分不要コードの効果を上げるのに役立つ．

変数の付替えが必要な変数は，VG を用いることによって，容易に見付けることができる．図 4.21(a) は，図 4.20(a) のプログラムを VG で表現したものである．z へ代入する計算式の降下後の VG を得るためには，前向き変形を適用する必要がある．しかし，図 4.21(a) の VG は，前向き変形の適用条件を満たしていない．そこで，図 4.21(b) のように，点線で示す ϕ 節を付加すると，図 4.7(a) のパターンに一致し，変形が可能になることが分かる．この ϕ 節の追加は，通常形式のプログラムで考えると，図 4.20(b) の節 1，2 のそれぞれに代入式 `t := a`，`t := b` を加えたことに相当し，元のプログラムの意味は保存される．

この付加する ϕ 節ごとに，一意の一時変数 t を生成し，変形を適用すると，図 4.22(a) の VG が得られる．この VG は，SSA 形式のプログラムに対応し，図 4.22(b) に示す変数の付



(a) 変形後の VG (b) 対応する SSA 形式

図 4.22 変数の付替えの結果

替えによる降下を行ったことになる.

第 5 章

拡張値グラフ

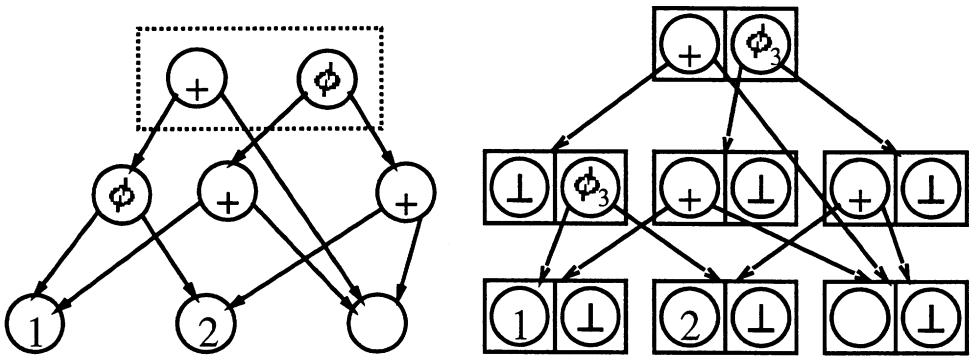
前章では，あるプログラム点における計算式を別のプログラム点へ移動する場合に，依存関係を表す構造が変わる場合があることを述べ，その変化した依存構造は，VG についても変形パターンの適用によって求められることを述べた．しかし，VG の表現形式では，変形前の VG と変形後の VG とに関して，それが同じ計算式の移動によって生じる依存構造であることを明示する方法がなく，そのままでは，コード移動と依存構造を関係付けることができない．そこで，本章では，この問題を解決するために，変形前後の構造を統一的に表現できる新しいグラフ表現，**拡張値グラフ**を導入する．拡張値グラフによって表現されるコード移動前後の計算の依存構造とそれらが表現する値の一致性によって，計算式がどのプログラム点に移動される場合でも，コード移動によって生じる副次的効果を依存構造から直接得ることができる．

この節では，まず拡張値グラフを定義してから，コード移動と拡張値グラフの関係を示し，その作成法を示す．

5.1 拡張値グラフの定義

プログラムの意味を変えずに，前向きまたは後向きに他のプログラム点に移動できる場合，その移動の伴う依存構造の変化は VG の変形で示したとおりである．しかし，変形を適用する前の依存構造と適用した後の依存構造は，互いに別のグラフとなってしまい，1 つの計算式から派生した同じ値を表現しているということは明示されない．図 5.1(a) に示す VG は，同じ計算式を表す図 4.16(a) と (b) の VG を重ねて示したものである．これらの VG の根はどちらも同じ値を表現しているが，独立した節となっているために，その関係は明らかではない．

値グラフの変形前後のグラフは，同じ値を表現するものであるので，図 5.1(a) において点線の矩形で示すように 2 つの根を 1 つの組にまとめて節として表現すると，構造が異なる等価式の表現手段が得られる．他の変形を適用していない節に関しても，静的に値を知ることができないことを意味する $\perp$ をラベルとする節と組にすることによって，図 5.1(b) に示



(a)　　(b)

図 5.1 VG と EVG の関係

すような 2 つの VG 節で 1 つの節を構成する新しいグラフ表現が得られる．このグラフ表現における辺の先は，2 つの VG 節からなる新しい節を指すものとする．このグラフ表現を拡張値グラフ（extended value graph，以降，EVG）と呼ぶ．EVG の定義を次に与える．以下，1 つの EVG 節を構成する 2 つの節をそれぞれ EVG 節の**副節**と呼ぶ．

定義 2 (拡張値グラフ) *EVG* は次の 3 つの集合からなる三つ組 (SV, V, A) である．

SV ： 副節の集合

V ： 副節の組の集合．ここで，$\mathbf{Op} \cup \mathbf{C} \cup \perp$ のラベルをもつ副節集合を SV_{op} とし，$\Phi \cup \perp$ のラベルをもつ副節集合を SV_ϕ とすると，節集合 V は，副節の組 $SV_{op} \times SV_\phi$ である．

A ： 副節から節への辺の集合．

EVG に対する操作として，次の関数を定義する．

$op : V \to SV_{op}$. $(sv_{op}, sv_\phi) \in V$ から sv_{op} を取り出す関数

$phi : V \to SV_\phi$. $(sv_{op}, sv_\phi) \in V$ から sv_ϕ を取り出す関数

$lb : SV \to OP \cup C \cup \Phi$. $sv \in SV$ からラベルを得る関数

$loc : SV_\phi \to N$. ϕ-関数 $sv_\phi \in SV_\phi$ に対応する *CFG* 節を得る関数

$child_i : SV \to V \cup \perp$. $sv \in SV$ の i 番目の後続節 v を得る関数．ここで，$(sv, v) \notin A$ ならば，$\perp$ とする．$sv_\phi \in SV_\phi$ のとき，$child_i(sv_\phi)$ は，$pred_i(loc(sv_\phi))$ から到達する値を表現する．

$Child_{op} : V \to \{V\}$ ． $v \in V$ から $op(v)$ の後続節の集合を得る関数．返値は，
$\{v' | v' = child_i(op(v)) \wedge v' \neq \perp,\ for\ i = 0,\ 1\}$ で表される．

$Child_{\phi} : V \to \{V\}$ ． $v \in V$ から $phi(v)$ の後続節の集合を得る関数．返値は，
$\{v' | v' = child_i(phi(v)) \wedge v' \neq \perp,\ for\ i = 0,\ 1\}$ で表される．

$Parent_{op} : V \to \{V\}$ ． $v \in V$ の先行節である sv が，左の副節 $op(v')$ である v' の集合を得る関数．返値は，$\{v' | op(v') = child_i^{-1}(v),\ for\ i = 0,\ 1\}$ で表される．

$Parent_{\phi} : V \to \{V\}$ ． $v \in V$ の先行節である sv が，右の副節 $phi(v')$ である v' の集合を得る関数．返値は，$\{v' | phi(v') = child_i^{-1}(v),\ for\ i = 0,\ 1\}$ で表される．

図 5.1(b) に示す EVG において，その中の節を v とすると，v によって表記される式は，CFG 節 $loc(phi(v))$ を境に次の異なった意味をもつ．

$loc(phi(v))$ とその後続の CFG 節において ： $op(v)$ を根とする依存構造をもつ．

$loc(phi(v))$ より先行する CFG 節において ： $Child_{\phi}(v)$ に含まれる節を根とする依存構造をもつ．

例： 図 5.2(a) に示す CFG において，節 6 にある計算式 `t := x + y` に注目した場合，この式の各節での依存構造は，図 5.3(a)-(e) に示すそれぞれの VG によって表現できる．CFG 節 1 と 3 における `t` の VG は，同じ構造をもつので，図 5.3(a) ではそれらをまとめて CFG 節集合として示している．このプログラムの依存構造を EVG で表現すると，図 5.2(b) のように表現される．図 5.3における各 VG との対応は次のとおりである．

EVG 節 1 を根とする副グラフ ：図 5.3(e) に示す VG

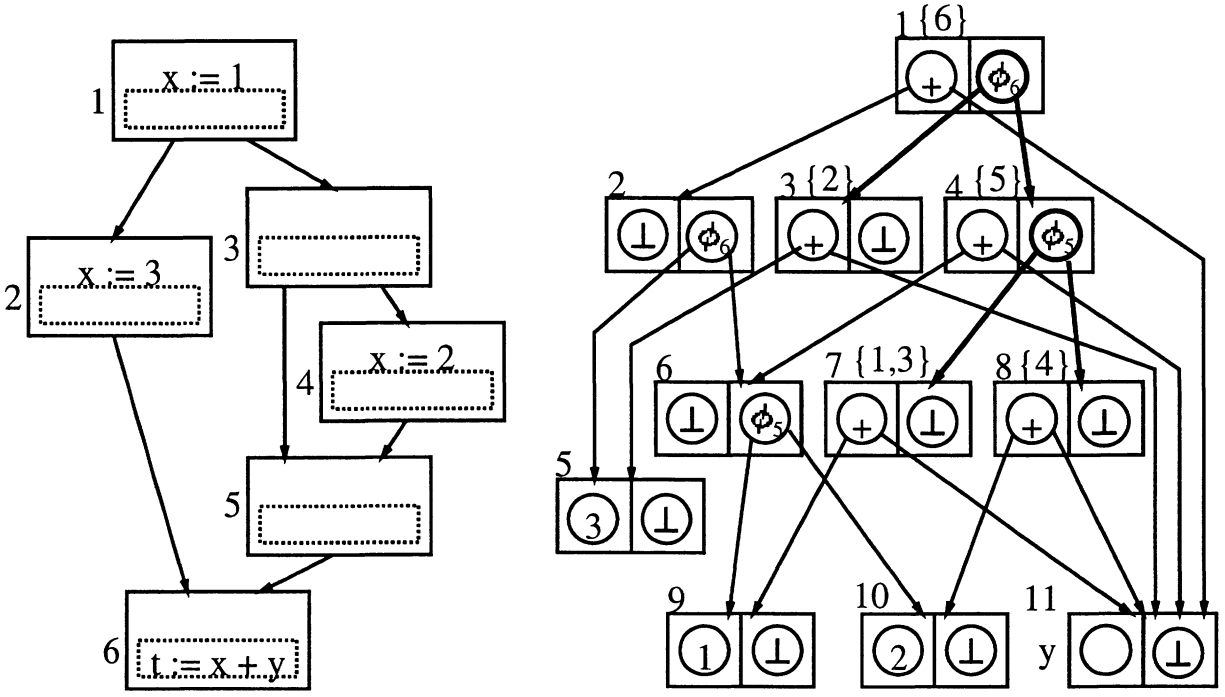
EVG 節 3 を根とする副グラフ ：図 5.3(c) に示す VG

EVG 節 4 を根とする副グラフ ：図 5.3(d) に示す VG

EVG 節 7 を根とする副グラフ ：図 5.3(a) に示す VG

EVG 節 8 を根とする副グラフ ：図 5.3(b) に示す VG ∎

この例が示すように，VG 表現では，同じ計算式について，各プログラム点における依存構造の関係を明示する表現手段をもたないのに対し，EVG 表現では，太線で表した ϕ 副節が，それらの依存構造を結び付ける働きをしていることが分る．



(a) CFG　　(b) CFG に対応する EVG

図 5.2 計算式の移動と EVG

5.2 拡張値グラフの作成

EVG を作成するには，まず VG から，元の計算の依存構造だけを表現するグラフを作ることから始める．ここで作成した EVG を**初期 EVG** と呼ぶ．次に，初期 EVG に対して，後述する変形を適用することによって最終的に用いる EVG が得られる．

VG から初期 EVG を作成する方法は次のとおりである．

1. 各 VG 節を副節 sv とし，それと対をなすもう一方の副節を $sv' \in SV$ $(lb(sv') = \perp)$ とすると，sv と sv' を次のように組にした EVG 節を作る．

   Case 1 ： $lb(sv) \in OP \cup C$ ならば，(sv, sv') を生成する．

   Case 2 ： $lb(sv) \in \Phi$ ならば，(sv', sv) を生成する．

2. VG 節を指している辺の先を，その VG 節を副節としてもつ EVG 節に付け替える．

初期 EVG は VG と同じ構造をもつので，初期 EVG に対して VG の場合と同様の変形を行うことができる．EVG における変形が VG における変形と異なる点は，変形は副節 sv

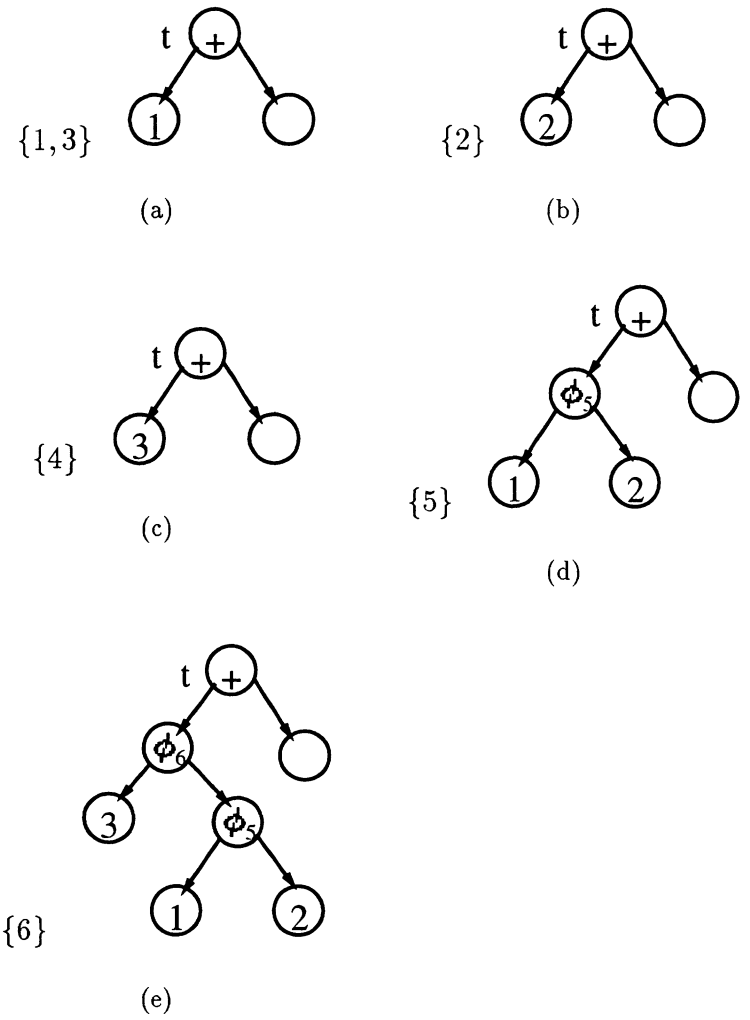


図 5.3 計算式の移動と VG

を根とする副グラフに対して適用し，変形後のグラフの根は，sv と変形後の副節 sv' 節を組にしたもので置き換えるという点である．初期 EVG に適用する変形の詳細を，後向き移動に基づく場合と前向き移動に基づく場合とに分けてそれぞれ示す．

以下では，変形候補の副グラフの根を vr とし，$vr = (sv_{op}, sv_{\phi})$ とする．表記を簡潔にするため，$i = 0, 1$ としたとき，$child_i(sv_{op})$ をそれぞれ $vopc_0$，$vopc_1$，$child_i(sv_{\phi})$ をそれぞれ $v\phi c_0$，$v\phi c_1$ で表す．

[初期 EVG の変形]

1. 左オペランドが複数の定義をもつ場合

後向き移動 ：

- 適用条件：図 5.4(a) に示すとおりである．すなわち，

$lb(sv_{op}) \neq \perp \wedge lb(sv_{\phi}) = \perp \wedge phi(vopc_0) \neq \perp \wedge phi(vopc_0) \neq phi(vopc_1)$.

