

静的単一代入形式上の 部分冗長性除去

東京工業大学 大学院
情報理工学研究科 数理・計算科学専攻

立川 英
(02M37207)

平成15年度 修士論文

指導教官 佐々 政孝 教授
脇田 建 助教授

目次

第 1 章 緒言	6
1.1 背景	6
1.2 概要	6
1.3 構成	7
第 2 章 制御フローグラフ	8
2.1 制御フローグラフの部品	9
2.2 制御フローグラフ内の性質	11
2.2.1 支配	11
2.2.2 深さ	12
第 3 章 静的単一代入形式	14
3.1 SSA 形式	14
3.2 SSA 変換	14
3.3 SSA 形式に基づく最適化	15
3.3.1 共通部分式除去	16
3.4 SSA 逆変換	16
3.5 SSA 形式の定義	18
3.6 いろいろな SSA 形式	18
第 4 章 並列化コンパイラ向け共通インフラストラクチャ COINS	20
4.1 背景	20
4.2 構成	20
第 5 章 部分冗長性除去	22
5.1 冗長性	22
5.2 部分冗長性	22
5.3 部分冗長性除去	23
5.4 SSA 形式上の部分冗長性除去	23
第 6 章 SSA 形式上の部分冗長性除去の新手法 Practical-PRE on SSA Form	25
6.1 背景	25
6.2 本手法のアルゴリズム	26
6.2.1 危険辺の除去	28
6.2.2 局所的な性質	28
6.2.3 コード移動	30
6.2.4 大域的な性質	30
6.2.5 計算の巻き上げ	32
6.2.6 コード移動 レベル 1	34
6.2.7 添字の解析	35

6.2.8	コード移動 レベル 1	44
6.2.9	変数の生存区間の短縮	44
6.2.10	コード移動 レベル 2	47
6.2.11	無駄なコード移動の除去	48
6.2.12	コード移動 レベル 3	50
第 7 章	評価	53
7.1	処理時間の計測	54
7.2	処理速度の考察	55
7.2.1	通常形式による部分冗長性除去との比較	55
7.2.2	他の SSA 形式上の部分冗長性除去との比較	57
7.2.3	COINS 上の他の最適化との比較	58
7.3	実行時間の計測	59
7.4	実行速度の考察	59
7.5	本手法のコード移動の比較	60
第 8 章	関連研究	64
8.1	Kennedy らの方法	64
8.2	Briggs らの方法	65
8.3	滝本らの方法	66
第 9 章	まとめ	67
9.1	結論	67
9.2	今後の課題	67

目次

2.1	制御フローグラフの例	9
2.2	図 2.1 の支配木	12
3.1	通常形式	18
3.2	図 3.1 の SSA 形式	18
4.1	COINS 構成図	21
5.1	冗長と冗長除去	22
5.2	部分冗長と冗長への変形	22
5.3	ループ不変コード移動	23
5.4	部分冗長性除去と SSA 形式の例	23
5.5	SSA 形式上の部分冗長性除去	24
6.1	例プログラムの最適化前と最適化後	26
6.2	例題のフローグラフ	27
6.3	危険辺の例とその除去	28
6.4	危険辺の除去の結果	29
6.5	「 $a + b$ 」に関する基本ブロックの入口と出口	30
6.6	基本ブロックの入口と出口	31
6.7	<i>DSafe</i> の結果	33
6.8	<i>USafe</i> の結果	34
6.9	<i>Earliest</i> の結果	35
6.10	最小 SSA 形式の例	36
6.11	<i>LiveSuffix</i> , <i>MinPhiInsert</i> の結果	38
6.12	<i>TempUseLater</i> の結果	39
6.13	<i>PrunPhiInsert</i> の結果	40
6.14	<i>TempReplaceSuffix</i> の結果	41
6.15	<i>CompSuffix</i> の結果 (a のみ表示)	42
6.16	<i>CompUseLater</i> の結果 (a のみ表示)	43
6.17	<i>PhiElimination</i> の結果 (a のみ表示)	44
6.18	計算の巻き上げによるコード移動 (レベル 1) の結果	45
6.19	<i>Delay</i> の結果	46
6.20	<i>Latest</i> の結果	47
6.21	変数の生存区間を短縮したコード移動 (レベル 2) の結果	48
6.22	<i>Live</i> の結果	49
6.23	<i>Insert</i> の結果	50
6.24	<i>Replace</i> の結果	51
6.25	無駄な移動を行わないコード移動 (レベル 3) の結果	52

7.1	処理時間 (表 7.2, 表 7.3 より)	56
7.2	最適化の方法	56
7.3	処理時間の割合	58
7.4	実行時間の割合 (表 7.6 より)	61
7.5	実行時間の割合 (表 7.12 より)	63
8.1	部分冗長性除去する部分式の選択ができる例	65
8.2	図 8.1 を部分冗長性除去した 2 つの結果	66

表 目 次

7.1 Sun Blade 1000 の主な仕様	53
7.2 SPECint 2000 181.mcf の処理時間 (<i>msec</i>)	54
7.3 6 つの小さいテストプログラムの合計の処理時間 (<i>msec</i>)	55
7.4 処理時間の割合 (%)	58
7.5 実行時間 (<i>sec</i>)	59
7.6 実行時間の割合 (%)	60
7.7 レベルごとの解析の数	61
7.8 SPECint 2000 181.mcf の処理時間 (<i>msec</i>)	62
7.9 6 つの小さいテストプログラムの合計の処理時間 (<i>msec</i>)	62
7.10 処理時間の割合 (%)	62
7.11 実行時間 (<i>sec</i>)	62
7.12 実行時間の割合 (%)	63

第1章 緒言

本論文では、部分冗長性除去の処理を SSA 形式上で行う方法を提案し、その評価を行う。

1.1 背景

プログラミング言語処理系が行うコード最適化の中で、部分冗長性除去の手法は、共通部分式の除去ばかりでなく、ループ不変コードのループ外移動をも統合的に行える優れた最適化であり、以前から多くの研究がなされてきた。

その研究の中で、部分冗長性除去を SSA 形式上で処理しようという試みがある。SSA 形式は、変数の定義の唯一性などからプログラミング言語処理における最適化に適しているとされているプログラムの表現方法であり、近年さかんに研究が行われている。

しかし、部分冗長性除去をこの SSA 形式上で処理しようとすると、

- 通常形式上で同一の変数として扱えたものが異なる変数として認識されてしまう。
- SSA 形式での変数には満たすべき条件があるので、異なる基本ブロックにコードを移動する際に変数をそのまま移動できない。

といった困難がある。そのため、SSA 形式上で部分冗長性除去を行う従来の手法 [Kennedy et al. 1998; 滝本・原田 1997] では制御フローグラフ上のみで処理が行えず、制御フローグラフとは別の特別なグラフの作成と、そのグラフ上における複雑な解析が必要となり、その実装はかなり困難であった。また、その処理にかかるコンパイル時間もかかっていた。

また、従来のアルゴリズムの中には、SSA 形式を対象に解析を行ってから部分冗長性除去の処理を行うと、処理中に通常形式に戻ってしまうものも存在する [Briggs and Cooper 1994]。一般に、SSA 形式を対象に最適化を行うコンパイラでは、SSA 形式のまま複数の最適化を繰り返し処理させる方式が望ましく、ある最適化で通常形式に戻ってしまうような方式は非効率的である。

1.2 概要

本研究では、SSA 形式を扱う部分冗長性除去として、次の条件を満たすような、新たなアルゴリズムを考案した。

- SSA 形式のまま処理ができる (出力も SSA 形式である)。
- 単方向のデータフロー解析で実現できる。
- 制御フローグラフ上のみで処理ができる (特別なグラフの作成が不要)。
- ビットベクトル方式により多数の冗長式を同時に処理できる。

本研究では、通常形式を扱う部分冗長性除去のアルゴリズムである怠けたコード移動 [Knoop et al. 1992; Knoop et al. 1994] を変形することで、これらの条件を満たす SSA 形式上の部分冗長性除去の見通しのよいアルゴリズムを開発した。また最適化効果についても、通常形式上の [Knoop et al. 1994]

や SSA 形式上の部分冗長性除去として最新の結果である [Kennedy et al. 1998] と同等に強力である . [Knoop et al. 1994] に対する利点は SSA 形式をサポートすることであり , [Kennedy et al. 1998] に対する利点はアルゴリズムの単純さ , 最適化にかかる処理時間の短縮 , 実装の簡潔さである . これをコンパイラ・インフラストラクチャ COINS の上で実装し , その処理時間と性能について考察を行う .

1.3 構成

本論文の構成を以下に示す .

- 2 章 : 本手法が対象にする制御フローグラフについて .
また , 以後の章で使用する用語の説明も兼ねる .
- 3 章 : 本手法の入力形式である SSA 形式について .
- 4 章 : 本手法を実装 , 実験を行う際に利用したコンパイラ・インフラストラクチャ COINS について .
- 5 章 : 部分冗長性除去について .
- 6 章 : 本研究で提案する SSA 形式上の部分冗長性除去 (本手法) について .
- 7 章 : 本手法に対して行った実験とその評価 .
- 8 章 : 関連研究について .
- 9 章 : 本研究のまとめと今後の課題について .

第2章 制御フローグラフ

本章では、本手法によって解析を行う 制御フローグラフ(*control flow graph*, *CFG*) に関する説明を行う¹。

以後の説明に用いるプログラムの例をプログラム 2.1 に示す。また、そのプログラムの制御フローグラフを図 2.1 に示す。

プログラム 2.1 (制御フローグラフの例題プログラム)

ブロック 1		<code>do {</code>	
		<code> a=1;</code>	
		<code> b=1;</code>	(2.1)
		<code> c=1;</code>	
		<code> if (...){</code>	
ブロック 2		<code> c=c+1;</code>	(2.2)
		<code> if (...){</code>	
ブロック 3		<code> b=b+3;</code>	(2.3)
		<code> } else {</code>	
ブロック 4		<code> b=b+2;</code>	(2.4)
		<code> }</code>	
ブロック 5		<code> a=b;</code>	(2.5)
		<code> } else {</code>	
ブロック 6		<code> a=b+c;</code>	(2.6)
		<code> }</code>	
		<code> do {</code>	
ブロック 7		<code> b=1;</code>	(2.7)
		<code> if (...){</code>	
ブロック 8		<code> b=b+2;</code>	(2.8)
		<code> c=c+1;</code>	
ブロック 9		<code> }</code>	(2.9)
		<code> }while (...);</code>	
ブロック 10		<code> a=a+3;</code>	(2.10)
		<code> }while (...);</code>	

¹本章での説明は本手法によって扱う制御フローグラフの説明である。正確な定義については [Hecht 1977; Appel 1998; 中田 1999; 佐々 1989] を参照。

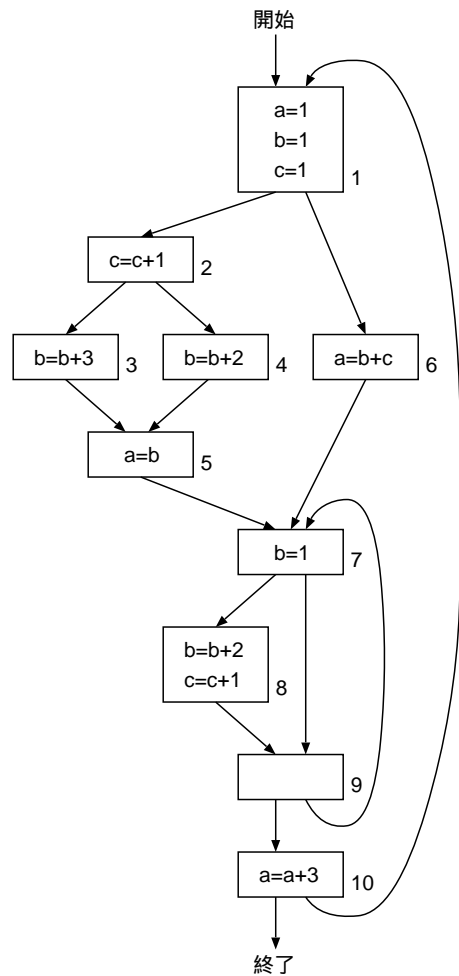


図 2.1: 制御フローグラフの例

2.1 制御フローグラフの部品

1 制御フローグラフ

プログラムの制御の流れをグラフで表したものを制御フローグラフとよぶ。制御フローグラフは基本ブロック (*basic block*) をノード (*node*) として、それらの間を分岐や合流を表す有向辺 (*directed edge*) で結んだ有向グラフである (図 2.1)。

2 基本ブロック

間に分岐も合流もない文 (*statement*) の列を基本ブロックとよぶ。基本ブロックの先頭には一般に合流があり、最後からは分岐がある。基本ブロック中の文は先頭から最後まで一直線に実行される。以後、基本ブロックを単にブロックとよぶ。

3 空のブロック

文のないブロックを空のブロックとよぶ。

4 有向辺

制御フローグラフにおいて、制御がブロック X からブロック Y に流れるとき、 $X \rightarrow Y$ と表記し、これを有向辺とよぶ。以後、有向辺のことを単に辺 (*edge*) とよぶ。

5 パス

ノードの列 $X_0, X_1, X_2, \dots, X_i$ ($i \geq 0$) に対して, $e_1 : X_0 \rightarrow X_1, e_2 : X_1 \rightarrow X_2, \dots, e_i : X_{i-1} \rightarrow X_i$ ($i \geq 0$) なる辺が存在すると仮定する. このとき X_0 から X_i にパス (*path*) が存在するという.

6 先行ブロック

$X \rightarrow Y$ であるとき, X は Y の先行ブロック (*predecessor block*) とよぶ. X の先行ブロックの集合を $pred(X)$ と表す. 図 2.1 のブロック 7 の場合, 先行ブロックはブロック 5, ブロック 6, ブロック 9 であり, $pred(7) = \{5, 6, 9\}$ と表す.

7 後続ブロック

$X \rightarrow Y$ であるとき, Y は X の後続ブロック (*successor block*) とよび, X の後続ブロックの集合を $succ(X)$ と表す. 図 2.1 のブロック 7 の場合, 後続ブロックはブロック 8 とブロック 9 であり, $succ(7) = \{8, 9\}$ と表す.

8 結合ブロック

先行ブロックが複数あるブロックを 結合ブロック (*join block*) とよぶ, 図 2.1 の結合ブロックはブロック 1, ブロック 5, ブロック 7, ブロック 9 である.

9 分岐ブロック

後続ブロックが複数あるブロックを 分岐ブロック (*branch block*) とよぶ, 図 2.1 の分岐ブロックはブロック 1, ブロック 2, ブロック 7, ブロック 9, ブロック 10 である.

10 開始ブロック

先行ブロックがないブロックを 開始ブロック (*start block*) とよぶ. 本論文ではアルゴリズムの説明上, 開始ブロックは空ブロックであるとし, プログラムの入口と同義語として扱う (図 2.1 最上部).

11 終了ブロック

後続ブロックがないブロックを 終了ブロック (*end block*) とよぶ. 本論文ではアルゴリズムの説明上, 終了ブロックは空ブロックであるとし, プログラムの出口と同義語として扱う (図 2.1 最下部).

12 先行パス

プログラムの入口からブロック X に至るすべてのパスを, X の 先行パス (*predecessor path*) とよぶ.

13 後続パス

ブロック X からプログラムの出口に至るすべてのパスを, X の 後続パス (*predecessor path*) とよぶ.

14 後退辺

X が Y の後続パスであるとき, $X \leftarrow Y$ を 後退辺 (*back edge*) とよぶ. 図 2.1 のブロック $9 \leftarrow$ ブロック 7 は後退辺である.

2.2 制御フローグラフ内の性質

2.2.1 支配

15 支配

2つのブロック X , Y について, プログラムの入口から Y に達するどのパスも必ず X を通るとき, X は Y を支配 (*dominate*) する, または X は Y の支配ブロック (*dominator*) であるとよぶ. 図 2.1 のブロック 2 はブロック 5 を支配するといひ, ブロック 2 はブロック 5 の支配ブロックであるとよぶ.

16 厳密に支配

2つのブロック X , Y について, X が Y を支配し, $X \neq Y$ であるとき, X は Y を厳密に支配 (*strictly dominate*) するといひ. 図 2.1 のブロック 2 はブロック 5 を厳密に支配するといひ,

17 直接支配

2つのブロック X , Y について, X が Y を厳密に支配し, X から Y へのパスにそれ以外に Y を厳密に支配するブロックがないとき, X は Y を直接支配 (*immediately dominate*) するといひ. 図 2.1 のブロック 2 はブロック 4 を直接支配するといひが, のブロック 2 はブロック 5 を直接支配するといひわない.

18 支配境界

ブロック X の支配境界(*dominance frontier*, $DF(X)$) は次の式で表される.

$$\begin{aligned} DF^+(X) &= \bigcup_{x \in X} DF(x) \\ &= \left\{ Y \mid \begin{array}{l} U \in \text{pred}(Y) \text{ が存在し,} \\ X \text{ は } U \text{ を支配し, } X \text{ は } Y \text{ を厳密な支配はしない} \end{array} \right\} \end{aligned}$$

図 2.1 の場合, $DF(8) = \{9\}$ である. 支配境界を求めるアルゴリズムの計算のオーダーは, 基本ブロック数を N , 辺の数を E とすると, 最悪で $O(E+N^2)$ となる [Appel 1998]. しかし, だいたいのプログラムにおいては, 基本ブロック数に線形で計算できるようである [中谷 2001].

19 繰り返し支配境界

ブロック X の繰り返し支配境界(*iterated dominance frontier*, DF^+) は次の式を繰り返し計算により求められるブロックの集合の最大解となる.

$$\begin{aligned} DF^+_1 &= DF(X) \\ DF^+_{i+1} &= DF(X \cup DF_i) \end{aligned}$$

図 2.1 の場合, $DF^+(3, 6, 8) = \{2, 7, 9\}$ である.

20 支配木

直接支配の関係を辺とした作成した木 (*tree*) を、支配木 (*dominator tree*) とよぶ。支配木の根 (*root*) は開始ブロックで、 X が Y を直接支配する場合には X は Y の親となっている。図 2.1 の支配木を図 2.2 に示す。支配木を求めるアルゴリズムの計算のオーダーは、制御フローグラフ内の基本ブロック数を N 、辺の数を E とすると、その定義どおりに行ったときには $O(N^2)$ となるが、効率のよいアルゴリズムでは $O(N \cdot \log N)$ まで下げられる。これは一般に支配木を求めるのに利用されるアルゴリズムである。また [Tarjan 1979] のアルゴリズムを利用することによって、アッカーマン関数 (Ackermann function, Ack) [Lengauer and Tarjan 1979] の逆関数 Ack^{-1} を用いて計算のオーダーが $O(N \cdot Ack^{-1}(E, N))$ まで下げられるアルゴリズムも提案されている。Ackermann 関数の逆関数は一定の値に収束することはないが、実際のプログラムでは実質 $O(N)$ であると考えてよいと思われる。

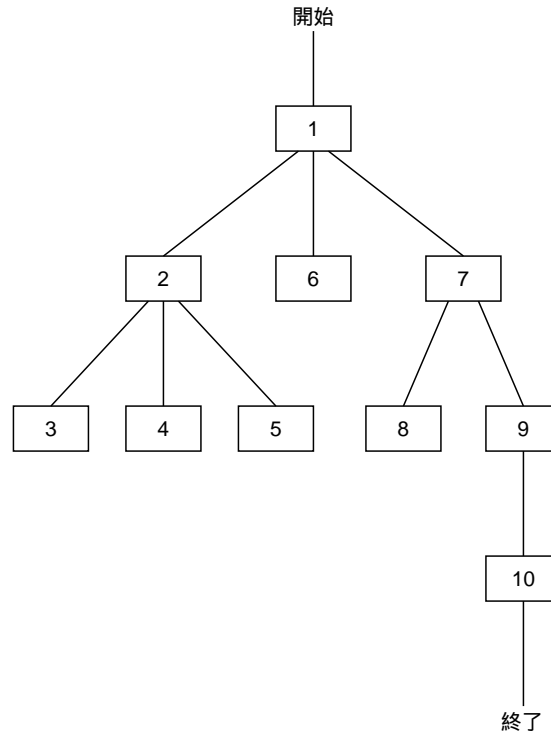


図 2.2: 図 2.1 の支配木

2.2.2 深さ

21 サイクル

X が Y を支配しているが厳密に支配していないとき、 X から Y までのパスをサイクル (*cycle*) といい、 Z が X から Y までのパス上にあるとき Z はサイクル上にあるという。図 2.1 のブロック 7、ブロック 8、ブロック 9 はサイクル上にある。

22 サイクルの入口

サイクルの入口とは、そのサイクルに属さない先行ブロックを持つブロックの入口を指す。図 2.1 のブロック 7 の入口は上記の例のサイクルの入口である。

23 サイクルの出口

サイクルの出口とは、そのサイクルに属さない後続ブロックを持つブロックの出口を指す。図 2.1 のブロック 9 の出口は上記の例のサイクルの出口である。

24 制御フローグラフの深さ

サイクルを含まない任意のパス上での後退辺の最大個数を 制御フローグラフの深さ (*depth*) という。以後、単に深さとよぶ [Hecht 1977; Rosen et al. 1988a; Appel 1998] 。

25 制御フローグラフの区間

制御フローグラフを有効に階層化するために、グラフをいくつかに分割したものを 区間 (*interval*) とよぶ。区間は自然なループであり、それに、そのループ内のブロックから達するサイクルのない構造を付加したものである。以後、単に区間とよぶ。

26 区間の深さ

[Knuth 1971] の統計によると、通常の区間の深さは非常に浅く、平均で 2.75 という報告がある。また、「深さ」は「区間の深さ」よりも決して小さくならないことが証明されている [Hecht 1977] 。

27 制御フローグラフの幅

[Dhamdhere and Khedker 1993] で定義されている制御フローグラフの幅 (*width*) とは、制御フローグラフの性質を表すものではなくデータフロー解析を行う際の制御フローグラフの複雑さを定義したものである。彼らによって、双方向のデータフロー解析でも単方向のデータフロー解析でも、解析処理に要する計算のオーダーは、基本ブロックの数を N 、制御フローグラフの幅を w とすると $\mathcal{O}(N \cdot w)$ となることが証明されている。また、単方向のデータフロー解析においては、 w は制御フローグラフの区間の深さ d となる。したがって、単方向のデータフロー解析の処理に要する計算のオーダーは $\mathcal{O}(N \cdot d)$ となる [Dhamdhere and Khedker 1993; Dhamdhere 1991; Dhamdhere and Patil 1993]。ここで上記より、区間の深さ d は一般に 2.75 よりも小さいことが示されている。単方向のデータフロー解析に必要な繰り返し計算の回数は $d + 2$ であり、最悪の場合でも制御フローグラフ全体を $d + 2$ 回繰り返し計算すれば不動点が求まる。したがって一般のプログラムでは繰り返し計算の回数は 5 回以内で収まると考えられるため、単方向のデータフロー解析にかかる計算のオーダーは実質 $\mathcal{O}(N)$ であると考えてよいと思われる。

第3章 静的単一代入形式

本章では、静的単一代入形式 (*static single assignment form* , 以後 *SSA 形式* とよぶ) について簡単に説明を行う。

3.1 SSA 形式

SSA 形式 [Alpern et al. 1988; Rosen et al. 1988b; Cytron et al. 1989; Appel 1998; 中田 1999] とは、変数の定義がプログラムの字面上で唯一になるようにしたプログラムの表現形式である。静的とは、プログラムの字面上で、という意味である。変数の定義が唯一になるように変数の名前替えを行うが、これはふつう変数に添字をつけて表す。この結果、SSA 形式では、プログラムあるいは中間表現の上で、各変数の定義¹が1箇所だけになる。以下に SSA 形式の例をあげる。

プログラム 3.1 (SSA 形式の簡単な例)

$$a_1 = x_0 + y_0; \tag{3.1}$$

$$a_2 = a_1 + 3; \tag{3.2}$$

$$b_1 = x_0 + y_0; \tag{3.3}$$

3.2 SSA 変換

通常形式²上から SSA 形式へ変換することを *SSA 変換*(*translation to SSA form*) という。次のような通常形式のプログラムがあったとする。

