コンパイラ・インフラストラクチャ COINS における SSA 形式最適化の実現

Implementation of an Optimizer Using SSA Form on Compiler Infrastructure COINS

福岡 岳穂 † 高橋 正人 † 中谷 俊晴 †† 佐々 政孝 ††
Takeaki FUKUOKA Masahito TAKAHASHI Toshiharu NAKAYA Masataka SASSA
†(株) 管理工学研究所
Kanrikogaku Kenkyusho, Ltd.
†† 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
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

コンパイラ・インフラストラクチャ COINS の中間表現を SSA 形式に変換し，最適化を行うシステムを実現した．本システムでは，SSA 形式への変換手法を 3 種類，SSA 形式から通常形式への変換手法を 2 種類，および SSA 形式上での条件分岐付定数伝播や共通部分式除去などの基本的な最適化手法を実装した．システムの機能としてはまだ基本的なものしか実装されていないが，インフラストラクチャとして，新たな手法の追加が容易で，さまざまな手法を比較することが可能である．

1 はじめに

すぐれたコンパイラを開発することは，ソフトウエアの高性能化に必須である．しかし，コンパイラを開発することは容易ではない．そこで，コンパイラを部品化し，インフラストラクチャとして提供することで，あらたなコンパイラ開発のコストを軽減することが考えられる．

COINS (COmpiler INfraStructure) [1] とは,「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」として平成 12 年度より進められている研究プロジェクトである．COINS ではコンパイラで利用される基本的な解析や最適化などを部品化し，提供することを目標としている．

本論文では，COINS コンパイラ上で実現した SSA 形式 (Static Single Assignment Form : 静的単一代入形式) 最適化システム (以下本システム) について述べる．本システムでは，SSA 形式最適化と，それに関わる種々の基本的な部品をインフラストラクチャとして提供しているので，SSA 形式に関するさまざまな手法を容易に比較・評価することができる．また，SSA 形式の新たな手法や最適化を容易に追加することができる．

以下，第 2 章で COINS コンパイラの概要を，第 3 章で SSA 形式の概要を述べ，第 4 章で実装したシステムについて述べる．第 5 章で本システムの実行例を挙げ，第 6 章でまとめと今後の課題を述べる．

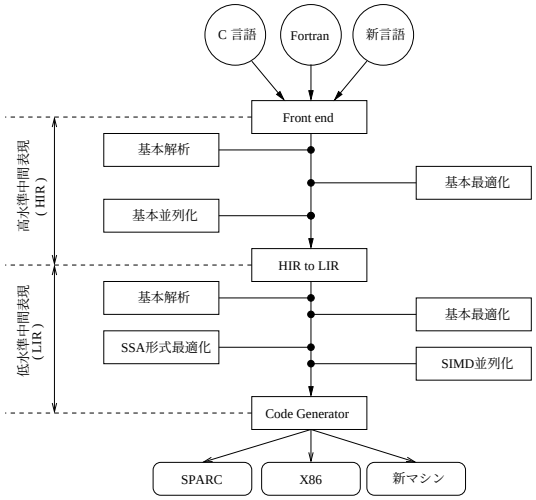

図 1: COINS コンパイラの概要

2 COINS コンパイラ

COINS コンパイラはコンパイラのインフラストラクチャであり，複数言語対応かつリターゲッタブルなコンパイラを目指し，Java 言語で記述されている．図 1 に COINS コンパイラの概要を示す．

COINS コンパイラでは高水準中間表現 (High Level Intermediate Representation : HIR) と低水準中間表現 (Low Level Intermediate Representation : LIR) を利用する．HIR はソース言語の情報を保持したデータ構造である．つまり，ソース言語の情報

を元に最適化を行ったり，HIR からソースプログラムを生成することができる．LIR は，必要最小限のメモリ参照だけを明示し，残りの変数を仮想レジスタであるとみなした低水準中間表現である．LIR では，実行されるコードを意識した最適化を行うことができる．

COINS コンパイラにおいて行われる最適化や解析は「パス」と呼ばれる．本システムもパスのひとつである．利用者は，コンパイラのメインプログラムであるコンパイラドライバにパスを追加することにより，最適化や解析を容易に COINS コンパイラに組み込むことができる．