- 変形：図 5.4(a) から図 5.4(c) に示す構造に変形する．具体的には，次の操作を行う．
  (a) vr のコピーによって 2 つの新しい節 nvr_0 と nvr_1 を生成する．
  (b) $lb(sv_{\phi})$ を $lb(phi(vopc_0))$ で置き換え，sv_{ϕ} から $nvr0$ と $nvr1$ への辺を生成する．
  (c) $op(nvr_i)$，$i = 0,1$ から，$child_i(phi(vopc_0))$ への辺と $vopc_1$ への辺を生成する．

前向き移動 ：

- 適用条件（図 5.5(b)）：$lb(sv_{op}) = \perp \wedge lb(sv_{\phi}) \neq \perp \wedge lb(v\phi c_0) = lb(v\phi c_1) \wedge child_1(op(v\phi c_0)) = child_1(op(v\phi c_1))$，かつ次の条件を満たす EVG 節 vp が存在する．
  $lb(sv_{\phi}) = lb(phi(vp)) \wedge child_i(phi(vp)) = child_0(op(v\phi c_i))$，$i = 0,1$
- 変形（図 5.5(b) から図 5.5(c)）：
  (a) $lb(op(v\phi c_0)) = lb(op(v\phi c_1))$ で $lb(sv_{op})$ を置き換える．
  (b) sv_{op} から，vp への辺と $child_1(op(v\phi c_0))$（$child_1(op(v\phi c_1))$ と同じ）への辺を生成する．

2. 右オペランドが複数の定義をもつ場合

後向き移動 ：

- 適用条件（図 5.5(a)）：
  $lb(sv_{op}) \neq \perp \wedge lb(sv_{\phi}) = \perp \wedge phi(vopc_1) \neq \perp \wedge phi(vopc_0) \neq phi(vopc_1)$.
- 変形（図 5.5(a) から図 5.5(c)）：
  (a) vr をコピーすることによって 2 つの新しい節 nvr_0 と nvr_1 を生成する．
  (b) $lb(sv_{\phi})$ を $lb(phi(vopc_0))$ で置き換え，sv_{ϕ} から $nvr0$ と $nvr1$ への辺を生成する．
  (c) $op(nvr_i)$，$i = 0,1$ から，$child_i(phi(vopc_1))$ への辺と $vopc_0$ への辺を生成する．

前向き移動 ：

- 適用条件（図 5.5(b)）：$lb(sv_{op}) = \perp \wedge lb(sv_{\phi}) \neq \perp \wedge lb(v\phi c_0) = lb(v\phi c_1) \wedge child_0(op(v\phi c_0)) = child_0(op(v\phi c_1))$，かつ次の条件を満たす EVG 節 vp が存在する．
  $lb(sv_{\phi}) = lb(phi(vp)) \wedge child_i(phi(vp)) = child_1(op(v\phi c_i))$，$i = 0,1$

- 変形（図 5.5(b) から図 5.5(c)）：
  (a) $lb(op(v\phi c_0)) = lb(op(v\phi c_1))$ で $lb(sv_{op})$ を置き換える.
  (b) sv_{op} から，vp と $child_0(op(v\phi c_0))$（$child_0(op(v\phi c_1))$ と同じ）への辺を生成する.

3. 左右両方のオペランドが複数の定義をもつ場合

後向き移動：

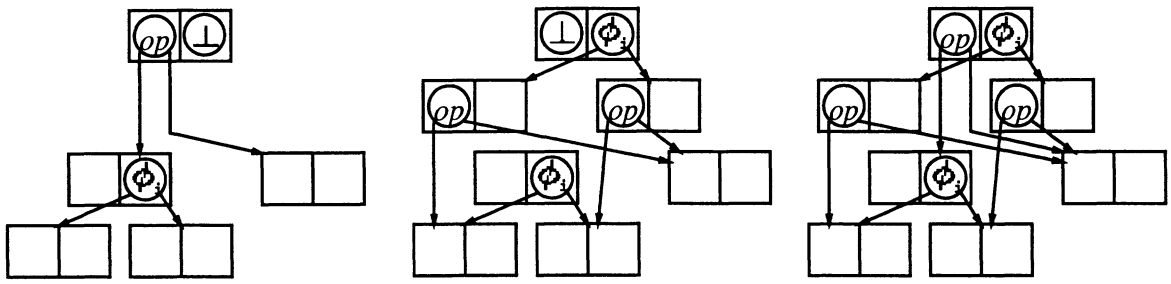
- 適用条件（図 5.6(a)）：
  $lb(sv_{op}) \neq \perp \wedge lb(sv_\phi) = \perp \wedge phi(vopc_0) = phi(vopc_1) \neq \perp$
- 変形（図 5.6(a) から図 5.6(c)）：
  (a) vr をコピーすることによって 2 つの新しい節 nvr_0 と nvr_1 を生成する.
  (b) $lb(sv_\phi)$ を $lb(phi(vopc_0))$（$lb(phi(vopc_1))$ と同じ）で置き換え，sv_ϕ から，$nvr0$ への辺と $nvr1$ への辺を生成する.
  (c) $op(nvr_i)$，$i = 0,1$ から $child_i(phi(vopc_j))$，$j = 0,1$ への辺を生成する.

前向き移動：

- 適用条件（図 5.6(b)）：$lb(sv_{op}) = \perp \wedge lb(sv_\phi) \neq \perp \wedge lb(v\phi c_0) = lb(v\phi c_1)$,
  かつ $lb(sv_\phi) = lb(phi(vp_0)) = lb(phi(vp_1))$,
  かつ $i = 0,1$ と $j = 0,1$ について，$child_i(phi(vp_j)) = child_j(op(v\phi c_i))$ である EVG 節 vp_0，vp_1 が存在する.
- 変形（図 5.6(b) から図 5.6(c)）：
  (a) $lb(op(v\phi c_0)) = lb(op(v\phi c_1))$ で $lb(sv_{op})$ を置き換える.
  (b) sv_{op} から，vp_0 への辺と vp_1 への辺を生成する.

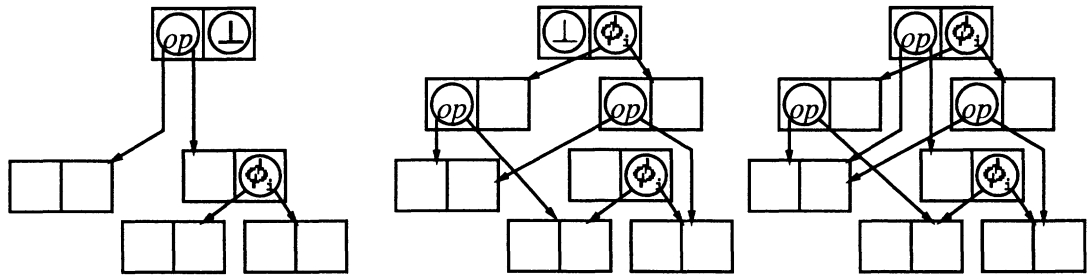
4. 左右両方のオペランドが同じ定義をもつ場合
VG における変形と同様に，この変形が生じるのは，前向き移動の場合だけである.

- 適用条件（図 5.7(a)）：$lb(sv_{op}) = \perp \wedge lb(sv_\phi) \neq \perp \wedge$
  $child_0(op(v\phi c_0)) = child_0(op(v\phi c_1)) \wedge child_1(op(v\phi c_0)) = child_1(op(v\phi c_1))$.
- 変形（図 5.7(a) から図 5.7(b)）：
  (a) $lb(op(v\phi c_0)) = lb(op(v\phi c_1))$ で $lb(sv_{op})$ を置き換える.
  (b) sv_{op} から，$child_0(op(v\phi c_0))$（$child_0(op(v\phi c_1))$ と同じ）への辺と $child_1(op(v\phi c_0))$（$child_1(op(v\phi c_1))$ と同じ）への辺を生成する.



(a) 後向き変形前の初期 EVG　(b) 前向き変形前の初期 EVG　(c) EVG

図 5.4 左オペランドが複数の節に依存している場合の変形



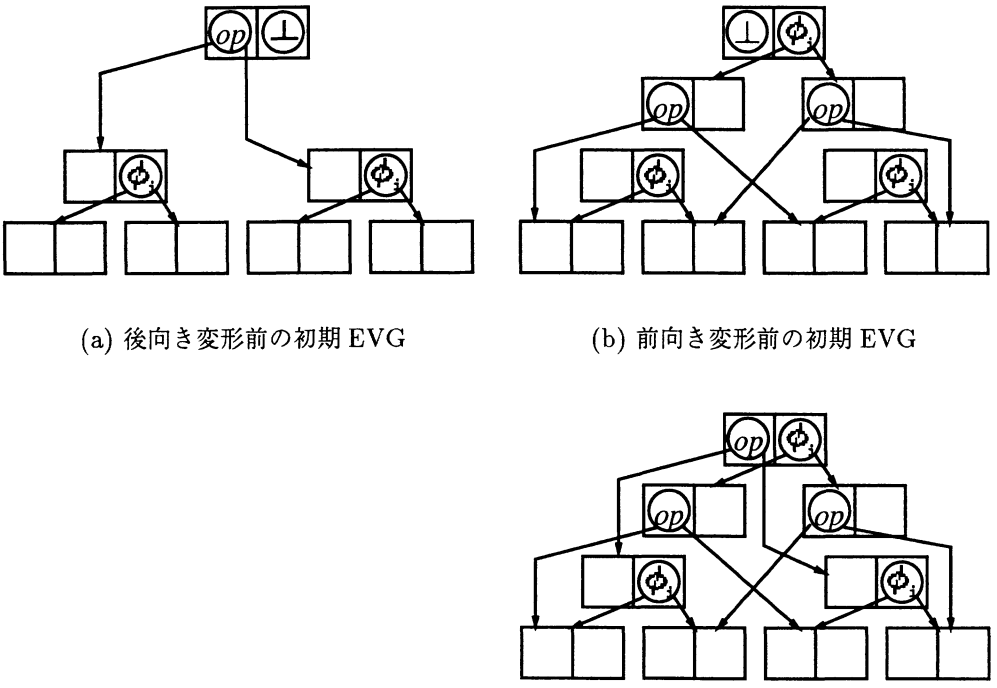
(a) 後向き変形前の初期 EVG　(b) 前向き変形前の初期 EVG　(c) EVG

図 5.5 右オペランドが複数の節に依存している場合の変形

4.2.2節で述べたように，変形の適用条件を満たす副グラフに対して変形を繰返し適用していくと，変形が停止しないことがある．EVG の目的は，各プログラム点における依存構造を表現することであるので，CFG の各節を巡回しながら，その間に出会った ϕ 節について変形パターンの適用を 1 回行うだけでよい．初期 EVG から EVG を作成する際に適用する変形が前向き変形である場合も後向き変形である場合も，変形の適用条件となる EVG の副グラフパターンには，副節として ϕ 節が含まれることが分かる．このことから，初期 EVG への変形の適用は次のように行う．

後向き変形 ：CFG 節を終了節から後向きに巡回する．その際，各節は 1 度だけ訪れることにする．この過程で ϕ- 関数に出会うたびに，対応する ϕ 節に依存する節を根とする EVG の副グラフに対して後向き変形を適用する．

前向き変形 ：CFG 節を開始節から前向きに巡回する．その際，各節は 1 度だけ訪れることにする．この過程で，ϕ- 関数に出会うたびに，対応する ϕ 節に依存する節を根と



(a) 後向き変形前の初期 EVG　　(b) 前向き変形前の初期 EVG

(c) EVG

図 5.6 左右両方のオペランドが複数の節に依存している場合の変形

する EVG の副グラフに対して前向き変形を適用する.

CFG を 1 巡する間に，初期 EVG を変形することによって得られる EVG は，各プログラム点の依存構造をすべて表現していることになる.

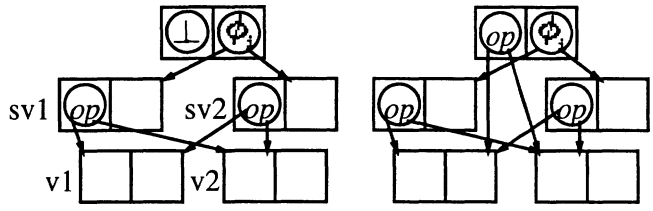
5.3 EVG を用いた定数畳込みと等価式発見

この節では，EVG に基づいて等価式を発見する手法を述べる. EVG は，その作成過程で VG の変形を適用するので，同じ値を表現する異なる依存グラフ構造を一緒に保持している. この性質を用いると，VG に変形を適用しながら行う等価式発見の効果を，後向き変形で作成した EVG 上で得ることができる. また，変形によって定数畳込みが可能になる依存グラフ構造についても，その変形済のグラフ構造を一緒にもっているので，VG に変形を適用しながら行う定数畳込みを等価式発見と同時に行うことができる.