プログラム 3.2 (通常形式)

$$a = x + y; \tag{3.4}$$

$$a = a + 3; \tag{3.5}$$

$$b = x + y; \tag{3.6}$$

これを SSA 形式に変換すると次のようになる。

プログラム 3.3 (プログラム 3.2 の SSA 形式)

$$a_1 = x_0 + y_0; \tag{3.7}$$

$$a_2 = a_1 + 3; \tag{3.8}$$

$$b_1 = x_0 + y_0; \tag{3.9}$$

¹代入文や read 文など

²SSA 形式に対して、もとのプログラムの形式を通常形式とよぶ

たとえば a について見ると、代入があるごとに変数の新しい version を作り、 a_1, a_2 のように添字をつけて区別する。以下は制御の合流がある例である。次のような通常形式のプログラムがあったとする。

プログラム 3.4 (制御の合流がある例)

$x = \dots;$ (3.10)

if($x > 0$){ (3.11)

$a = x;$ (3.12)

else{ (3.13)

$a = -x;$ (3.14)

 } (3.15)

print(a); (3.16)

これを SSA 形式に変換すると次のようになる。

プログラム 3.5 (プログラム 3.4 の SSA 形式)

$x_1 = \dots;$ (3.17)

if($x_1 > 0$){ (3.18)

$a_1 = x_1;$ (3.19)

else{ (3.20)

$a_2 = -x_1;$ (3.21)

 } (3.22)

$a_3 = \phi(a_1, a_2);$ (3.23)

print(a_3); (3.24)

文 (3.23) が属するブロックでは、 a_1 と a_2 の値が合流するので、それ以降に a が使われていると、 a_1 か a_2 かが決められなくなる。そこで、SSA 形式では ϕ 関数 (ϕ function) とよばれる仮想的な関数を導入する。 $a_3 = \phi(a_1, a_2)$ により、これ以後使われる a は a_3 であることを表す。また、この ϕ 関数 $\phi(a_1, a_2)$ は、制御が文 (3.19) が属するブロックから来たときは a_1 の値を返し、文 (3.21) が属するブロックから来たときは a_2 の値を返す関数を表す。

SSA 変換についての効率的なアルゴリズムは、代表的なものに

1. Cytron らの方法 [Cytron et al. 1989; Cytron et al. 1991]
2. Brandis らの方法 [Brandis and Mössenböck 1994]
3. Sreedhar らの方法 [Sreedhar and Gao 1994; Sreedhar and Gao 1995b; Sreedhar and Gao 1995a]
4. Aycock らの方法 [Aycock and Horspool 2000]

が知られている。これらのうち、1 と 3 とを後の 4 章で述べる COINS 上で実装し比較したものが [中谷 2001; 中谷他 2001] である。

3.3 SSA 形式に基づく最適化

SSA 形式では、変数の使用に対する定義が 1 つだけなので、SSA 形式を用いると、コンパイラにおけるデータフロー解析や種々の最適化が見通しよく行なえる。

3.3.1 共通部分式除去

次のような通常形式のプログラムがあったとする。

プログラム 3.6 (通常形式)

$$a = x + y; \tag{3.25}$$

$$a = a + 3; \tag{3.26}$$

$$b = x + y; \tag{3.27}$$

これを SSA 形式に変換すると次のようになる。

プログラム 3.7 (SSA 形式)

$$a_1 = x_0 + y_0; \tag{3.28}$$

$$a_2 = a_1 + 3; \tag{3.29}$$

$$b_1 = x_0 + y_0; \tag{3.30}$$

これに共通部分式除去を適用すると次のようになる。

プログラム 3.8 (プログラム 3.7 の SSA 形式に共通部分式除去を適用した結果)

$$a_1 = x_0 + y_0; \tag{3.31}$$

$$a_2 = a_1 + 3; \tag{3.32}$$

$$b_1 = a_1; \tag{3.33}$$

プログラム 3.8 の 3 行目の右辺では、共通部分式 $x_0 + y_0$ が a_1 に置き換えられている。ちなみに、SSA 形式でない通常形式 (プログラム 3.6) では、3 行目の右辺の $x + y$ を a に置き換えようとしても、 a の値が 2 行目で変えられているので、プログラム 3.8 のような最適化は簡単にはできない。

3.4 SSA 逆変換

SSA 形式を通常形式に戻すことを SSA 逆変換 (*normalization, translating out of SSA form*) という。φ 関数がない SSA 形式を通常形式に戻すのは簡単である。次に簡単な SSA 逆変換の例を示す。次のような SSA 形式のプログラムがあったとする。

プログラム 3.9 (SSA 形式)

$$a_1 = x_0 + y_0; \tag{3.34}$$

$$a_2 = a_1 + 3; \tag{3.35}$$

$$b_1 = a_1; \tag{3.36}$$

これを SSA 逆変換すると次のようになる。

プログラム 3.10 (プログラム 3.9 の SSA 逆変換の結果)

$$a_1 = x_0 + y_0; \tag{3.37}$$

$$a_2 = a_1 + 3; \tag{3.38}$$

$$b_1 = a_1; \tag{3.39}$$

ϕ 関数をそのまま実行できるアーキテクチャはないので，目的コードを生成する前に ϕ 関数の除去を行う必要がある． ϕ 関数がある場合の SSA 逆変換は，あらし次のように行う．次は ϕ 関数がある場合の SSA 逆変換の例である．以下のような SSA 形式のプログラムがあったとする．

プログラム 3.11 (SSA 形式)

$x_1 = \dots;$ (3.40)

if($x_1 > 0$) { (3.41)

$a_1 = x_1;$ (3.42)

else { (3.43)

$a_2 = -x_1;$ (3.44)

 } (3.45)

$a_3 = \phi(a_1, a_2);$ (3.46)

print(a_3); (3.47)

これを SSA 逆変換すると次のようになる．

プログラム 3.12 (プログラム 3.11 の SSA 逆変換の結果)

$x_1 = \dots;$ (3.48)

if($x_1 > 0$) { (3.49)

$a_1 = x_1;$ (3.50)

$a_3 = a_1;$ (3.51)

else { (3.52)

$a_2 = -x_1;$ (3.53)

$a_3 = a_2;$ (3.54)

 } (3.55)

print(a_3); (3.56)

前述したとおり，プログラム 3.11 の文 (3.46) 「 $a_3 = \phi(a_1, a_2)$ 」は，どちらの基本ブロックからこの合流点へ入って来たかによって a_3 の値を決めるものであった．そこで，上の例では，通常形式でそれを復元するために，合流点に入ってくるそれぞれの基本ブロックの最後に文 (3.51) と文 (3.54) のコピー文を置き， ϕ 関数を消去する．

ただし，このままだと無駄なコピー文があるので，合併(*coalescing*) [Chaitin 1982; Briggs et al. 1994; George and Appel 1996; Park and Moon 1998] という手法を使って，それらを取り除くことが多い．プログラム 3.12 で a_1 , a_2 , a_3 を合併すると，元の通常形式である 3.4 と同じものが得られる．

しかし SSA 逆変換では，上記の素朴な方法がうまくいかない場合が知られている [Briggs et al. 1998]．それらの問題点を解決した SSA 逆変換についての効率的なアルゴリズムは，代表的なものに，

1. Briggs らの方法 [Briggs et al. 1998]
2. Sreedhar らの方法 [Sreedhar et al. 1999]

が知られている．これらを後の 4 章で述べる COINS 上で実装し比較したものが [小濱 2002; 小濱他 2002; 小濱 2004] である．

3.5 SSA 形式の定義

定義 3.1 元プログラムの全ての変数 v に対して、次の三条件を満たすとき、そのプログラムは SSA 形式になっているという。

1. もしブロック Z が空でないパス $X \rightarrow Z$ と $Y \rightarrow Z$ (ブロック X とブロック Y は v の定義を含んでいる) の最初の結合ブロックならば、 v に対する ϕ 関数が Z の先頭に挿入されている。
2. v に対する新しい名前 v_i はプログラムテキスト上で唯一の定義を持つ。
3. 任意の制御フローグラフのパスにおいて、 v に対する新しい名前 v_i の使用と、元プログラムでそれに対応する v の使用を考える。このとき v と v_i は同じ値を持つ。

3.6 いろいろな SSA 形式

通常 ϕ 関数を挿入する場所は支配辺境³とよばれるブロックである。この方法により得られる SSA 形式は最小 SSA 形式 (*minimal SSA form*) とよばれる。

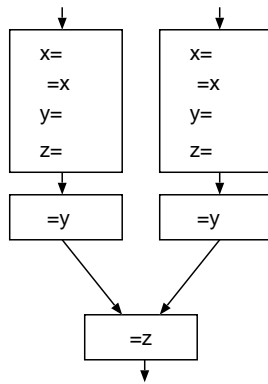


図 3.1: 通常形式

図 3.1 の通常形式に対して、最小 SSA 形式に変換したものが図 3.2 (左) である。

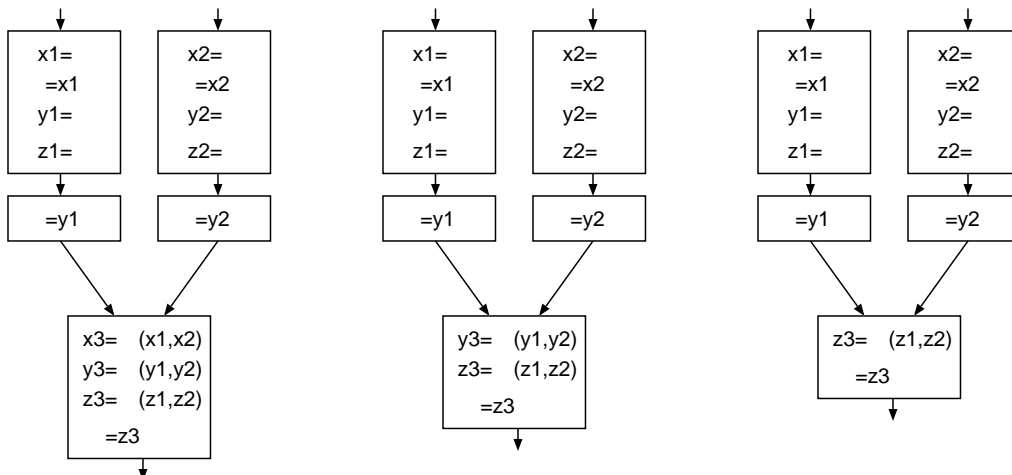


図 3.2: 図 3.1 の SSA 形式

³2 章 2.2.1 節参照。

ただし、それは定義のことだけを考えた場合の最小である。もし、 ϕ 関数によって定義されたものがその後で使わないのであれば、その定義は実際には必要ないと考えられる。そのようなものを除いた SSA 形式を刈り込んだ SSA 形式 (*pruned SSA 形式*) [Choi et al. 1991] という。図 3.1 を刈り込んだ SSA 形式に変換したものが図 3.2 (右) である。

刈り込んだ SSA 形式の方が最小 SSA 形式に比べて、 ϕ 関数の数がずっと少なくなって変数名の数も減るから、SSA 形式でのその後の各種の最適化の処理の時間も空間も効率的になる。また、レジスタ割り付けのときなどのように生存区間を求める解析もする必要がある場合は刈り込んだ SSA 形式の方がよい。ただし、刈り込んだ SSA 形式を作るためには生存変数解析 (*liveness analysis*) [Kennedy 1981] もする必要がある。また、[Cytron et al. 1991] によると、最小 SSA 形式で一目無駄な ϕ 関数があることがある種の最適化を可能にする場合もある。

最小 SSA 形式と刈り込んだ SSA 形式の中間的な SSA 形式としてなかば刈り込んだ SSA 形式 (*semi pruned SSA form*) [Briggs et al. 1998] も提案されている。それは、1 つのブロックの中だけで定義され使用されている変数については ϕ 関数を作らないとするものである。これは、アルゴリズムが簡単で、 ϕ 関数を作らない効果もかなりあるようである。また、SSA 形式を作るのにかかる時間は、最小 SSA 形式の場合がアルゴリズムが一番単純であるにもかかわらず、 ϕ 関数の個数が多くなることによって、一番長いようである [Briggs et al. 1998]。図 3.1 をなかば刈り込んだ SSA 形式に変換したものが図 3.2 (中) である。

第4章 並列化コンパイラ向け 共通インフラストラクチャ COINS

本章では、本手法を実装、実験を行う際に利用した並列化コンパイラ向け共通インフラストラクチャ (*a compiler infra structure*, 以後 COINS とよぶ) について簡単に説明を行う。

4.1 背景

COINS は、コンパイラ研究の基盤となる共通のコンパイラの作成をテーマに 2000 年度より研究が進められている [文部科学省; 渡邊他 1999]。つまり、組み合わせ可能なコンパイラ部品で構成される共通インフラを作り、その上に各企業や研究者がそれぞれの目的に合う機能部品を加えることができるようにすることを目的としている。COINS では SSA 最適化を行っている。現在、多くの最適化コンパイラ [MachSUIF; Fitzgerald et al. 2000; GCC; Intel; IBM] で SSA 最適化の実験が行われているが、それらはテスト段階であり実用的に使える段階には至っていない。現段階の COINS もまだ開発段階であり、虫 (bug) を多く含んでいる。このような状況のため、本手法の実装や実験を行うにあたり様々な制約を受けるのは否めないが、

- 本研究室の佐々政孝教授が COINS の研究プロジェクトに携わっているため、本研究室では COINS をインフラとして利用した研究が多く行われており、データの蓄積が行いやすい。
- 数年後に COINS の完成度が高まったときに本研究の評価の精度も高められる。
- 今後、本研究室やその他の研究機関により最適化などの研究が行われていったときに、それらの比較として本手法も利用できる。

といった理由のため、本手法を実装を行うインフラに COINS を選択した。

4.2 構成

一般にコンパイラは、フロントエンド (*front end*) とバックエンド (*back end*) から構成される。フロントエンドは、原始プログラム (*source program*) を中間コード (*immediate code*) とよばれる内部形式に変換する。バックエンドは、中間コードを計算機の機械コード (*machine code*) に変換する。フロントエンドはさらに、字句解析器 (*lexical analyzer*)、構文解析器 (*syntax analyzer*)、意味解析器 (*semantic analyzer*) に分けられる。バックエンドは、最適化器 (*optimizer*) とコード生成器 (*code generator*) に分けられる。これら各部分は、コンパイラのフェーズとよばれる。

COINS では、複数の入力言語、複数の対象機種に対応する 2 つの中間コードがある (図 4.1)。入力言語の論理構造に近いレベルの中間コードを、高水準共通中間表現 (*high-level intermediate representation*, *HIR*) とよび、機械語に近いレベルの中間コードを、低水準共通中間表現 (*low-level intermediate representation*, *LIR*) とよぶ。

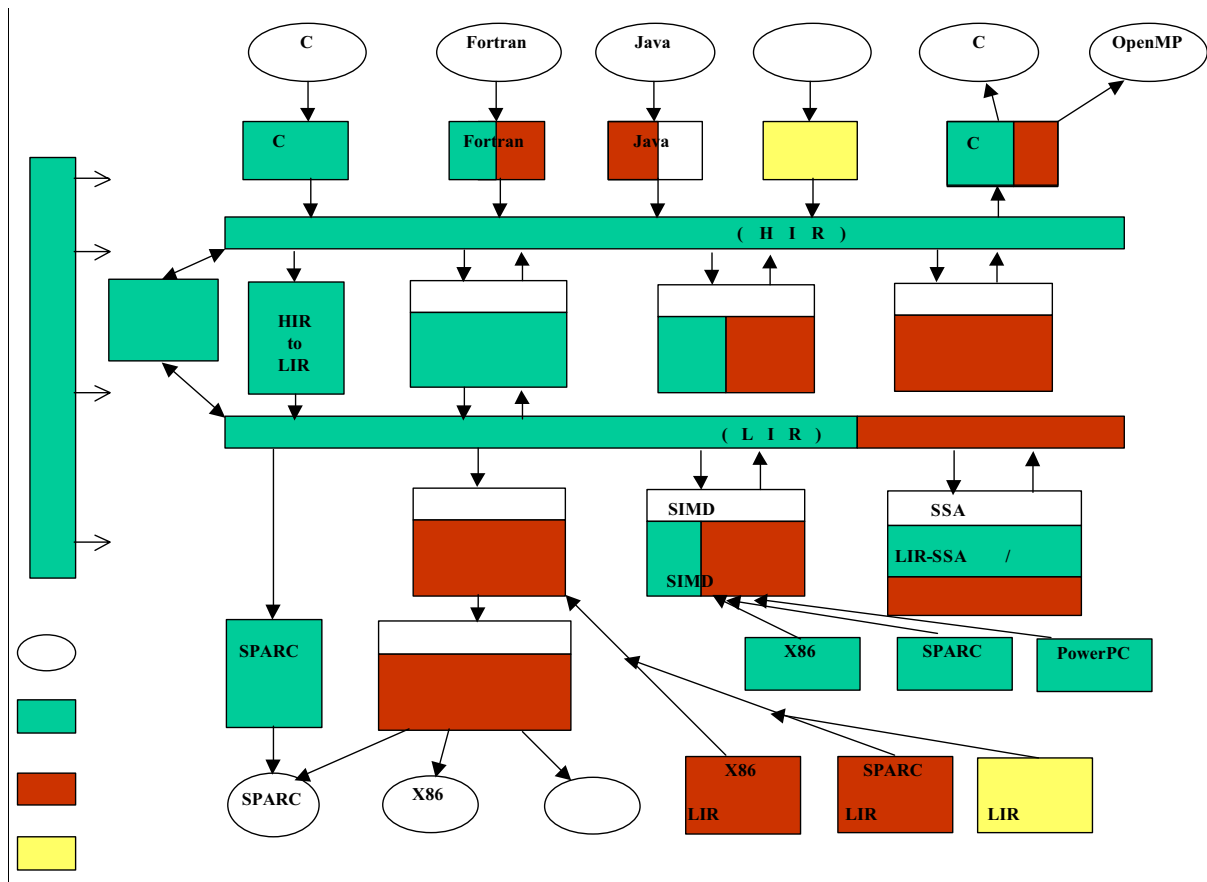


図 4.1: COINS 構成図

今年度 (2003 年度) にて第 1 期が終了し, 来年度 (2004 年度) より第 2 期が開始される予定である. 現状は, それぞれのフェーズではエラーは出ないが, HIR と LIR, 複数の最適化など, フェーズを組み合わせることによってエラーが出てしまうため, 虫の位置を特定しようと努力している段階である.

第5章 部分冗長性除去

本章では、部分冗長性除去(*partial redundancy elimination*, *PRE*)について簡単に説明を行う。

5.1 冗長性

冗長な計算を除去する手法は、コンパイラのコード最適化において強力な手法であり、以前から多くの研究がなされてきた。

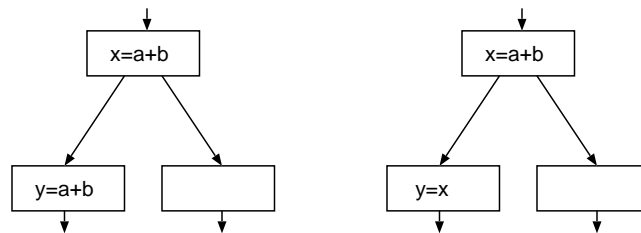


図 5.1: 冗長と冗長除去

図 5.1 に示す 2 つの制御フローグラフは、説明を簡単にするために「 $a + b$ 」の計算に関係のある文（「 $a + b$ 」の計算と、 a や b の変更文）以外の文は省略してあるものとする（以下同様）。

図 5.1 (左) に示す制御フローグラフにおいて、下側の式の「 $a + b$ 」のように、プログラムの開始点から式に到達するすべての経路上に同じ計算式が存在するとき、その式は冗長 (*redundant*) であるという。このとき、図 5.1 (右) のように、冗長な式を除去することによってプログラムの意味を変えずに実行時の効率をあげることができる。これを 共通部分式の除去 (*common subexpression elimination*) という。

5.2 部分冗長性

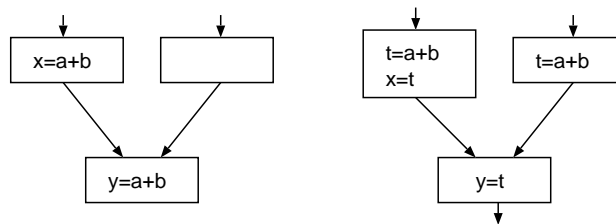


図 5.2: 部分冗長と冗長への変形

図 5.2 (左) の場合、左上のブロックからの経路上では、「 $a + b$ 」が 2 回実行される。しかし、下側の式を除去してしまうと、右上からの経路上に「 $a + b$ 」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。この場合には、図 5.2 (右) のように、右上のブロックに「 $a + b$ 」

を挿入することによって、下側の「 $a + b$ 」は冗長になり、除去可能になる。このように上方に式を挿入することによって、冗長になる式は部分冗長 (*partial redundant*) であるという。

5.3 部分冗長性除去

部分冗長な式は、直前に式を挿入することによって、冗長なものに変えることができる。新たな式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。

データフロー解析を用いて部分冗長を取り除く手法を部分冗長性除去 (*partial redundancy elimination, PRE*) [Morel and Renvoise 1979; Knoop et al. 1992; Knoop et al. 1994] という。図 5.3 (左) に対し部分冗長性除去を行うと図 5.3 (右) のようになる。

部分冗長性除去は共通部分式の除去に加えて ループ不変コード移動 (*loop invariant code motion*) をも統一的に扱うことができる。



図 5.3: ループ不変コード移動

図 5.3 (左) で、ループの直前のブロックに「 $a + b$ 」を挿入することによって、ループ内部の「 $a + b$ 」は冗長となる。すなわち、ループ内部の「 $a + b$ 」は部分冗長であり、部分冗長性除去によってループ不変コード移動が実現できる。結果は図 5.3 (右) のようになる。

5.4 SSA 形式上の部分冗長性除去

部分冗長性除去を SSA 形式上のプログラムに適用するのは簡単ではない。

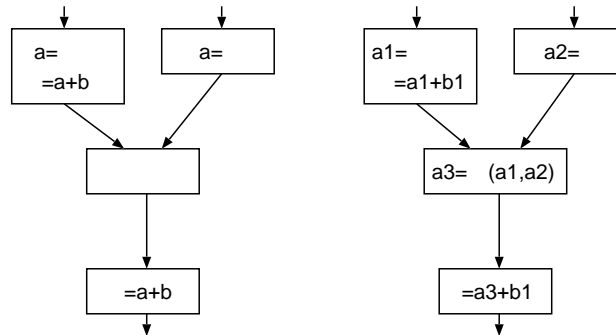


図 5.4: 部分冗長性除去と SSA 形式の例

たとえば、図 5.4 (左) の「 $a + b$ 」は部分冗長性除去の対象となるものであり一番下のブロックの「 $a + b$ 」を右上のブロックに移動することが望ましいが、図 5.4 (右) のような SSA 形式にすると 2 つの「 $a + b$ 」が別々の形になってしまうため、同一の式で冗長であることが認識できなくなってしまうのである。また仮に $a_1 + b_1$ と $a_3 + b_1$ が同じだと認識できたとしても、図 5.4 (右) の

一番下のブロックの $a_3 + b_1$ をそのままの形で右上のブロックに移動することはできない。 a_3 の使用を定義の先行ブロックに移動することはできないからである。

つまり具体的には、部分冗長性除去をこの SSA 形式上で処理しようとするとき、

- 通常形式上で同一の変数として扱えたものが異なる変数として認識されてしまう。
- 異なる基本ブロックにコードを移動する際に変数がそのまま使用できない。

といった困難があるのである。

この問題をなんとか解決して、SSA 形式に対しても部分冗長性除去を行うアルゴリズムがいくつか考えられている。

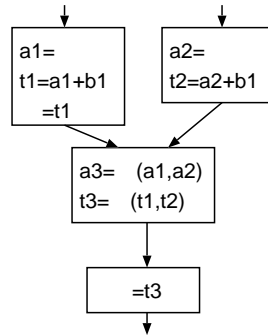


図 5.5: SSA 形式上の部分冗長性除去

たとえば、その中の 1 つを 図 5.4 (右) に対して適用すると、図 5.5 のような SSA 形式が得られる。しかしこれらのアルゴリズムは特別なグラフの作成やそのグラフ上における複雑な解析を必要とするものである。またアルゴリズムの中には、入力は SSA 形式であるが出力されたコードは SSA 形式が崩れてしまうなどのものもある。前者のアルゴリズムは実装が困難であり、後者のアルゴリズムでは途中で SSA 形式でなくなるため、その後 SSA 形式上の最適化を続行することができない。詳しくは、関連研究として第 8 章で述べる。