現状では，COINS コンパイラは C 言語を入力とし，SPARC プロセッサに対応するアセンブラ言語を出力する．今後，入力言語としては Fortran や 新言語，出力は X86 や 新マシンにも対応することを目標としている．また，パスは，HIR と LIR 上での制御フローおよびデータフロー解析，共通部分式除去などの基本的な最適化が利用できる．

3 SSA 形式

SSA 形式 [2, 9, 12] とは，プログラム中の変数の定義が唯一になるように，変数にインデクスをつけたものである．ひとつの変数の使用に対して，異なる定義が到達する制御フローグラフ (Control Flow Graph : CFG) 上の合流点には，ϕ 関数と呼ばれる仮想的な関数を挿入し，それらの定義をまとめる．一般的に，SSA 形式を用いることにより，データフロー解析や逐次実行の最適化などが見通しよく行なえる．

SSA 形式に変換される前の表現形式を通常形式と呼ぶことにすると，通常形式から SSA 形式への変換手法 (以下 SSA 変換) は，Cytron らが提案した手法 [8, 12] と，Sreedhar らが提案した手法 [10] が広く知られている．一般的に，SSA 変換は ϕ 関数の挿入，および変数の名前替えという 2 つのフェーズからなる．変換された SSA 形式には，minimal SSA 形式 [8, 12] ，semi-pruned SSA 形式 [4, 12]，pruned SSA 形式 [7, 12] の 3 種類がある．これらの形式への変換アルゴリズムは，生存変数解析を行う部分のみが異なる．3 種類の SSA 形式の違いを図 2 に示す．

SSA 形式から通常形式への変換手法 (以下 SSA 逆変換) は，Briggs らが提案した手法 [4] や Sreedhar らが提案した手法 [11] などが知られている．

一般的に，SSA 逆変換は，ϕ 関数が行う処理を先行ブロックで分担して行わせればよい．よって多くの場合，ϕ 関数が入っている基本ブロックの先行ブロックに，ϕ 関数のパラメタで使われている変数のコピー文を挿入し，ϕ 関数を消去することで通常形式に戻すことができる．図 3 に SSA 逆変換の例を示す．(a) のブロック L3 にある ϕ 関数でパラメタとして使われている変数 x1 と x2 は，それぞれブロック L1 とブロック L2 から到達した定義である．SSA 逆変換によって，ブロック L3 の ϕ 関数であらたに定義される変数 x3 を，L3 の先行ブロックである L1 と L2 で定義し，ϕ 関数を除去したものが (b) である．

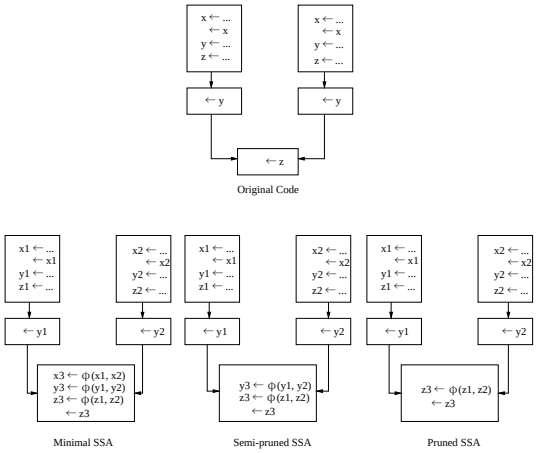

図 2: 3 種類の SSA 形式

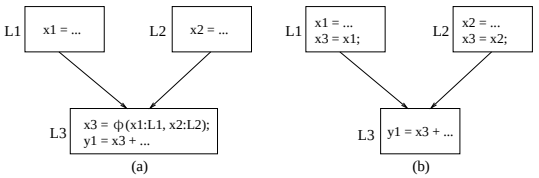

図 3: SSA 逆変換

4 SSA 形式最適化システムの構成

中谷ら [13] は，COINS コンパイラでの SSA 形式最適化プロトタイプとして，HIR 上で SSA 形式最適化を行うシステムを実装し評価した．そこで得られた知見をもとに，本システムでは，対象を LIR とし，図 4 に示す流れに従って SSA 形式最適化を行う．本システムの構成にあたっては，今後のインフラストラクチャとしての利用を考慮して，複数の表現形式およびフェーズを選択できるようにした．