EVG に基づく等価式の発見の手順は次のとおりである.

- EVG 節を深さ優先順序の逆順で訪れながら（同じ節は 2 回訪れない），各 EVG 節について次の操作を行う.

  1. 依存先がすべて定数の場合は，定数畳込みを行う.



(a) 前向き変形前の初期 EVG　　(b) EVG

図 5.7 左右両方のオペランドが同じ節に依存している場合の変形

2. ハッシュ表を検査して，等価な式を発見する（ハッシュ表は EVG 節をそのまま登録するものとする）．

畳込みによって，新たに定数となった節から出ている依存辺は取り去るようにする．

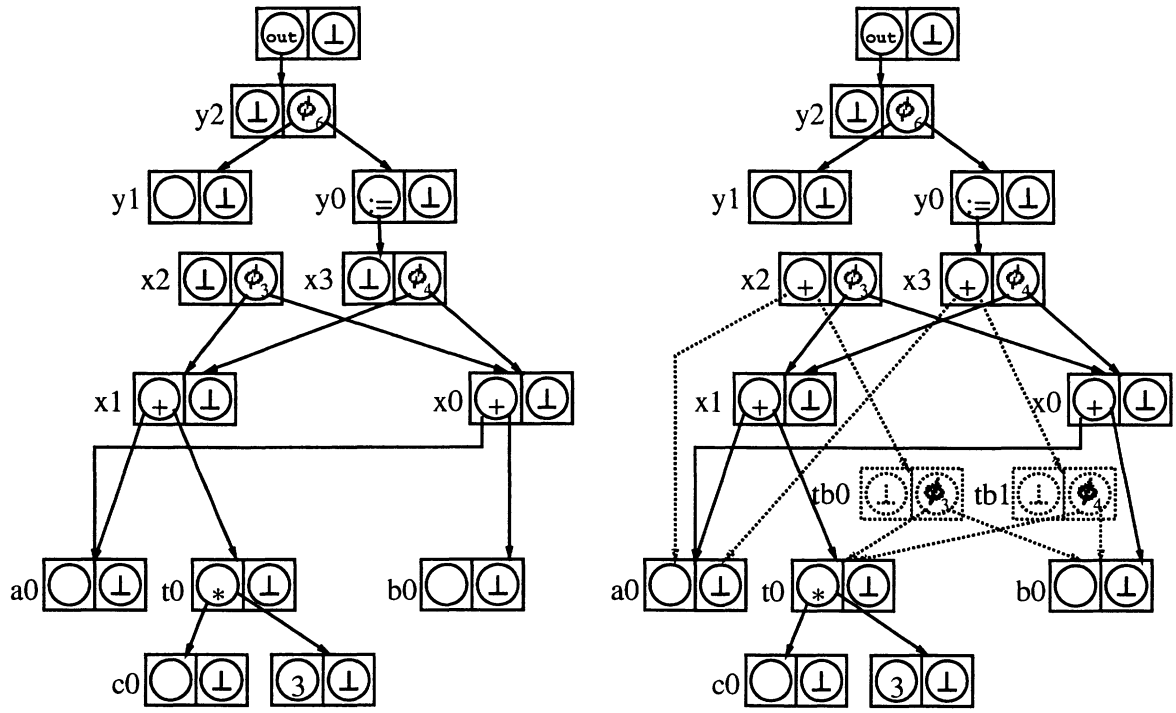
EVG 節を x としたとき，ハッシュ表による等価式の発見は，$(op(x), x')$，$(op(x), x'')$，$(phi(x), y')$，$(phi(x), y'')$ を辺として，次の 3 通りの場合に分けることができる．

1. $\mathbf{lb(op(x))} \neq \bot \wedge \mathbf{lb(phi(x))} = \bot$ の場合
   $(lb(op(x))\ x'\ x'')$ をキーとして，ハッシュ表を検査する．もし，表中に x と等価な式を表す節（以下，等価節と呼ぶ）v が存在すれば，EVG 節 x を v で置き換える．存在しなければ，x をハッシュ表に登録する．

2. $\mathbf{lb(op(x))} = \bot \wedge \mathbf{lb(phi(x))} \neq \bot$ の場合
   $(lb(phi(x))\ y'\ y'')$ をキーとして，ハッシュ表を検査する．もし，表中に等価節 v が存在すれば，EVG 節 x を v で置き換える．存在しなければ，x をハッシュ表に登録する．

3. $\mathbf{lb(op(x))} \neq \bot \wedge \mathbf{lb(phi(x))} \neq \bot$ の場合
   $(lb(op(x))\ x'\ x'')$ と $(lb(phi(x))\ y'\ y'')$ をそれぞれキーとして検査し，少なくとも片方の検査で，表中に等価節 v が存在すれば，EVG 節 v を x で置き換え，v に依存する節をすべて登録し直す．存在しなければ，x をハッシュ表に登録する．

場合 1，2では，EVG 節 x を表中の v で置き換え，場合 3では，表中の v を x で置き換えている理由は，節 x を処理したのちの検査において，$(lb(op(x))\ x'\ x'')$ と $(lb(phi(x))\ y'\ y'')$ の 2 つのキーの検査結果が同一の EVG 節である必要性があるからである．



(a) 変形前　　(b) 変形後

図 5.8 変形適用条件の緩和

5.4 EVG を用いた変数付替え

初期 EVG に対する変形適用条件を緩和することによって，4.3.5節で述べた VG における変形を用いた変数の付替えは，次の例に示すとおり，EVG についても行うことができる．

例： 図 3.11(a) に示したプログラムから初期 EVG を作成すると，図 5.8(a) に示す結果が得られる．`x2` の値に対応する節や `x3` の値に対応する節は，それぞれ辺の先が同じラベル `+` をもつが，それらのオペランドは変形の適用条件を満たさないので，変形の対象にならない．その理由は，`x0` に対応する節と `x1` に対応する節に関して，左から出る辺の先はそれぞれ同じ `a0` に対する節であるが，右から出る辺の先は異なっていて，それらを結合させる ϕ 節もないからである．図 2.7からも，`x1 := a0 + t0` と `x0 := a0 + b0` は結合できないことが分かる．

この場合，図 5.8(b) の点線で示した節のように，一時変数 `tb0`，`tb1` を表す節を挿入する．これらの節をそれぞれ (sv, sv') とし，`x2` に対応する節を $vx2$ とすると，(sv, sv') は $lb(sv) = \bot \wedge lb(sv') = lb(phi(vx2)) \wedge loc(sv') = loc(phi(vx2))$ の条件を満たす．それぞれの sv' からは `b0` と `t0` に対応する節へ辺を生成する．この操作は，通常形式の代入文に

おいて，異なる代入先の変数名を同じ名前に付け替えることに対応し，さらなる変形が可能になる．∎

第 6 章

EVG に基づくデータフロー解析

本章では，EVG が表現する各プログラム点における計算の依存情報を利用して，コード移動に基づく最適化の副次的効果を 1 回のデータフロー解析によって予測し，直接効果として得る方法を示す．データフロー解析では，異なる依存構造をもつ式ごとに，それぞれ異なるスロットを割り当てることによって，各プログラム点における依存情報を解析に反映させる．ただし，この方法を直接用いる場合には，本来 1 つの値を生成する式であっても，移動した先のプログラム点によって依存構造が異なる式ごとに別のスロットを割り当ててしまうという問題が生じてしまう．

この問題の解決のために，EVG の ϕ 節に着目し，それらの辺が同じ式の異なった依存構造を繋ぐという性質を利用する．計算式にスロットを割り当てる場合，各プログラム点におけるその計算式の依存構造はスロットどうしの関係として関連付けることができるので，ϕ 節に対応するスロットと ϕ 節によって繋がれる依存構造に対応するスロットを同一のフロー情報のために用いることができる．

本章では，ϕ 節を利用することによって，異なるスロットを関連付ける方法を示し，制御フローと依存構造を組み合わせた EVG 上のデータフロー解析の枠組を与える．そのあと，コード移動に基づく最適化の代表的な例として，部分冗長除去と部分不要コード除去をとり上げ，EVG 上のデータフロー解析によるそれらの実現法について述べる．

6.1 データフロー解析とデータスロット

データフロー解析には，2.2.1節で示したとおり，ワークリストを用いて各データスロットごとにフロー情報を伝播させるスロットワイズ法を用いる．各スロットは，計算式を表現する EVG 節と，CFG 節との組によって識別することにし，各プログラム点における依存構造を参照できるようにする．このスロット集合を $SL \subset V \times N$ で表す．

一般に，コード移動に基づく最適化においては，計算式の移動を解析するので，同一の字句形式で表現される計算式ごとにスロットを割り当てる． SL で定義したスロットについても，同様に考えると，通常のプログラム形式では，1 つのスロットで表現されたものが，

SSA 形式では複数のスロットによって表現される場合がある.

例： 図 6.1(a) は, `a + b` の計算を行う CFG を表してる. この `a + b` に対する EVG を作成すると, 図 6.1(b) のようになる. スロットが CFG 節と EVG 節の組みによって識別されるという性質から, `a + b` に対するスロットを考えると, 異る EVG 節に対応するスロットは, 別々に用意しなければならない. 図 6.1(b) の EVG において, `a + b` の EVG 節は, 節 1, 3, 4 の 3 つが存在するので, 図 6.1(a) の CFG 節の右に示したように, スロットも 3 つ必要になる. 図 6.1(a) の点線矩形の上あるいは下に示した番号は, 対応する EVG 節を表す. 各 CFG 節において, 適切な依存構造を表す EVG に対応するスロットは, CFG 節 1, 2 においては, 左から 2 番目のスロットであり, CFG 節 3 においては, 3 番目のスロットであり, CFG 節 4 においては, 1 番目のスロットである. ∎

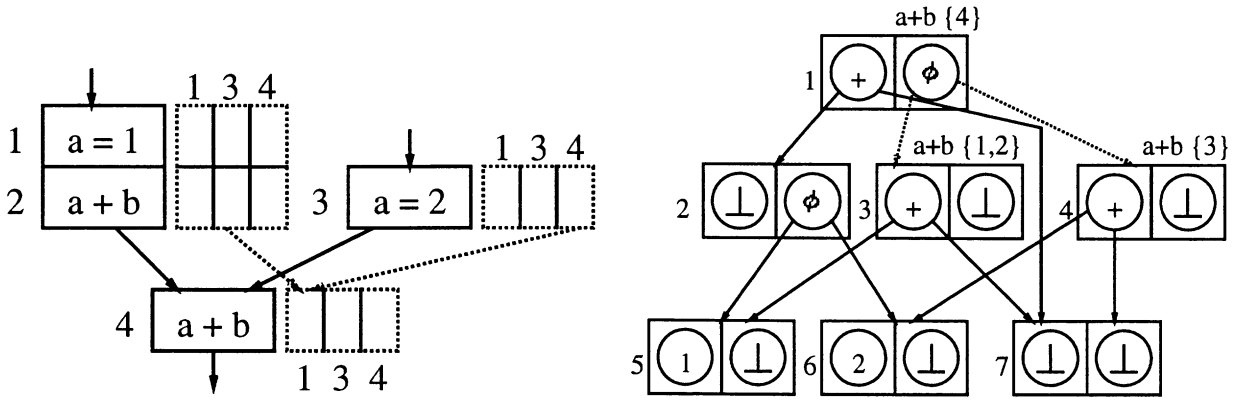
上記の例で, 1 つの計算式に対して複数のスロットが必要になるのは, 4.2節で示したように, ϕ- 関数のプログラム点の前後で, EVG 節の左の副節の依存構造と右の副節の辺の先の依存構造とが入れ替わり, 式としては別のものとして取り扱う必要があるからである.

EVG 節の右の副節が表す ϕ 節に着目すると, 5.1節で述べたように, この ϕ 節から出る辺が, 依存構造が変わる前後の値グラフを繋いでいる. この ϕ 節の依存関係を用いると, ϕ 節に対応する ϕ- 関数が存在するプログラム点において, ϕ 節が繋ぐ EVG 節をもつスロットに互いにデータを伝播させることによって, 従来と同様にデータフロー解析を行うことができる.

例： 図 6.1(b) において, EVG 節 1 の右の副節 (ϕ 節) から出る辺によって, EVG 節 1 は EVG 節 3, 4 を繋いでいる. この関係から, 図 6.1(a) の CFG 節 4 にある 1 番目のスロットと CFG 節 2, 3 にある 2, 3 番目のスロットを, 互いにデータを伝播させる先として扱うことができる. ∎

スロット $sl1$ の情報を次にスロット $sl2$ に伝播させなければならないとき, $sl1$ (あるいは $sl2$) を $sl2$ (あるいは $sl1$) の**隣接スロット**と呼ぶ. データを伝播させるべきスロットは, 各スロットに対する隣接スロットとして次のように定義することができる.

EVG 節 v とそれに対応する CFG 節 n によって識別されるスロットを $sl \equiv (v, n)$ とするとき, sl の隣接スロットは, その先行スロット $pred_{SL}(sl)$ と後続スロット $succ_{SL}(sl)$ を定義することによって定めることができる. まず, sl が対応する CFG 節について, i 番目の先行節に含まれるスロット $pred_{SLi}(sl)$ と後続節に含まれるスロット $succ_{SLi}(sl)$ を次のように定義する.



(a) データスロット　　(b) 対応する EVG

図 6.1 EVG とデータスロット

- 先行スロット $pred_{SLi}(sl)$:
  - もし，$loc(phi(v)) = n$ ならば，$v' = child_i(phi(v))$ として $(v', pred_i(n))$ である（図 6.2）.
  - もし，$\exists(v' \neq v).loc(phi(v')) = n \wedge v \in Child_\phi(v')$ ならば，存在しない（図 6.3）.
  - さもなければ，$(v, pred_i(n))$ である.
- 後続スロット $succ_{SLi}(sl)$:
  - もし，$\exists(v' \neq v).v' \in Parent_\phi(v) \wedge loc(phi(v')) = succ_i(n)$ ならば，$(v', succ_i(n))$ である（図 6.4）.
  - もし，$succ_i(n) = loc(phi(v))$ ならば，存在しない（図 6.5）.
  - さもなければ，$(v, succ_i(n))$ である.

$pred_{SLi}(sl)$ と $succ_{SLi}(sl)$ によって，$pred_{SL}(sl)$ と $succ_{SL}(sl)$ を次のように定義する.

