

# 静的単一代入形式における 正規化アルゴリズムの比較と評価

## A Comparison of SSA Normalization Algorithms

伊藤 陽<sup>†</sup>

Yo ITO

小濱 真樹<sup>†</sup>

Masaki KOHAMA

佐々 政孝<sup>†</sup>

Masataka SASSA

<sup>†</sup> 東京工業大学 大学院情報理工学専攻 数理・計算科学専攻

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

{Yo.Ito, Masaki.Kohama, Masataka.Sassa }@is.titech.ac.jp

SSA 形式で表された中間表現は、 $\phi$  関数のない形に変換しなければならない (正規化)。本研究では、正規化アルゴリズムとして、Briggs らの方法と、Sreedhar らの方法の長所と短所を明確にした。また、Briggs らの方法において、改良案を提案した。さらに、それらの方法を実装し、比較した。その結果、Sreedhar らの方法が実証的に優れていること、改良案は若干ながら効果があることなどが判明した。

## 1 はじめに

SSA 形式 (Static Single Assignment Form, 静的単一代入形式) とは各種最適化にとって有用なプログラム中間表現である。しかし、SSA 形式で表されたプログラムはそのままではアセンブリ言語や機械語に変換することができないため、通常形式に戻す必要がある。これを正規化 (SSA 逆変換) と呼ぶ。

正規化は、そのアルゴリズム [1, 7, 5] によって変換後のプログラムの実行効率が異なってくるため、コンパイラを設計する際にどのアルゴリズムを選択するかは重要な問題である。しかし、これらのアルゴリズムは考え方がまったく異なっているので、理論的に比べるのは困難である。Sreedhar らは変換結果が最小であるかどうかの理論的議論を少し行っているが、彼らの方法が最小であることは言えない [7]。そのため、正規化アルゴリズムの比較はほとんどされておらず、その選択の手助けとなるものがなかった。

本研究では、主要な正規化アルゴリズムである、Briggs らの方法 [1] と Sreedhar らの方法 [7] の長所と短所を明確にした。また、Briggs らの方法において、改良案を提案した。さらに、それらの方法を実装し、比較した。Sreedhar らの方法によるコードの実行効率は、Briggs らの方法によるものに比べて 1% ~ 7% 良いことが分かった。これは中程度の最適化に匹敵するので、良い正規化アルゴリズムを選ぶことが重要である。

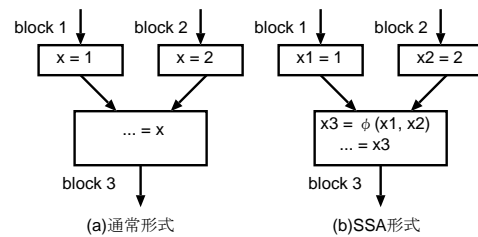


図 1: SSA 形式の例

## 2 SSA 形式

SSA 形式はプログラム中間表現の一つで、プログラム上の各変数の定義を一ヶ所になるように表現したものである [4]。また、異なる定義の合流点には  $\phi$  関数という仮想的な関数を挿入することで定義を一つにまとめる (図 1)。このようにすることで変数の定義と使用の関係が明確になるため、最適化に適した形であるといわれている。

しかし、SSA 形式で表されたプログラム中に含まれる  $\phi$  関数は一般的な計算機では実行不可能である。そこで実行可能なプログラムを得るためには、SSA 形式で表されたプログラムから  $\phi$  関数を削除し通常形式 (通常形式) に変換する必要がある。この変換を正規化 (normalization) または、SSA 逆変換と呼ぶ。