第6章 SSA 形式上の部分冗長性除去の新手法

Practical-PRE on SSA Form

本章では、著者が考案した SSA 形式上において部分冗長性除去を制御フローグラフ上のみの解析で行う方法 (以後、本手法とよぶ) に関して、そのアルゴリズムを説明する。

SSA 形式は、変数の定義の唯一性などからプログラミング言語処理における最適化に適した表現とされている。しかし、第5章5.4節で述べた通り、部分冗長性の除去を SSA 形式上で処理しようとすると、通常形式上で同一の変数として扱えたものが、異なる変数として認識されてしまったり、異なる基本ブロックにコードを移動する際に変数がそのまま使用できなかつたり、といった問題がある。そのため従来のアルゴリズム [Chow et al. 1997; Kennedy et al. 1999; 滝本・原田 1997] は、制御フローグラフ上のみで処理ができず、複雑な解析と特別なグラフの作成が必要であり、実装がかなり困難であった。

本章は、部分冗長性の除去をきれいな SSA 形式のまま制御フローグラフ上のみで処理する方法を説明する。きれいな SSA 形式とは、SSA 変換がなされた直後では起こり得ない同一 ϕ 関数内の変数の生存区間の重複や、無駄な ϕ 関数のない刈り込んだ SSA 形式のことを指し、きたない SSA 形式とはきれいでない SSA 形式のことを指すことにする。詳しい説明は第7章で説明を行うが、きたない SSA 形式で出力を行うと、後に最適化効果が打ち消されてしまうような例が多く見つけており、きたない SSA 形式で出力する最適化により、実用的なテストプログラムの実行時間が遅くなる結果も示すことができた。本手法は、通常形式上で同一の変数として扱えた変数を一時的に同一の変数と見なして解析し、異なる基本ブロックにコードを移動する際に変数を適切な添字に変えることにより部分冗長性除去を行う。したがって、 ϕ 関数に阻まれることなく ϕ 関数を飛び越えたコード移動も可能になり、通常形式上の部分冗長性除去と同様の成果を得ることができるのである。本手法の最適化効果は SSA 形式上の部分冗長性除去として最新の結果である [Kennedy et al. 1999] と同等である¹。

6.1 背景

コード移動 (*code motion*) と呼ばれる最適化の技術は、実行時の不必要な再計算の回避によりプログラムの効率を改善する技術である。この最適化において求められることは、「もとのプログラムの意味を変えないこと (安全性)」と「より効率のよいコードを出すこと (最適性)」である。計算を移動する場合における安全性とは、計算を移動するそのパスに、その計算で使用される変数の値を変更するような計算を挿入させてはならないということである。また余分な副作用を起こさないことも求められる。コード移動ではこの安全性を維持しながら最適性を考慮する必要がある。

本手法では最適性を考慮する際にレジスタの負担に注目した、怠けたコード移動 (*lazy code motion*) [Knoop et al. 1992; Knoop et al. 1994] とよばれる部分冗長性の除去のアルゴリズムを基にし、これを SSA 形式に対応するような変形を行う。コード移動を簡単に実現するためには、プログラム中のできるだけ前の方に計算式を置けばよいが、これはレジスタの生存区間 (*live range*) を長くすることでレジスタ圧力 (*register pressure*) が増え、ひいては必要以上の負担をかけ、プログラムの実行時間

¹くわしくは第8章参照

を遅くする．そこで怠けたコード移動では計算式を置く場所をできるだけ遅くにし，レジスタの負担を軽減している．また，部分冗長性除去のアルゴリズムの多くが双方向のデータフロー解析を行うのに対し，怠けたコード移動は後ろ向きと前向きの単方向のデータフロー解析を行えばよいアルゴリズムとなっている．本手法でも，単方向のデータフロー解析のみで行えるようにしたため，処理の計算のオーダーは元の通常形式上の怠けたコード移動と同等になる．また最適化効果も通常形式上の怠けたコード移動 [Knoop et al. 1994] と同様である．

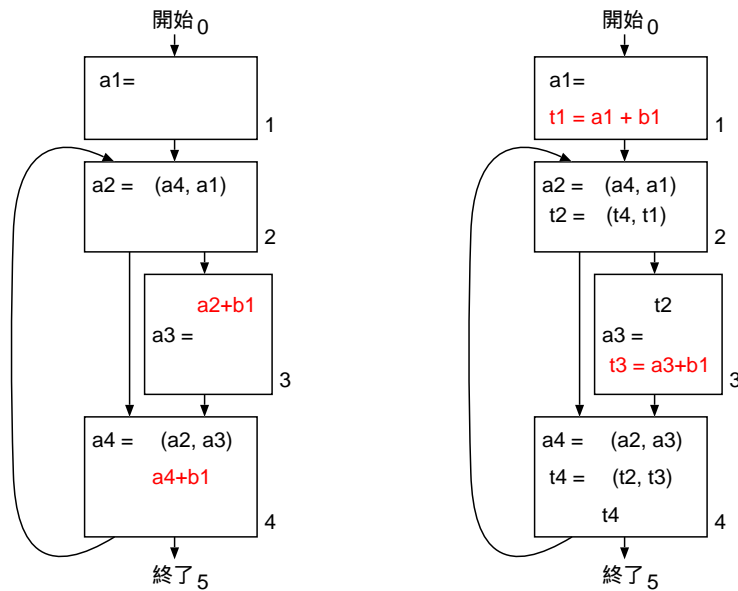


図 6.1: 例プログラムの最適化前と最適化後

図 6.1 は，例プログラム図 6.1 (左) に対し本手法による最適化をかけたもの図 6.1 (右) をそれぞれ制御フローグラフで示している．元の図 6.1 (左) のプログラムは，ループ内で，ブロック 2 → ブロック 3 → ブロック 4 のパスで二項演算が 2 回行われ，ブロック 2 → ブロック 4 のパスで二項演算が 1 回行われている．これを図 6.1 (右) のように，本手法により最適化すると，ブロック 2 → ブロック 3 → ブロック 4 のパスでは二項演算が 1 回と変わらないが，ブロック 2 → ブロック 4 のパスでは二項演算が 0 回となり，ループ外であるブロック 1 へ演算が巻き上げられている．

6.2 本手法のアルゴリズム

本手法では複数種類の単方向のデータフロー解析を行うものである．単方向のデータフロー解析の処理にかかる計算のオーダーは，基本ブロックの数を N ，制御フローグラフの深さを d とすると， $\mathcal{O}(N \cdot d)$ で表される．これは一般のプログラムでは $\mathcal{O}(N)$ と考えてよい²．

本手法によるアルゴリズムは，データフロー方程式をプログラムの制御の流れに沿って解くものであり，データが流れる前向きの流れと，その反対の後ろ向きの流れの両方を含む．また，データフロー方程式には真偽値を扱うものと整数値を扱うものが存在する．

以下で述べる計算は，各 1 種類ずつの式についてそれぞれ，真偽値を扱うデータフロー方程式では 1 ビットで，整数値を扱うデータフロー方程式では 32 ビットで行うことができる．これらに対してそれぞれビットベクトルやイントベクトルを使えば，対象とする式を複数種類にしてそれらを 1 度に計算することができる．

ここで本手法の処理時間の短縮や最適化効果を高めるために，本手法への入力形式にいくつかの

²制御フローグラフの深さや単方向のデータフロー解析に要する計算のオーダーについては，第 2 章 2.2.2 節参照．

制約を設ける．本手法への入力形式は，刈り込んだ SSA 形式³となっているものとする．本手法のアルゴリズムは刈り込んだ SSA 形式に限らなくても正しく動作を行うが，刈り込んだ SSA 形式がもっとも本手法の処理に時間はかからず最適化効果も高い．また本手法では，本手法後の出力形式を刈り込んだ SSA 形式と最小 SSA 形式の両方から選択することができる．しかし，特に制約のない場合には最適化効果の高い刈り込んだ SSA 形式で出力することを推奨する．また，同一 ϕ 関数内の変数の生存区間は重なっていないものとする．通常，SSA 変換をした直後では SSA 形式の性質上この生存区間は重なることはないが，下手な SSA 最適化を行うとこの生存区間が重なってしまう．[小濱 2004] によると，同一 ϕ 関数内の変数の生存区間が重なっている状態で SSA 逆変換を行うとコピー文が残ってしまい，その後に行われる合併によってもそのコピー文が除去されない．本論文でも第 7 章で実験を行ったデータを掲載し考察を行っている．それによると，生存区間が重なっている場合には SSA 最適化を行っているにもかかわらずプログラムの実行時間が増えてしまっている場合も多くあった．したがって，本論文では同一 ϕ 関数内の変数の生存区間が重なるような SSA 最適化は用いないことを推奨するため，このような制約を設ける．この制約により，コード移動する際に必要となる変数の添字の決定や ϕ 関数の挿入に要する支配木の作成や支配辺境を求める計算を行う必要なく計算量が少ないアルゴリズムでそれらを求めることができる．

本章では，図 6.2 の制御フローグラフを例としてアルゴリズムの説明を行う．説明を簡単にするために，冗長性を除去する対象とする式は「 $a+b$ 」の 1 種類のみとし，「 $a+b$ 」と記した場合には通常形式の「 $a+b$ 」ではなく SSA 形式の「 a_1+b_1 」や「 a_2+b_3 」などをすべて含むものとする．

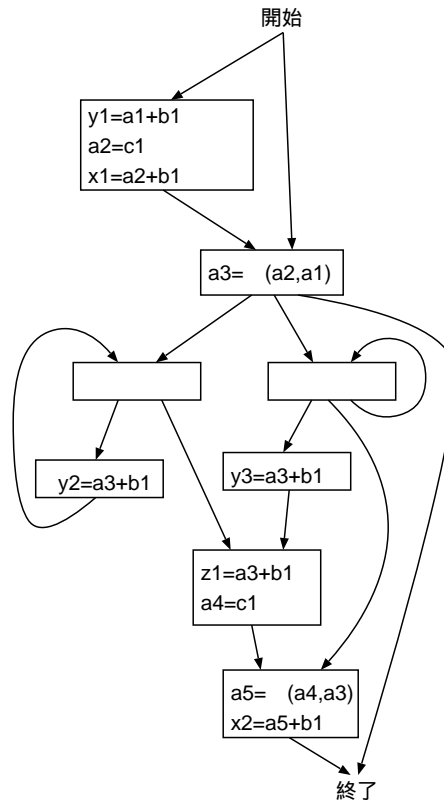


図 6.2: 例題のフローグラフ

³SSA 形式の種類については，第 3 章 3.6 節参照．

6.2.1 危険辺の除去

本手法では、危険辺 (*critical edge*) とよばれる辺が邪魔になるため、前処理としてこれを除去しておく。この危険辺の除去 (*edge splitting*) は、一般の最適化の前によく行われる処理である。任意の制御フローグラフでは、コードの移動がその過程で危険辺によって阻まれることがある。

危険辺とは図 6.3 (左) の ブロック 3 → ブロック 2 の辺のように、後続ブロックを複数持つブロックから先行ブロックを複数持つブロックへの辺のことである。これは分岐ブロックの一方の後続ブロックが、別のパスのブロックとの結合ブロックである場合に起こる。

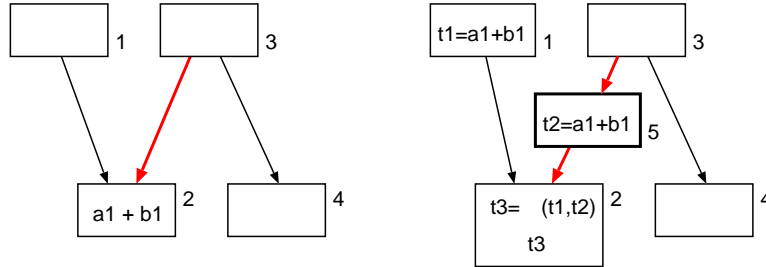


図 6.3: 危険辺の例とその除去

図 6.3 (左) の「 $a + b$ 」の計算をその先行ブロックたちに移動したい場合、ブロック 1 には移動してもプログラムの意味は変わらないが、ブロック 2 に計算を移動すると、ブロック 3 → ブロック 4 へのパスに影響を与えてしまうため、この移動ができない。そこであらかじめ、危険辺の除去をしておくことで、この問題を回避できる。具体的には、図 6.3 (右) のように危険辺上に空のブロック 5 を挿入する。これにより、ブロック 5 に計算を置くことで、ブロック 3 → ブロック 4 へのパスに影響を与えることなく、先行ブロックに計算を移動できるようになった。

図 6.2 を例に危険辺の除去をした例を、図 6.4 に示す。

6.2.2 局所的な性質

本手法では、まず基本ブロック内で局所的な性質 (*local property*) を求め、それからその他の基本ブロックの局所的な性質との関係を解析し大域的な性質 (*global property*) を求める。

本節では、まず局所的な性質について求める。基本ブロック内では、共通部分式の削除はしてあるものとする。通常 SSA 変換する際に簡単な最適化を行うものであり、本手法を実装した COINS でも SSA 変換の際に基本ブロック内の共通部分式の除去は行われているため、この仮定は本手法を実装する際の特別な制約にはならない。その場合、「 $a + b$ 」の計算が連続して現れることはない。以下に、1つの基本ブロックの中で、この「 $a + b$ 」の計算に関係がある部分 (a や b の使用や、 a と b の変更文) だけに注目した例を示す。

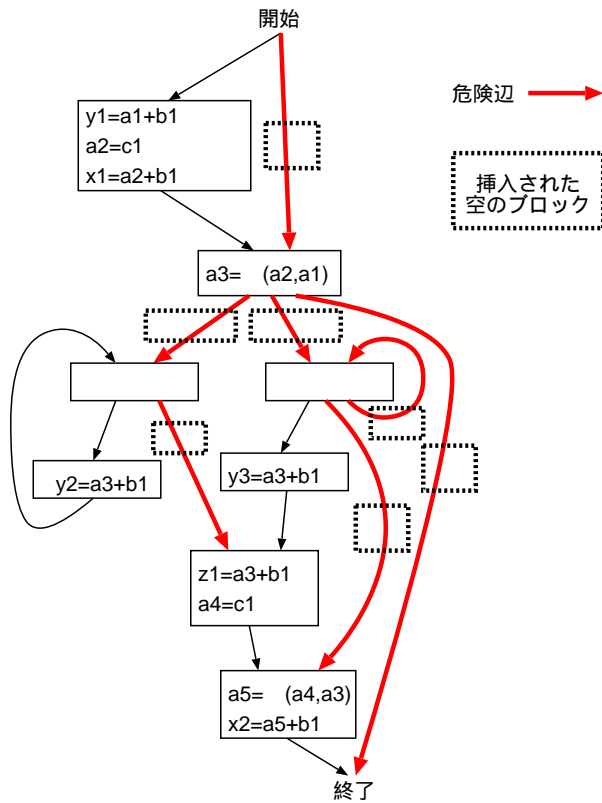


図 6.4: 危険辺の除去の結果

プログラム 6.1 (基本ブロックの内部)

$$a_2 = \phi(a_0, a_1); \tag{6.1}$$

$$b_2 = \phi(b_0, b_1); \tag{6.2}$$

$$c_2 = \phi(c_0, c_1); \tag{6.3}$$

$$x_1 = a_2 + b_2; \tag{6.4}$$

$$a_3 = c_2; \tag{6.5}$$

$$y_1 = a_3 + b_2; \tag{6.6}$$

$$b_3 = d_1; \tag{6.7}$$

$$z_1 = a_3 + b_3; \tag{6.8}$$

基本ブロック内を以下の3つの部分にわけて考えることにする。まず、 ϕ 関数の部分(6.1~6.3)をわけて考え、その部分を ϕ 関数部分とよぶ。残りのうち他の基本ブロックの「 $a+b$ 」と冗長になる可能性がある計算は、 a や b の値が変更する文(ϕ 関数を除く。以後、変更文とよぶ)より前ある最初の「 $a+b$ 」(6.4)と、 a や b の変更文より後にある最後の「 $a+b$ 」(6.8)だけである。その2つを分けた方が考えやすいので、基本ブロックを ϕ 関数の部分と残り2つの部分に分けて考えることにする。残り2つの部分の分け目は ϕ 関数以外の a や b の変更文の最後のもの(6.7)の直後とする。分けたものの最初の方を入口部分とよび、後の方を出口部分とよぶ。変更文がないときは、その基本ブロックの ϕ 関数以外を入口部分とし、出口部分は空とする。入口部分の a や b に代入する文より前にある「 $a+b$ 」の計算(6.4)を入口計算とよび、出口部分の計算(6.8)を出口計算とよぶ。本アルゴリズムでは「 $a+b$ 」の計算を適当なところに挿入するが、その挿入点は各部分に1つずつ設ける。挿入点は、入口計算や出口計算があるときはその計算の直前、入口計算がなく変更

文があるときは最初の変更文の直前，計算と変更文もどちらもなければそのブロックの最後とする．
 また，新しい変数の ϕ 関数の挿入点は， ϕ 関数部分の最後の部分とする．例を図 6.5 に示す．

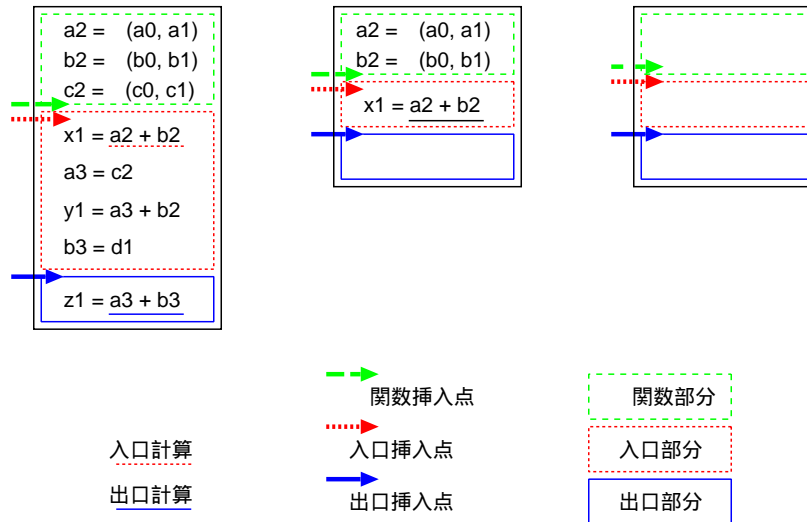


図 6.5: 「 $a + b$ 」に関する基本ブロックの入口と出口

図 6.2 の例に対して，基本ブロックの入口と出口を求めたものを図 6.6 に示す．
 以上により求められる局所的な性質を次のように定義する．

局所的な性質 定義 6.1

- $\mathcal{N}Comp(B)$: (ブロック) B に入口計算がある．
 (あれば *true* , なければ *false* . 以下同様)
- $\mathcal{X}Comp(B)$: B に出口計算がある .
- $Transp(B)$: B に変更文はない .

6.2.3 コード移動

コード移動では冗長な 「 $a + b$ 」 の計算をその計算の先行パスに持ち上げて，適切な位置で一時変数 t に保持し，その t の値を基のブロックの 「 $a + b$ 」 と置換することで冗長性を除去する．
 以後の説明に使用する一時変数 t についても a や b の場合と同様に， t と記した場合には通常形式の t ではなく SSA 形式の t_1 や t_2 などをすべて含むものとする．

コード移動には次の 4 つの手順が必要である .

- ステップ 1 「 $t = a + b$ 」 を挿入する .
- ステップ 2 「 $a + b$ 」 の計算を t で置き換える .
- ステップ 3 t の ϕ 関数を挿入する .
- ステップ 4 a および b の不要な ϕ 関数を除去する .

6.2.4 大域的な性質

ここでコード移動のステップ 1 を行うべき地点を *Insert* , ステップ 2 を行うべき地点を *Replace* とすると，次のように定義できる .

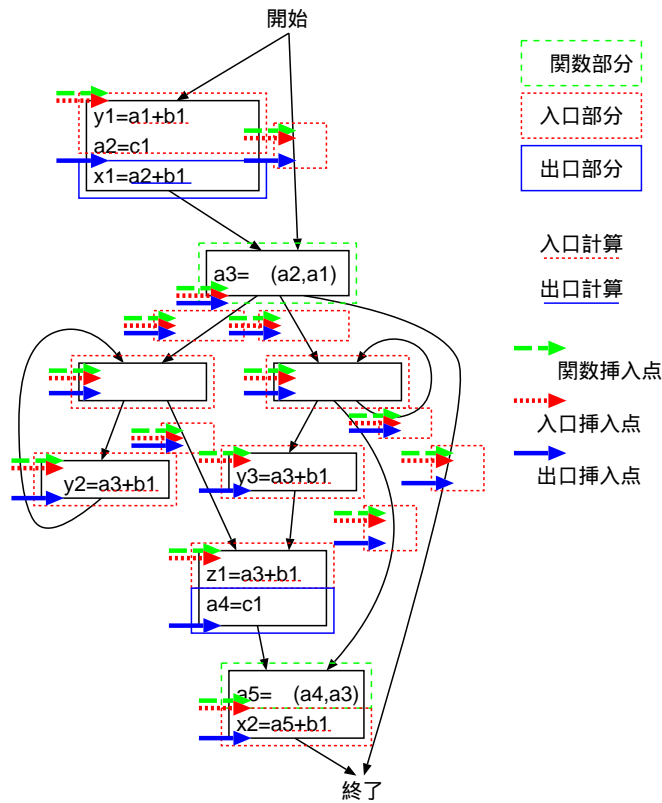


図 6.6: 基本ブロックの入口と出口

大域的な性質 定義 6.1

- $\mathcal{N}Insert(B)$: B の入口挿入点に計算「 $t = a + b$ 」を挿入すべきである .
- $\mathcal{X}Insert(B)$: B の出口挿入点に計算「 $t = a + b$ 」を挿入すべきである .
- $\mathcal{N}Replace(B)$: B の入口計算を t で置き換えるべきである .
- $\mathcal{X}Replace(B)$: B の出口計算を t で置き換えるべきである .

ここで、ステップ 1 で t の定義文を挿入すると、 t の後続パスで ϕ 関数を必要に応じて挿入する必要がある。ステップ 3 では、必要なブロックの ϕ 関数挿入点に t の ϕ 関数を挿入する。 ϕ 関数を挿入すべきブロックを $\mathit{PhiInsert}$ とする。最後にステップ 4 では、 a および b の不要になった ϕ 関数を除去する。これはステップ 2 によって a や b が t に置換されたために、その後続パスでその使用がなければ生存区間を短縮できるので、不要な ϕ 関数は除去すべきである。 ϕ 関数を除去すべきブロックを $\mathit{PhiElimination}$ とする。 $\mathit{PhiElimination}$ は、 a について除去すべきかどうかの真偽値と、 b についてのその真偽値との組となっているものとする。ここでの説明の場合は、コード移動の対象とする式を「 $a + b$ 」のみとしているため a と b についての組としているが、実際の場合は、コード移動の対象とする式に使用されるすべての変数の組として扱う。そうすることで、1 度を使用されている変数の添字の解析を行うことができる。以後、組としたときには同様の意味である。

すると、 $\mathit{PhiInsert}$ と $\mathit{PhiElimination}$ は次のように定義できる。

大域的な性質 定義 6.2

- $\mathit{PhiInsert}(B)$: B の ϕ 関数挿入点に t の ϕ 関数を挿入すべきである .
- $\mathit{PhiElimination}(B)$: B の a, b の ϕ 関数を除去すべきである .

