静的単一代入形式における 正規化アルゴリズムの比較

A Comparison of SSA Normalization Algorithms

小濱 真樹 [†] 中谷 俊晴 [†] 佐々 政孝 [†]
Masaki KOHAMA Toshiharu NAKAYA Masataka SASSA
[†] 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
{Masaki.Kohama, Toshiharu.Nakaya, sassa }@is.titech.ac.jp

SSA 形式で表された中間表現は，ϕ 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 らの方法を比較した．その結果，概ね Sreedhar らの方法で正規化した方が実行効率のよいコードを得ることができることがわかった．また，よりよいアルゴリズムの可能性と今後の方針についても述べる．

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，通常の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, 6, 4] 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撰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違っているので，理論的に比べるのは困難である．そのため，正規化アルゴリズムの比較はそれ程されておらず，その撰択の手助けとなるものはあまりなかった．

そこで今回，正規化の代表的なアルゴリズムである Sreedhar らのアルゴリズム [6] と Briggs らのアルゴリズム [1] を同一のコンパイラ上で実装し，その変換結果について実証的な比較をした．また，よりよい正規化アルゴリズムの可能性についても議論する．

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 [3]．また，異なる定義の合流点には **ϕ 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 ϕ 関数は一般的な計算機では実行不可能である．よって実行可能なプログラムを得るためには，SSA

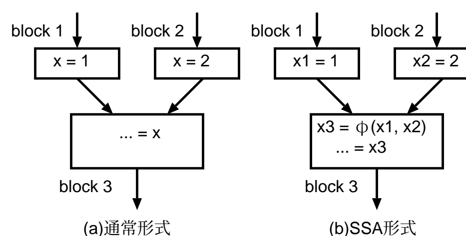

図 1: SSA 形式の例

形式で表されたプログラムから ϕ 関数を削除し通常の形式 (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．

3 正規化と問題点

最初に，Cytron らの示した正規化の基本的な方法 [3] 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，ϕ 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，ϕ 関数を除去することができる (block2 の時も同様)．このように，ϕ 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，ϕ 関数を除去する方法を素朴な方法と呼ぶことにする．

しかし，SSA 形式上で様々な最適化を行ったあと，正規化は困難なものになる．実際，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

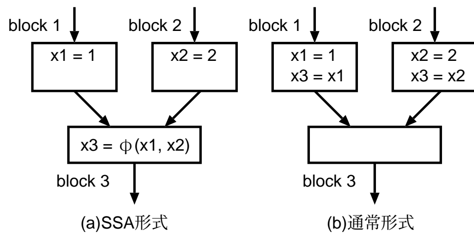

図 2: 素朴な正規化

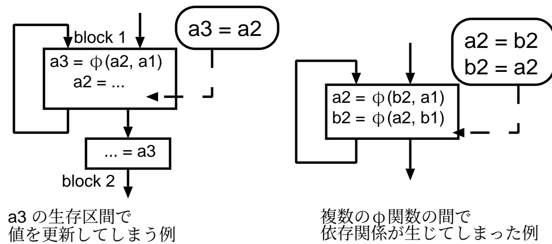

図 3: 危うい例

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，ϕ 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 `destination` (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 ϕ 関数がある場合である．図 3 右の例だと，`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)．ふたたび，素朴な方法でこれを正規化すると，図3右のように，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代入されるため，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，ϕ 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 正規化アルゴリズムの比較

第3節に挙げた問題点を解決した代表的な正規化アルゴリズムとして，Briggs らの方法と Sreedhar らの方法がよく知られている．これら二つのアルゴリズムは考え方が大きく異なっているため，変換結果も同一のものではない．特に，コピー文の数と挿入される場所に差が生じることがある．そこで本研究では，コピー文に関して両者のアルゴリズムを比較した．