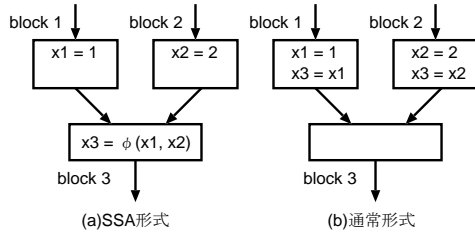


図 2: 素朴な正規化

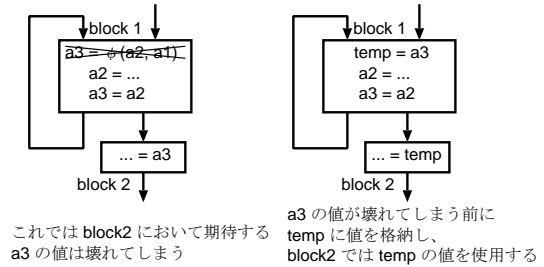


図 4: Briggs らの方法

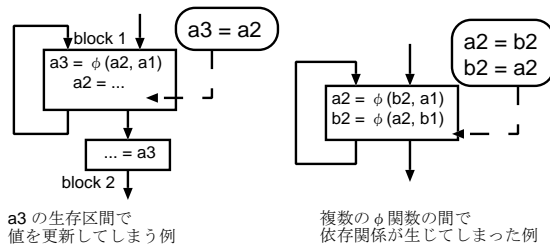


図 3: 危うい例

### 3 SSA 正規化

最初に, Cytron らの示した正規化の基本的な方法 [4] について述べる. 図 2 の左側の制御フローグラフを考えると, block1 から block3 へ制御が移った時,  $\phi$  関数によって  $x3$  には  $x1$  の値が代入される. そこで, block1 中の命令列の最後部に「 $x3 = x1$ 」を挿入し,  $\phi$  関数を除去することができる (block2 の時も同様). このように,  $\phi$  関数のあるブロックの先行ブロックに, 適当なコピー文を挿入し,  $\phi$  関数を除去する方法を素朴な方法と呼ぶことにする.

素朴な方法ではプログラムの意味を変えてしまうような危うい例がいくつか知られている. 素朴な正規化を行うときに, そのような問題を引き起こす要因として次の二つが考えられる [1].

一つは, 図 3 左のような制御フローグラフで表されるプログラムを考える. これを素朴な方法で正規化すると,  $\phi$  関数に対して, 先行ブロックとしての block1 の最後尾に「 $a3 = a2$ 」を挿入する. しかし, このようにすると, block2 で使用されている  $a3$  の値はコピー文の挿入前と挿入後で変わってしまう. これは, 挿入するコピー文のターゲット (代入文の左辺, この場合は  $a3$ ) が挿入する場所で生きているためである.

二つめは, 同一ブロック内に複数の  $\phi$  関数がある場合である. 図 3 右の例では,  $a2$  と  $b2$  への代入は同時におこると考えるべきである (同時代入性). ぶ

たたび, 素朴な方法でこれを正規化すると, 図 3 右の丸枠のように, たがいに依存関係のあるコピー文が挿入されてしまう. これらのコピー文は順番に実行されるため, 変換の前後でプログラムの意味が変わってしまう. これは,  $\phi$  関数の同時代入性をきちんと考慮していなかったためである.

### 4 Briggs らによる SSA 正規化

Briggs らの正規化アルゴリズム [1] は素朴な方法を拡張して, 安全な正規化を行う.

例えば, 図 3 左に対し図 4 左のように素朴な方法で正規化を行った場合, block2 で使用している  $a3$  は, 挿入されたコピー文「 $a3 = a2$ 」によって期待する値を壊されてしまう. そこで Briggs らの方法では, 図 4 右のように, block1 の先頭で「 $temp = a3$ 」とし,  $a3$  の値を  $temp$  に格納し, block1 以降で用いられている  $a3$  は  $temp$  で置き換えるということをする. このように, 「 $temp = a3$ 」のようなコピー文を挿入することで素朴な方法の危うさを補い, プログラムの意味を変えずに正規化を行うことができる.

上記のように Briggs らの正規化アルゴリズムでは多くのコピー文が挿入される. しかし彼らはコアレッシング (生存区間の合併, もとはレジスタ割り当てで用いられた [2]) を行うことでその大部分を除去できると主張している.

### 5 Sreedhar らによる SSA 正規化

Sreedhar らの正規化アルゴリズム [7] では, 素朴な方法や Briggs らの方法とはまったく異なったアプローチをとる. 後者の二つのアルゴリズムは,  $\phi$  関数と同等の役割をするコピー文を挿入して  $\phi$  関数を除去する. それに対し, Sreedhar らのアルゴリズムでは  $\phi$  関数内の変数間に生存区間の干渉があるかど

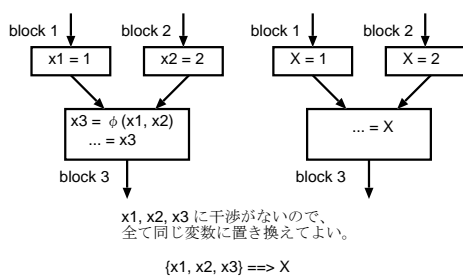


図 5: Sreedhar らの方法による  $\phi$  関数の除去

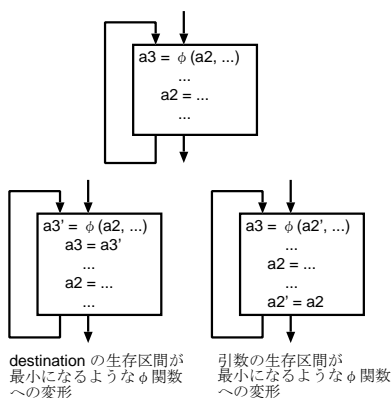


図 6: Sreedhar らの方法による  $\phi$  関数の書き換え

うかを調べ、干渉がなければそれらの変数を同じ変数で置き換えることで  $\phi$  関数を除去する。

例えば、図 5 左を考える。ここで、 $\phi$  関数内の変数 (ターゲットを含む) である  $x_1, x_2, x_3$  の間に干渉はない。そこで、 $x_1, x_2, x_3$  を全て同じ変数  $X$  に置き換える (合併する)。すると、図 5 右のように、コピー文を挿入しなくても、 $\phi$  関数を除去し、通常形式に変換することができる。