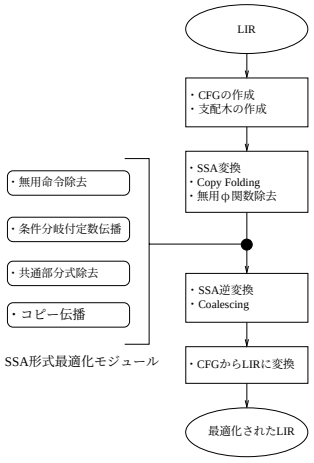

図 4: SSA 形式最適化の流れ

4.1 SSA 変換

SSA 変換は Cytron らが提案した手法 [8] を実装した．本システムでは，ϕ 関数挿入の際に，異なる精度の変数生存区間解析を行うことにより，minimal, semi-pruned，pruned の 3 種類の SSA 形式に変換することができる．また SSA 変換における変数の名前替えの際に，同時に Copy Folding [4, 5] と無用 ϕ 関数除去 [4, 5] を行うことが可能である．

変換する SSA 形式，および Copy Folding と無用 ϕ 関数除去の有無は選択的に行うことができる．

4.2 SSA 形式最適化

SSA 形式上での最適化は，現在以下のものが実装されている [3, 12].

- 無用命令除去
- 条件分岐付定数伝播
- 共通部分式除去
- コピー伝播

図 4 で示すように，これらの最適化はすべてモジュール化されており，どの最適化をどのような順序で実行するかは利用者が指定できる．

4.3 SSA 逆変換

SSA 形式から通常形式への変換は Sreedhar らが提案した手法 [11] を実装した．この手法は，以下の手順で実行される．

Step 1: プログラム中の全ての ϕ 関数に対して，そこで使用されている変数間の干渉を除去するための変数の名前替えとコピー文の挿入.

Step 2: 無駄なコピー文の除去.

Step 3: ϕ 関数を除去して通常形式に変換.

ここで，Sreedhar らは，Step 1 のために，3 種類のアルゴリズム，Method I，Method II，および Method III を提案している．Method I は ϕ 関数で使用される全ての変数に対してナイーブにコピー文を挿入する．Method II は Method I を拡張し，干渉のある変数にのみコピー文を挿入する．Method III では，挿入されるコピー文の数を Method II よりもさらに減らすために，干渉情報だけでなく変数の生存区間情報も利用する．本システムでは， Method I と Method III を実装した．これらはコンパイル時に選択可能である．また，Step 2 では，Sreedhar らが [11] で提案している SSA-based Coalescing を実行することで，無駄なコピー文を除去する．本システムでは，これを選択的に実行できる．

さらに，本システムでは，Chaitin が (もともとレジスタ割り当てのために) 提案した干渉グラフに基づく Coalescing アルゴリズム [6] も実装した．これは，SSA 逆変換後の通常形式上で選択的に利用することができる．

小濱らは，本システムの上に別の SSA 逆変換アルゴリズムを実装し，比較・検討を行っている [14]．これは，本システムをインフラストラクチャとして利用した一例と言える．

5 実行例

本システムの実行例として，図 5 のプログラム (以下テストプログラム) を最適化する過程を示す．テス

```
01: int main(){
02:   int i;
03:   int j;
04:   int k;
05:   int l;
06:
07:   i=6;
08:   j=1;
09:   k=1;
10:
11:   while(i!=j){
12:     if(i==6) k=0;
13:     else i=i+1;
14:
15:     i=i+k;
16:     j=j+1;
17:   }
18:
19:   printf("%d %d %d",i,j,k);
20: }
```