- $pred_{SL}(sl) =_{def} \{sl' | sl' \equiv pred_{SLi}(sl),\ for\ i = 0, 1\}$.
- $succ_{SL}(sl) =_{def} \{sl' | sl' \equiv succ_{SLi}(sl),\ for\ i = 0, 1\}$.

さらに，$pred_{SL}(sl)$ と $succ_{SL}(sl)$ を用いて，開始スロット SL_{s} と終了スロット SL_{e} を次のように定義する.

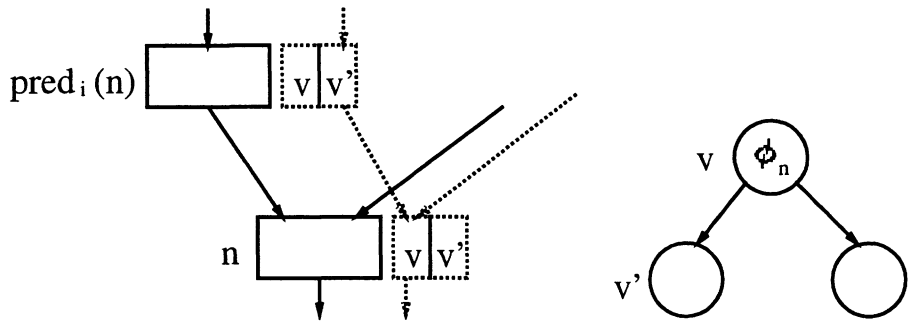


図 6.2 先行スロット 1

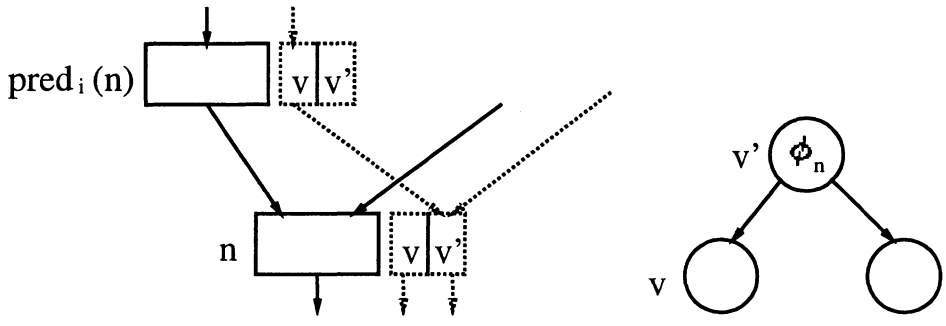


図 6.3 先行スロット 2

- $SL_{\mathrm{s}} =_{def} \{sl \mid pred_{SL}(sl) = \emptyset\}$.
- $SL_{\mathrm{e}} =_{def} \{sl \mid succ_{SL}(sl) = \emptyset\}$.

例： 図 6.6(a) に対する PDE のデータフロー解析のスロットの一部を図 6.6(b) に示す．図 6.6(b) において，点線の矩形は各 CFG 節を表し，実線の矩形は各 CFG 節に対応するスロットの集合を表す．また，点線矢印は CFG 辺を表し，実線矢印はスロットの隣接関係を表す．ただし，それ以外の隣接スロットは，隣接 CFG 節上の同じ位置（同じ変数名）で表現し，実線矢印は省略してある．例えば，CFG 節 3 の `x2` スロットの先行スロットは節 33 の `x1` スロットと節 2 の `x0` スロットであり，後続スロットは節 34 の `x2` スロットである． ∎

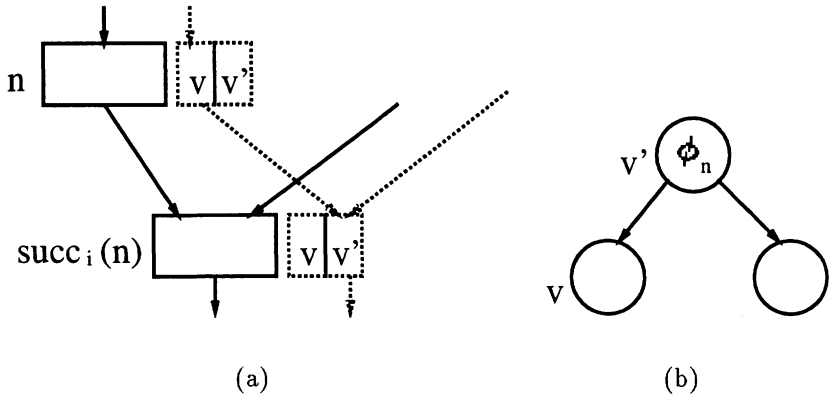


図 6.4 後続スロット 1

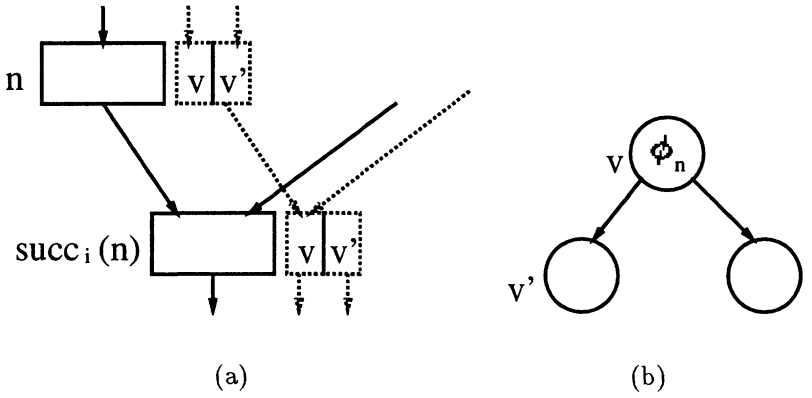
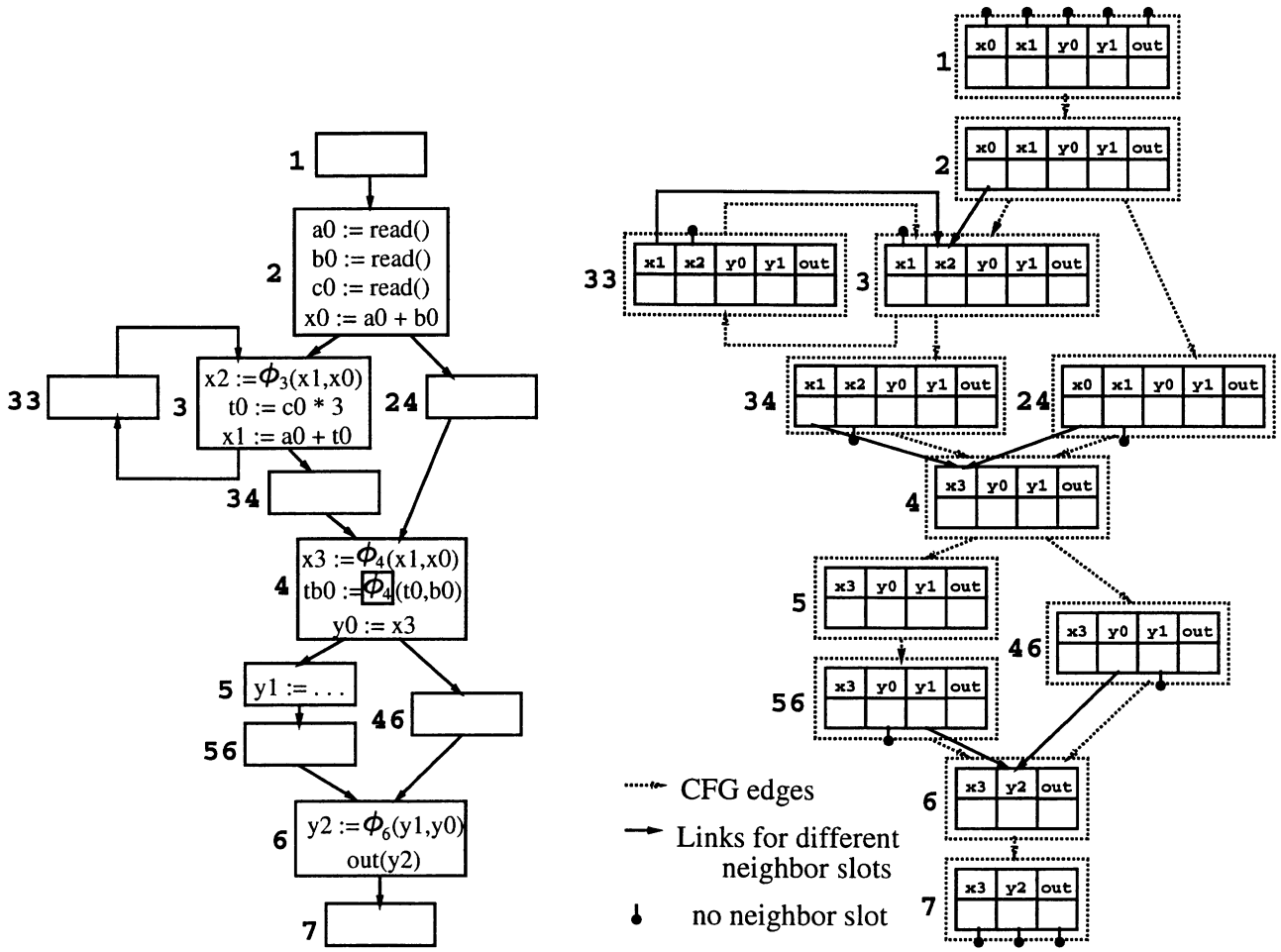


図 6.5 後続スロット 2

6.1.1 部分冗長除去

この節では，従来の PRE が繰返し適用によって得ていた副次的効果を，前節で定義した隣接スロットと EVG の依存情報を用いて，直接効果とし見付け出す PRE について述べる．ここで用いる PRE 法は，Knoop，Rüthing，Steffen らの手法 [KRS94a, KRS92] を変形したものである．PRE 法の多くが双方向のデータフロー解析であるのに対し，本手法は，巻上げと遅延をそれぞれ，後向きと前向きの単方向のデータフロー解析によって実現する．双方向の解析を単方向にしたことによって，各式はまず，$EARLIEST_{(v,n)} = true$ であるプログラム開始点に最も近い巻上げ可能な点 n へ巻上げられるので，巻上げによる効果（定数畳込みなど）を確実に得ることができる．最終的に決まる巻上げ点は，$LATEST_{(v,m)} = true$ の無効な巻上げを巻き戻した点 m として求められる．



(a) プログラム例　　(b) PDE におけるスロット

図 6.6 隣接スロット

PRE の解析に用いるデータフロー方程式を図 6.7～図 6.10に示す．この方程式は，基本ブロックをプログラム点とする述語からなる．基本ブロックの入口，出口におけるそれぞれのフロー情報に対応する述語は，接頭語 “N-”，“X-” によって区別している．このデータフロー解析を解くことによって，従来の PRE を繰返し適用しなければ得られなかった式の挿入点を求めることができる．

巻上げ可能なプログラム点の計算

各方程式中の各述語の意味は次のとおりである．

$COMP_{(v,n)}$ ： EVG 節 v に対応する式が基本ブロック n に存在するときだけ $true$，それ以外の場合は $false$ とする．

N-（あるいは X-）$HOISTABLE_{(v,n)}$ ：EVG 節 v によって表される式が，CFG 節 n の入口（N-），出口（X-）において，それぞれ巻上げ可能である場合に $true$，それ以外の場合は $false$ とする．

N-（あるいは X-）$REDUNDANT_{(v,n)}$ ：EVG 節 v によって表される式が，CFG 節 n の入口，出口において，それぞれ存在するか，あるいは冗長になる場合に $true$，それ以外の場合は $false$ とする．

$DEFINED_{(v,n)}$ ：EVG 節 v によって表される式のオペランドに対する定義が，基本ブロック n で使用できる場合に $true$，それ以外の場合は $false$ とする．

図 6.7は，どの実行パスにも新しい式を導入しないで，プログラム中に現れるすべての式について巻上げ可能な最大範囲と，冗長になる範囲を求めるための方程式である．

N-$HOISTABLE_{(v,n)}$，X-$HOISTABLE_{(v,n)}$ は，それぞれ後向きのデータフロー解析によって決定される述語であり，巻上げ可能なプログラム点を計算する．また，N-$REDUNDANT_{(v,n)}$，X-$REDUNDANT_{(v,n)}$ は，それぞれ前向きのデータフロー解析で決定される述語であり，巻上げによって冗長になる式を見付けるために用いる．

$DEFINED_{(v,n)}$ は，対象とする式 v の依存先 $v' \in Children(op(v))$ が，1 つでも基本ブロック n に対応するスロット (v', n) について N-$HOISTABLE_{(v',n)} = false$ であれば，$DEFINED_{(v,n)} = false$ になる．この $DEFINED_{(v,n)}$ によって，v に対応する式の移動の情報 N-$HOISTABLE$ が定義から使用へ伝播され，計算順序を保持しながら，すべての式の巻上げ可能なプログラム点を同時に計算することが可能になる．

図 6.7に示す巻上げに関するデータフロー解析は，SL_{e} と基本ブロックに固定 (pinned) のコードが存在する節に限って $false$，他のすべての節を $true$ とした初期状態に対する最大解の計算である．冗長性に関する解析は，SL_{s} だけ $false$ で，他のすべての節を $true$ とする初期状態における最大解の計算である．

最大巻上げ点の計算

次に，N-$HOISTABLE$，X-$HOISTABLE$，N-$REDUNDANT$，X-$REDUNDANT$ の情報を用いて，式の最大巻上げ点のうちで冗長にならないもの，すなわち最低限挿入の必要な場所を決定する．この挿入を表す述語が図 6.8の $EARLIEST_{(v,n)}$ である．この値は各スロットについての局所的な計算によって求めることができる．

遅延可能なプログラム点の計算

最後に，巻上げすぎた式を遅延させるためのデータフロー解析 (図 6.9) を行う．各述語の意味は次のとおりである．

N-（あるいは X-）$DELAYED_{(v,n)}$ ： EVG 節 v によって表される式が，CFG 節 n の入口（あるいは出口）において，安全に遅延させることができる場合（以降，**遅延可能**であるという）に $true$，それ以外の場合は $false$ とする．

$EFFECTVE_{(v,n)}$ ： EVG 節 v によって表される式が，CFG 節 n に巻き上げられることによって，冗長性が除去される以外の有利な効果が期待できる場合に $true$，それ以外の場合は $false$ とする．