これらは大域的な性質であり，前に求めた局所的な性質やそれらを利用して求めた別の大域的な性質を利用しながら制御フローグラフを解析して求めるのである．

6.2.5 計算の巻き上げ

部分冗長性を除去するには，その計算をもとあったところの先行パスに移動して一時的に値を保持し，もとの計算をその保持した値で置き換えればよい．したがって，まず計算をできるだけ前に移動する計算の巻き上げを行う．

まず，「 $a + b$ 」の計算を安全に移動できる場所を調べる．ある場所 P に計算を移動しても安全であるのは，

1. P からプログラムの出口へのどのパスにも同じ値を与える計算がある（変更文を通らずにその計算に達する）．
2. プログラムの入口から P に達するどのパスにも同じ値を与える計算がある

のどちらかが成り立つ場合である．1 の場合を下安全 (*down safe*) ，2 の場合を上安全 (*up safe*) とよぶ．

大域的な性質 定義 6.3

$\mathcal{N}dSafe(B)$: B の入口挿入点で下安全である．

$\mathcal{X}dSafe(B)$: B の出口挿入点で下安全である．

$\mathcal{N}uSafe(B)$: B の入口挿入点で上安全である．

$\mathcal{X}uSafe(B)$: B の出口挿入点で上安全である．

下安全性 (*DSafe*) に関しては，以下のデータフロー方程式が成り立つ．

データフロー方程式 6.1

$$\begin{aligned} \mathcal{N}dSafe(B) &= \mathcal{N}Comp(B) \cup \{Transp(B) \cap \mathcal{X}dSafe(B)\} \\ \mathcal{X}dSafe(B) &= \mathcal{X}Comp(B) \cup \begin{cases} false & \text{if } B = \text{終了} \\ \bigcap_{S \in succ(B)} \mathcal{N}dSafe(S) & \text{otherwise} \end{cases} \end{aligned}$$

この方程式は「 B の入口挿入点で下安全であるのは， B に入口計算があるか， B の出口挿入点で下安全でかつ B に変更文がない場合である」，「 B の出口挿入点で下安全であるのは， B に出口計算があるか， B のすべての後続ブロックの入口挿入点で下安全である場合である．ただし，後続ブロックがない場合は，後続ブロックでは下安全でないと考える」ことを意味する．この方程式を解いたもの（最大解）を $\mathcal{N}dSafe$ ， $\mathcal{X}dSafe$ とする．

この最大解を求めるのに必要な繰り返し計算は最悪 $d + 2$ 回である．よってこのデータフロー方程式を解く計算のオーダーは $\mathcal{O}(N \cdot d)$ である．

図 6.2 を例に， $DSafe$ の計算の結果を図 6.7 に示す．

ブロックの上（下）の部分に $DSafe$ があるのは，そのブロックの入口挿入点（出口挿入点）で $\mathcal{N}dSafe(\mathcal{X}dSafe)$ が *true* であることを示す（以下同様）．たとえば図 6.7 の左上のブロックでは，入口計算「 $y_1 = a_1 + b_1$ 」があるので入口挿入点で下安全，出口計算「 $x_1 = a_2 + b_1$ 」があるので出口挿入点で下安全である．

上安全性 (*USafe*) に関しては，以下のデータフロー方程式が成り立つ．

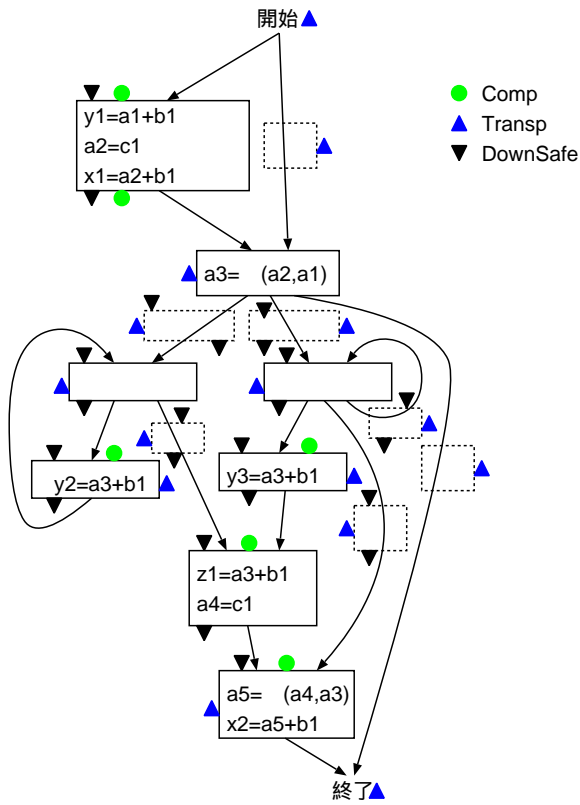


図 6.7: $DSafe$ の結果

データフロー方程式 6.2

$$\begin{aligned}
 NuSafe(B) &= \begin{cases} false & \text{if } B = \text{開始} \\ \bigcap_{P \in pred(B)} \{ \mathcal{X}Comp(P) \cup \mathcal{X}uSafe(P) \} & \text{otherwise} \end{cases} \\
 \mathcal{X}uSafe(B) &= Transp(B) \cap \{ \mathcal{N}Comp(B) \cup NuSafe(B) \}
 \end{aligned}$$

この方程式は、「 B の入口挿入点で上安全であるのは、 B のすべての先行ブロックが、出口計算を持つか出口挿入点で上安全である場合である。ただし、開始ブロックでは上安全でない」、「 B の出口挿入点で上安全であるのは、 B に変更文がなく、かつ、 B に入口計算があるか B の入口挿入点で上安全である場合である」ことを意味する。この方程式を解いたもの (最大解) を $NuSafe$, $\mathcal{X}uSafe$ とする。

この最大解を求めるのに必要な繰り返し計算も最悪 $d+2$ 回である。よってこのデータフロー方程式を解く計算のオーダーも $\mathcal{O}(N \cdot d)$ である。

$USafe$ の計算の結果を図 6.8 に示す。

次に、安全に計算をその先行パスのなるべく開始ブロックに近くまで移動できる場所を求める。すなわち下安全でできるだけ開始ブロックに近い場所は、次の定義で与えられる。

大域的な性質 定義 6.4

- $\mathcal{N}Earliest(B)$: B の入口挿入点に計算を置いてても下安全であるが、それを B のどれかの先行ブロックに移すことはできない (安全でなくなる)。
- $\mathcal{X}Earliest(B)$: B の出口挿入点に計算を置いてても下安全であるが、それを B の入口挿入点に移すことはできない (安全でなくなる)。

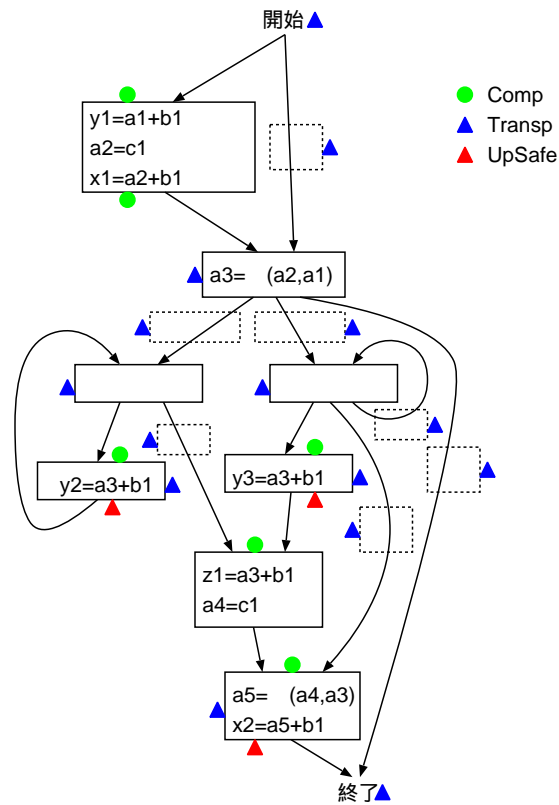


図 6.8: *USafe* の結果

Earliest は次の式で計算できる .

データフロー方程式 6.3

$$\mathcal{N}Earliest(B) = \mathcal{N}dSafe(B) \cap \bigcap_{P \in pred(B)} \overline{\{\mathcal{X}uSafe(P) \cup \mathcal{X}dSafe(P)\}}$$

$$\mathcal{X}Earliest(B) = \mathcal{X}dSafe(B) \cap \overline{Transp(B)}$$

これらの式の右辺は、「 B の入口挿入点で下安全であり、かつ、 B のすべての先行ブロックの出口挿入点で下安全でも上安全でもない」、「 B の出口挿入点で下安全であり、かつ、 B に変更文がある」ことを意味する .

このデータフロー方程式は繰り返し計算による解析を必要としない . したがって全ての基本ブロックを 1 度解析すればよいので、計算のオーダーは $\mathcal{O}(N)$ である .

Earliest の計算の結果を図 6.9 に示す .

たとえばこの左上のブロックでは、入口計算で下安全かつ先行ブロックで上安全でも下安全でもないので、入口計算で *Earliest* である .

6.2.6 コード移動 レベル 1

Earliest まで計算を行うと、この時点でコード移動を正しく行うことができる . この計算の巻き上げをした時点においてコード移動を行う場合は、 $\mathcal{N}Earliest(B)$ が *true* である B の入口挿入点と $\mathcal{X}Earliest(B)$ が *true* である B の出口挿入点でコード移動のステップ 1 を行い、 $\mathcal{N}Comp(B)$ が *true* である B の入口計算と $\mathcal{X}Comp(B)$ が *true* である B の出口計算についてステップ 2 を行えばよい .

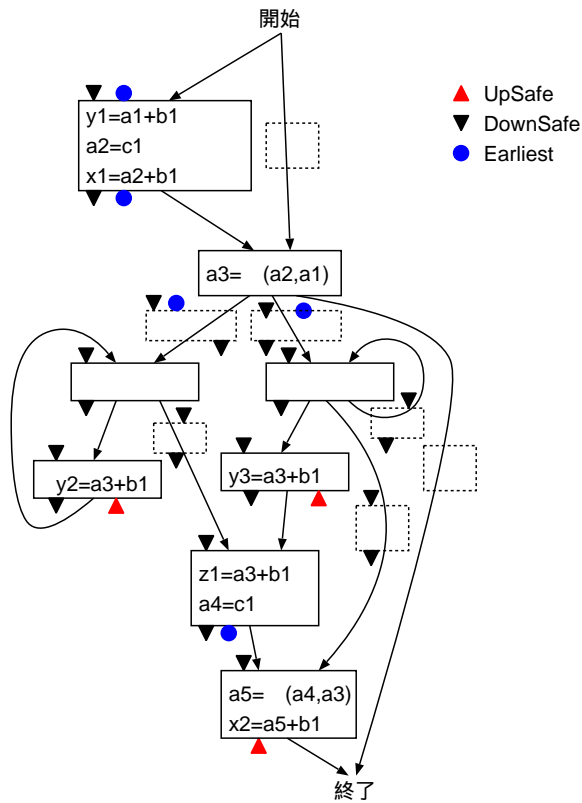


図 6.9: Earliest の結果

すなわち,

$$\mathcal{N}Insert(B) = \mathcal{N}Earliest(B)$$

$$\mathcal{X}Insert(B) = \mathcal{X}Earliest(B)$$

$$\mathcal{N}Replace(B) = \mathcal{N}Comp(B)$$

$$\mathcal{X}Replace(B) = \mathcal{X}Comp(B)$$

としてコード移動を行うのである。これを本手法のレベル1のコード移動とする。

ここで、先に述べたとおり、ステップ1で t の定義を挿入すると、その変数の後続のブロックで必要に応じて ϕ 関数を挿入しなければならないため、ステップ3を行わなければならないのであった。また、最後のステップ4で a および b の不要になった ϕ 関数を除去すべきであった。

次の節でこれらのステップについて必要となる添字や ϕ 関数について説明を行う。

6.2.7 添字の解析

コード移動のステップ1から4を行うためには、ステップ1で挿入する t, a, b の添字や、ステップ2で置き換える t の添字、ステップ3で ϕ 関数を挿入するブロックの場所や、その左辺と右辺の添字について求めなくてはならない。

本手法では、ステップ1で挿入する t, a, b の添字やステップ2で置き換える t の添字を、支配木を利用して求めることはせず、ステップ3で ϕ 関数を挿入するブロックについても繰り返し支配境界を利用して求めない。支配木や繰り返し支配境界を利用した添字の求め方は、ステップ1やステップ2でいったん通常形式のまま入れておき、その後 t の繰り返し支配境界を求めて ϕ 関数を挿入し、 t, a, b の変数の添字を支配木を辿って求めるという方法である。支配木や繰り返し支配辺

境は SSA 最適化の前に SSA 変換のフェーズでいったん求められるものであるが、その後に行われる最適化や危険辺の除去により制御フローグラフが変わるため、再び計算して求める必要がある。支配木を求める計算のオーダーは、一般に利用されるのアルゴリズムで $\mathcal{O}(N \log N)$ であるが、さらに努力をすればアッカーマン関数の逆数を用いて $\mathcal{O}(N \cdot \text{Ack}^{-1}(N))$ まで下げられる。繰り返し支配辺境を求める計算のオーダーは、最悪 $\mathcal{O}(N^2)$ である⁴。

以下に説明する本手法の解析は単方向のデータフロー解析を行うものであり、その計算のオーダーは繰り返し解析が必要なもので $\mathcal{O}(N \cdot d)$ 、その必要がないもので $\mathcal{O}(N)$ となる。繰り返し解析が必要なデータフロー解析に必要な繰り返し計算の回数は、この添字を求める場合においても最悪 $d + 2$ である。そして、先に述べたとおり $\mathcal{O}(N \cdot d)$ も実際のプログラムでは $\mathcal{O}(N)$ と考えてよい。

まず最初に、ステップ 1 により挿入する「 $t = a + b$ 」の左辺 t の添字と 3 により挿入する t の ϕ 関数の左辺の添字を求める。そこで次のように定義する。

局所的な性質 定義 6.2

$NTempSetSuffix(B)$: B の入口挿入点に「 $t = a + b$ 」を挿入するときの t の添字
(挿入の必要がなければ $null$ を入れる。以下同様)。

$XTempSetSuffix(B)$: B の出口挿入点に「 $t = a + b$ 」を挿入するときの t の添字。

$TempPhiSuffix(B)$: B の ϕ 関数挿入点に t の ϕ 関数を挿入する時の左辺の t の添字。

これらは t を定義する添字であるため、SSA 形式の規則により重複してはならない。そこで、オートナンバー型の変数 v を設けておく。オートナンバー型の変数とは、最初は 0 でその変数が呼ばれるたびに 1 つずつ増えていく (オートナンパリング) 整数型のことをいう。 $TempSetSuffix$ に関しては挿入する場所が求められているため、順番にこだわらず適当な順で v を入れていく。 $TempPhiSuffix$ に関しては ϕ 関数を挿入する場所がまだ求められていないので、これを求めることにする。

本手法では、まず最初に最小 SSA 形式⁵を求め、その後刈り込んだ SSA 形式を求める。すなわち、 t の添字が異なる先行ブロックを持つブロックをすべて求め、その後無駄に ϕ 関数を挿入するブロック (図 6.10 の下のブロック) を求めてこれを除去する。これらの ϕ 関数を挿入する場所を次のように定義する。

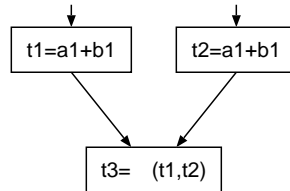


図 6.10: 最小 SSA 形式の例

大域的な性質 定義 6.5

$MinPhiInsert(B)$: 最小 SSA 形式となるように、
 B の ϕ 関数挿入点に t の ϕ 関数を挿入すべき。

$PrunPhiInsert(B)$: 刈り込んだ SSA 形式となるように、
 B の ϕ 関数挿入点に t の ϕ 関数を挿入すべき。

そこで、各ブロックの ϕ 関数挿入点において定義が到達している t の添字を次のように定義する。

⁴ 支配木や繰り返し支配辺境、およびその計算のオーダーについては、第 2 章 2.2.1 節参照。

⁵ 3 章 3.6 節参照。

大域的な性質 定義 6.6

$LiveSuffix(B)$: B の ϕ 関数挿入点における t の添字 .

ここで i を整数型としたとき $\uplus i$ を, i が異なる 2 種類以上のものである場合 v を返し, i が 1 つもしくは 2 つ以上の同一のものである場合 i を返すものと定義する .

また $x \in X, y \in Y$ のとき, x が $null$ でなければ x , y が $null$ なら y をとる演算を, $X \odot Y$ で表すことにする . すると $LiveSuffix$ は次の式で求められる .

データフロー方程式 6.4

$$LiveSuffix(B) = \begin{cases} null & \text{if } B = \text{開始} \\ \uplus_{P \in pred(B)} \{ \mathcal{X}TempSetSuffix(P) \odot \mathcal{N}TempSetSuffix(P) \odot LiveSuffix(P) \} & \text{otherwise} \end{cases}$$

この式の意味は, どの先行ブロックの出口における t の添字も同一のもの, もしくは先行ブロックが 1 つしかなければ, ϕ 関数を挿入する必要がないためそれらの先行ブロックの出口における添字が ϕ 関数挿入点における添字となり, 先行ブロックのうちの 2 つの出口の添字が異なるならば, ϕ 関数を挿入する必要があるため新たな t の添字 v が ϕ 関数挿入点における添字となる (t_v が ϕ 関数の左辺) .

このデータフロー解析は解が収束するまで計算するものであるが, これまでの真偽値を扱ったデータフロー方程式と同様に不動点が求まるのに必要な繰り返し計算の回数は最悪 $d + 2$ ですむ . 以下にわかりやすく説明する . このデータフロー方程式の計算と同時に, \uplus が v を返したブロックの $MinPhiInsert$ を随時 $true$ に変更していく . すると, この値が再び $false$ に戻ることはない . つまりこのデータフロー方程式は $MinPhiInsert$ の最大解を求めるこれまでと同様の真偽値のデータフロー方程式と同様であり, その繰り返し計算の回数は最悪 $d + 2$ である . すなわち計算のオーダーは $O(N \cdot d)$ である .

また, このデータフロー方程式の $\mathcal{X}TempSetSuffix(P) \odot \mathcal{N}TempSetSuffix(P)$ の部分は値が変わらないため, この部分の繰り返し計算の必要はなく 1 度計算して値を固定しておけばよい .

$LiveSuffix$ と $MinPhiInsert$ を求めた結果を図 6.11 に示す . $null$ が入っている値は非表示とする (以下同様) .

これにより $MinPhiInsert$ が $true$ のブロックの ϕ 関数挿入点に t の ϕ 関数を挿入すれば最小 SSA 形式が求まる . 挿入する ϕ 関数の左辺の添字は, その場所の $LiveSuffix$ を用いればよい (右辺の添字については, 後に求めるものとする) .

しかし, この形式には無駄がある . 挿入した ϕ 関数のうち, 入口挿入点の後続パスで t が使用されなければ ϕ 関数を挿入する必要はない (図 6.10) . そこで各ブロックの ϕ 関数挿入点の後続パスで t が使用されるかどうかを調べる . 各ブロックの入口挿入点と出口挿入点において, その点の後続パスの t の使用の有無を次のように定義する .

大域的な性質 定義 6.7

$\mathcal{N}TempUseLater(B)$: B の入口挿入点の後続パスで t が使用される .

$\mathcal{X}TempUseLater(B)$: B の出口挿入点の後続パスで t が使用される .

これらは次の式で求められる .

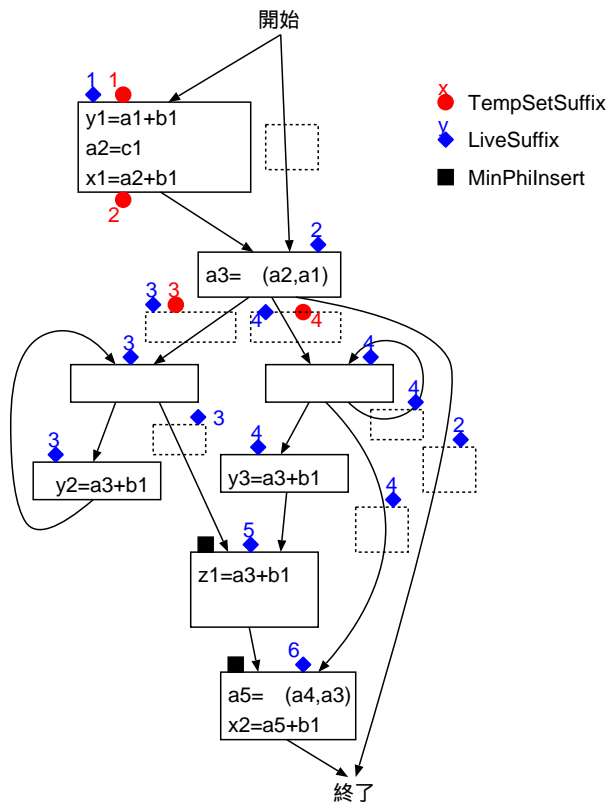


図 6.11: *LiveSuffix* , *MinPhiInsert* の結果

データフロー方程式 6.5

$$\begin{aligned}
 \mathcal{N}TempUseLater(B) &= \begin{cases} false & \text{if } B = \text{終了} \\ \overline{\mathcal{N}Earliest(B) \cap \{\mathcal{N}Comp(B) \cup \mathcal{X}TempUseLater(B)\}} & \text{otherwise} \end{cases} \\
 \mathcal{X}TempUseLater(B) &= \overline{\mathcal{X}Earliest(B) \cap \{\mathcal{X}Comp(B) \cup \bigcup_{S \in succ(B)} \mathcal{N}TempUseLater(S)\}}
 \end{aligned}$$

これらの式の意味は、「 B の入口挿入点以降に t が使用されるのは、 B の入口挿入点に計算を挿入しない、かつ、 t に置換される入口計算があるか出口挿入点以降に t が使用される場合であり、 B の出口挿入点以降に t が使用されるのは、 B の出口挿入点に計算を挿入しない、かつ、 t に置換される出口計算があるか後続ブロックのどれか 1 つで入口挿入点以降に t が使用される」ことを意味する。

この最大解を求めるのに必要な繰り返し計算も最悪 $d+2$ 回である。よってこのデータフロー方程式を解く計算のオーダーも $\mathcal{O}(N \cdot d)$ である。

TempUseLater を求めた結果を図 6.12 に示す。

MinPhiInsert と *TempUseLater* によって刈り込んだ SSA 形式を求める。最小 SSA 形式の ϕ 関数 (*MinPhiInsert*) のうち後続パスで t の使用があるものだけ残せばよいので、次の式で表される。

データフロー方程式 6.6

$$PrunPhiInsert(B) = \mathcal{N}TempUseLater(B) \cap MinPhiInsert(B)$$

この式の意味は、「刈り込んだ SSA 形式の ϕ 関数の挿入点は、最小 SSA 形式の ϕ 関数の挿入点から以降で使用されないものを除いたものである」ことを意味する。

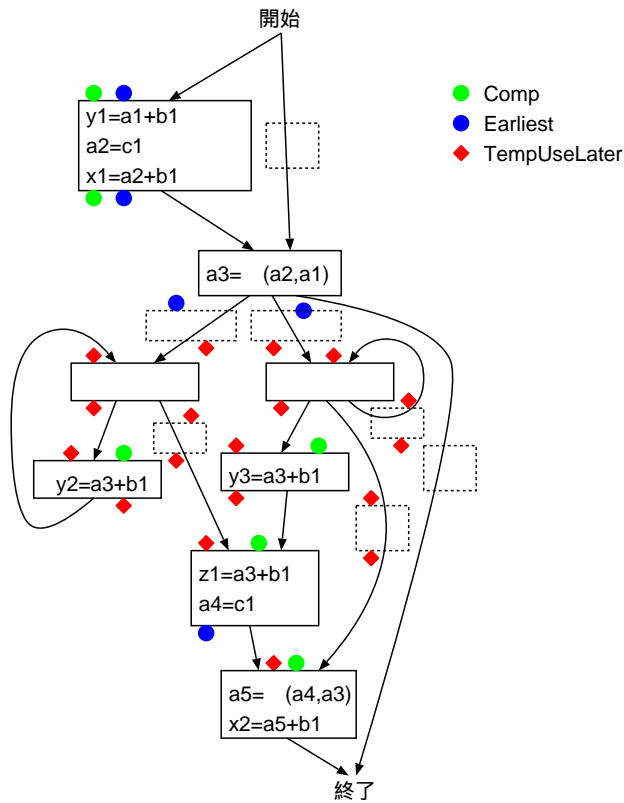


図 6.12: *TempUseLater* の結果

このデータフロー方程式は繰り返し計算による解析を必要としない．したがって全ての基本ブロックを1度解析すればよいので，計算のオーダーは $O(N)$ である．

PrunPhiInsert を求めた結果を図 6.13 に示す．この *PrunPhiInsert* が *true* の ϕ 関数挿入点に ϕ 関数を挿入すれば刈り込んだ SSA 形式となり，無駄な ϕ 関数は挿入されなくてすむ．挿入する ϕ 関数の左辺の添字 *TempPhiSuffix* は，その場所の *LiveSuffix* を用いればよい(右辺の添字については，後に求めるものとする)．