図 5: テストプログラム (C 言語)

```
01: (PROLOGUE "main")
02: (LABELDEF "_lab1")
03: (SET:I32 (REG:I32 %ar1$ti0) (CONST:I32 6))
04: (SET:I32 (REG:I32 %ar2$ti1) (CONST:I32 1))
05: (SET:I32 (REG:I32 %ar3$ti2) (CONST:I32 1))
06: (LABELDEF "_lab8")
07: (JUMPC (TSTNE:I32 (REG:I32 %ar1$ti0) (REG:I32 %ar2$ti1))
           (LISTD (LABEL "_lab9")
           (LISTD (LABEL "_lab4"))))
08: (LABELDEF "_lab9")
09: (JUMPC (TSTEQ:I32 (REG:I32 %ar1$ti0) (CONST:I32 6))
           (LISTD (LABEL "_lab5")
           (LISTD (LABEL "_lab6"))))
10: (LABELDEF "_lab5")
11: (SET:I32 (REG:I32 %ar3$ti2) (CONST:I32 0))
12: (JUMP (LABEL "_lab7"))
13: (LABELDEF "_lab6")
14: (SET:I32 (REG:I32 %ar1$ti0)
             (ADD:I32 (REG:I32 %ar1$ti0) (CONST:I32 1)))
15: (LABELDEF "_lab7")
16: (SET:I32 (REG:I32 %ar1$ti0)
             (ADD:I32 (REG:I32 %ar1$ti0) (REG:I32 %ar3$ti2)))
17: (SET:I32 (REG:I32 %ar2$ti1)
             (ADD:I32 (REG:I32 %ar2$ti1) (CONST:I32 1)))
18: (LABELDEF "_lab3")
19: (JUMP (LABEL "_lab8"))
20: (LABELDEF "_lab4")
21: (CALL:I32 "printf" (LISTA:I32 (CONST:I32 ""%d %d %d"")
                       (LISTA:I32 (REG:I32 %ar1$ti0)
                       (LISTA:I32 (REG:I32 %ar2$ti1)
                       (LISTA:I32 (REG:I32 %ar3$ti2))))))
22: (LABELDEF "_lab2")
23: (EPILOGUE "main")
```

図 6: LIR に変換後のテストプログラム (LIR)

```
01: (PROLOGUE "main")
02: (LABELDEF "_lab1")
03: (SET:I32 (REG:I32 %ar1_1$ti6) (CONST:I32 6))
04: (SET:I32 (REG:I32 %ar2_1$ti7) (CONST:I32 1))
05: (SET:I32 (REG:I32 %ar3_1$ti8) (CONST:I32 1))
06: (LABELDEF "_lab8")
07: (PHI:I32 (REG:I32 %ar3_2$ti9) ((%ar3_1 :: block_0)
                                   ((%ar3_3 :: block_6))))
08: (PHI:I32 (REG:I32 %ar2_2$ti10) ((%ar2_1 :: block_0)
                                    ((%ar2_3 :: block_6))))
09: (PHI:I32 (REG:I32 %ar1_2$ti11) ((%ar1_1 :: block_0)
                                    ((%ar1_5 :: block_6))))
10: (JUMPC (TSTNE:I32 (REG:I32 %ar1_2$ti11) (REG:I32 %ar2_2$ti10))
           (LISTD (LABEL "_lab9")
           (LISTD (LABEL "_lab4"))))
11: (LABELDEF "_lab9")
12: (JUMPC (TSTEQ:I32 (REG:I32 %ar1_2$ti11) (CONST:I32 6))
           (LISTD (LABEL "_lab5")
           (LISTD (LABEL "_lab6"))))
13: (LABELDEF "_lab5")
14: (SET:I32 (REG:I32 %ar3_4$ti17) (CONST:I32 0))
15: (JUMP (LABEL "_lab7"))
16: (LABELDEF "_lab6")
17: (SET:I32 (REG:I32 %ar1_3$ti12)
             (ADD:I32 (REG:I32 %ar1_2$ti11) (CONST:I32 1)))
18: (LABELDEF "_lab7")
19: (PHI:I32 (REG:I32 %ar3_3$ti13) ((%ar3_4 :: block_3)
                                    ((%ar3_2 :: block_4))))
20: (PHI:I32 (REG:I32 %ar1_4$ti14) ((%ar1_2 :: block_3)
                                    ((%ar1_3 :: block_4))))
21: (SET:I32 (REG:I32 %ar1_5$ti15)
             (ADD:I32 (REG:I32 %ar1_4$ti14) (REG:I32 %ar3_3$ti13)))
22: (SET:I32 (REG:I32 %ar2_3$ti16)
             (ADD:I32 (REG:I32 %ar2_2$ti10) (CONST:I32 1)))
23: (LABELDEF "_lab3")
24: (JUMP (LABEL "_lab8"))
25: (LABELDEF "_lab4")
26: (CALL:I32 "printf" (LISTA:I32 (CONST:I32 ""%d %d %d"")
                       (LISTA:I32 (REG:I32 %ar1_2$ti11)
                       (LISTA:I32 (REG:I32 %ar2_2$ti10)
                       (LISTA:I32 (REG:I32 %ar3_2$ti9))))))
27: (LABELDEF "_lab2")
28: (EPILOGUE "main")
```

