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

Implementain of a Prototype Optimizer Using SSA Form on Coins Compiler Infrastructure

中谷 俊晴† 加藤 吉之介† 佐々 政孝† 脇田 建†
Toshiharu NAKAYA Yoshinosuke KATO Masataka SASSA Ken WAKITA
† 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
{Toshiharu.Nakaya, Yoshinosuke.Kato, sassa, wakita}@is.titech.ac.jp

コンパイラ・インフラストラクチャにおける，プログラム中間表現の SSA 形式への変換，SSA 形式を用いた最適化，通常形式への変換のプロトタイプシステムを実装した．SSA 形式への変換については，2 種類の方法を実装して評価した．プロトタイプシステムの実装で得られた知見より，今後の実装方針についても議論する．

1 はじめに

コンパイラ技術の基盤を固めることは，将来のコンピュータ産業発展のために必須である．そのためには，研究の基盤となる共通のコンパイラを作り，その上に高度な積み重ねができるようにする必要がある．このような認識から,「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」（以下 Coins と呼ぶ）が 2000 年度より開始された．

Coins コンパイラ・インフラストラクチャの逐次部分では，コンパイラの性能向上に向けた最新の手法として，**SSA 形式**（Static Single Assignment Form, 静的単一代入形式）を用いた最適化を行なう．

今回，事前評価のために SSA 形式最適化プロトタイプシステムを実装したので，SSA 形式への 2 種類の変換方法の評価を含め，その内容を報告する．

2 SSA 形式

SSA 形式 [3] はコンパイラ内部の中間表現の 1 つであり，プログラム上の各変数の使用に対して，その使用に対する定義が一ヶ所しかないように表現したものである．つまり，変数への代入は一度しかない．制御フローグラフ上で 1 つの変数の使用に対して異なる定義が到達する合流点には，単一代入性を保つために 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（図 1）．SSA 形式を中間表現として用いると，変数の定義と使用の関係が明確に表現され，最適化の実現容易性と最適化の実行効率が向上するといわれている．以降，SSA 形式に変換される前の表現形式を**通常形式**と呼び，通常形式から SSA 形式への変換のことを **SSA 変換**と呼ぶことにする．

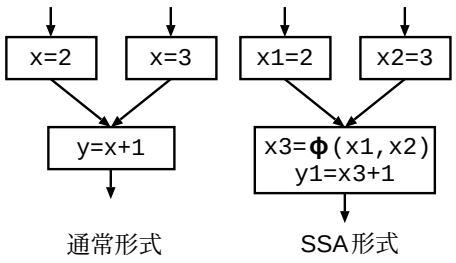

図 1: SSA 形式

3 プロトタイプシステムの構成

Coins コンパイラ・インフラストラクチャ（図 2）には，複数の入力言語，複数の対象機種に対応する 2 種類の中間表現がある．入力言語の論理構造に近いレベルの中間表現を高水準共通中間表現（High Level Intermediate Representation，略して HIR），機械語に近いレベルの中間表現を低水準共通中間表現（Low Level Intermediate Representation，略して LIR）と呼ぶ．HIR は抽象構文木であり，LIR は gcc の RTL に近い．

今回実装したプロトタイプシステムは，HIR レベルでの SSA 形式最適化を行なう（図 3）．ただし，HIR に対して次のような制限を加えた．HIR で扱うデータ型は整数型のみ，制御構造は if，for，while，do-while，break，continue，goto である．また日程上，Coins のフロントエンドが間に合わなかったので，フロントエンドには Cmm プロジェクト [9] のフロントエンドを用い，Cmm コンパイラの中間表現を変換することで HIR を生成した．