次に，コード移動のステップ1で挿入する「 $t = a + b$ 」の右辺に出てくる変数 a ， b の添字のつけ方，およびステップ2で置換する変数 t の添字のつけ方について説明する．

まず，ステップ2で「 $a + b$ 」の計算を t で置き換えるときの t の添字を求める．

大域的な性質 定義 6.8

$\mathcal{N}TempReplaceSuffix(B)$: B の入口計算「 $a + b$ 」と置換すべき t の添字．

$\mathcal{X}TempReplaceSuffix(B)$: B の出口計算「 $a + b$ 」と置換すべき t の添字．

TempReplaceSuffix は，置換が行われるかどうかには拘わらず求めておくものであり，その場所で置換が行われる場合にこの値は使用されるが，行われない場合にはこの値は使用されないことになる．ここで $x \in X, y \in Y$ のとき， x が *null* でなければ x ， x が *null* なら y をとる演算 $X \odot Y$ を再び適用すると，以下のデータフロー方程式が成り立つ．

データフロー方程式 6.7

$$\mathcal{N}TempReplaceSuffix(B) = \mathcal{N}TempSetSuffix(B) \odot LiveSuffix(B)$$

$$\mathcal{X}TempReplaceSuffix(B) = \mathcal{X}TempSetSuffix(B) \odot \mathcal{N}TempReplaceSuffix(B)$$

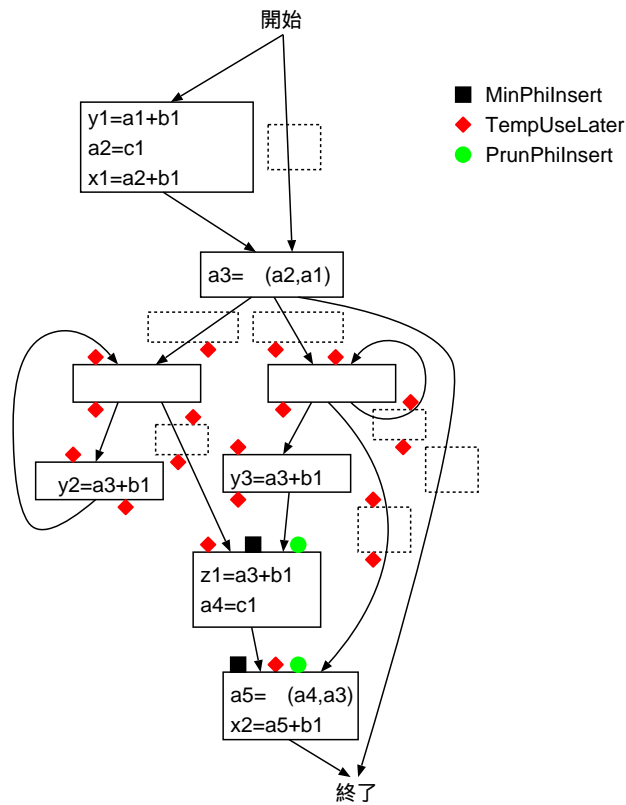


図 6.13: *PrunPhiInsert* の結果

これらの式の意味は、「入口計算と置換すべき t の添字は、 t の入口挿入が行われればその t の添字、行われなければ ϕ 関数挿入点での t 添字」であり、また「出口計算と置換すべき t の添字は、 t の出口挿入が行われればその t の添字、行われなければ入口計算と置換すべき t の添字」である。

このデータフロー方程式は繰り返し計算による解析を必要としない。したがって全ての基本ブロックを1度解析すればよいので、計算のオーダーは $O(N)$ である。この計算は *LiveSuffix* の計算と同時に求めてしまうこともできる。

ここで、ステップ3のときに挿入する t の ϕ 関数の右辺の添字は、その先行ブロックの $\mathcal{X}TempReplaceSuffix$ を使用すればよい。

TempReplaceSuffix の計算の結果を図 6.14 に示す。

次に、ステップ1の右辺の a および b の添字を求める。これらを求めるために、次の情報を使用する。ここでは a の添字を i 、 b の添字を j であるとしたとき、 (i, j) というように添字の組にして考えるものとする。

局所的な性質 定義 6.3

$CompPhiSuffix(B)$: B の a の ϕ 関数の左辺の添字と b の ϕ 関数の右辺の添字の組
(なければ *null* . 以下同様) .

$CompSetSuffix(B)$: B の a の最後の変更文と b の最後の変更文の添字の組 .

これらを利用して次の値を求める。

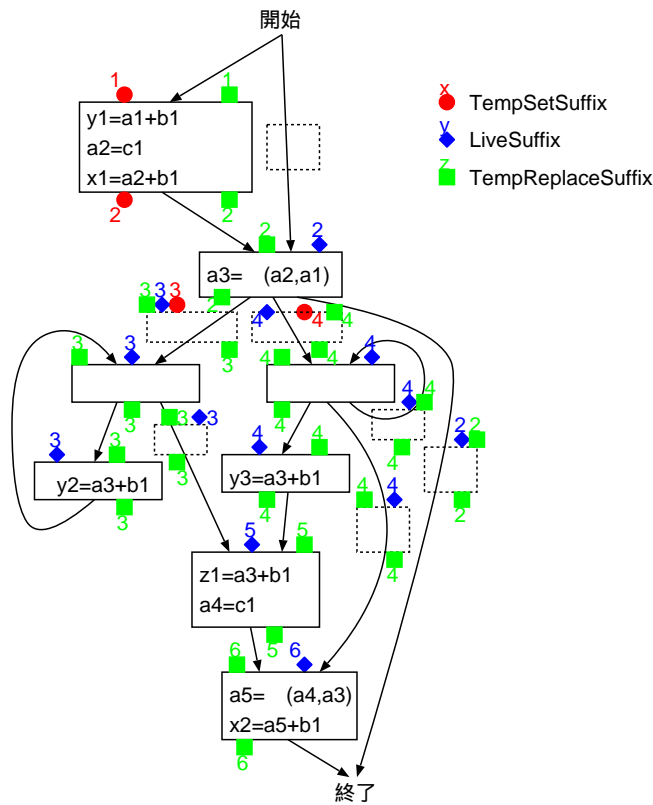


図 6.14: *TempReplaceSuffix* の結果

大域的な性質 定義 6.9

$\mathcal{N}CompSuffix(B)$: B の入口挿入点に「 $t = a + b$ 」を挿入すべき a, b の添字の組 .

$\mathcal{X}CompSuffix(B)$: B の出口挿入点に「 $t = a + b$ 」を挿入すべき a, b の添字の組 .

$CompSuffix$ は、挿入が行われるかどうかには拘わらず求めておくものであり、その場所で挿入が行われる場合にはこの値は使用されるが、行われない場合にはこの値は使用されないことになる . 中でも、 $x \in X, y \in Y$ のとき x が $null$ でなければ x , y が $null$ なら y をとる演算 $X \odot Y$ を適用する . $(s, t) \odot (v, w)$ は $(s \odot v, t \odot w)$ と計算する .

すると、 $CompSuffix$ のデータフロー方程式は、次の式で計算できる .

データフロー方程式 6.8

$$\mathcal{N}CompSuffix(B) = CompPhiSuffix(B) \odot \mathcal{X}CompSuffix(\forall P \in pred(B))$$

$$\mathcal{X}CompSuffix(B) = CompSetSuffix(B) \odot \mathcal{N}CompSuffix(B)$$

これらの式は、「 B の入口挿入点での添字の組はそれぞれ、 ϕ 関数があればその左辺の添字、 ϕ 関数がないければ任意の先行ブロックの出口挿入点での添字、の組」、「 B の出口挿入点での添字の組はそれぞれ、変更文があればその左辺の添字、変更文がないければ入口挿入点での添字、の組」であることを意味する .

ここで、 $\mathcal{N}CompSuffix$ の右辺が任意の先行ブロックでよい理由は、 ϕ 関数があれば $CompPhiSuffix$ が $null$ にはならず、 $CompPhiSuffix$ が $null$ の場合は ϕ 関数がないので先行ブロックから合流する変数の添字はどの先行ブロックのものを取っても同じになるためである .

このデータフロー方程式は繰り返し計算による解析を必要としない．したがって全ての基本ブロックを1度解析すればよいので，計算のオーダーは $O(N)$ である．この計算は，先に出てきた *Earliest* の計算と同時にしてしまうことができる．

CompSuffix の計算の結果を図 6.15 に示す．この例題のプログラムでは b には変更文が存在しないため b の添字はどのブロックの入口と出口においても 1 である．したがって図 6.15 には a の添字のみ扱った結果を示した (以下同様) ．

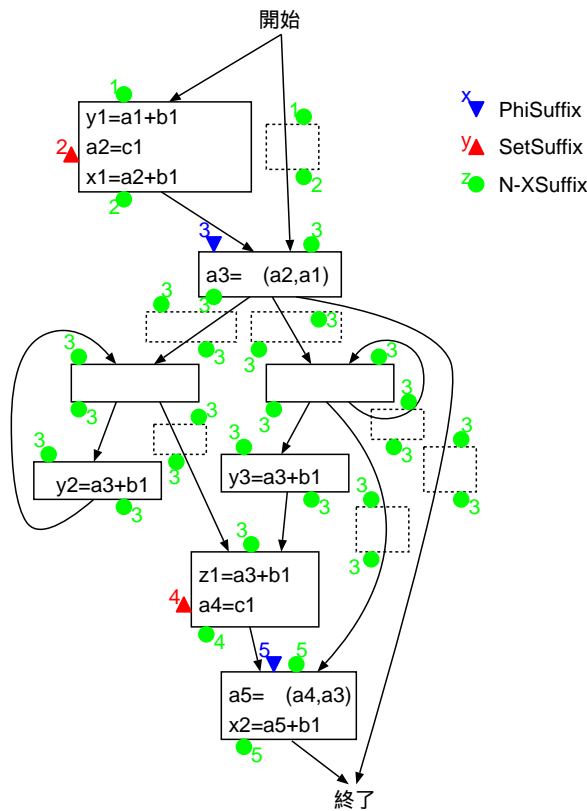


図 6.15: *CompSuffix* の結果 (a のみ表示)

ここで，図??の終了ブロックの左側の先行ブロックには a の ϕ 関数があるが， $a_5 + b_1$ の計算が t に置換されてしまうため，その後に a_5 が使用されることはない．このような ϕ 関数は変数の生存区間を長くしてしまうので除去する．

a や b の ϕ 関数のうち後続パスで使用があるものだけ残せばよい．そこで a と b の組になっている *EachUse* , *EachTransp* , *EachPhi* を定義する．

局所的な性質 定義 6.4

EachUse(B) : B の入口部分において $a(b)$ の最初の変更文までに $a(b)$ の使用がある．
 (*Replace*(B) が *true* となる B の入口 (出口) 計算での使用を除く．)
 (レベル 1 のコード移動では B の入口 (出口) 計算での使用を除く．)

EachTransp(B) : B に $a(b)$ の変更文がない．

EachPhi(B) : B の ϕ 関数部分に $a(b)$ の ϕ 関数がある．

この *EachUse* と *EachTransp* を使用し，そのブロックの ϕ 関数挿入点の後続パスに使用があるかどうかを示す *CompUseLater* を定義する．この値もまた a と b の組になっている．

大域的な性質 定義 6.10

$CompUseLater(B)$: B の ϕ 関数挿入点の後続パスに $a(b)$ の使用がある .

$CompUseLater$ は次のデータフロー方程式の最大解で求められる .

データフロー方程式 6.9

$$CompUseLater(B) = \begin{cases} false & \text{if } B = \text{終了} \\ \mathcal{N}Insert(B) \cup EachUse(B) \cup \{ \overline{EachTransp(B)} \cap \bigcup_{S \in succ(B)} CompUseLater(S) \} & \text{otherwise} \end{cases}$$

この式の意味は、「 B の ϕ 関数挿入点の後続パスで $a(b)$ が使用されるのは、 B の入口挿入点に「 $t = a + b$ 」が挿入されるか、 B の最初の変更文までに $a(b)$ の使用があるか、 B に変更文がない、かつ、 B の後続ブロックの ϕ 関数挿入点の後続パスで $a(b)$ の使用がある」である .

この最大解を求めるのに必要な繰り返し計算は最悪 $d + 2$ 回である . よってこのデータフロー方程式を解く計算のオーダーは $\mathcal{O}(N \cdot d)$ である .

$CompUseLater$ の計算の結果を図 6.16 に示す .

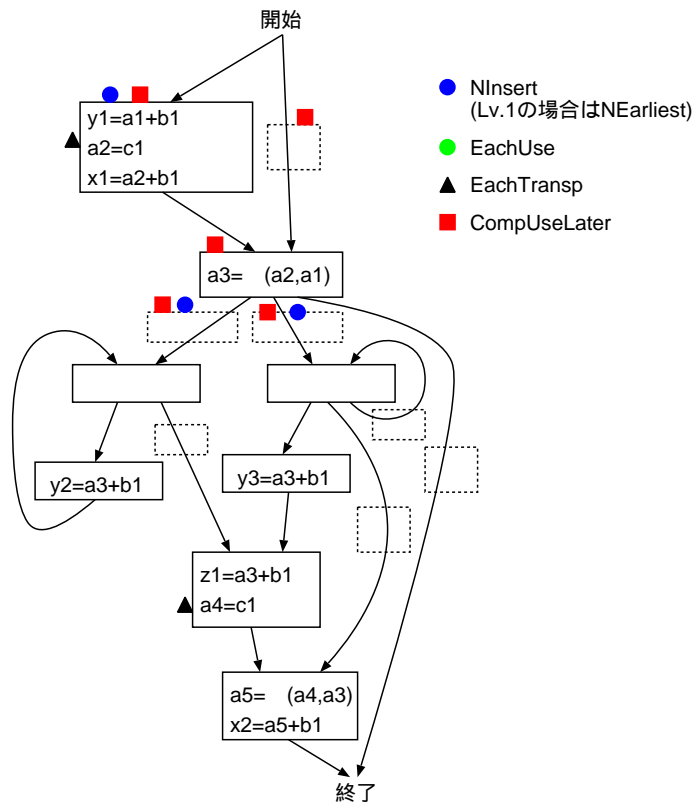


図 6.16: $CompUseLater$ の結果 (a のみ表示)

$CompUseLater$ を利用して、無用な $a(b)$ の ϕ 関数を除去すべきブロック $PhiElimination$ を求める . $PhiElimination$ は次のように定義する .

大域的な性質 定義 6.11

$PhiElimination(B)$: B の $a(b)$ の ϕ 関数を除去すべき .

PhiElimination は次の式で求められる。

データフロー方程式 6.10

$$\text{PhiElimination}(B) = \text{EachPhi}(B) \cap \overline{\text{CompUseLater}(B)}$$

この式の意味は、「 B の ϕ 関数を除去すべきであるのは、 B に $a(b)$ の ϕ 関数が存在し、かつ、その ϕ 関数の後続パスで $a(b)$ の使用がない」である。

このデータフロー方程式は繰り返し計算による解析を必要としない。したがって全ての基本ブロックを1度解析すればよいので、計算のオーダーは $O(N)$ である。この計算は、先に出てきた PrunPhiInsert の計算などと同時に行うことができる。

PhiElimination の計算の結果を図 6.17 に示す。

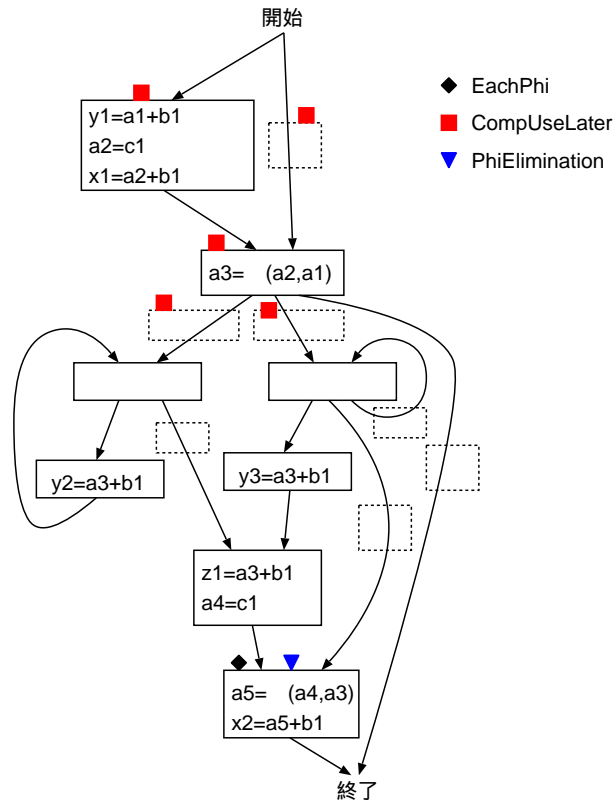


図 6.17: PhiElimination の結果 (a のみ表示)

6.2.8 コード移動 レベル 1

ここまでの解析の結果を用いて計算の巻き上げを行った結果を図 6.18 に示す (空のブロックは除去する)。

6.2.9 変数の生存区間の短縮

図 6.18 の点線がかこったブロック中の 2 つの計算のうち、左側のブロックでの計算はこれ以上遅れさせることはできないが、右側のブロックのものはもっと遅れさせることができる。遅れさせれ

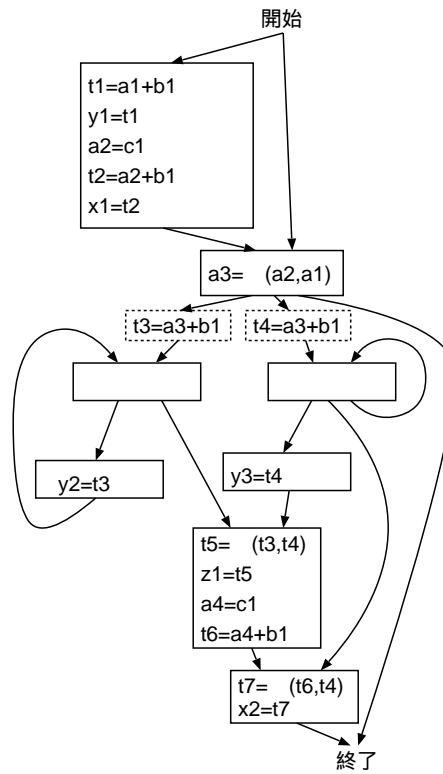


図 6.18: 計算の巻き上げによるコード移動 (レベル 1) の結果

ば, t の生存区間が短くなってレジスタ割当てがうまくいく可能性がある. 次の節でこれについて改善を行う.

まず, 計算を *Earliest* の場所から計算を遅れさせてもよい場所を調べる. *Earliest* は計算をプログラムの意味が変わらない範囲でできるだけ上に引き上げたものである. この計算を遅れさせるときに, その計算がもともとあった場所よりも遅れさせることはプログラムの意味を変えてしまうためできない.

定義 6.1

$\mathcal{N}Delay(B)$: プログラムの入口から B の入口挿入点に達するどのパスにも, $\mathcal{N}Earliest(P)$ または $\mathcal{X}Earliest(P)$ である挿入点があり, かつ, その計算を B の入口挿入点まで遅れさせてもよい.

$\mathcal{X}Delay(B)$: プログラムの入口から B の出口挿入点に達するどのパスにも, $\mathcal{N}Earliest(P)$ または $\mathcal{X}Earliest(P)$ である挿入点があり, かつ, その計算を B の出口挿入点まで遅れさせてもよい.

これらは, 次のデータフロー方程式を解くことによって得られる. その解 (最大解) を $\mathcal{N}Delay$, $\mathcal{X}Delay$ と書くことにする.

データフロー方程式 6.11

$$\mathcal{N}Delay(B) = \mathcal{N}Earliest(B) \cup \begin{cases} false & \text{if } B = \text{開始} \\ \bigcap_{P \in \text{pred}(B)} \{\mathcal{X}Delay(P) \cap \overline{\mathcal{X}Comp(P)}\} & \text{otherwise} \end{cases}$$

$$\mathcal{X}Delay(B) = \mathcal{X}Earliest(B) \cup \{\mathcal{N}Delay(B) \cap \overline{\mathcal{N}Comp(B)}\}$$

この最大解を求めるために必要な繰り返し計算の回数は最悪 $d + 2$ 回であり、計算のオーダーは $\mathcal{O}(N \cdot d)$ となる。

$Delay$ の計算の結果を図 6.19 に示す。

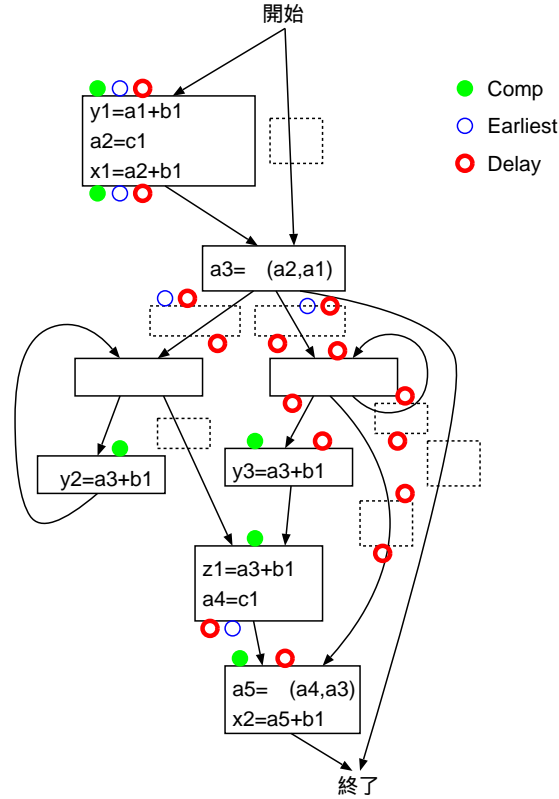


図 6.19: $Delay$ の結果

この結果を使って、 $Earliest$ である場所から遅れさせていって、それ以上遅れさせることのできない場所を求める。

定義 6.2

$\mathcal{N}Latest(B)$: $\mathcal{N}Delay(B)$ であるがこれ以上遅れさせることはできない。

$\mathcal{X}Latest(B)$: $\mathcal{X}Delay(B)$ であるがこれ以上遅れさせることはできない。

計算 (入口計算か出口計算) に遭遇したらそれ以上遅れさせることはできない。また、後続ブロックのどれかが $Delay$ でなかったとしたらそれ以上遅れさせることはできない。このことから、次のような計算をすればよいことがわかる。

データフロー方程式 6.12

$$\mathcal{N}Latest(B) = \mathcal{N}Delay(B) \cap \mathcal{N}Comp(B)$$

$$\mathcal{X}Latest(B) = \mathcal{X}Delay(B) \cap \{ \mathcal{X}Comp(B) \cup \bigcup_{S \in succ(B)} \overline{\mathcal{N}Delay(S)} \}$$

このデータフロー方程式は繰り返し計算による解析を必要としない。したがって全ての基本ブロックを 1 度解析すればよいので、計算のオーダーは $\mathcal{O}(N)$ である。