図 7: SSA 変換後のテストプログラム (LIR)

```
01: (PROLOGUE "main")
02: (LABELDEF "_lab1")
03: (SET:I32 (REG:I32 %ar2_1$ti7) (CONST:I32 1))
04: (SET:I32 (REG:I32 %ar3_1$ti8) (CONST:I32 1))
05: (LABELDEF "_lab8")
06: (PHI:I32 (REG:I32 %ar3_2$ti9) ((%ar3_1 :: block_0)
                                   ((%ar3_3 :: block_5))))
07: (PHI:I32 (REG:I32 %ar2_2$ti10) ((%ar2_1 :: block_0)
                                    ((%ar2_3 :: block_5))))
08: (SET:I32 (REG:I32 %ar1_2$ti11) (CONST:I32 6))
09: (JUMPC (TSTNE:I32 (REG:I32 %ar1_2$ti11) (REG:I32 %ar2_2$ti10))
           (LISTD (LABEL "_lab7")
           (LISTD (LABEL "_lab4"))))
10: (LABELDEF "_lab7")
11: (SET:I32 (REG:I32 %ar3_3$ti13) (CONST:I32 0))
12: (SET:I32 (REG:I32 %ar2_3$ti16)
             (ADD:I32 (REG:I32 %ar2_2$ti10) (CONST:I32 1)))
13: (JUMP (LABEL "_lab8"))
14: (LABELDEF "_lab4")
15: (CALL:I32 "printf" (LISTA:I32 (CONST:I32 ""%d %d %d"")
                       (LISTA:I32 (REG:I32 %ar1_2$ti11)
                       (LISTA:I32 (REG:I32 %ar2_2$ti10)
                       (LISTA:I32 (REG:I32 %ar3_2$ti9))))))
16: (LABELDEF "_lab2")
17: (EPILOGUE "main")
```

図 8: 最適化後のテストプログラム (LIR)

```
01: (PROLOGUE "main")
02: (LABELDEF "_lab1")
03: (SET:I32 (REG:I32 %ar2_2$ti10) (CONST:I32 1))
04: (SET:I32 (REG:I32 %ar3_2$ti9) (CONST:I32 1))
05: (LABELDEF "_lab8")
06: (SET:I32 (REG:I32 %ar1_2$ti11) (CONST:I32 6))
07: (JUMPC (TSTNE:I32 (REG:I32 %ar1_2$ti11) (REG:I32 %ar2_2$ti10))
           (LISTD (LABEL "_lab7")
           (LISTD (LABEL "_lab4"))))
08: (LABELDEF "_lab7")
09: (SET:I32 (REG:I32 %ar3_2$ti9) (CONST:I32 0))
10: (SET:I32 (REG:I32 %ar2_2$ti10)
             (ADD:I32 (REG:I32 %ar2_2$ti10) (CONST:I32 1)))
11: (JUMP (LABEL "_lab8"))
12: (LABELDEF "_lab4")
13: (CALL:I32 "printf" (LISTA:I32 (CONST:I32 ""%d %d %d"")
                       (LISTA:I32 (REG:I32 %ar1_2$ti11)
                       (LISTA:I32 (REG:I32 %ar2_2$ti10)
                       (LISTA:I32 (REG:I32 %ar3_2$ti9))))))
14: (LABELDEF "_lab2")
15: (EPILOGUE "main")
```