しかし  $\phi$  関数内の変数の間に干渉があった場合、合併を行うことができないため、これをなくす必要がある。そこで Sreedhar らの方法では、 $\phi$  関数を書き換えることで干渉を取り除くことをする。以下の作業を組み合わせて行い、 $\phi$  関数内の変数の生存区間を短くすることで干渉を避けることができる。

- $\phi$  関数のターゲットは、その関数が定義されているブロックの入口で生きていると考える。そこでターゲットが最小の生存区間を持つようにするには、図 6 左下のように書き換えればよい。
- $\phi$  関数の引数は、その対応する先行ブロックの出口で生きていると考える。そこで、引数が最

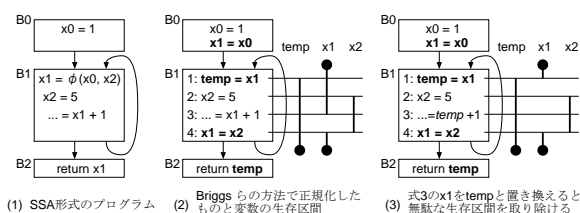


図 7: 無駄な生存区間が出来てしまう例

小の生存区間を持つようにするには図 6 の右下のように書き換えればよい。

## 6 問題の提起と改良案

### 6.1 Briggs らの方法における欠点と改良案

#### 6.1.1 無駄な生存区間によるもの

Briggs らの方法では、 $\phi$  関数のターゲットの値を保持するために「temp = ターゲット」というコードを挿入することがある。このとき、 $\phi$  関数のターゲットと temp が同じ値を持っているのに生存区間が重なってしまう場合があり、これは無駄な生存区間となる。例えば、図 7 のような例を考える。

(2) において、 $x_1$  の値が temp に入っているにも関わらず式 3 で  $x_1$  を使用しているため  $x_1$  の生存区間は式 3 までとなってしまう。また、このような無駄な生存区間があるために  $x_1$  は  $x_2$  と干渉してしまっている。

この場合、式 3 の  $x_1$  を temp で置き換える改良を行うと図 7 の (3) のようになる。このようにすると、式 4 のコピー文もコアレスニング可能となる。

このような状況は必ずループの内側で生じるため、実行効率に与える影響が大きいと考える。

#### 6.1.2 条件式によるもの

ブロックの最後尾に挿入されるコピー文は条件分岐命令等の条件式を含む命令の直前に挿入されることがある。このとき、挿入されたコピー文のソースが条件式内で使用されているとコピー文のターゲットとソースの生存区間が重なってしまう。例えば、図 8 のような例を考える。

(2) において、 $x_1$  と  $x_2$  の生存区間に重なりが生じている。これは  $x_2$  と同じ値が  $x_1$  に入っているにも関わらず、条件式で  $x_2$  を使用してしまうことによるものである。そこで、(3) のように条件式を書き換える改良を行うと、「 $x_1 = x_2$ 」はコアレスニング可能

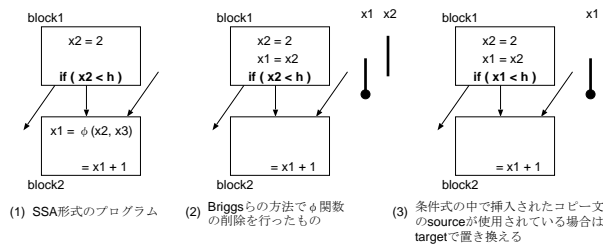


図 8: 条件式によってできる生存区間の重なり

なものとなる。

## 6.2 Sreedhar らの方法における欠点

Sreedhar らの方法では  $\phi$  関数内の変数を全て同じ変数に合併することで  $\phi$  関数を除去する。 $\phi$  関数をコピー文と見なすと、レジスタアロケーションにおけるコアレスティングと同様の議論ができる。つまり、合併した変数の生存区間が長くなりレジスタ数の少ない環境では不利になる可能性がある。

## 7 実験

Coins コンパイラ・インフラストラクチャの SSA モジュールを用い、同一の枠組みで実験を行った [3, 6].