$USED_{(v,n)}$ ： EVG 節 v によって表される式の代入先になっている変数の使用が，CFG 節 n よりも後に存在する場合に $true$，それ以外の場合は $false$ とする．

述語 $N\text{-}DELAYED_{(v,n)}$，$X\text{-}DELAYED_{(v,n)}$ は，対象にしている式 v に関して，それぞれ前向きデータフロー解析式によって決定される述語であり，最大巻上げ点から元の v の位置までの範囲内で，遅延可能な点を計算する．遅延は，巻上げによって得られる効果を損なわないようにして行われなければならない（ループの外に出た式は元に戻さない）．

冗長性の除去以外に巻上げによって生じる効果は，$EFFECTVE_{(v,n)}$ の情報を用いて遅延を止めることによって得られる．図 6.8に示すように，$EFFECTVE_{(v,n)}$ は，5.3節で示した巻上げによって定数の畳込みができるようになる効果と，異なる値の式が巻上げによって 1 つになるため，片方を冗長にできるという効果を表現する．遅延は，この $EFFECTVE_{(v,n)}$ によって，巻上げで得られた効果を損なわないようにブロックされる．

また，$USED_{(v,n)}$ は，$DEFINED_{(v,n)}$ と同様に，EVG をデータフロー解析に反映させるためのもので，対象とする式 v に依存している式 $v' = Parent_{op}(v)$ が，基本ブロック n に対応するスロット (v', n) について $X\text{-}DELAYED_{(v,n)} = false$ となるものがあれば，$USED_{(v,n)} = false$ になる．この $USED_{(v,n)}$ によって，v に対する式の移動の情報 $X\text{-}DELAYED$ が，使用から定義に伝播され，計算順序を保持しながら，すべて式の遅延可能なプログラム点を同時に計算することが可能になる．

[挿入点の計算]

最終的な挿入点の計算は，図 6.10の方程式で与えられる．$LATEST_{(v,n)}$ は n が巻上げを最も遅延できる場所であることを表していて，遅延が安全でなくなるプログラム点でブロックされる場合を除けば，n は元の v の位置と同じになる．

$$
\begin{aligned}
N\text{-}HOISTABLE_{(v,n)} &= DEFINED_{(v,n)} \wedge X\text{-}HOISTABLE_{(v,n)} \\
X\text{-}HOISTABLE_{(v,n)} &= COMP_{(v,n)} \\
&\quad \vee \begin{cases} false & if\ (v,n) = SL_{\mathbf{e}} \\ \prod_{m \in succ(n)} \sum_{\substack{(v',n') \in succ(v,n) \\ n'=m}} N\text{-}HOISTABLE_{(v',n')} & otherwise \end{cases} \\
DEFINED_{(v,n)} &= \begin{cases} false & if\ v\ is\ pinned\ on\ n \\ \prod_{v' \in Children(op(v))} N\text{-}HOISTABLE_{(v',n)} & otherwise \end{cases} \\
N\text{-}REDUNDANT_{(v,n)} &= \\
&\quad \begin{cases} false & if\ (v,n) = SL_{\mathbf{s}} \\ \prod_{(v',n') \in pred_{SL}(v,n)} (COMP_{(v',n')} \vee X\text{-}REDUNDANT_{(v',n')}) & otherwise \end{cases} \\
X\text{-}REDUNDANT_{(v,n)} &= (COMP_{(v,n)} \vee N\text{-}REDUNDANT_{(v,n)})
\end{aligned}
$$

図 6.7 巻上げ部のデータフロー方程式

$$
\begin{aligned}
EARLIEST_{(v,n)} &= N\text{-}HOISTABLE_{(v,n)} \\
&\quad \wedge \prod_{(v',n') \in pred_{SL}(v,n)} \neg(X\text{-}REDUNDANT_{(v',n')} \vee X\text{-}HOISTABLE_{(v',n')}) \\
&\quad \vee X\text{-}HOISTABLE_{(v,n)} \wedge \neg DEFINED_{(v,n)} \\
EFFECTIVE_{(v,n)} &= \sum_{m \in succ(n)} \Big(\sum_{\substack{(v',n') \in succ(v,n) \\ n'=m}} lb(op(v)) \in \mathbf{C} \wedge lb(op(v'))) \notin \mathbf{C} \\
&\quad \vee \begin{cases} true & if\ more\ than\ one\ X\text{-}HOISTABLE_{\{(v',n')|n'=m\}}\ are\ true \\ false & otherwise \end{cases} \Big)
\end{aligned}
$$

図 6.8 効果的な巻上げ

例： 図 3.11(a) に対するデータフロー解析の結果の一部を図 6.11 に示す．点線矩形は，CFG 節を表し，実線の矩形は，その節に対応するスロットの集合を表す．実線矩形の各列は，一番上の行に示した式に対応するスロットである．2 行目のスロットと 3 行目のスロットは，それぞれ CFG 節の入口のスロットと出口のスロットを表している．各スロットは，述語ごとに用意しなければならないが，ここでは図を単純にするために，すべての述語を 1 つの箱で表し，*true* となった述語だけを異なった形状で示している．図中の灰色の矩形，R 印の矩形，黒色の矩形，E 印の矩形は，それぞれ接頭字 *N-* あるいは *X-* をもつ *HOISTABLE*, *REDUNDANT*, *DELAYED*, *EFFECTIVE* が *true* であることを意味している．

結果として，*LATEST* = *true* となるのは，節 1 の出口における `x1 + y1`，同じく入口における `i1 + 1`，節 6 の出口における `i2 + 1`，節 3 の出口における `a1 + b1`，節 4 の入口における `a2 + b1` となる．この結果から，例えば `x + y` については，部分冗長性の除去とループ外移動を行うことができ，`i + 1` については，節 1 において定数へ畳み込めることが分る．

$$
\begin{aligned}
N\text{-}DELAYED_{(v,n)} &= EARLIEST_{(v,n)} \\
&\vee \begin{cases} false & if\ (v,n) \in SL_{\mathbf{S}} \\ \prod_{(v',n') \in pred_{SL}(v,n)} X\text{-}DELAYED_{(v',n')} & otherwise \end{cases} \\
X\text{-}DELAYED_{(v,n)} &= N\text{-}DELAYED_{(v,n)} \wedge \neg COMP_n \wedge USED_{(v,n)} \wedge \neg EFFECTIVE_{(v,n)} \\
USED_n &= \prod_{v' \in Parent_{op}(v)} X\text{-}DELAYED_{(v',n)}
\end{aligned}
$$

図 6.9 遅延部のデータフロー方程式

$$
\begin{aligned}
LATEST_{(v,n)} &= N\text{-}DELAYED_{(v,n)} \wedge \neg X\text{-}DELAYED_{(v,n)} \vee X\text{-}DELAYED_{(v,n)} \\
&\wedge \sum_{(v',n') \in succ_{SL}(v,n)} \neg N\text{-}DELAYED_{(v',n')}
\end{aligned}
$$

図 6.10 挿入点の計算

データフロー解析におけるスロットの値の更新は，述語に変化があったスロットの隣接スロットに限られる．さらに，述語は $true$ から $false$ に変更されるだけであり，$false$ になった後は変化することがない．したがって，データフロー解析の計算量は高々スロットの数で抑えられる [DRZ92, SKO90]．スロットは，EVG を変形させる際に生じる合成節およびその合成節と ϕ 節で繋がれる元の節に関して，重ならない CFG の範囲に作成されることから，スロット上のデータフロー解析の計算量は変形 EVG の作成と同様に $O(CN)$ で抑えられる．

6.1.2 通常形式のプログラムへの変換

部分冗長除去の解析に基づくプログラム変換結果として，EVG で表現されているプログラムを制御フローで表現した通常形式プログラムに変換しなければならない．この変換では，EVG が値グラフの性質をもつことから，基本ブロック n に対応するスロットを (v,n) とすると，基本ブロック n ごとに EVG を深さ優先順序で訪問し，$LATEST_{(v,n)} = ture$ を満たす EVG 節 v に対応するコードを出力していく．ある EVG 節 v に対する三アドレスコードは，各 EVG 節が区別できるように番号がふられているとして次のように表される．ここで，v から，その一意の番号を添字とする変数を生成する関数を id とする．

$$id(v) \quad ":= " \quad id(child_1(op(v))) \quad lb(op(v)) \quad id(child_2(op(v)))$$

ϕ- 関数をラベルとする副節 ϕ_i をもつ場合には，基本ブロック i の先行節の出口に次の式を挿入する．

$$id(v) \quad ":=" \quad id(child_i(phi(v)))$$

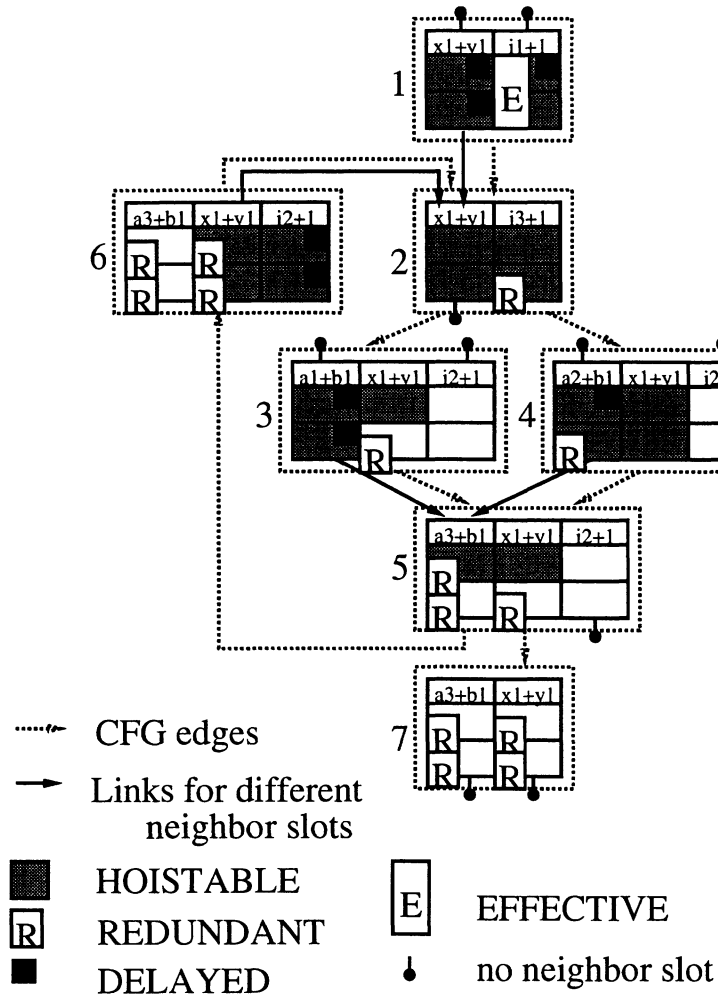


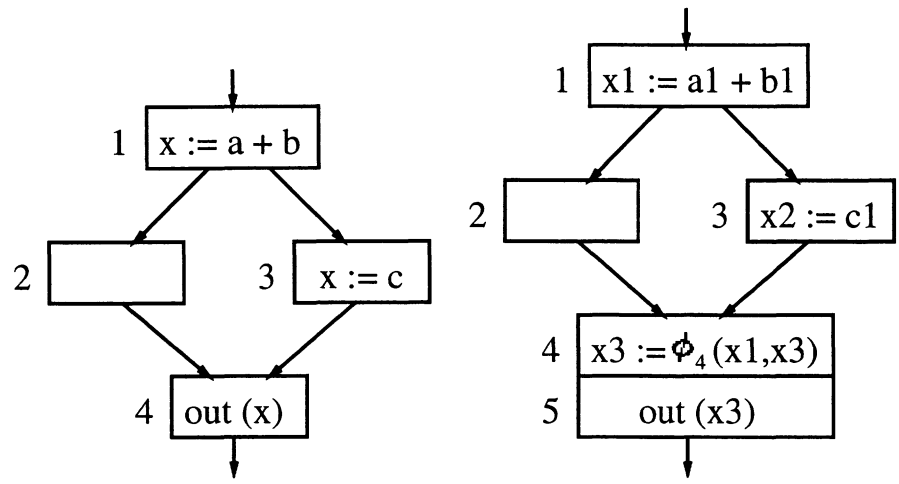
図 6.11 データフロー解析の結果

これらのコピー代入の多くは，レジスタ彩色アルゴリズム [CH90] による後の変数合体によって，除かれることが期待できる．

以上の変換の際，不要コード除去の効果を同時に得ることができる．関数の引数や，配列の添字に用いられる式のコード，または必ずメモリに格納しなければらないコード（レジスタに値を置くことができないコード）など，確実に使用されるコードに対応する EVG 節については，それを根として上述の深さ優先順序でコードを出力することによって，不要コードの生成自体を防ぐことができる．

6.2 部分不要コード除去

この節では，PRE と同様，従来の PDE が繰返し適用によって得ていた副次的効果を，隣接スロットと EVG の依存情報を用いて，直接効果として見付け出す PDE について述べる．



(a) 通常形式　　(b) SSA 形式

図 6.12 部分不要性をもつプログラム

まず，不要な代入文と降下可能な代入文が副次的効果として生じるという相互関係を取り除くために，絶対不要変数と絶対不要代入という概念を導入する．これによって，最も降下できるプログラム点と，不要になる代入を独立に決定することが可能になる．

次に，絶対不要変数とコード降下のデーターフロー方程式を示し，PDE の実現法を例を用いて示す．

6.2.1 絶対不要

代入文 $x := t$ が除去されると，t の中で使用している変数への代入文が不要になる可能性がある．このように，副次的に不要コードとなる代入文を，特に弱体代入 [KRS94b] という．

不要代入と弱体代入は，すべての重要文から走査を始めて，使用 - 定義関係によって到達できる文の集合に含まれない代入として求めることができる．ここで，弱体代入は，使用と定義が元のプログラムの位置にあるものとして計算されるので，コード降下の効果を反映することができない．コード降下によって新たに弱体となった文の除去は，次の部分不要コード除去[1]の適用まで待たなければならない．

この問題を解決するために弱体代入を拡張した，**絶対不要代入** (absolute dead assignment) という概念を導入する．絶対不要代入の定義を示す前に，次の 2 つの用語を導入する．