$Latest$ の計算の結果を図 6.20 に示す。

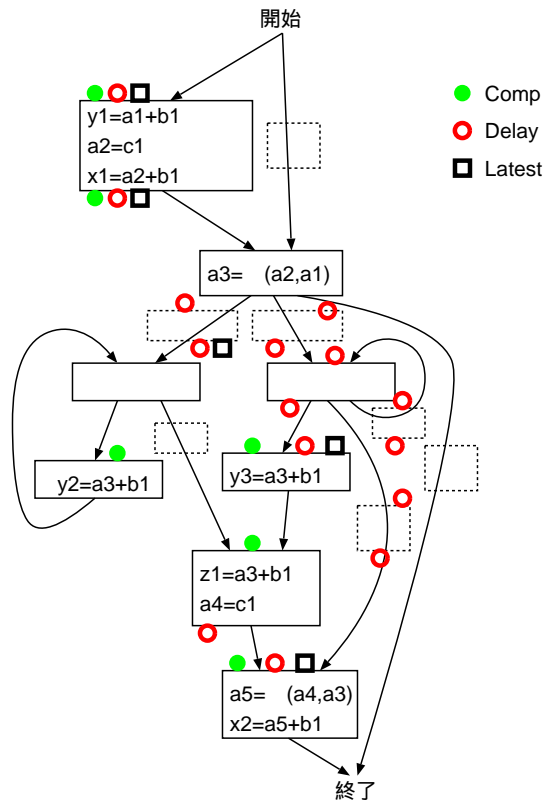


図 6.20: Latest の結果

ここで，前項の計算の巻き上げにおけるコード移動に使われた $\mathcal{N}Earliest(B)$ と $\mathcal{X}Earliest(B)$ の代わりに，上記の $\mathcal{N}Latest(B)$ と $\mathcal{X}Latest(B)$ を使ってコード移動のステップ 1 を行う．例題の図 6.2 について， $Delay$ ， $Latest$ を求めた結果を図 6.20 に，また，

6.2.10 コード移動 レベル 2

$Latest$ まで計算を行うと，この時点でもコード移動を正しく行うことができる．この生存区間を短縮した時点においてコード移動を行う場合は， $\mathcal{N}Latest(B)$ が $true$ である B の入口挿入点と $\mathcal{X}Latest(B)$ が $true$ である B の出口挿入点でコード移動のステップ 1 を行い， $\mathcal{N}Comp(B)$ が $true$ である B の入口計算と $\mathcal{X}Comp(B)$ が $true$ である B の出口計算についてステップ 2 を行えばよい．すなわち，

$$\mathcal{N}Insert(B) = \mathcal{N}Latest(B)$$

$$\mathcal{X}Insert(B) = \mathcal{X}Latest(B)$$

$$\mathcal{N}Replace(B) = \mathcal{N}Comp(B)$$

$$\mathcal{X}Replace(B) = \mathcal{X}Comp(B)$$

としてコード移動を行うのである．これを本手法のレベル 2 のコード移動とする．

レベル 2 のコード移動では，この時点でレベル 1 のコード移動の際に行った添字の解析を行う（図による説明は割愛）．変数の生存区間を短縮したコード移動を行った結果を図 6.21 に示す（空のブロックは除去する）．

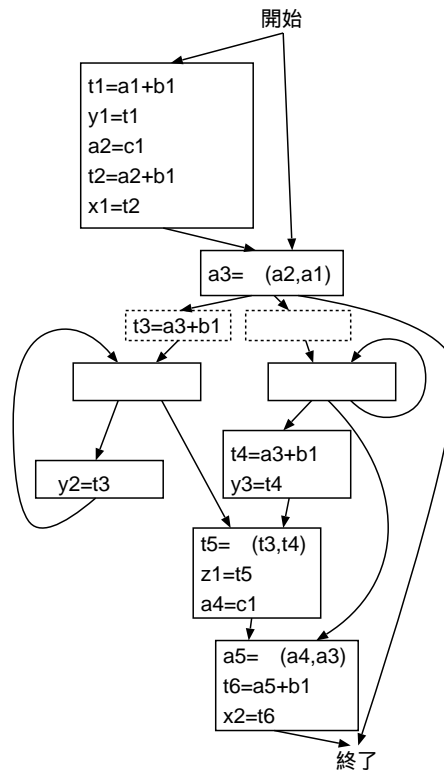


図 6.21: 変数の生存区間を短縮したコード移動 (レベル 2) の結果

6.2.11 無駄なコード移動の除去

図 6.21 には,

$$t = a + b;$$

$$y = t;$$

のようになっていて, t の値が直後に 1 回使われるだけで, その後は使われない場合が, 開始ブロックの直後のブロックに 2ヶ所, 終了ブロックの直前のブロックに 1ヶ所ある. このような場所での t への置き換えは無駄であるから, もとの

$$y = a + b;$$

のままとすることにする.

t がそれ以上使われないのは, t がそこで生きていない場合である. したがって, t が生きているかどうかの計算をすることにする.

大域的な性質 定義 6.12

$\mathcal{N}Live(B)$: B の入口部分の出口で t が生きている.

$\mathcal{X}Live(B)$: B の出口部分の出口で t が生きている.

計算をしてその値を t に代入するのは *Latest* が *true* である場所であり, その値 t を実際に使用するのは, 計算がある *Comp* であるから, 以下のデータフロー方程式が得られる.

データフロー方程式 6.13

$$\mathcal{N}Live(B) = \{\mathcal{X}Comp(B) \cup \mathcal{X}Live(B)\} \cap \overline{\mathcal{X}Latest(B)}$$

$$\mathcal{X}Live(B) = \begin{cases} false & \text{if } B = \text{終了} \\ \bigcup_{S \in succ(B)} [\{\mathcal{N}Comp(S) \cup \mathcal{N}Live(S)\} \cap \overline{\mathcal{N}Latest(S)}] & \text{otherwise} \end{cases}$$

これらの方程式の意味は「 B の入口で生きている変数は、 B で使われる変数か、 B の出口で生きている変数、ただし、出口で定義される場合をのぞく」、 B の出口で生きている変数は、後続ブロックのどれかの、入口で変数が使われているか生きている変数、ただし、そのブロックの入口で定義されている場合をのぞく」である。このデータフロー方程式の解(最小解)を $\mathcal{N}Live(B)$ 、 $\mathcal{X}Live(B)$ とする。

この最小解を求めるために必要な繰り返し計算の回数は最悪 $d + 2$ 回であり、計算のオーダーは $\mathcal{O}(N \cdot d)$ となる。

$Live$ を求めた結果を図 6.22 に示す。

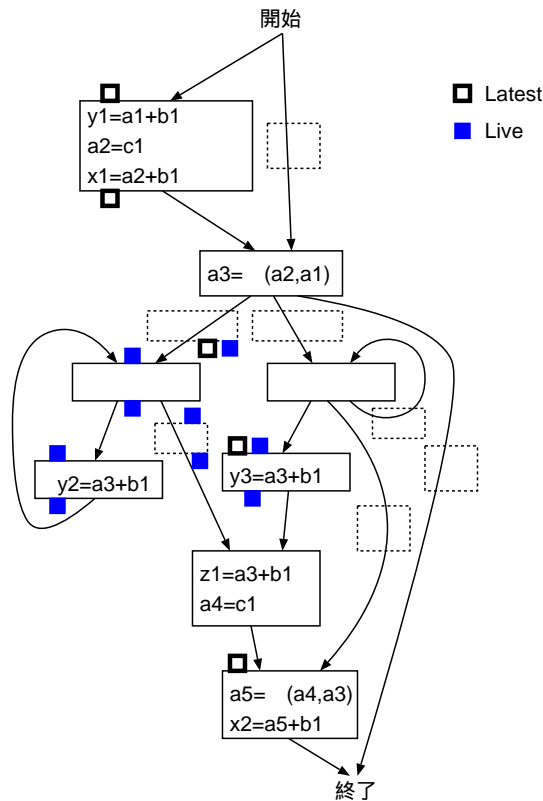


図 6.22: Live の結果

コード移動のステップ 1 で、無駄なコード移動は行わずに必要な計算を挿入すべき場所は、レベル 2 により計算を遅れさせたところのうち、その変数がそこで生きている場所である。

データフロー方程式 6.14

$$\mathcal{N}Insert(B) = \mathcal{N}Latest(B) \cap \mathcal{N}Live(B)$$

$$\mathcal{X}Insert(B) = \mathcal{X}Latest(B) \cap \mathcal{X}Live(B)$$

この式の意味は、「計算を挿入すべき場所は、*Latest* であつ *Live* であるところ」である。
 このデータフロー方程式は繰り返し計算による解析を必要としない。したがって全ての基本ブロックを1度解析すればよいので、計算のオーダーは $O(N)$ である。
Insert を求めた結果を図 6.23 に示す。

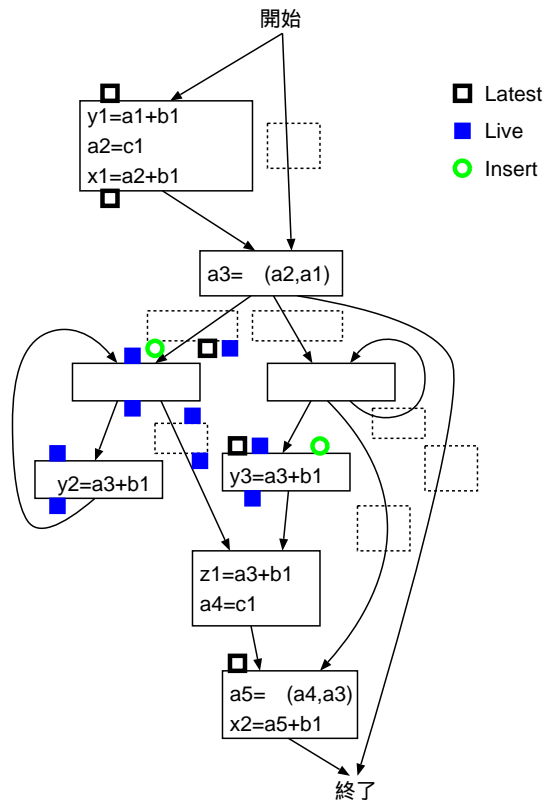


図 6.23: *Insert* の結果

また、ステップ2で、「 $a + b$ 」の計算を t と置換すべき場所は無駄な計算の移動にならない「 $a + b$ 」の計算である。これらを求める式は次のようになる。

データフロー方程式 6.15

$$\begin{aligned} \mathcal{N}Replace(B) &= \mathcal{N}Comp(B) \cap \{\mathcal{N}Insert(B) \cup \overline{\mathcal{N}Latest(B)}\} \\ \mathcal{X}Replace(B) &= \mathcal{X}Comp(B) \cap \{\mathcal{X}Insert(B) \cup \overline{\mathcal{X}Latest(B)}\} \end{aligned}$$

この式の意味は、「計算を置き換えるべき場所は、もとの計算のあったところのうち、計算を挿入する場所が、*Latest* でない場所」である。

このデータフロー方程式も繰り返し計算による解析を必要としない。したがって全ての基本ブロックを1度解析すればよいので、計算のオーダーは $O(N)$ である。この計算は、*Insert* と同時に行うことができる。

Replace を求めた結果を図 6.24 に示す。

6.2.12 コード移動 レベル3

これらの *Insert* および *Replace* を使用してコード移動のステップ1から4を行うコード移動は、本手法において最適化効果のもっとも高いコード移動である。これを本手法のレベル3のコード移動とする。

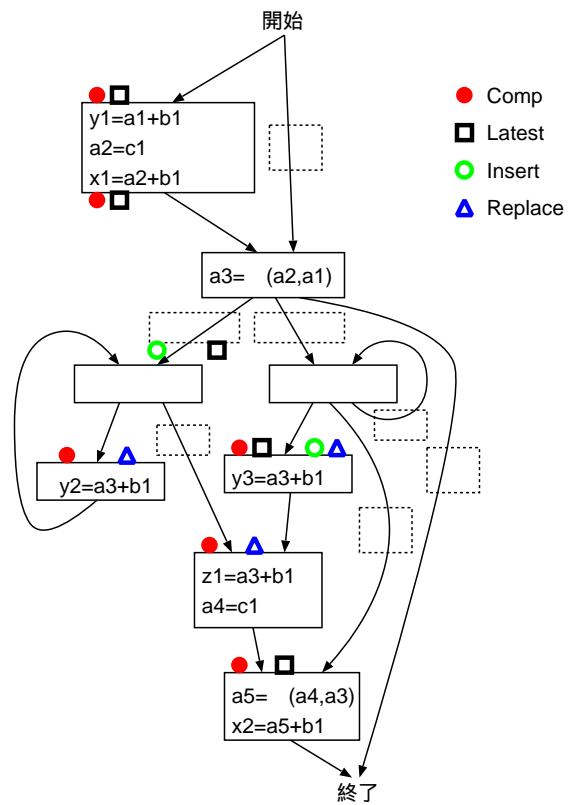


図 6.24: *Replace* の結果

レベル3のコード移動でも、この時点においてレベル1のコード移動の際に行った添字の解析を行う(図による説明は割愛)。

その結果を図 6.25 に示す(空のブロックは除去する)。

このレベル3のコード移動により、変数の生存区間は最短であり、かつ、無駄なコード移動も行わない部分冗長性除去ができる。

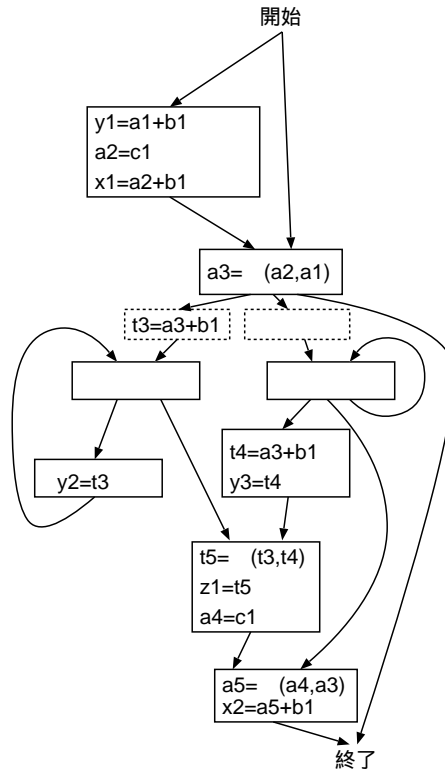


図 6.25: 無駄な移動を行わないコード移動 (レベル3) の結果

第7章 評価

本章では、本手法の最適化にかかるコンパイル時間（以後、処理時間とよぶ）、および最適化効果、すなわち本手法の目的コードの実行時間（以後、実行時間とよぶ）について評価する。

実験には COINS[文部科学省] の SSA 形式最適化コンパイラ用モジュールを用い、SUNW, Sun Blade 1000 (表 7.1) で処理時間と実行時間を測定した。

現在 COINS は開発段階であり、コンパイラ全体を通して十分なテストプログラムが正常にコンパイルできないため、正常に動作が確認されているテストプログラムのみを用いた。評価は次の最適化の比較で行った。

- 最適化 0 OPT_0 : SSA 変換 → SSA 逆変換
- 最適化 1 OPT_1 : SSA 変換
→ ループ不変式のループ外移動 → 共通部分式の除去
→ SSA 逆変換
- 本手法 1 NEW_1 : SSA 変換
→ 危険辺の除去 → 本手法による部分冗長性除去
→ SSA 逆変換
- 最適化 2 OPT_2 : SSA 変換
→ ループ不変式のループ外移動 → 定数伝播
→ 共通部分式の除去 → 無効コードの除去
→ SSA 逆変換 → 合併
- 本手法 2 NEW_2 : SSA 変換
→ 危険辺の除去 → 本手法による部分冗長性除去
→ 定数伝播 → 無効コードの除去
→ SSA 逆変換 → 合併

SSA 変換は刈り込んだ SSA 形式¹に変換するものを、SSA 逆変換は Sreedhar らのアルゴリズム [Sreedhar et al. 1999] を用いた。合併には Chaitin のアルゴリズム [Chaitin 1982] を用いた。

表 7.1: Sun Blade 1000 の主な仕様

アーキテクチャ	Superscalar SPARC TM Version 9
プロセッサ種別	750MHz UltraSPARC III
プロセッサ数	2
1 次キャッシュ	64KB データ, 32KB インストラクション
2 次キャッシュ	8MB 外部キャッシュ
メモリ容量	1GByte
オペレーティング環境	SunOS 5.8

¹第 3 章 3.6 節参照。

表 7.2: SPECint 2000 181.mcf の処理時間 (msec)

フェーズ	最適化 0 OPT ₀ ^{mcf}	最適化 1 OPT ₁ ^{mcf}	本手法 1 NEW ₁ ^{mcf}	最適化 2 OPT ₂ ^{mcf}	本手法 2 NEW ₂ ^{mcf}
SSA 変換	614	678	707	709	733
本手法による部分冗長性除去 <i>PRE</i>	-	-	508	-	527
ループ不変式のループ外移動 <i>HLI</i>	-	174	-	186	-
共通部分式の除去 <i>CSE</i>	-	140	-	148	-
SSA 逆変換	570	1103	868	1138	1122
SSA 最適化部の全体	1423	2551	2675	3443	3556

ループ不変式のループ外移動と共通部分式の除去を行ったものが最適化 1 であり、その 2 つの最適化の代わりに本手法を用いたものが本手法 1 である。また、最適化 2 と本手法 2 では、現 COINS で単独で正常に動作するその他の最適化を、最適化 1 と本手法 1 にそれぞれ組み合わせて、最適化効果が高いと思われる順番に並べたものである (現段階の COINS では、単独では正常に動く最適化でも、組み合わせる順番によっては正常に動かないものもある。本手法においては、きれいな SSA 形式を出力するためそのような制約はない)。

7.1 処理時間の計測

本手法は複数種類の単方向のデータフロー解析を行うものである。その処理にかかる計算のオーダーは、基本ブロックの数 N 、制御フローグラフの区間の深さを d とすると $O(N \cdot d)$ である²。

実際のプログラムで評価するために、ベンチマークとして SPEC の CINT2000 から 181.mcf³を使用した際の処理時間を表 7.2、および図 7.1 (左) に示す。各フェーズごとの処理時間については、そのフェーズに制御が移り入る直前と制御が抜け出た直後で時間を測り、その差をもってそのフェーズの処理時間とした。また、同様に SSA 最適化部の全体の時間についても、SSA 最適化部に制御が移り入る直前と制御が抜け出た直後で時間を測り、その差をもって SSA 最適化部の全体の時間とした。SSA 最適化部の全体の時間は、SSA 最適化を行うことによって起こる処理をすべて行った時間である。すなわち、SSA 最適化部の全体の時間には、SSA 変換や SSA 逆変換、SSA 変換に各最適化の処理や、SSA 最適化部に必要なその他の処理 (たとえば、前処理にあたるループ解析やループ構造の変形など) にかかる時間も含まれる。また、本手法による部分冗長性除去の処理時間には、危険辺の除去にかかる処理時間も含めた。

また、小さいテストプログラムとして次のプログラムを使用した。

- 13 女王問題
- バブルソート
- 挿入ソート
- 安定な結婚の問題

²第 2 章 2.2.2 節参照。

³COINS で、現段階で正常に動作する SPEC ベンチマークは 181.mcf のみである。

表 7.3: 6 つの小さいテストプログラムの合計の処理時間 (*msec*)

フェーズ	最適化 0 OPT_0^{tiny}	最適化 1 OPT_1^{tiny}	本手法 1 NEW_1^{tiny}	最適化 2 OPT_2^{tiny}	本手法 2 NEW_2^{tiny}
SSA 変換	454	481	504	489	469
本手法による部分冗長性除去 <i>PRE</i>	-	-	272	-	298
ループ不変式のループ外移動 <i>HLI</i>	-	133	-	154	-
共通部分式の除去 <i>CSE</i>	-	96	-	123	-
SSA 逆変換	191	300	211	281	277
SSA 最適化部の全体	990	1327	1401	1595	1580

- クイックソート
- 選択ソート

これらは小さいプログラムであるため、各プログラムごとに処理時間を測ると誤差の割合が大きくなり正確な評価がしにくい。したがってこれらのプログラムごとの処理時間の合計の処理時間を表 7.3，および 図 7.1 (右) に示す。

7.2 処理速度の考察

SSA 変換にかかる処理時間と SSA 逆変換にかかる処理時間を足し合わせた時間は、多くの最適化を行った 最適化 2 や本手法 2 においても SSA 最適化部の全体にかかる時間の 40%以上を占めている。すなわち、SSA 変換や SSA 逆変換の処理と比べると、各最適化にかかる処理時間の比率は少ないと言える。

本手法では、処理の途中で一部分の変数に関して SSA 形式を求める計算を行うが、SSA 変換のフェーズで用いる繰り返し支配境界や支配木の計算を使用せず単方向のデータフロー解析により SSA 形式を求めている。本手法を処理する時間の全体を見ても SSA 変換にかかる処理時間よりも少ないため、本手法が効率よく SSA 形式を求めているといえる。

7.2.1 通常形式による部分冗長性除去との比較

この節では、通常形式による部分冗長性除去の COINS 上での実装が現在、伊藤 [伊藤 2004] により行われているが、未完成であるため考察のみを述べる。

最適化では一般に、同じ最適化やいくつかの異なる種類の最適化を繰り返し処理する。COINS 上で通常形式による部分冗長性除去を用いる方法には、次の 2 つが考えられる。

方法 1 SSA 最適化を処理した後で通常形式に戻し、通常形式上の部分冗長性除去を行う (図 7.2 左)。

方法 2 SSA 最適化を処理する中で、いったん通常形式に戻して通常形式上の部分冗長性除去をかけ、再度 SSA 形式に直して繰り返し最適化を行う (図 7.2 右)。

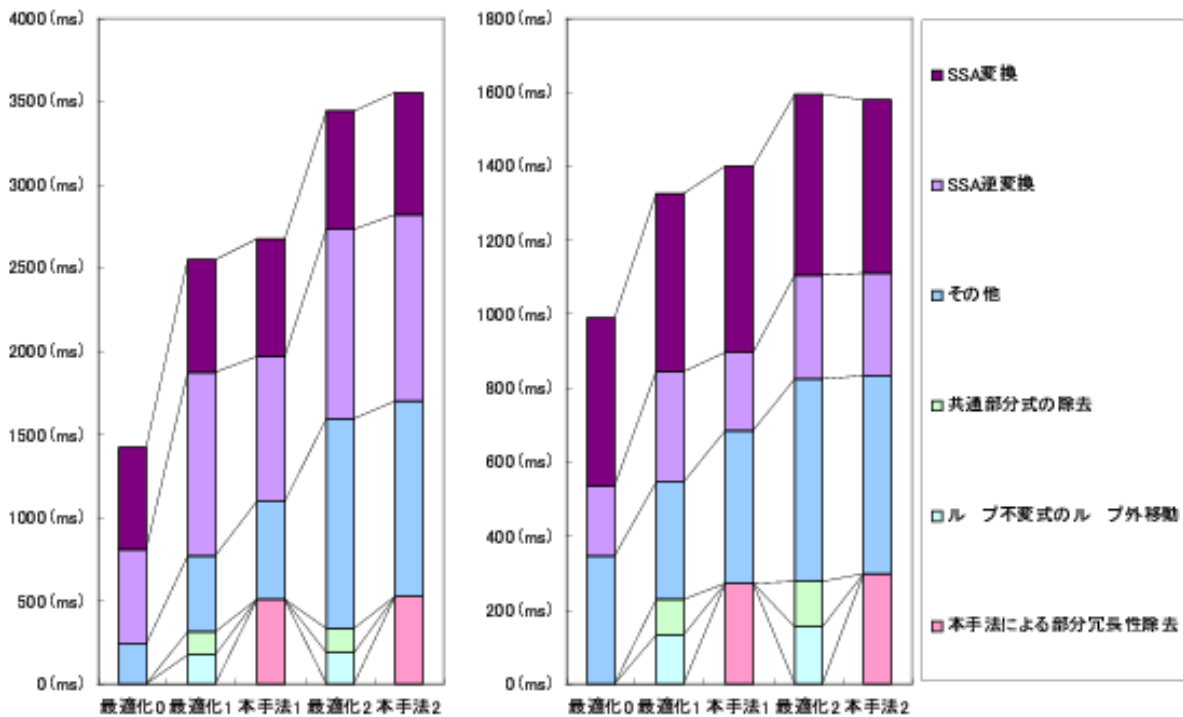


図 7.1: 処理時間 (表 7.2 , 表 7.3 より)