### 7.1 レジスタ数が 8 本の場合

レジスタ数が 8 本の場合の実行時間の相対比を図 9 に示す。相対比からわかるように、Sreedhar らの方法は Briggs らの方法と比べて、gzip では最適化 1 を除いて 3~4%程度、mcf では最適化 2 を除いて 4~5%程度速い。

また、本研究で提案した改良案については、Briggs らの方法と比べて、gzip で 1%、mcf でも最適化 3 を除いてある程度の性能向上が見られる。

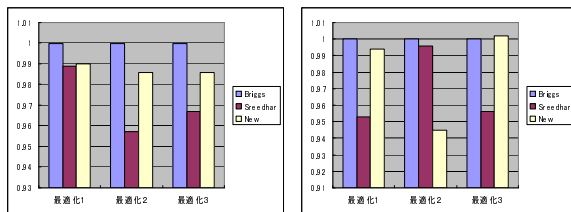


図 9: レジスタ数 8 本での実行時間における相対比 (左 gzip, 右 mcf)(New は改良案)

### 7.1.1 レジスタ数が 20 本の場合

次にレジスタ数が 20 本の場合の実行時間の相対比を図 10 に示す。相対比からわかるように、Sreedhar らの方法は Briggs らの方法と比べて、mcf において 1%程度、gzip において 2~7%程度速い。

本研究で提案した改良案については、Briggs らの方法と比べて、gzip で 1%程度、mcf でも 1~3% 程度の向上があった。

また、レジスタ数が 8 本の場合と同様、多少のばらつきも見られた。

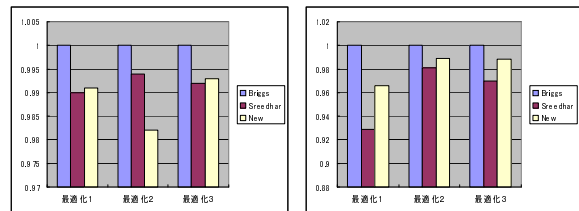


図 10: レジスタ数 20 本での実行時間における相対比 (左 gzip, 右 mcf)(New は改良案)

## 8 おわりに

SSA 正規化アルゴリズムには、主要な 2 つのアプローチが存在するものの、これまでそれらと比較する研究はされてこなかった。そのため、初期のアルゴリズムの誤りを最初に解決した Briggs らの方法をそのまま採用するケースが多かった。そこで本研究では、SSA 正規化アルゴリズムの異なるアプローチに対して、長所と短所を明確にし、検証を行なった。これにより、今まで注目されていなかった SSA 正規化アルゴリズムがプログラムの実行効率に少なからず影響を与えることを明かにし、今後、SSA 正規化アルゴリズムを選択する際の指標を示したことが、本研究の最大の意義であると考えられる。

得られた知見として、

- Briggs らの方法ではコアレスティングできないコピー文が大量に挿入され、Sreedhar らの方法によって懸念されるレジスタ圧力の増加以上に影響を与えること。
- レジスタ数の比較的少ない環境では、スピルによる動的なコストと、コピー文の実行数という 2

つの観点から,  $\phi$  関数に対してある種のコアレスシング処理をするアプローチである Sreedhar らの方法の方が有利となること.

- レジスタ数の比較的多い環境でも, コピー文の実行数という観点から見て, Sreedhar らの方法がやや有利であること.

などが挙げられる. 詳しい解析は [8] にある.

また,  $\phi$  関数をコピー文で代用するアプローチである Briggs らの方法の改良案を与えた. これにより, Briggs らの方法と比べてコアレスシング不可能なコピー文を 4 割程度削減し, スピルのコストについても多少の改善が見られたが, Sreedhar らの方法の優位性を崩す程ではなかった.

今後, より多くのベンチマークで評価を行い, 解析結果をまとめる予定である.

## 9 謝辞

本研究の一部は, 文部科学省科学技術振興調整費および科学研究費の補助を受けた.

## 参考文献

- [1] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, July 1998.
- [2] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, 1982.
- [3] Coins Project Homepage. <http://www.coins-project.org/>.
- [4] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [5] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.
- [6] M. Sassa, T. Nakaya, M. Kohama, T. Fukuoka, and M. Takahashi. Static Single Assignment Form in the COINS Compiler Infrastructure. Proc. SSGRR 2003w, 2003.
- [7] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. 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*, pp. 194–210. Springer-Verlag, 1999.

- [8] 小濱 真樹. SSA 正規化アルゴリズムの比較と評価. 東京工業大学 数理・計算科学専攻 修士論文, 2004.