[1]弱体コードの概念を用いたものを，特に部分弱体コード除去 (partial faint code elimination) と呼ぶ．

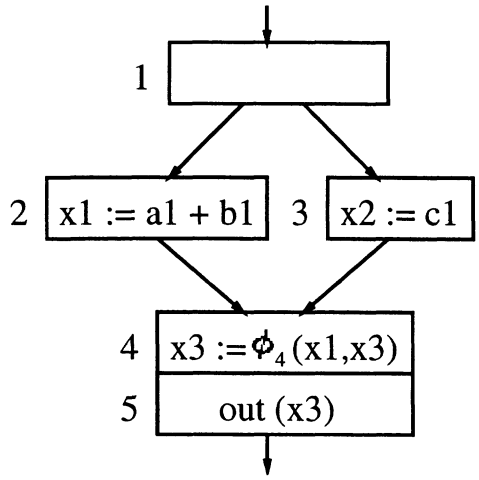


図 6.13 最下点へ移動した例

定義 3 (関係する重要文) 代入文 $s \equiv x := t$ に関係する重要文 r とは，s で定義する値を，最終的に使用する重要文のことである．

s と r には，EVG においてそれぞれに対応する節 v と vr が存在し，vr は v の祖先という関係をもつ．

定義 4 (最下点) 代入文 $s \equiv x := t$ の文 s' に対する最下点を次のように定義する．s，s' が存在する CFG 節をそれぞれ n，n' として，

1. もし，n から終了節 $\mathbf{e}$ へのパス上に x を使用している文 us が存在すれば，us の s' に対する最下点が s の s' に対する最下点である．
2. もし，n から n' へのパス P 上に x を引数とする ϕ-関数 s_ϕ が現れれば，s_ϕ の存在する CFG 節 n_ϕ の P 上の先行節が最下点である．
3. 上記のどの条件にも該当しない場合は，n' が最下点である．

s の s' に対する最下点は，s に対応する EVG の構造を保存したままで降下可能なプログラム点のうち，s' に最も近いプログラム点を意味する．

EVG を用いて，絶対不要変数と絶対不要代入を次のように定義する．

定義 5 (絶対不要変数と絶対不要代入) 絶対不要変数とは，各重要文を r として，r に対応する EVG 節 vr から下向きに到達可能な節に対応する文だけが，r に対する最下点で使用されていると仮定した場合の不要変数である．絶対不要変数を左辺にもつ代入文を，絶対不要代入という．

- すべての $DEAD_{(v,n)}$ （$USED_{(v,n)}$）を $true$ （$false$）に初期化しておく（指定したものを除く）．

$$USED_{(v,n)} =_{def} \begin{cases} true & \text{if } lb(op(v)) \text{ is relevant} \\ & \quad \wedge\, loc(op(v)) = n \\ \sum_{\substack{(v',n') \in succ_{SL}(v,n) \\ \wedge\, v' \neq v}} \neg DEAD(v',n') \vee \sum_{v' \in Parent_{op}(v)} USED_{(v',n)} & \text{otherwise} \end{cases}$$

$$DEAD_{(v,n)} =_{def} \neg USED_{(v,n)} \wedge \prod_{(v',n') \in succ_{SL}(v,n)} DEAD_{(v',n')}$$

図 6.14 絶対不要変数の解析

例： 図 6.12(a) を SSA 形式に変換した図 6.12(b) において，代入文 `x1 := a1 + b1`，`x2 := c1` の `out(x3)` に対する最下点は，それぞれ ϕ- 関数の節 4 の先行節 2，3 となる（図 6.13）．したがって，節 1，3 においては，変数 `a1` と `b1` が絶対不要変数となり，節 1，2 においては，変数 `c1` が絶対不要変数となる． ∎

絶対不要代入は，原始プログラムに現れる各文が，関係する重要文に最も近い点に降下されていると仮定して弱体代入を計算したものに相当する．これによって，コード降下を行ったときに不要となるべき文とプログラム点とを，降下に関係なく計算しておくことができる．

スロット上のデータフロー解析は，次の 3 つのステップからなる．

1. 絶対不要変数の計算
2. 降下可能なプログラム点の計算
3. 挿入点の決定

部分不要コード除去の解析で用いるデータフロー方程式を，文を CFG 節とする述語に基づいて図 6.14～図 6.16に示す．

絶対不要変数の計算

各変数が絶対不要変数となるプログラム点を計算する方程式（図 6.14）において，各述語の意味は次のとおりである．

$DEAD_{(v,n)}$ ： CFG 節 n において，EVG 節 v に対応する式の代入先の変数が絶対不要である場合に $true$，それ以外は $false$ とする．

- $COMP(v,n)$: v に相当する代入文が n 上に存在する.
- すべての $SUNK_{(v,n)}$ ($BLOCKED_{(v,n)}$) を $true$ ($false$) に初期化しておく (指定したものを除く).

$$BLOCKED_{(v,n)} =_{def}$$

$$\begin{cases} true & if\ lb(op(v))\ is\ relevant \\ & \wedge loc(op(v)) = n \\ \sum_{v' \in Parent_{op}(v)} (\neg DEAD_{(v',n)} \wedge SUNK_{(v',n)} & \\ \quad \wedge \sum_{(v'',n') \in succ_{SL}(v',n)} \neg SUNK_{(v'',n')}) & otherwise \end{cases}$$

$$SUNK_{(v,n)} =_{def}$$

$$\begin{cases} false & if\ (v,n) \in SL_{\mathbf{S}} \vee \\ & lb(op(v)) = \bot \wedge loc(phi(v)) = n \\ COMP_{(v,n)} \vee \prod_{(v',n') \in pred_{SL}(v,n)} (SUNK_{(v',n')} & \\ \quad \wedge \neg DEAD_{(v',n')} \wedge \neg BLOCKED_{(v',n')}) & otherwise \end{cases}$$

図 6.15 可能な降下の解析

$$INSERT_{(v,n)} =_{def} \neg DEAD_{(v,n)} \wedge SUNK_{(v,n)} \wedge \sum_{(v',n') \in succ_{SL}(v,n)} \neg SUNK_{(v',n')}$$

図 6.16 挿入点

$USED_{(v,n)}$: EVG 節 v に対応する式の代入先変数が CFG 節 n で使用されている場合に $true$, それ以外は $false$ とする.

$DEAD$ は, 各 EVG 節に対応する式の代入先が絶対不要変数となるプログラム点の集合を求める方程式である. $DEAD$ には, 初期値としてすべて $true$ を設定しておく. 以降の計算において, その値が $false$ に変更された場合, その影響は CFG 上を後向きに伝播される. 重要文から, SV_{op} の辺だけを用いて到達できる文は, 重要文と同じ CFG 節で使用されていることを $USED$ によって伝播する. また, スロットを後向きに辿る際に, $DEAD = false$ が, 異なった EVG 節をもつスロットに伝播したとき, その情報は, やはり $USED$ を用いて伝播される. この $USED$ による $DEAD = false$ の伝播によって, 各文の関係する重要文に使用される範囲が決定し, 不要な範囲はその他の部分となる.

降下可能なプログラム点の計算

降下可能なプログラム点を計算する方程式において，各述語は次のとおりである．

$SUNK_{(v,n)}$ ：EVG 節 v に対応する代入文が CFG 節 n に降下することができる場合に $true$，それ以外は $false$ とする．

$BLOCKED_{(v,n)}$ ：EVG 節 v に対応する文の定義が n と n の後続する節で使用されている場合に $true$，それ以外は $false$ とする．

$SUNK$ の方程式を図 6.15に示す．$SUNK$ の初期値は，開始スロットと，EVG 上で変形が起こらなかった ϕ- 関数（例えば，図 6.6において，四角で囲んでいない ϕ- 関数）のスロットを除いて，$true$ とする．$false$ の影響は CFG 上を前向きに伝播する．$BLOCKED$ は，コード降下の際，定義が使用を追い越して降下し，プログラムの意味を変えてしまうのを防ぐために，降下をブロックする働きをする．重要文に対しては，$BLOCKED = false$ であり，重要文を移動させない働きもする．図 6.15に現れる $\neg DEAD$ は，定義側が不要になった際にブロックを行わない効果を与える．実際の $SUNK$ の計算は，図 6.14中の $\neg DEAD$ によって，絶対不要代入になったところで降下は終了する．

挿入点の決定

すべてのデータフロー解析が終了した後，図 6.16に示すように $INSERT$ が $true$ である CFG 節に文を挿入する．これによって，最も降下したプログラム点での，不要でない文の挿入が実現できる．

例： 図 3.10(a) に対するデータフロー解析の結果の一部を図 6.17に示す．PRE の例と同様に図を単純にするために，すべての述語を 1 つの箱で表し，$true$ となった述語だけをその中に示す．図中の灰色の矩形，u 印の矩形，黒の矩形はそれぞれ $DEAD$，$USED$，$SUNK$ が $true$ であることを意味している．結果として，`x3` および `y0` の定義を CFG 節 46 に降下できることを示している．

6.2.2 通常形式のプログラムへの変換

PRE と同様に，PDE の後処理として，EVG で表現されているプログラムを通常のプログラム形式に変換しなければならない．この際，同じ CFG 節に挿入する計算式の計算順序を正しく保証するために，CFG 節 n ごとに EVG を深さ優先順序で訪問しながら，$INSERT_{(v,n)} = true$ の EVG 節 v に対応する文を生成する．三アドレスコードを生成するのであれば，各 EVG 節 v に一意の変数 $id(v)$ が割り当てられているとして，次の文を挿入する．

$$id(v) \quad \text{“:= ”} \quad id(child_1(op(v))) \quad lb(op(v)) \quad \text{“}id(child_2(op(v)))\text{”}$$

v が ϕ- 関数をラベルとする副節 ϕ_i をもつ場合には，PRE の場合と同様に $loc(phi(v))$ の CFG 節（ϕ- 関数固定の CFG 節 i）の先行節の出口に次の文を挿入する．

$$id(v) \quad \text{“:= ”} \quad id(child_i(phi(v)))$$

これらのコピー代入の多くは，PRE の場合と同様に，後のレジスタ彩色アルゴリズムの適用によって除かれることが期待できる．

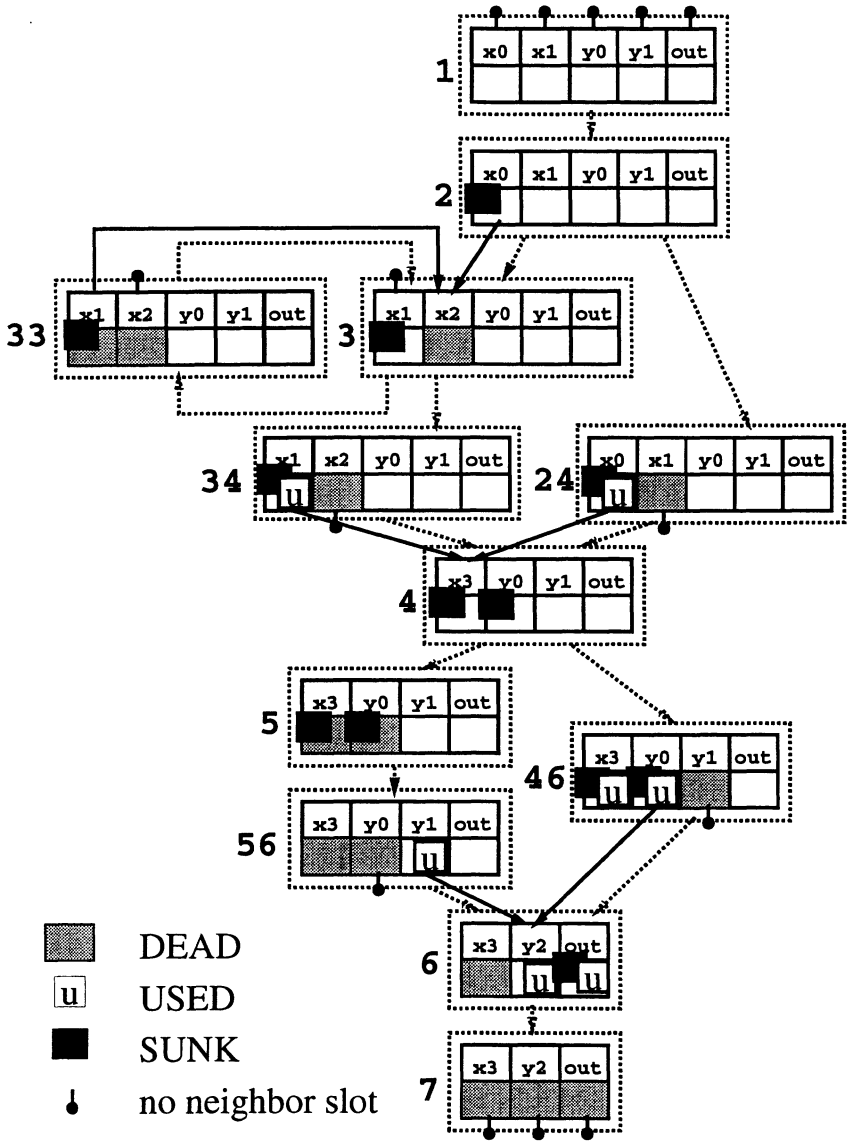


図 6.17 データフロー解析の結果

第 7 章

評価

この章では，本提案手法の有効性を示すために，従来手法と EVG を用いた最適化を行う C コンパイラを実装し，それぞれのコンパイラが生成するコードの実行時間を測定した実験の結果を示す．

このコンパイラは次の 3 つの部分からなる．

1. 中水準中間表現である MIR を生成するフロントエンドプロセッサ
2. MIR に対して最適化を適用する最適化部
3. 最適化部から受け取った MIR を C プログラムに変換するトランスレータ

コンパイラの構成の概略を図 7.1に示す．評価を行う最適化アルゴリズムは，フロントエンドと C コードへのトランスレータに挟まれた最適化部として実装する．

MIR は，すべての計算に必要なだけの個数のレジスタをもつものとする．また，命令のオペランドとしてメモリへのアクセスができるのは，ロード命令と格納命令だけとし，他の命令のオペランドはレジスタへのアクセスだけとする．

生成される C プログラムには， MIR 命令の実行回数を記録するための文を挿入している [BC94, Cli95] ので，このプログラムをコンパイル，実行することによって， MIR 命令操作に関する実際のコストを知ることができる．

7.1 部分冗長除去

評価は，本手法および本手法に近い効果が得られると考えられる従来法の組合せ 2 つの 3 通りの最適化を実装し，実行コストの比較を行った．これらの最適化に含まれる具体的な手法は次のとおりである．