方法 1 では SSA 変換および SSA 逆変換はそれぞれ 1 度ずつ行えばよいが、方法 2 では通常形式上の部分冗長性除去を行うたびに、SSA 変換と SSA 逆変換を行わなければならない。一方、方法 2 では SSA 最適化の中の好きな場所で部分冗長性除去を行うことができるため、効果のある最適化の組み合わせを選ぶことができるが、方法 1 ではできない。すなわち、方法 2 は方法 1 と比べると処理時間は多くかかるが、最適化効果は高いと言える。

方法 1 については、現在 COINS 上で通常形式上の部分冗長性除去を実装中である [伊藤 2004] が完成され次第、これを利用して処理時間を計測できる。ただし、[伊藤 2004] の実装では、実装しやすい部分冗長性除去のアルゴリズム [Paleri et al. 1998] が用いられている。このアルゴリズムは単純さを目的に考案されたものであり、危険辺の除去を行わなくてもよかったり、単方向のデータフロー方程式を解析する数が少なかったりといった特徴がある。本手法のアルゴリズムと比較すると、レジスタの生存区間などが本手法よりも長くなることなどから、最適化効果はやや劣ると考えられる。

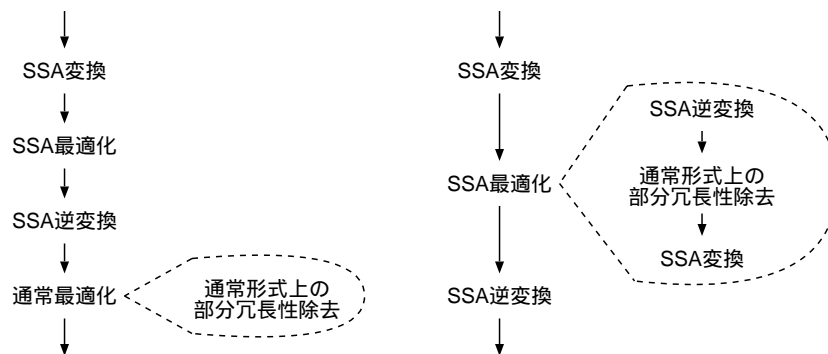


図 7.2: 最適化の方法

処理にかかる時間については、本手法が制御フローグラフ上で基本ブロック単位の解析をしているのに対し、[Paleri et al. 1998] のアルゴリズムは文 1 つ 1 つを制御フローグラフのノードとし解析をしている。このため処理にかかる計算のオーダーは、プログラムの文の数を n とすると $O(n \cdot d)$ であり、本手法の $O(N \cdot d)$ と比較するとやや劣る。また彼らのアルゴリズムはベクトル方式のものであるが、[伊藤 2004] の実装ではベクトル方式では実装されていない。

また、本手法と同等な最適化効果を持つ通常形式上の部分冗長性除去のアルゴリズム [Knoop et al. 1992; Knoop et al. 1994] の実装、および処理時間と性能の評価は、[立川 2002; 立川他 2002] で行っているが、これは COINS 上での実装ではなく、属性文法コンパイラ上での定式化を行ったものである。また、その実装も式 1 つ 1 つを制御フローグラフのノードとし解析をしている。以上の理由により、本手法と方法 1 との処理時間と性能を単純に比較することは現段階ではできない。

方法 2 については、表 7.2 と 7.3、および図 7.1 の結果を見ると、何も SSA 最適化をかけない最適化 0 の SSA 変換および SSA 逆変換の処理時間でさえ本手法による部分冗長性除去の処理時間より長いことより、本手法よりも処理時間は多くかかると言える。一方、通常形式上の部分冗長性除去は本手法の最適化効果と同等であり、方法 2 では複数の最適化を繰り返し処理する中の好きな場所で処理できるため、方法 2 は本手法と同等の最適化効果が得られるはずである。しかし、実際には SSA 逆変換をするたびに大量のコピー文が発生してしまい、実行速度は遅くなってしまうと予想される [小濱 2004]。

以上をまとめると、最適化効果に関しては

$$\text{方法 1} \leq \text{方法 2} \leq \text{本手法}$$

が成り立ち、処理時間に関しては、

$$\text{方法 2} \geq \text{本手法}$$

$$\text{方法 2} \geq \text{方法 1}$$

が成り立つ。方法 1 と本手法の処理時間の違いは実験を行っていないが、本手法が通常形式上の部分冗長性除去よりも多くの解析をしていることより、方法 1 よりも多くの処理時間を要すると予想される。

方法 1 と比べて本手法がどれだけの最適化効果を出すことができるのかについても実験してみたが、COINS 上で [Knoop et al. 1992; Knoop et al. 1994] の実装を行っていないことや、現段階の COINS では正常に動作する最適化の組み合わせが限られているために、効果の高い最適化の組み合わせで実験ができないこと、などの理由で実験が現段階ではできない。

しかし、計算のオーダーは通常形式上の部分冗長性除去と同等であり、実際の実験でも本手法にかかる処理時間が SSA 最適化部の全体の処理時間に対して少ないので、処理時間の差は問題にはならない。よって、コンパイラに部分冗長性除去を採用するときには最適化効果の高い方を選んだ方がよい。方法 1 よりも方法 2 よりも最適化効果の高い本手法を採用した方がよいといえる。すなわち、通常形式上の部分冗長性除去よりも本手法の方を採用した方がよいといえる。

7.2.2 他の SSA 形式上の部分冗長性除去との比較

現段階で、COINS 上に本手法以外の SSA 形式上の部分冗長性除去は実装されていないので、処理時間の実測による比較は不可能である。しかし、計算量やアルゴリズムなどの比較については可能であり、第 8 章に記述した。

表 7.4: 処理時間の割合 (%)

フェーズ	$\frac{NEW_1^{mcf}}{OPT_1^{mcf}}$	$\frac{NEW_1^{tiny}}{OPT_1^{tiny}}$	$\frac{NEW_2^{mcf}}{OPT_2^{mcf}}$	$\frac{NEW_2^{tiny}}{OPT_2^{tiny}}$
SSA 変換	110.4	104.7	103.3	95.9
$\frac{PRE}{HLI + CSE}$	161.7	118.7	157.7	107.5
SSA 逆変換	78.6	70.3	98.5	98.5
SSA 最適化部の全体	104.8	105.5	103.2	99.0

7.2.3 COINS 上の他の最適化との比較

表 7.2 と 表 7.3 に示した共通部分式の除去とループ不変コードのループ外移動の両者 (最適化 1 および最適化 2) を行った処理時間を基準にした本手法 (本手法 1 および本手法 2) の処理時間の割合を表 7.4 , および図 7.3 に示す .

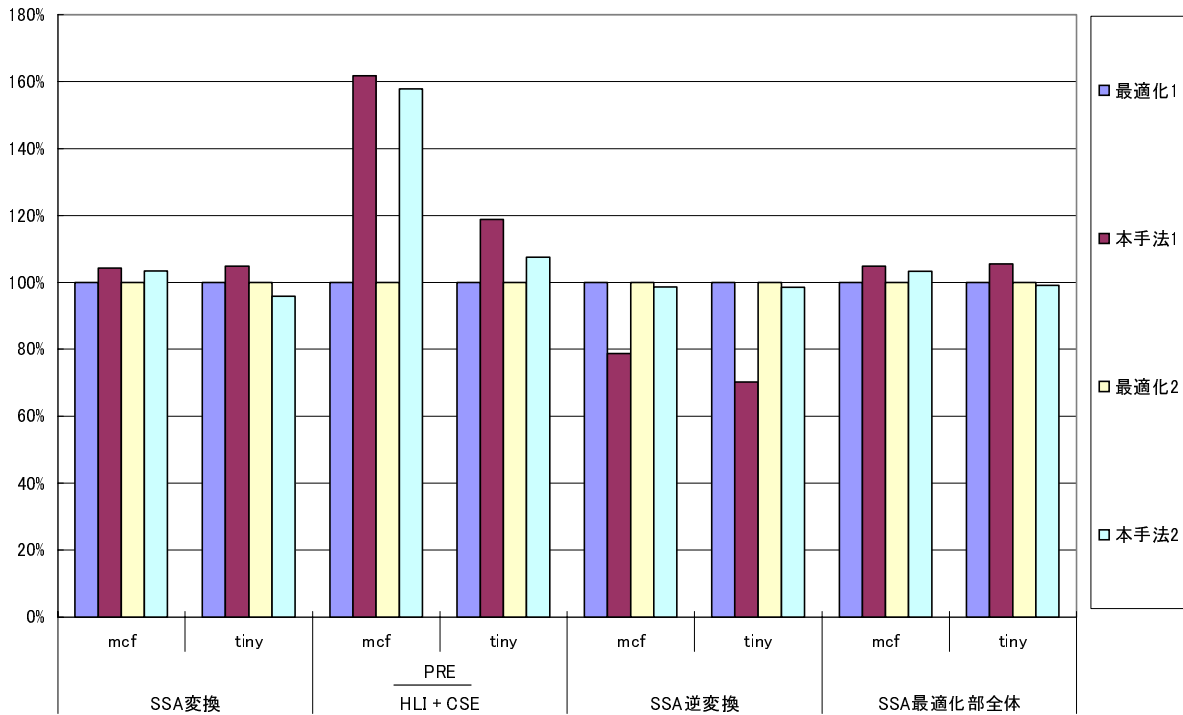


図 7.3: 処理時間の割合

SSA 最適化部の全体の時間の結果では, 本手法にかかる処理時間は共通部分式の除去とループ不変コードのループ外移動の両者を行った場合の処理時間より, 数%遅くなっている. 各フェーズごとの処理時間を見てみると, SSA 逆変換では最適化 1 と比べると, 本手法 1 の場合は 20%以上早くなっている. これは, ループ不変コードのループ外移動の際に SSA 形式を複雑なものにしているためだと思われる. SSA 形式が複雑であるということの意味は, 同一 ϕ 関数内の変数の生存区間が重なっている状態を指す. SSA 変換をした直後では, それらの生存区間は重なることはないが, 下手な最適化の中にはこれらの生存区間を重ねてしまうものもあるようである. 生存区間が重なっている SSA

表 7.5: 実行時間 (sec)

プログラム	gcc -O0 GCC ₀	最適化 0 OPT ₀	最適化 1 OPT ₁	本手法 1 NEW ₁	最適化 2 OPT ₂	本手法 2 NEW ₂	gcc -O2 GCC ₂
181.mcf	561.23	543.41	489.09	458.20	463.31	462.10	456.32
13 女王問題	8.49	6.60	6.69	6.53	6.68	6.64	6.42
バブルソート	6.99	2.22	2.07	2.04	2.06	2.00	1.49
挿入ソート	5.97	1.61	1.61	1.59	1.61	1.61	0.73
安定な結婚の問題	6.15	6.65	6.55	6.35	6.54	6.46	4.87
クイックソート	5.23	1.76	1.80	1.69	1.79	1.77	1.19
選択ソート	6.83	2.83	3.21	2.45	2.83	2.45	2.43

形式の SSA 逆変換は、処理に多くの時間がかかる上、目的コードの実行速度も遅い。本手法は、処理後の SSA 形式として生存区間が重ならない、きれいな SSA 形式を出力するため、このような結果が得られたと思われる。また、本手法 2 の場合の SSA 逆変換の処理時間が最適化 2 と僅差であるのは、本手法の後の最適化フェーズによって SSA 形式が複雑なものになったためであると思われる。

ループ不変コードのループ外移動にかかる処理時間と共通部分式の除去にかかる処理時間を単純に足し合わせたものに比べると、本手法にかかる処理時間は 150%ほどかかっているものもあるが、SSA 最適化部の全体の時間を見ると、その差が 3 から 4%ぐらいに少なくなっている。これは、本手法のアルゴリズムがきれいな SSA 形式を出力することにより、後の処理が速くなることや、ループ不変コードのループ外移動と共通部分式の除去が、SSA 形式や制御フローグラフを複雑にすることなどが考えられる。

7.3 実行時間の計測

本手法の最適化効果は、通常形式上の部分冗長性除去のアルゴリズムである怠けたコード移動 [Knoop et al. 1994] と同等に強力であるため、この節での実行時間の計測は確認のために行った。

効果を確認するための実験用ベンチマークとして、SPEC の CINT2000 から 181.mcf と、前節でも使用した 6 つの小さいテストプログラムを使用しそれぞれの最適化を行った。それらの出力された目的コードの実行時間を表 7.5 に示す。

7.4 実行速度の考察

表 7.5 を基に、gcc -O0 でコンパイルしたものを基準とした最適化の割合を表 7.6 および図 7.4 に示す。

本手法 1 と最適化 1 を比較すると、本手法 1 の方が最適化効果が現れている。これは、部分冗長性除去が共通部分式の除去とループ不変コードのループ外移動の効果を包括していることから、当然と言える。COINS を使用した全ての最適化で、gcc -O2 には及んでいない。

気になる点としては、本手法 1 と本手法 2 を比較すると、本手法 2 が本手法 1 に加えて様々な最適化を行っているにもかかわらず本手法 1 よりも劣るものが多いことである。また、最適化 0 と最適化 1 を比較したときにも、最適化 1 の方が劣っているものがあるということである。これら原因として、本手法以外の SSA 最適化の処理中に、同一 ϕ 関数内の変数の生存区間が重なってしまったことが考えられる。[小濱 2004] によると、同一 ϕ 関数内の変数の生存区間が重なると SSA 逆変換を

表 7.6: 実行時間の割合 (%)

プログラム	$\frac{OPT_0}{GCC_0}$	$\frac{OPT_1}{GCC_0}$	$\frac{NEW_1}{GCC_0}$	$\frac{OPT_2}{GCC_0}$	$\frac{NEW_0}{GCC_0}$	$\frac{GCC_2}{GCC_0}$
181.mcf	96.8	87.1	81.6	82.5	82.3	81.3
13 女王問題	77.7	78.7	76.9	78.6	78.2	75.6
バブルソート	31.7	29.6	29.1	29.4	28.6	21.3
挿入ソート	26.9	26.9	26.6	26.9	26.9	12.2
安定な結婚の問題	108.1	106.5	103.2	106.3	105.0	79.1
クイックソート	33.6	34.4	32.3	34.2	33.8	22.7
選択ソート	41.4	46.9	35.8	41.4	35.8	35.5

するときコピー文が残ってしまい、その後に行われる合併によってもそのコピー文が除去されないとのことである。また、この重複がない場合にはレジスタに値をうまくのせスケジューリングすることができることである。コピー文のコストは小さいものであるが、小さいテストプログラムのように、ループ内の計算を多く含むような例では、コストが大きくなってしまっているようである。

本手法のみを適用した本手法 1 の場合は、本手法が同一 ϕ 関数内の変数の生存区間が重ならないように処理しているため、本手法 2 よりも SSA 最適化本来の効果がよく現れたのだと思われる。

本実験以外でも、COINS の目的コードの実行速度実験は行われているが、一般に多くの最適化をかければかけるほどその実行速度は向上するものであるのに、多くの最適化をかければかけるほど逆に悪化していくことが確認されている。これは、下手な最適化が SSA 形式のきたなくしてしまっているのが原因だと思われる。きたない SSA 形式とは、同一 ϕ 関数内の変数の生存区間が重なったり、無駄な ϕ 関数が多く存在しているなどの状態を指す。最適化効果が薄れてくるまで最適化を続けると、下手な最適化により、あとは一方的に悪化していくことが原因だと考えられる。最適化効果が薄れてくるまで最適化をかけなかったとしても、本来の最適化の効果が相殺されてしまうので、きたない SSA 形式を出力するような最適化は望ましくない。

また、6 つの小さいプログラムの中に最適化効果が悪化しているものがある。これは、Ultra Spack III のようなスーパスカラマシンでは、パイプライン、キャッシュライン機能などの影響で、小さなプログラムではこのような逆転現象がときに起こることが知られている。この問題も、きれいな SSA 形式を扱うことで軽減されるようである [小濱 2004]。

7.5 本手法のコード移動の比較

本手法では、

- レベル 1 計算の巻き上げ
- レベル 2 変数の生存区間の短縮
- レベル 3 無駄なコード移動の除去

を段階的に行う。レベル 1 は、計算をなるべく先行パスに移動したものである。レベル 1 からレベル 2 では、レジスタの生存区間を短くすることでレジスタ圧力を軽減している。レベル 2 からレベル 3 では、無駄なコード移動を消している。

各レベルに必要な繰り返し計算を要する単方向のデータフロー解析の数を、表 7.7 に示す。

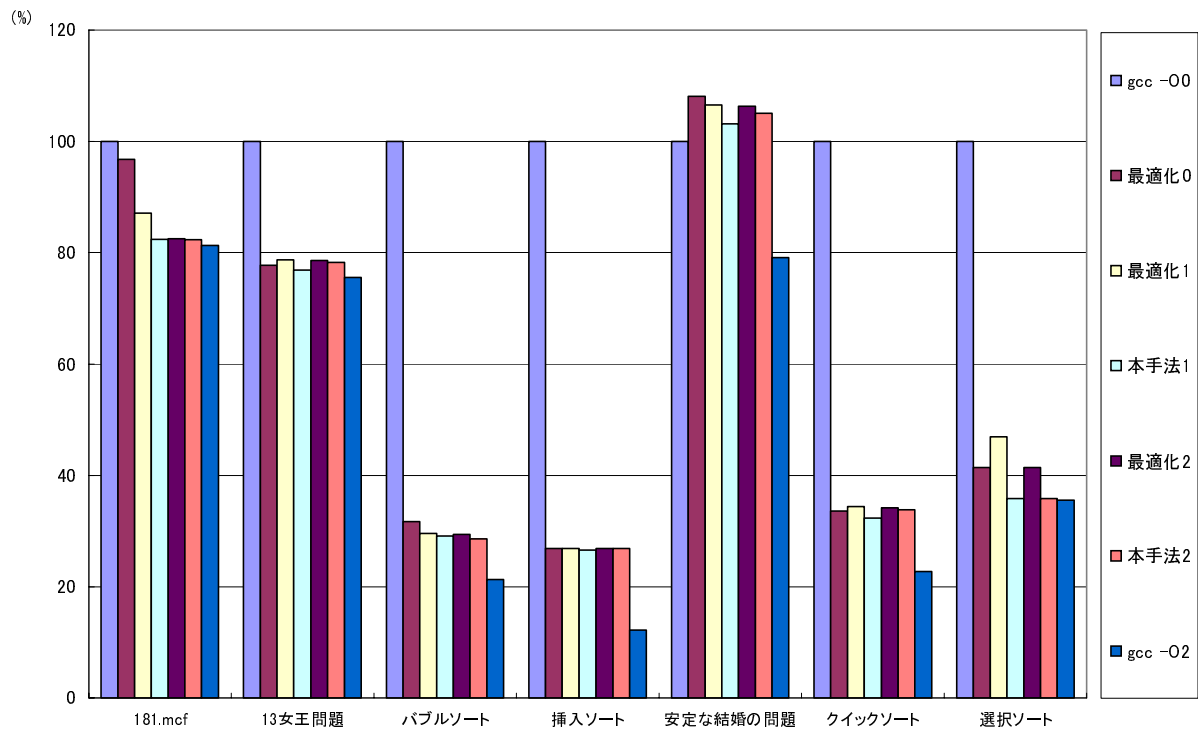


図 7.4: 実行時間の割合 (表 7.6 より)

表 7.7: レベルごとの解析の数

解析	Lv.1	Lv.2	Lv.3	計算のオーダー
繰り返し計算を必要とする解析の数	5	6	7	$\mathcal{O}(N \cdot d)$
繰り返し計算を必要としない解析の数	2	2	2	$\mathcal{O}(N)$

費用的効果に興味あったので、レベルごとの処理時間と効果の関係を測定してみた。181.mcf と小さい6つのテストプログラムのレベルごとの処理時間を、表 7.8 と表 7.9 に、レベル3を基準にした処理時間の割合を表 7.10，図 7.5 に示す。

レベル1では最適化1の効果を含んでいるものの、レジスタ圧力のため最適化1よりも効果が劣るものが多いが、レベル2では最適化1と同等もしくはそれ以上の効果を得ている。レベル2とレベル3の効果の差が3%あるものもあるため、処理時間の規制がなければやはりレベル3を使用するのが望ましいと思われる。

表 7.8: SPECint 2000 181.mcf の処理時間 (msec)

フェーズ	最適化 0	本手法 1		
	OPT ₀	NEW _{1Lv.1} ^{mcf}	NEW _{1Lv.2} ^{mcf}	NEW _{1Lv.3} ^{mcf}
SSA 変換	614	693	712	707
本手法による部分冗長性除去	-	430	459	508
SSA 逆変換	570	853	848	868
SSA 最適化部の全体	1423	2577	2620	2675

表 7.9: 6 つの小さいテストプログラムの合計の処理時間 (msec)

フェーズ	最適化 0	本手法 1		
	OPT ₀	NEW _{1Lv.1} ^{tiny}	NEW _{1Lv.2} ^{tiny}	NEW _{1Lv.3} ^{tiny}
SSA 変換	454	502	512	504
本手法による部分冗長性除去	-	224	258	272
SSA 逆変換	191	214	209	211
SSA 最適化部の全体	990	1353	1369	1401

表 7.10: 処理時間の割合 (%)

フェーズ	$\frac{NEW_{1Lv.1}^{mcf}}{NEW_{1Lv.3}^{mcf}}$	$\frac{NEW_{1Lv.1}^{tiny}}{NEW_{1Lv.3}^{tiny}}$	$\frac{NEW_{1Lv.2}^{mcf}}{NEW_{1Lv.3}^{mcf}}$	$\frac{NEW_{1Lv.2}^{tiny}}{NEW_{1Lv.3}^{tiny}}$
	SSA 変換	98.0	99.6	100.7
本手法による部分冗長性除去	84.6	82.3	90.3	94.8
SSA 逆変換	98.2	102.3	97.6	99.0
SSA 最適化部の全体	96.3	96.5	97.9	97.7

表 7.11: 実行時間 (sec)

プログラム	最適化 1	本手法 1		
	OPT ₁	NEW _{1Lv.1}	NEW _{1Lv.2}	NEW _{1Lv.3}
181.mcf	489.09	465.46	462.34	458.20
13 女王問題	6.69	6.78	6.69	6.53
バブルソート	2.07	2.11	2.07	2.04
挿入ソート	1.61	1.61	1.60	1.59
安定な結婚の問題	6.55	6.64	6.54	6.35
クイックソート	1.80	1.75	1.69	1.69
選択ソート	3.21	2.46	2.45	2.45

表 7.12: 実行時間の割合 (%)

プログラム	$\frac{NEW_1^{Lv.1}}{OPT_1}$	$\frac{NEW_1^{Lv.2}}{OPT_1}$	$\frac{NEW_1^{Lv.3}}{OPT_1}$
181.mcf	95.1	94.5	93.6
13女王問題	101.3	100.0	97.6
バブルソート	101.9	100.0	98.5
挿入ソート	100.0	99.3	98.7
安定な結婚の問題	101.3	99.8	96.9
クイックソート	97.2	93.8	93.8
選択ソート	76.6	76.3	76.3