図 9: SSA 逆変換後のテストプログラム (LIR)

```
01: int main(){
02:   int ar2_2;
03:   int ar3_2;
04:   int ar1_2;
05:
06: _lab1:
07:   ar2_2=1;
08:   ar3_2=1;
09: _lab8:
10:   ar1_2=6;
11:   if(ar1_2!=ar2_2) goto _lab7;
12:   else goto _lab4;
13: _lab7:
14:   ar3_2=0;
15:   ar2_2=ar2_2+1;
16:   goto _lab8;
17: _lab4:
18:   printf("%d %d %d",ar1_2,ar2_2,ar3_2);
19: _lab2:
20: }
```

図 10: SSA 逆変換後のテストプログラム (C 言語)

トプログラムは条件分岐付定数伝播などの最適化の微妙な例を確かめるものである．

図 5 に示す C 言語で書かれたコードを LIR に変換したものを図 6 に示す．図 6 のコードを pruned SSA 形式に変換した LIR を図 7 に示す．また，図 7 の LIR に対して最適化 (条件分岐付定数伝播と無用命令除去) を行った結果を図 8 に示す．例えば，図 7 の 09: 行目の ϕ 関数や 12: 行目の条件ジャンプ文が，最適化の結果，除去されているのがわかる．

図 8 の LIR に対して Sreedhar の Method III を用いて SSA 逆変換したものを 図 9 に示す．また，図 9 の LIR から，本システムのオプションを用いて生成した C 言語のコードを図 10 に示す．

6 まとめと今後の課題

本論文では，COINS コンパイラ上で実現した SSA 形式最適化システムについて述べた．本システムは，COINS コンパイラ の LIR 上で実現し，以下の機能を実装した．

- SSA 変換 (minimal SSA，semi-pruned SSA，pruned SSA から選択可能)
- SSA 逆変換 (Sreedhar らの Method I と Method III から選択可能)
- SSA 変換時における Copy Folding と無用 ϕ 関数除去
- SSA 形式での最適化における，共通部分式除去，条件分岐付定数伝播，無用命令除去，コピー伝播
- 2 種類の Coalescing

これらのほとんどは，COINS コンパイラのコンパイラオプションとして指定することが可能であり，さまざまな変換の評価データが容易に得られるようになっている．システムの機能としてはまだ基本的なものしか実装されていないが，[14] での利用に見られるように，インフラストラクチャおよび種々の新しい最適化のベースとしての機能に十分考慮を払っている．

今後は，ループを扱った最適化など，より多くの最適化を実装する予定である．さらに，最適化システムとしての評価を行い，インフラストラクチャとしての利便性も追求する予定である．

謝辞

本研究は科学技術振興調整費「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」による．

参考文献

[1] COINS. http://www.coins-project.org/.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. of the 15th ACM POPL*, pp. 1–11, January 1988.

[3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge Univ. Press, 1998.

[4] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859–881, July 1998.

[5] P. Briggs, T. J. Harvey, and L. T. Simpson. Static single assignment construction, version 1.0, January 1996. ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps.

[6] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Notices 17(6), Proc. of the ACM SIGPLAN '82 Symp. on Compiler Construction*, pp. 98–105, 1982.

[7] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proc. of 18th ACM POPL*, pp. 55–66, January 1991.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 4, pp. 461–486, October 1991.

[9] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proc. of the 15th ACM POPL*, pp. 12–27, January 1988.

[10] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proc. of the 22nd ACM POPL*, pp. 62–73, January 1995.

[11] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. *SAS'99 LNCS 1694*, pp. 194–210, 1999.

[12] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[13] 中谷俊晴, 加藤吉之介, 佐々政孝, 脇田建. コンパイラ・インフラストラクチャにおける SSA 形式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大会論文集, 第 18 回, 3D-2, September 2001.

[14] 小濱真樹, 中谷俊晴, 佐々政孝. 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大会論文集, 第 19 回, September 2002.