最適化 1: 定数伝播 → 定数畳込み → PRE → 不要コード除去 → 変数合体

従来の構文等価な式に対する PRE と，定数伝播，定数畳込み，不要コード除去，変数合体を組み合わせたもの．

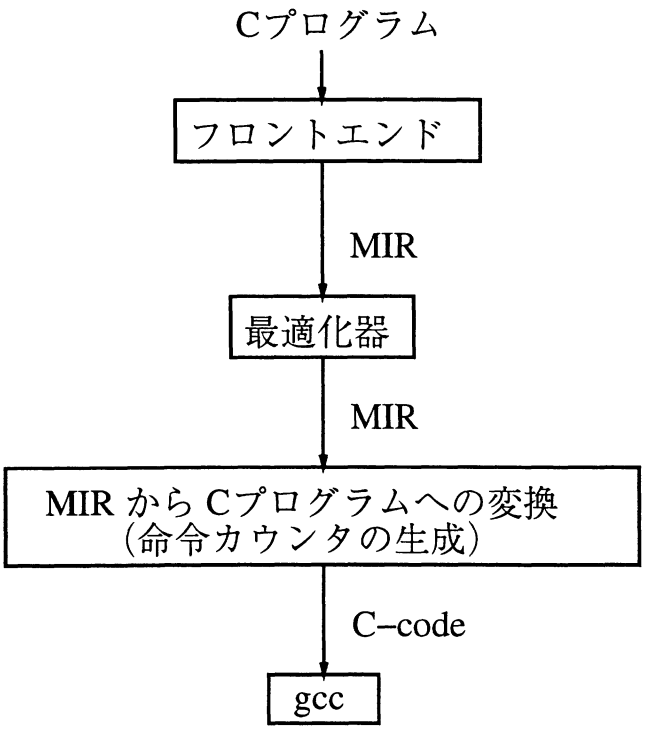


図 7.1 評価用コンパイラの構成

最適化 2: 定数伝播 → 定数畳込み → 大域値番号付け → PRE → 不要コード除去 → 変数合体

最適化 1 に加えて，PRE の前に，VG 上で意味等価な式を発見し，等価な変数を同一の変数名に付け替えるための大域値番号付け（大域一致発見を用いている）を行ったもの．

本手法: EVG による PRE → 変数合体

EVG による PRE を適用した後，変数合体を行ったもの．

各最適化によって得られる目的コードの実行時間を予測するために，ここではいくつかのサンプルプログラムを用いて，MIR レベルでの実行サイクルをカウントした結果を表 7.1に示す．また，最適化 1 と最適化 2 を基準とした高速化の達成率を表 7.2に示す．

表 7.1に示す結果から，本手法は通常の PRE による最適化 1 よりも高い効果が得られることが分る．最適化 2 と比べても，1.1 倍前後の高速化が達成されている．

この 3 通りの手法について，図 3.10(a) を例に効果の相違を述べる．まず，通常の PRE の適用である最適化 1 の場合は，字句形式が等しい式だけを等価な式として扱うので，部分

[1]入力として最大公約数を求めるプログラムを用いた．

表 7.1 実行サイクル数の比較

プログラム名	最適化 1 X	最適化 2 Y	本手法 Z
8 女王問題	448863	400134	372057
ガウス消去法	1739	1249	1173
クイックソート	89490	79275	68379
マージソート	18998	18638	16731
ヒープソート	27145	25149	21205
最短距離問題	2249	2054	1833
Pascal (サブセット) フロントエンド[1]	77195	62019	57042

表 7.2 高速化の達成率

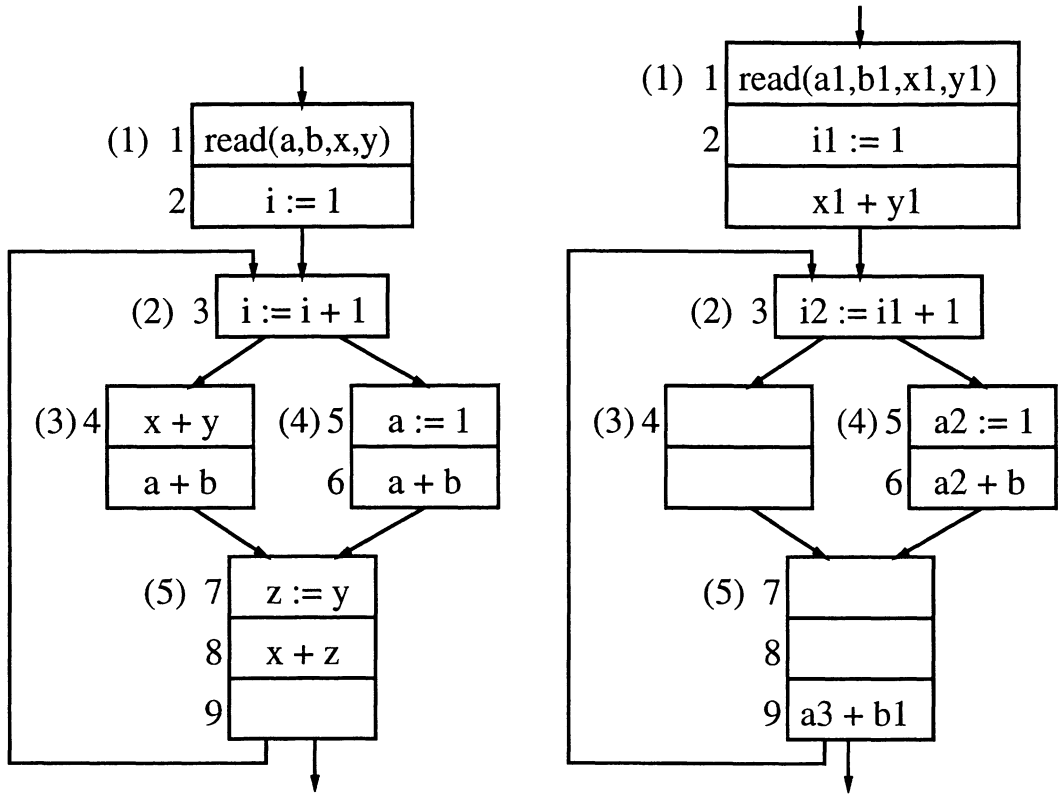
プログラム名	$\frac{X}{Z}$	$\frac{Y}{Z}$
8 女王問題	1.206	1.075
ガウス消去法	1.483	1.065
クイックソート	1.309	1.159
マージソート	1.135	1.114
ヒープソート	1.280	1.186
最短距離問題	1.227	1.121
Pascal (サブセット) フロントエンド	1.353	1.087

冗長である `a + b`（9 行目）だけが除去される．結果は図 7.2(a) のようになる．

これに対して，最適化 2 では，前処理として行った大域値番号付けによって，`x + y`（4 行目）と `x + z`（8 行目）の字句形式が同じになるので，これらの計算式は除去され，ループの外へ移動される．一方，最適化 1 で字句形式が同じとみなされた `a + b` は，1 つの定義しか到達しない 6 行目と，2 つの定義が到達する 9 行目とで構造が異なるので，最適化 2 では字句形式が同じにならない．したがって，9 行目の式は除去されず，図 7.2(b) の結果になる．

本手法では，図 3.10(b) の結果が得られる．この図からは，最適化 1 で得られる効果と最適化 2 で得られる効果の両方が得られていることがわかる．さらに，3 行目の `i := i + 1` は，ループの外側の先行節に移動したものが畳み込まれて定数になる．

本手法が，もとの PRE の効果を損うことなしに，意味等価の発見と PRE を組み合わせていること，さらに定数の畳込みを効果的に統合していることが示された．



(a) 最適化 1　　(b) 最適化 2

図 7.2 最適化 1 と最適化 2 の結果

7.2 部分不要コード除去

本手法の効果を示すために，従来の部分無効コード除去法と本手法を最適化部にもつ C コンパイラを実装し，評価を行った．評価には，2 つの有名なベンチマークと 4 つの簡単なアルゴリズムのプログラム [石畑 89] を用い，プログラム中に現れる各関数ごとに評価を行った．表 7.3に実行結果を示す．表の各列の意味は次のとおりである．

non-PDE：最適化を施さなかった場合の命令実行回数

PDE：従来の部分無効コード除去法を適用した場合の命令実行回数

new：本手法を適用した場合の命令実行回数

non-PDE/PDE：PDE に対する non-PDE の割合

PDE/new：new に対する PDE の割合

表 7.3 実行サイクル数の比較

programs	functions	non-PDE	PDE	new
linpack	main	7190	6661	6256
	print_time	368	368	360
	epslon	23	20	19
	dmxpy	147774	146361	146361
	dgesl	213252	192504	187330
	dgefa	6685068	6286202	6154928
	daxpy	184404012	181715066	181715066
	dscal	1791452	1751698	1751698
	idamax	2071602	1932138	1803438
	matgen	12219498	10853028	10853028
whetstone	main	1419378	1371296	1341777
	p0	190960	190960	190960
	p3	206770	188790	188790
n-queen	main	280	233	231
	backtrack	483671	450759	447875
quicksort	main	90403	84147	84146
hilbert curves	main	219	202	191
	C	2076	2028	1916
	B	3172	3091	2902
	A	4612	4486	4192
	D	3172	3091	2902
shortest path	main	2035	1942	1939
	init	896	812	812

non-PDE/new : new に対する non-PDE の割合

PDE/new の結果から，従来の部分無効コード比べて，性能を低下させることはなく，表 7.4に示すとおり，最大で実行時間は 1.071 倍の高速化が達成されているのが分かる．non-PDE と比べると，最大で 1.211 倍の高速化が達成された．

表 7.4 高速化の割合

programs	functions	non-PDE/PDE	PDE/new	non-PDE/new
linpack	main	1.079	1.065	1.149
	print_time	1.000	1.022	1.022
	epslon	1.150	1.053	1.211
	dmxpy	1.010	1.000	1.010
	dgesl	1.108	1.028	1.138
	dgefa	1.063	1.021	1.086
	daxpy	1.015	1.000	1.015
	dscal	1.023	1.000	1.023
	idamax	1.072	1.071	1.149
	matgen	1.126	1.000	1.126
whetstone	main	1.035	1.022	1.058
	p0	1.000	1.000	1.000
	p3	1.095	1.000	1.095
n-queen	main	1.202	1.009	1.212
	backtrack	1.073	1.006	1.080
quicksort	main	1.074	1.000	1.074
hilbert curves	main	1.084	1.058	1.147
	C	1.024	1.058	1.084
	B	1.026	1.065	1.093
	A	1.028	1.070	1.100
	D	1.026	1.065	1.093
shortest path	main	1.048	1.002	1.050
	init	1.103	1.000	1.103

第 8 章

結論

8.1 まとめと考察

本稿では，コード移動に基づく最適化を効率的に実現するための共通の枠組を提案した．計算式の移動は，計算順序を保持しなければならない必要性から，依存関係にある計算式を越えて行うことはできない．従来のコード移動に基づく最適化では，この条件を満たすためにブロッキングを移動前のプログラム点で行っていた．したがって，ある計算式に関して，その移動をブロックしている計算式が移動した場合には，さらに最適化を繰り返す必要があり，この繰返しによってコンパイル時間が長くかかっていた．

本研究では，計算式が移動した際の各プログラム点における依存情報を利用することによって，すべての計算式を，依存関係にある計算式の移動先までブロックを行わずに移動させることができるコード最適化の枠組を提案した．多くのコード移動に基づく最適化は，本枠組上で実現することによって，1 回の適用で，すべての計算式の可能な移動点を同時に計算することができるようになる．

本枠組を実現するために，計算式が各プログラム点へ移動した際の依存構造を統一的に表現することができる拡張値グラフを提案した．拡張値グラフは，移動先によって異なる依存構造をもつ計算式を，1 つの節を根とするグラフ構造で表現することができる．本枠組は，拡張グラフと制御フローグラフにおけるデータフロー解析として実現することができ，データフロー方程式を入れ替えるだけで，必要な解析の定義ができるので，コード移動に基づく最適化の実現を容易にすることができる．

また，本枠組を用いた部分冗長除去と部分不要コード除去を実現することによって，各コード移動に基づく最適化の効果をより発揮できることを示した．

部分冗長除去の実現においては，拡張値グラフが，各計算点に計算式を移動したときの等価関係を構造的に表現できる性質を利用することによって，意味等価に基づく部分冗長除去が可能となった．

さらに，拡張値グラフ上での定数畳込みは，実際に計算式を移動した際に畳込みの効果が

得られるかどうかの情報を部分冗長除去に組み込むことが可能となった．その計算量は，計算式の数C，基本ブロックの数Nとすると，悲観的にみて$O(CN)$であり，従来の方法に比べて効率が良いことを示した．

拡張値グラフは，計算を移動させるときの依存構造を適切に表現できるので，部分不要コード除去の実現においては，複数の先行節から降下してくる代入文の右辺が同じオペランドをもっていなかったとしても，意味的に等価であれば，プログラムの意味を変えることなく，オペランドの変数名を同じものに置き換えて，さらに降下を促進させることができる．この性質を利用して，従来法よりも広い範囲での部分無効コード除去法を実現した．

さらに本提案手法では，拡張値グラフが変数の定義 - 使用関係をもつことを利用して，文が不要になったという情報と降下したという情報を，使用側の節から定義側の節へ伝播させることができるので，従来繰返し適用を行わなければ除去できなかった不要代入を一度に除去することを可能にした．計算量の点でも，部分冗長除去と同様に悲観的にみて$O(CN)$であり，従来の方法に比べて効率が良いことを示した．

本手法の効果を評価するために，C 言語のサブセットに対して，本手法による最適化を行うコンパイラを試作し，実験を行った．結果として，従来の最適化の効果的な組合せに比べて，さらに効果が得られることを示した．

8.2 今後の課題

本稿で提案した EVG を用いた枠組は，コード移動に基づく多くの最適化に適用することができる．本稿では，部分冗長冗長除去と部分不要コード除去の 2 つの最適化について，それらを本枠組で実現し，評価を行ったが，今後その他の最適化についても，本枠組における実現方法を示し，その効果を確かめることが課題である．