まずは，それぞれのアルゴリズムについて簡単に紹介し，第 4.3 節にて二つの比較について述べる．

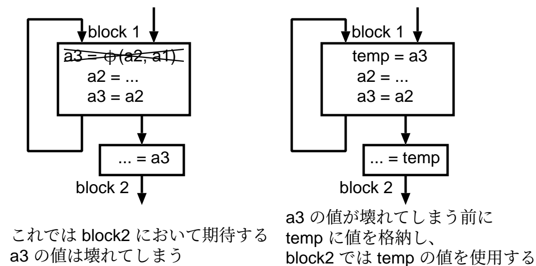

図 4: Briggs らの方法

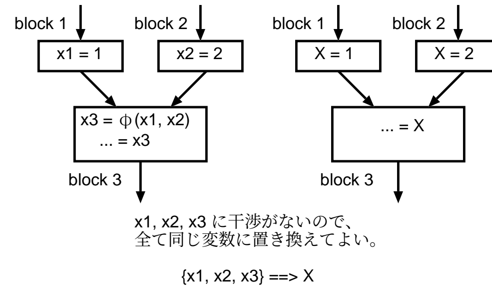

図 5: Sreedhar らの方法による ϕ 関数の除去

4.1 Briggs らによる解決法

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 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,「素朴な方法で挿入されるコピー文 + 危うさを補うために挿入されるコピー文」といった，たくさんのコピー文が挿入されてしまう．しかし，彼らは生存区間の合併を行うことでその多くを除去することができると主張している．

4.2 Sreedhar らによる解決法

Sreedhar らの正規化アルゴリズム [6] 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，ϕ 関数と同等の役割をするコピー文を挿入して ϕ 関数を除去する．それに対し，Sreedhar らのアルゴリズムでは ϕ 関数内の変数間に生存区間の干渉があるかを

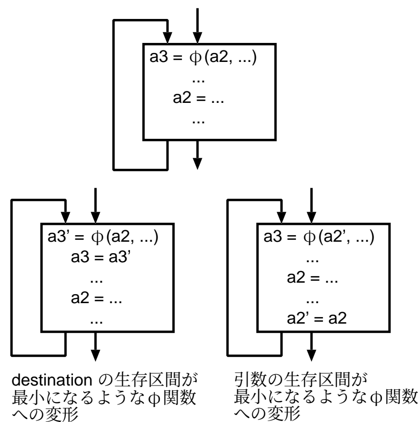

図 6: Sreedhar らの方法による ϕ 関数の書き変え

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 ϕ 関数を除去する．

例えば，図 5左を考える．ここで，ϕ 関数内の変数 (`destination`) である `x1`，`x2`，`x3` の間に干渉はない．そこで，`x1`，`x2`，`x3` 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，ϕ 関数を除去し，通常形式に変換することができる．

しかし ϕ 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，ϕ 関数を書き変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，ϕ 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．

- ϕ 関数の `destination` 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 `destination` が最小の生存区間を持つようにするには，図 6左下のように書き変えればよい．
- ϕ 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変えればよい．

4.3 比較と評価

正規化アルゴリズムによって，変換後のプログラムの実行効率が変わってくるため，どのアルゴリズムを撰択するかは重要な問題である．しかし，これまで正規化アルゴリズムを比較する研究はほとんどなされていない．理論的に変換結果が最小であるかどうかは，Sreedhar らが少し議論している [6] が，最小であるとは言えず，他の文献では議論がない．また，歴史的には Briggs らの方法が先に発表されたため，多くのシステムが単にその当時知られていたという理由で Briggs らの方法を用いる傾向にあり，Sreedhar らの方法を用いたシステムは少ない．

そこで本研究では，正規化アルゴリズムとして Briggs らの方法と Sreedhar らの方法を同一のコンパイラ上で実証的に比べ，コンパイラにとってどちらのアルゴリズムを撰択する方が有利であるかを調べた．また，その過程で，よりよいアルゴリズムのヒントになる例を作り上げることができたので，それについて 4.4節で述べる．

コンパイラとしては，現在開発中の COINS コンパイラ・インフラストラクチャ [5] を利用した．Sreedhar らの正規化アルゴリズムと Chaitin のコアレッシング (もとはレジスタ割り当てに用いられる)[2] は，COINS の SSA 最適化コンパイラ用フレームワークに実装されたもの [7] を用いた．今回我々は，同一のモジュールを使って Briggs らの正規化アルゴリズムを実装した．

比較方法としては，Briggs の方法で正規化した後に Chaitin のコアレッシングをかけたものと，Sreedhar の方法で正規化をしたものとで，それぞれコピー文の数に差があるのか，挿入される場所に違いがあるのかを調べた．

表 1 に結果の一部を示す．この表でのコピー文はすべて正規化後に挿入されたコピー文である．入力プログラムの lost copy，simple ordering，swap は正規化について議論する際によく用いられる例題である [1, 6]．hige swap については4.4節にて触れる．その他のプログラムは，COINS コンパイラの開発に利用されている例題プログラムである．

表 1 を見るとわかるように，多くの例題においてコピー文の数は変わらなかった．また，ループ中のコピー文の数にも差はでなかった．しかし，swap など，いくつかの例題で Sreedhar らの方法で正規化を行った方がコピー文の数が少なくてすむことがわかった．さらに，それらの差はループ中に生じていることから，実行時にコピー文が実行される回数はもっと多いと考えられる．