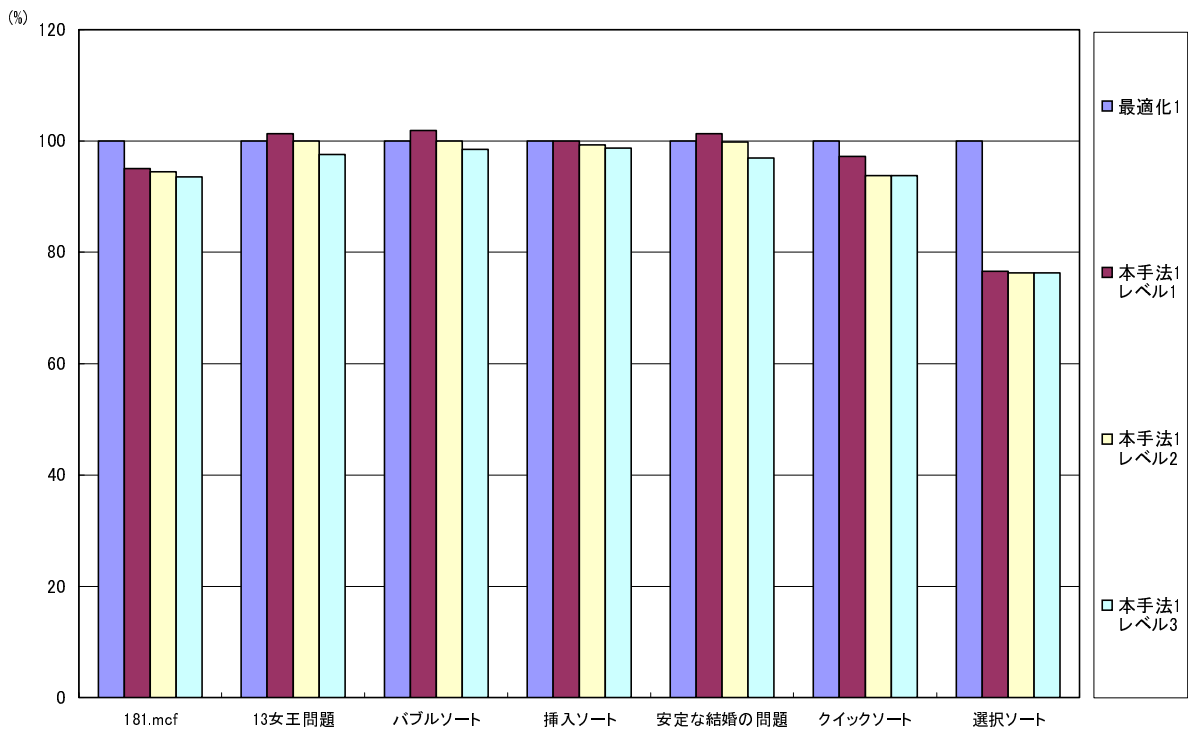


図 7.5: 実行時間の割合 (表 7.12 より)

第8章 関連研究

ここでは、本研究の関連研究について述べる。

8.1 Kennedy らの方法

[Kennedy et al. 1999; Chow et al. 1997] は、現在 SSA 形式を入力形式とした部分冗長性除去の最新の結果となっている。これは、[Intel] のコンパイラにも採用されており、現在、主流のものとなっている。彼らのアルゴリズムと同じ土俵で比較できるように、本研究ではアルゴリズムを考案した。

彼らの論文に掲載されている実験データは、処理速度に関するもののみであり、目的コードの実行速度については掲載されていない。アルゴリズムから判断すると、意味等価な式は扱わず構文等価な式のみを扱っている点や、レジスタの生存区間を考慮している点、無駄な計算の移動を行わない点や、無駄な ϕ 関数を取り除いている点から、本手法の最適化効果と同等の効果が見られると考えられる。

彼らは、本手法のようなベクトル方式ではなく、離散的アルゴリズムを採用している。ベクトル方式は全部の式に対する解析を一度に並列に行なうことができるが、離散アルゴリズムでは式ごとに処理を繰り返す必要がある。

彼らの方法は、除去の対象となる部分冗長な式を1つとりあげ、その式の値を格納する架空の変数を設ける。そして、それを含んだ SSA 形式に対して部分冗長性除去を行うものである。式1つ1つに対してそれぞれ異なった括った冗長グラフ (*factored redundancy graph*, *FRG*) というグラフを作成し、そのグラフ上で部分冗長性除去の処理を行うため、ベクトル方式のように、一度に並列に処理を行うことはできない。その処理にかかる計算量は、計算式の数を C 、括った冗長グラフのノードの数を n 、辺の数を e とすると、 $O(C \cdot (n + e))$ となる。

彼らは、離散的アルゴリズムを用いた彼らの方法と、ベクトル方式である部分冗長性除去の当初のアルゴリズム [Morel and Renvoise 1979; Chow 1983] を、Silicon Graphics の MIPSpro コンパイラに組み込んで実験を行っている。両者を SGI の 195MHz の R10000 上で SPEC95 で比べてみると、その処理速度には有意差がなかった。

しかし、この実験で使われたベクトル方式の部分冗長性除去のアルゴリズムは、双方向のデータフロー解析を複数含むものであり、その計算量は、基本ブロックの数を N 、制御フローグラフの区間の深さを d とすると $O(N^2 \cdot d)$ である。現在は、単方向のデータフロー解析を行う部分冗長性除去のアルゴリズムが提案されており、こちらの方が格段に処理時間は速い。したがって彼らの比較実験はフェアではない。彼らがこのアンフェアなアルゴリズムを採用した理由は、そのアルゴリズムが単に彼らによって過去に研究されたものであったということのみだと予想される。

ここで本手法の処理速度と彼らの実験に使用されたアルゴリズムの処理速度について比較する。どちらもベクトル方式で処理するものであるが、本手法は単方向のデータフロー解析を行うものである。その計算のオーダーは $O(N \cdot d)$ であり、彼らが実験に使ったアルゴリズムと比べて処理にかかる計算量は格段に少なくてすむ。すなわち、彼らの SSA 形式上の部分冗長除去の処理が、彼らが比較したアルゴリズムと有意差ないのであれば、本手法における処理速度の方が圧倒的に速いと予想される。

彼らは目的コードの実行速度についての評価は行わなかったため、アルゴリズムから本手法の最適

化効果と同等だと判断したが、厳密に述べると、彼らの方法が離散的アルゴリズムであるのに対し、本手法はベクトル方式であるため、対象とする部分式を処理する順番などによって効果は異なってくる事が予想される。しかし、本手法でも対象とする式を1つずつ選択していきながら、繰り返し処理を行えば、同等の効果が得られる。

また、彼ら自身によってこのアルゴリズムによる部分冗長性除去と同時に、演算の強さの軽減を行うアルゴリズム [Kennedy et al. 1998] も考えられている。

8.2 Briggsらの方法

[Briggs and Cooper 1994] の方法は、字面が同じ式どうしの中でそれらのオペランドへの代入が存在しないという関係(構文等価)を満たすものだけでなく、構文上は異なっても同じ値を計算する式(意味等価)も扱えるように事前に SSA 形式上で解析を行い、その上で部分冗長性除去を行っている。また、部分冗長性除去にはどの部分式を対象とするかによって、その効果が変わってしまう場合があるが、この問題について事前に解析を行い、部分冗長性除去の効果の高い部分式を選択して処理を行っている。以下にその例を示す。

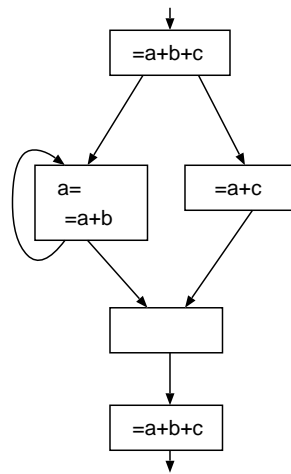


図 8.1: 部分冗長性除去する部分式を選択ができる例

たとえば、図 8.1 では部分冗長性除去の対象となる部分式としては、「 $a+b$ 」と「 $a+c$ 」の2つが存在する。図 8.1 ではループ内に二項演算が1回、右側を通してプログラムの入口から出口へ抜けるパス上に二項演算が5回存在する。これを「 $a+b$ 」と「 $a+c$ 」を対象にそれぞれ部分冗長性除去した結果が図 8.2 (左) と (右) である。

図 8.2 の (左) と (右) を比較すると、ループ内の二項演算はそれぞれ1回ずつ存在する。これはループ不変式のループ外移動を行えない例である。しかし右側のパス上の二項演算は、図 8.2 (左) では4回と、図 8.1 と比べて1回しか減っていないのに対し、図 8.2 (右) では3回と、図 8.1 と比べて2回減っている。

彼らの処理の方法は、部分冗長性除去の処理を行う前に位(rank)というものを計算が存在するループのネストの深さなどによって求め、除去効果の高い部分式を選択してからその部分式に対して部分冗長性除去を行っている。この方法による部分式を選択は最適ではないが、何も考えずに処理するものに比べるとある程度の効果が期待できる。

ただし彼らの処理では、SSA 形式を保持したまま部分冗長性除去を行うことはできず、処理中に通常形式に戻ってしまう。

一般に、SSA 形式を対象に最適化を行うコンパイラでは、SSA 形式のまま複数の最適化を繰り返

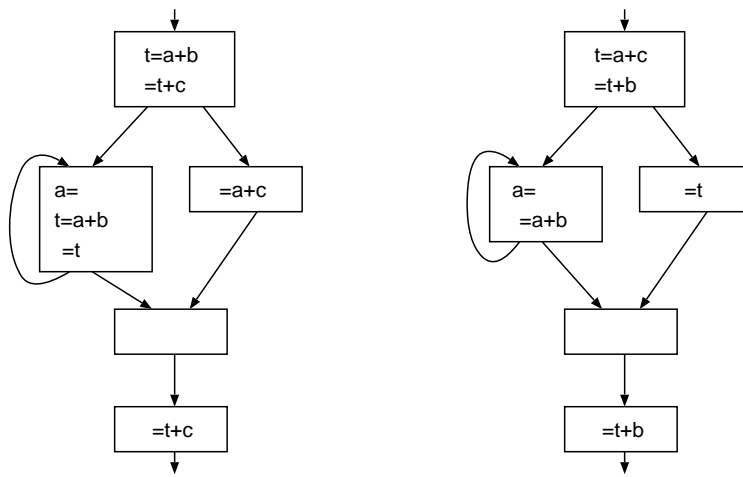


図 8.2: 図 8.1 を部分冗長性除去した 2 つの結果

し処理させる方式が望ましく、ある最適化で通常形式に戻ってしまうと、そのつど SSA 形式に SSA 変換しなくてはならない。本実験¹ で確認したとおり、そのつど SSA 変換を行うことは処理に時間がかかる上、SSA 逆変換によるコピー文が増え最適化効果も悪い。SSA 形式のまま部分冗長性除去を処理できないことは非効率的であるといえる。

しかし、彼らの事前の解析と同様なものを本手法に適用すれば、構文等価な式だけでなく意味等価な式も扱えるようになる上、最適化効果の高い式を選択できるようになり、さらなる効果が期待できる。

8.3 滝本らの方法

[滝本・原田 1997] でも、構文等価だけでなく意味等価も扱った SSA 形式上の部分冗長性除去のアルゴリズムが考えられている。これは、拡張値グラフ (*extended value graph*, *EVG*) というグラフを作成し、これを解析することで部分冗長性除去を行うものである。

彼らの方法や [Kennedy et al. 1999] の方法は、制御フローグラフとは別の特別なグラフの作成が必要であり、解析が複雑になってしまう。また処理の最後にそれぞれの特別なグラフを制御フローグラフに戻す必要があるため、コンパイル時間が多くかかり、その実装は容易ではない。

彼らの方法による部分冗長性除去の処理の計算量は $O(C \cdot N)$ であり、計算量のオーダーとしては少ないが、実際のプログラムの例においては、拡張値グラフの作成やそれを制御フローグラフへ戻す際に処理時間が多くかかるのではないかと予想される。論文中には部分冗長性除去の処理速度に関する実験データは掲載されていない。

彼らの実験は、仮想機械上で行われている。他の最適化と部分冗長性除去の処理を 1 回ずつ組み合わせたものと、彼らの部分冗長性除去を 1 回行った結果のコードとを実行サイクルで比較し、10% 程度の性能向上を示している。

ちなみにまだ未発表であるが、滝本らは最近、効率的な質問伝播を用いた SSA 形式上の部分冗長性除去の新しいアルゴリズムを考案したとのことである。

¹ 第??参照。

第9章 まとめ

本章では、本研究のまとめを行う。

9.1 結論

本研究では、SSA 形式のまま部分冗長性除去を行う新たな手法を提案した。このアルゴリズムは解析に制御フローグラフのみを扱った、単方向のデータフロー方程式のみで表現されている。その最適化効果は、すべての冗長な演算を取り除き、かつ変数の生存区間を最短にしている上、無駄なコード移動はいっさい行わない。出力コードもきれいな SSA 形式を出力し、無駄な ϕ 関数もいっさいなく、同一 ϕ 関数内の変数の生存区間も重ならないため、最適化フェーズの処理速度の向上と、目的コードの実行速度の両方で最大限に効果を発揮する。またその処理は、単方向のデータフロー解析をベクトル方式で 1 度に行えるため、従来の方法に比べ格段に処理時間が減少すると予想される。また、本手法を COINS 上に実装し処理時間の計測を行ったところ、同 COINS 上に実装された共通部分式の除去やループ不変コードのループ外移動などほぼ同等であり、その実用性を示した。

9.2 今後の課題

まず、早速に行える課題をあげる。COINS 上に従来の SSA 形式上の部分冗長性除去として、最新の結果であり、かつ主流の方法とされている [Kennedy et al. 1999] を実装し、処理時間や効果について比較を行い、本手法の優位性を正確な形で示したい。また、[Briggs and Cooper 1994] の行った事前の解析と同等なものを本手法でも行い、除去効果の高い演算の選択と意味等価な式に対して除去できるようにしたい。

次に、研究を重ねて達成できる課題をあげる。現在、SSA 形式では配列を SSA 形式に変換しない。しかし [Collard 1999; Knobe and Sarkar 1998] など、配列も SSA 形式にする研究が行われている。本手法を配列も SSA 形式で扱えるようにすることにより、さらに最適化効果を高めたい。

謝辞

本研究を行うにあたり多くの方々にお世話になりいろいろのことを教えていただいた。特に、東京工業大学数理・計算科学専攻教授の佐々政孝先生，同助教授の脇田建先生には，多くの助言をいただいた。また，本研究のための文献調査に当たって，COINS 関係者の皆様，特に東京理科大学情報科学専攻助手の滝本宗宏氏に参考文献についてのご教示をいただいた。

本研究室の仲間たちには，貴重なアドバイスと，共に学生生活を過ごした貴重な時間をいただいた。ここに深く心からの感謝の意を表す。

平成 16 年 1 月

立川 英

参考文献

- ALPERN, BOWEN, MARK N. WEGMAN, AND F. KENNETH ZADECK 1988. “Detecting Equality of Variables in Programs.” In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*.
- APPEL, ANDREW W. 1998. *Modern Compiler Implementation in Java*. Cambridge University Press.
- AYCOCK, JOHN AND NIGEL HORSPOOL 2000. “Simple Generation of Static Single-Assignment Form.” In *9th International Conference on Compiler Construction*. Vol. 1781 of *Lecture Notes in Computer Science*. Springer-Verlag.
- BRANDIS, MARC M. AND HANSPETER MÖSSENBÖCK 1994. “Single-Pass Generation of Static Single-Assignment Form for Structured Languages.” *ACM Transactions on Programming Languages and Systems*. Vol. 16. No. 6. pp. 1684–1698.
- BRIGGS, PRESTON AND KEITH D. COOPER 1994. “Effective Partial Redundancy Elimination.” In *SIGPLAN Conference on Programming Language Design and Implementation*.
- BRIGGS, PRESTON, KEITH D. COOPER, AND LINDA TORCZON 1994. “Improvements to Graph Coloring Register Allocation.” *ACM Transactions on Programming Languages and Systems*. Vol. 16. No. 3. pp. 428–455.
- BRIGGS, PRESTON, KEITH D. COOPER, TIMOTHY J. HARVEY, AND L. TAYLOR SIMPSON 1998. “Practical Improvements to the Construction and Destruction of Static Single Assignment Form.” *Software – Practice and Experience*. Vol. 28. No. 8. pp. 859–881.
- CHAITIN, G. J. 1982. “Register Allocation & Spilling via Graph Coloring.” In *Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction*.
- CHOI, JONG-DEOK, RON CYTRON, AND JEANNE FERRANTE 1991. “Automatic Construction of Sparse Data Flow Evaluation Graphs.” In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*.
- CHOW, FRED C., SUN CHAN, ROBERT KENNEDY, SHIN-MING LIU, RAYMOND LO, AND PENG TU 1997. “A New Algorithm for Partial Redundancy Elimination based on SSA Form.” In *SIGPLAN Conference on Programming Language Design and Implementation*.
- CHOW, F. 1983. “A portable machine independent optimizer - design and measurements.” Ph.D. dissertation. Stanford University, Dept. of Electrical Engineering, Stanford, CA.
- COLLARD, JEAN-FRANCOIS 1999. “Array SSA for Explicitly Parallel Programs.” In *European Conference on Parallel Processing*.
- CYTRON, RON, JEANNE FERRANTE, BARRY K. ROSEN, MARK N. WEGMAN, AND F. KENNETH ZADECK 1989. “An Efficient Method of Computing Static Single Assignment Form.” In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*.

- CYTRON, RON, JEANNE FERRANTE, BARRY K. ROSEN, MARK N. WEGMAN, AND F. KENNETH ZADECK 1991. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.” *ACM Transactions on Programming Languages and Systems*. Vol. 13. No. 4. pp. 451–490.
- DHAMDHARE, DHANANJAY M. AND UDAY P. KHEDKER 1993. “Complexity of bi-directional data flow analysis.” In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press.
- DHAMDHARE, D. M. AND HARISH PATIL 1993. “An elimination algorithm for bidirectional data flow problems using edge placement.” *ACM Trans. Program. Lang. Syst.* Vol. 15. No. 2. pp. 312–336.
- DHAMDHARE, D. M. 1991. “Practical adaption of the global optimization algorithm of Morel and Renvoise.” *ACM Trans. Program. Lang. Syst.* Vol. 13. No. 2. pp. 291–294.
- FITZGERALD, ROBER, TODD B. KNOBLOCK, ERIK RUF, BJARNE STEENSGAARD, AND DAVID TARDITI 2000. “Marmot: an optimizing compiler for Java.” *Software – Practice and Experience*. Vol. 30. No. 3. pp. 199–232.
- GCC “GNU Compiler Collection Home Page.”. <http://gcc.gnu.org/>.
- GEORGE, LAL AND ANDREW W. APPEL 1996. “Iterated Register Coalescing.” In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- HECHT, MATTHEW S. 1977. *Flow Analysis of Computer Programs*. Elsevier Science Inc.
- IBM “Jikes Research Virtual Machine Home Page.”. <http://www-124.ibm.com/developerworks/oss/jikesrvml/>.
- INTEL “Intel Compilers Home Page.”. <http://developer.intel.com/software/products/compilers/>.
- 伊藤陽 2004. 「コンパイラ・インフラストラクチャ上の部分冗長除去の実装と評価」東京工業大学 情報科学科 卒業論文.
- KENNEDY, ROBERT, FRED C. CHOW, PETER DAHL, SHIN-MING LIU, RAYMOND LO, AND MARK STREICH 1998. “Strength Reduction via SSAPRE.” In *Comppile Construction*.
- KENNEDY, ROBERT, SUN CHAN, SHIN-MING LIU, RAYMOND LO, PENG TU, AND FRED CHOW 1999. “Partial redundancy elimination in SSA form.” *ACM Transactions on Programming Languages and Systems*. Vol. 21. No. 3. pp. 627–676.
- KENNEDY, KEN 1981. “A Survey of Data Flow Analysis Techniques.” In Steven S. Muchnick and Neil D. Jones, eds. *Program Flow Analysis: Theory and Applications*. Prentice-Hall. pp. 5–54.
- KNOBE, KATHLEEN AND VIVEK SARKAR 1998. “Array SSA Form and Its Use in Parallelization.” In *Symposium on Principles of Programming Languages*.
- KNOOP, J., O. RUETHING, AND B. STEFFEN 1992. “Lazy code motion.” *SIGPLAN Notices*. Vol. 27. No. 7. pp. 224–234.
- KNOOP, JENS, OLIVER RÜTHING, AND BERNHARD STEFFEN 1994. “Optimal Code Motion: Theory and Practice.” *ACM Transactions on Programming Languages and Systems*. Vol. 16. No. 4. pp. 1117–1155.
- KNUTH, D. E. 1971. “An empirical study of FORTRAN programs.” *Software Practice and Experience*. Vol. 1. No. 105–133.

- 小濱真樹 2002. 「静的単一代入形式の正規化における Briggs 法の実装および Sreedhar 法との比較」東京工業大学 数理・計算科学専攻 卒業論文.
- 小濱真樹 2004. 「SSA 正規化アルゴリズムの比較と評価」 Master's thesis 東京工業大学 情報理工学研究科 数理・計算科学専攻.
- 小濱真樹・中谷俊晴・佐々政孝 2002. 「静的単一代入形式における正規化アルゴリズムの比較」『日本ソフトウェア科学会第 19 回大会論文集』.
- LENGAUER, THOMAS AND ROBERT E. TARJAN 1979. “A Fast Algorithm for Finding Dominators in a Flowgraph.” *ACM Transactions on Programming Languages and Systems*. Vol. 1. No. 1. pp. 121–141.
- MACHSUIF “Harvard University MachSUIF Homepage.”. <http://www.eecs.harvard.edu/machsuiif/>.
- MOREL, E. AND C. RENVOISE 1979. “Global optimization by suppression of partial redundancies.” *Commun. ACM*. Vol. 22. No. 2. pp. 96–103.
- 中田育男 1999. 『コンパイラの構成と最適化』朝倉書店.
- 中谷俊晴 2001. 「コンパイラ・インフラストラクチャにおける静的単一代入形式変換器の実装と評価」東京工業大学 情報科学科 卒業論文.
- 中谷俊晴・加藤吉之介・佐々政孝・脇田建 2001. 「コンパイラ・インフラストラクチャにおける SSA 最適化プロトタイプシステムの実装」『日本ソフトウェア科学会第 18 回大会論文集』.
- 文部科学省 『科学技術振興調整費 総合研究「並列化コンパイラ向け共通インフラストラクチャの研究」』. <http://www.coins-project.org/>.
- PALERI, VINEETH KUMAR, Y. N. SRIKANT, AND PRITI SHANKAR 1998. “A simple algorithm for partial redundancy elimination.” *SIGPLAN Not.* Vol. 33. No. 12. pp. 35–43.
- PARK, JINPYO AND SOO-MOOK MOON 1998. “Oprimistic Register Coalescing.” In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.
- ROSEN, B. K., M. N. WEGMAN, AND F. K. ZADECK 1988a. “Global value numbers and redundant computations.” In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press.
- ROSEN, BARRY K., MARK N. WEGMAN, AND F. KENNETH ZADECK 1988b. “Global Value Numbers and Redundant Computations.” In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*.
- 佐々政孝 1989. 『プログラミング言語処理系』岩波書店.
- SREEDHAR, VUGRANAM C. AND GUANG R. GAO 1994. “Computing ϕ -nodes in Linear Time Using DJ-graphs.” Technical report McGill University School of Computer Science ACAPS Laboratory. ACAPS Technical Memo 75.
- SREEDHAR, VUGRANAM C. AND GUANG R. GAO 1995a. “Computing ϕ -nodes in linear time using DJ graphs.” *Journal of Programming Languages*. Vol. 3. pp. 191–213.
- SREEDHAR, VUGRANAM C. AND GUANG R. GAO 1995b. “A Linear Time Algorithm for Placing ϕ -Nodes.” In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- SREEDHAR, VUGRANAM C., ROY DZ-CHING JU, DAVID M. GILLIES, AND VASTA SANTHANAM 1999. “Translating Out of Static Single Assignment Form.” In *Proceedings of the 6th International Symposium on Static Analysis*. Vol. 1694 of *Lecture Notes in Computer Science*. Springer-Verlag.

- 立川英 2002. 「循環リモート属性文法による部分冗長性除去の記述」東京工業大学 情報科学科卒業論文.
- 立川英・佐々木晃・佐々政孝 2002. 「リモート参照が可能な循環属性文法を用いた怠けたコード移動の実現」日本ソフトウェア科学会第 19 回大会.
- 滝本宗宏・原田賢一 1997. 「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』第 38 巻, 第 11 号, 2237-2250 頁.
- TARJAN, ROBERT ENDRE 1979. “Applications of Path Compression on Balanced Trees.” *J. ACM*. Vol. 26. No. 4. pp. 690-715.
- 渡邊坦・藤波順久・中田育男 1999. 「コンパイラ研究の動向について」『コンピュータソフトウェア』第 16 巻, 第 2 号, 87-90 頁.