例：2.3.4節で示した演算子の強さ軽減は，例で用いた図 2.12(a) の節 2 と節 3 が入れ替わった図 8.1(a) のようなプログラムでは，直接適用することができない．これは，節 2 の `i` に到達する定義は，ループの外から到達する `i := 1` と 1 回前の繰返しから到達する `i + 1` が存在するので，単純に `i` を単に `i' + 1`（`i'` は代入前の `i` を表す）で置き換えることができないからである．この場合には，`p := i * 5` を巻き上げることによって，置換えを可能にすることができる．`p := i * 5` の巻上げを行った結果を図 8.1(b) に示す．図 8.1(b) の節 5 においては，`i` を `i' + 1` で置き換えることができるので，2.3.4節と同様に，`p := (i' + 1) * 5` を `p := (i' * 5 + 1 * 5)` と代数変形を行って，図 8.2のように，`p := p + 5` に演算子の強さ軽減を行うことができる．

この演算子の強さ軽減は，オペランド `i` を `i'+1` で置き換え，代数変形可能なプログラム点に巻上げを行うことによって実現することができる．EVG を用いると，節 5 において，

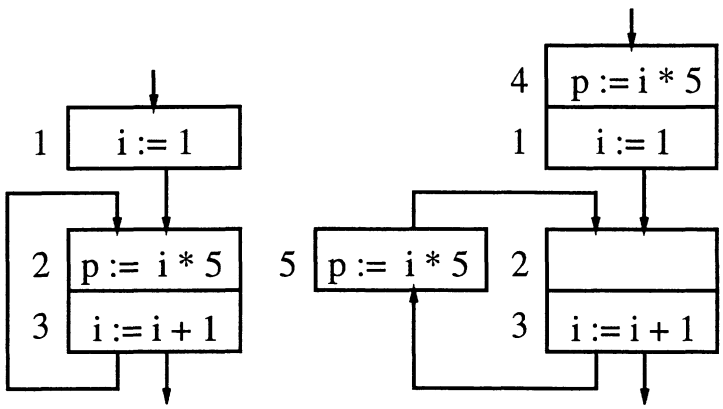


図 8.1 演算子の強さ軽減のための巻上げ

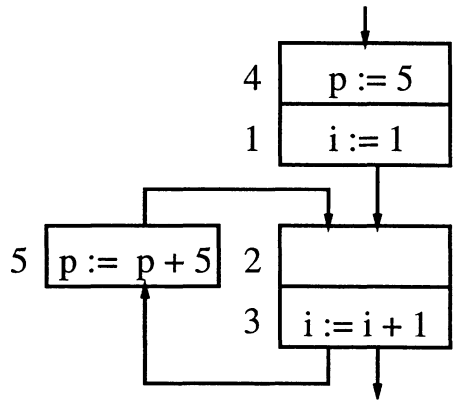
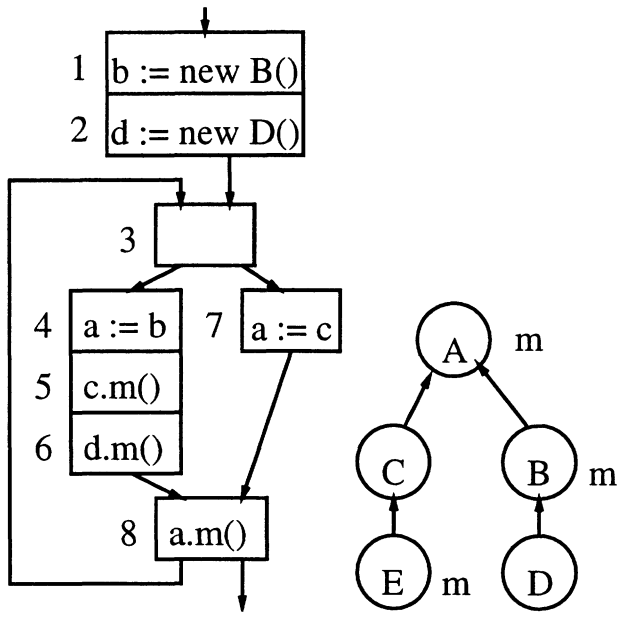


図 8.2 コード移動に基づく演算子の強さ軽減の結果

i が i'+1 の構造をもつことを容易に発見でき，代数変形が可能かどうかもグラフ構造から決定することができる．代数変形可能なプログラム点に対して，6.1.1節で示したように，$EFFECTIVE$ を *true* にすることによって，この巻上げを実現することができる． ∎

また，EVG が値の構造を表していたのに対し，同様の表現方法で，型の構造を示すことができる．この性質を用いて，本枠組を型情報を用いた最適化に対しても利用できるように拡張することも課題である．例えば，型情報は，オブジェクト指向プログラムにおける仮想メソッドの呼出しを効率化するのに役立つ．

例： 図 8.3(a) の CFG において，各変数 a，b，c，d はクラス A，B，C，D の型で宣言され，各クラスは，図 8.3(b) で示すクラス階層で関係付けられているものとする．また，



(a) CFG 表現 (b) グラス階層

図 8.3 オブジェクト指向プログラム

メソッド m が定義されているクラスは，図 8.3(b) の節のうち，右側に m と示した節に対応するクラスとする．メソッド検索の結果は，new A のようなオブジェクト生成の際に明示されるクラスの型をフローさせることによって，呼び出される可能性のあるメソッドの集合として得られる．この際，同一の型をもつ受信オブジェクトに対して同一のシグネチャをもつメソッド呼出しが行われると，メソッド検索が冗長になる可能性がある．冗長なメソッド検索は，先行して実行されたメソッド検索の結果を，後の呼出しで再利用することによって，冗長性を取り除くことができる．呼び出すメソッドのシグネチャが同一の場合，メソッド検索結果の一致性は，メソッド呼出しに対する受信オブジェクトの型の一致性に対応するので，EVG と同様の構造をもち，型の参照構造をグラフ化したものを用いれば，EVG 上で PRE を行うのと同様に冗長なメソッド検索を除去することができる．

本来，メソッド検索はメソッド呼出しの一部として行われるので，その操作がプログラム中に直接現れることはない．冗長性を扱うには，メソッド検索を明示する必要があるので，各メソッド呼出しの直前のプログラム点に，仮想関数 *lookup* を挿入する．*lookup* は，受信オブジェクトの型と呼び出すメソッドのシグネチャから，呼び出されるメソッドの本体を決定し，そのポインタを返す．一般に，オブジェクトからその型を返す関数を T で表すことにすると，メソッド呼出し $a.m()$ は，一時変数 h を用いて $h := lookup(T(a), m)$; $(*h)(a)$ に変換することができる．すべてのメソッド呼出しについてこの変換を適用すると，メソッ

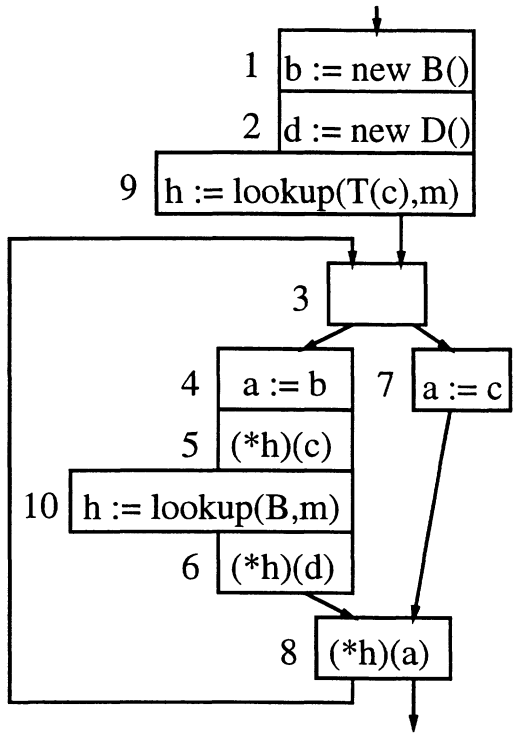


図 8.4 メソッド検索結果の再利用

ド検索に対する冗長性の除去は，冗長な *lookup* を除去することと等しくなる．

図 8.3(a) のプログラムに対して，メソッド検索に対する PRE を適用したときの結果を図 8.4に示す．`h := lookup(T(c),m)` の節 9 への巻上げは，従来の PRE で得られる冗長除去とループ不変コードのループ外移動の効果である．節 10 へ巻上げは，`lookup` の第 1 引数である受信オブジェクトの型が一意に決まるプログラム点への移動である．受信オブジェクトの型が一意に決まる場合，呼び出されるメソッドも一意に決まるので，`lookup` は，直接メソッド本体の先頭アドレスで置き換えることができる．この型を一意に変換する効果は，型の参照構造からあらかじめ計算しておくことができる． ∎

謝辞

本研究の機会を与えてくださり，絶えず御指導頂いた，慶應義塾大学理工学部 原田 賢一教授に深く感謝致します．原田教授には，研究の仕方から論文の書き方まで，研究者が研究者たるべき 1 から 10 を御教授頂きました．御教授頂いた 1 つ 1 つは，これまでに公私を問わず頂いた多くのアドバイスや励ましとともに，私の基盤であり何にも代えがたい宝物であります．本当にありがとうございました．また，原田研究室の皆様にも，公私ともに大変お世話になりました．

本論文の執筆に当たりまして，適切なコメントとアドバイスを頂いた慶應義塾大学理工学部の天野英晴教授，大野義夫教授，榎本彦衛教授に深く感謝致します．

多数の有用なコメントと援助を頂いた東京工業大学情報理工学研究科 佐々政孝教授に深く感謝致します．

また，快適な研究環境を提供して頂き，励まして下さった東京理科大学理工学部情報科学科の諸先生方に深く感謝致します．

これまで，無償の援助と励ましを与えてくれた両親と妹達に深く感謝致します．

最後に，常に支えてくれた妻友香に心から感謝致します．

2002 年 10 月
滝本 宗宏 (Munehiro Takimoto)

参考文献

[App98] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proc. Principles of Programming Languages (POPL'88)*, pp. 1–11. ACM, 1988.

[BC94] P. Briggs and K. D. Cooper. Effective Partial Redundancy Elimination. In *Proc. Programming Language Design and Implementation (PLDI'94)*, pp. 159–170. ACM, 1994.

[BG97] R. Bodik and Rajiv Gupta. Partial Dead Code Elimination using Slicing Transformations. In *Proc. Programming Language Design and Implementation (PLDI'97)*, pp. 159–170. ACM, 1997.

[CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently Computing Static Single Assignment Form and Control Dependence Graph. *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 4, pp. 451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 4, pp. 501–536, October 1990.

[Cli95] C. Click. Global Code Motion Global Value Numbering. In *Proc. Programming Language Design and Implementation (PLDI'95)*, pp. 246–257. ACM, 1995.

[Dha91] D. M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2, pp. 291–294, April 1991.

[DP93] D. M. Dhamdhere and H. Patil. An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement. *ACM Trans. Prog. Lang. Syst.*, Vol. 15, No. 2, pp. 321–336, April 1993.

[DRZ92] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to Analyze Large Programs Efficiently and Informatively. In *Proc. Programming Language Design and Implementation (PLDI'92)*, pp. 212–223. ACM, 1992.

[FCKLL97] S. Chan F. Chow, R. Kennedy, S. M. Liu, and R. Lo. A New Algorithm for Partial Reundancy Elimination Based on SSA Form. In *Proc. Programming Language Design and Implementation (PLDI'97)*, pp. 273–286. ACM, 1997.

[FKCX94] L. Feigen, D. Klappholz, R. Casazza, and X. Xue. The Revival Transformation. In *Proc. Principles of Programming Languages (POPL'94)*, pp. 421–434. ACM, 1994.

[KD94] U. P. Khedker and D. M. Dhamdhere. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Trans. Prog. Lang. Syst.*, Vol. 16, No. 5, pp. 1472–1511, September 1994.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy Code Motion. In *Proc. Programming Language Design and Implementation (PLDI'92)*, pp. 224–234. ACM, 1992.

[KRS94a] J. Knoop, O. Rüthing, and B. Steffen. Optimal Code Motion: Theory and Practice. *ACM Trans. Prog. Lang. Syst.*, Vol. 16, No. 4, pp. 1117–1155, July 1994.

[KRS94b] J. Knoop, O. Rüthing, and B. Steffen. Partial Dead Code Elmination. In *Proc. Programming Language Design and Implementation (PLDI'94)*, pp. 147–158. ACM, 1994.

[KRS95] J. Knoop, O. Rüthing, and B. Steffen. The Power of Assignment Motion. In *Proc. Programming Language Design and Implementation (PLDI'95)*, pp. 233–245. ACM, 1995.

[MR79] E. Morel and C. Renvoise. Global Optimization by Supperssion of Partial Redundancies. *Comm.ACM*, Vol. 22, No. 2, pp. 96–103, February 1979.

[Muc97] S. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[OKS99] O. Rüthing, J. Knoop, and B. Steffen. Detecting Equalities of Variables: Combining Efficiency with Precision. In *Proc. Int. Static Analysis Symposium (SAS'99), vol. 1694 of LNCS*, pp. 232–247, Venice, 1999. Springer-Verlag.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proc. Principles of Programming Languages (POPL'88)*, pp. 12–27. ACM, 1988.

[SCL+96] F. Chow S, S. Chan, S. Liu, R. Lo, and M. Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proc. Int. Compiler Construction (CC'96), LNCS*, Berlin, 1996. Springer-Verlag.

[SG95] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing ϕ-Nodes. In *Proc. Principles of Programming Languages (POPL'95)*, pp. 62–73. ACM, 1995.

[SKO90] B. Steffen, J. Knoop, and O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *Proc. Int. European Symposium on Programming (ESOP'90)*, pp. 389–405, Copenhagen, Denmark, 1990. Springer-Verlag.

[TH99] M. Takimoto and K. Harada. Partial Dead Code Elimination Using Extended Value Graph. In *Proc. Int. Static Analysis Symposium (SAS'99), vol. 1694 of LNCS*, Venice, 1999. Springer-Verlag.

[TH01] M. Takimoto and K. Harada. Eliminating May-aliases. In *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.

[Wei84] M. Weiser. Program Slicing. *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 4, pp. 352–357, July 1984.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2, pp. 181–210, April 1991.

[佐々 89] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.

[石畑 89] 石畑清. アルゴリズムとデータ構造. 岩波書店, 1989.

[滝本 97] 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, November 1997.

[滝本 00] 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, January 2000.

[滝本 02a] 滝本宗宏, 原田賢一. May 別名除去. 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG 8(PRO 15), pp. 11–22, September 2002.

[滝本 02b] 滝本宗宏, 原田賢一. 別名情報に基づくレジスタ促進. 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG 8(PRO 15), pp. 49–61, September 2002.