一方，意図的な例題ではあるが，hige swap のように Briggs らの方法の方が有利な結果を得ることがあることも確認した．

4.4 今後の展望

表 1 の入力プログラム swap は図 7 の 変数 `b` と変数 `c` のように，依存関係にサイクルが生じるような例である (swap 問題と呼ばれる)．このように単純なサイクルがある入力プログラムについては，Sreedhar らの方法で正規化した方が有利であると予想される．一方，図 7 の依存関係には，もう一つ髭のように，変数 `a` が含まれている．このように特殊な依存関係を持つ構造を，ここでは髭付き swap 問題 (hige swap) と呼ぶことにする．この髭付き swap 問題の例題を

	SSA form	Briggs	Briggs + Coalescing	Sreedhar
lost copy	0	3	**1(1)**	**1(1)**
simple ordering	0	5	**2(2)**	**2(2)**
swap	0	7	**5(5)**	**3(3)**
fib	0	4	**0(0)**	**0(0)**
gcd	0	9	**5(2)**	**5(2)**
insertion sort	0	4	**0(0)**	**0(0)**
selection sort	0	9	**0(0)**	**0(0)**
ssatest2	0	6	**0(0)**	**0(0)**
swap-lost	0	10	**7(7)**	**4(4)**
s-order	0	6	**4(4)**	**3(3)**
do	0	9	**6(4)**	**4(2)**
for-test	0	4	**0(0)**	**0(0)**
dce-test	0	10	**0(0)**	**0(0)**
hige swap	0	8	**3(3)**	**4(4)**

表 1: 実験結果

左側の列から順番に，入力プログラム，正規化する前の SSA 形式上でのコピー文の数，Briggs らの方法で正規化したときのコピー文の数，Briggs らの方法の後に Chaitin のコアレッシングをかけたときのコピー文の数，Sreedhar らの方法で正規化したときのコピー文の数である．また，括弧内の数は，ループ中のコピー文の数である．

正規化すると，表 1 のように，Briggs らの方法で正規化した方がコピー文の数が少なくてすんだ．これは，Briggs らの方法が，挿入するコピー文の依存関係に従って，挿入する順番を工夫していることがうまく働いたためである．しかも，その数の差はループの中で現れるため，Sreedhar らの方法で正規化するよりも有利な結果である．

このように，どちらの正規化アルゴリズムも常に最適な結果を得られるわけではない．今後，上記のような依存関係を持つ入力に対してもよい結果を得られるように Sreedhar らの方法を拡張することを考えている．

5 おわりに

最後に本研究のまとめを行う．本研究では SSA 形式における異なる 2 種類の正規化アルゴリズムを同一コンパイラ上で実証的に比較した．比較の対象は Briggs らの方法と Sreedhar らの方法で，比較の手段としては変換結果に残されたコピー文の数，場所などである．

多くのケースで Briggs らの方法と Sreedhar らの方法ではコピー文の数に違いはなかった．しかし，いくつかの場合において Sreedhar らの方法で正規化した方がコピー文の数が少なくてすむことがわかった．さらに，これらのコピー文の差はループ中に現れていたことから，実行時に処理される回数の差は大きいと予想される．

また，意図的ではあるが Briggs らの方法の方が Sreedhar らの方法よりも有利になる例も存在した．

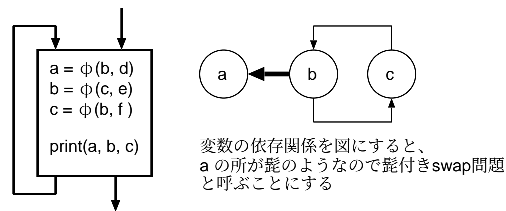

図 7: 変数の依存関係

このことから，Sreedhar らの方法が必ずしも最適な変換結果を得るわけではないこともわかった．

今後の課題としては，

- より大きなテストプログラムでの比較
- 変換したコードの実行時間での比較
- よりよい正規化アルゴリズムの提案

などがあげられる．

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] 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.

[4] R. Morgan. *Building an Optimizing Compiler*. Degital Press, 1998.

[5] COINS Project. `http://www.coins-project.org/`.

[6] 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.

[7] 福岡, 高橋, 中谷, 佐々. 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. 日本ソフトウェア科学会第 19 回大会論文集, September 2002.

[8] 中谷, 加藤, 佐々, 脇田. 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, September 2001.