これからの議論では，HIR で表現されたプログラムに対応した制御フローグラフと，その制御フローグラフの支配木が作成されているものとする．

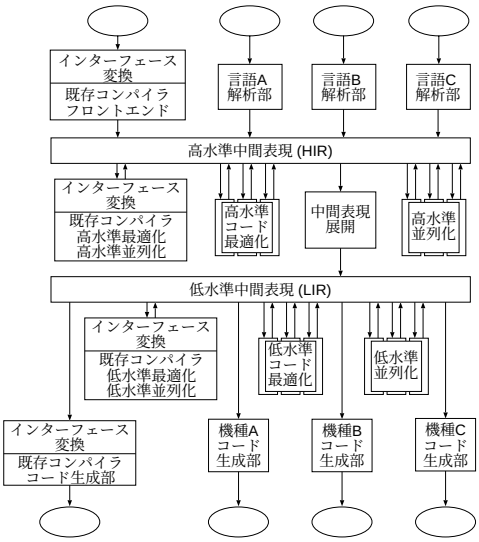

図 2: Coins コンパイラ・インフラストラクチャ

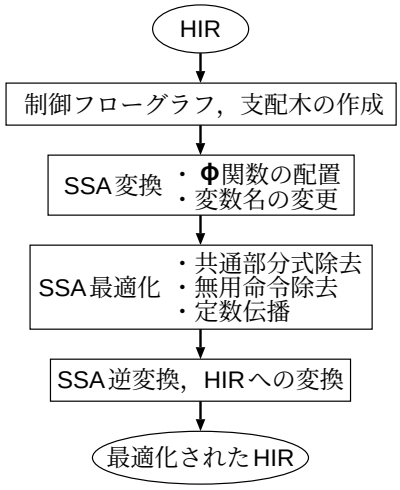

図 3: SSA 最適化の流れ

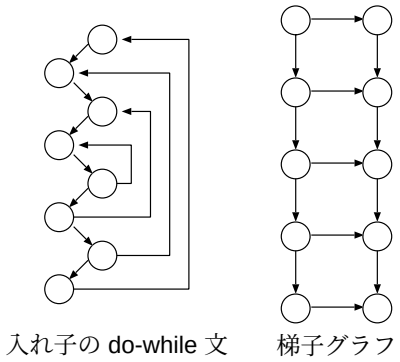

図 4: 入れ子の do-while 文と梯子グラフ

3.1 2 種類の SSA 変換方法とその評価

代表的な SSA 変換アルゴリズムである Cytron らによる方法 [3] と Sreedhar らによる方法 [5, 6] を実装した．どちらの方法も支配辺境と呼ばれるデータ構造を用いて，可約でない（irreducible）制御フローグラフに対しても効率的に SSA 形式を求めることができる．

一般に SSA 変換は（1）ϕ 関数の挿入,（2）変数名の変更，の 2 つのフェーズに分かれる．Cytron らの方法と Sreedhar らの方法の違いは，ϕ 関数の挿入の仕方である．Cytron らの方法での ϕ 関数の挿入は，まず支配辺境を求め，次にその求めた支配辺境を利用して，実際に ϕ 関数を挿入する．それに対して Sreedhar らの方法では，DJ-graph と呼ばれるデータ構造を用いて，支配辺境を求めながら ϕ 関数を挿入できる．Sreedhar らは Cytron らの方法よりも高速であると主張している [5].

Cytron らの方法は，大部分の現実的なプログラムに，その制御フローグラフの大きさに対して，線形時間で ϕ 関数を挿入できるが，do-while 文を繰り返し入れ子にしたプログラムや，制御フローグラフが梯子グラフと呼ばれる形になるプログラム（図 4）では 2 次の計算時間がかかってしまう．さらに，Cytron らの方法は，実際に ϕ 関数を挿入する前に，支配辺境を求めるが，その支配辺境を求める計算も，制御フローグラフの節数に対して 2 次の計算時間がかかる可能性がある．Sreedhar らの方法は，どのような制御フローグラフでも線形時間で ϕ 関数を挿入できる．

我々は，上記の 2 つの SSA 変換アルゴリズムを実装し，以下の 3 種類のプログラムの変換時間を計測した．ここでの変換時間は，支配木が与えられた状態から SSA 形式を得るまでの時間，つまり前述の 2 つのフェーズの合計時間を指す．変数名の変更には，同じアルゴリズムを用いた．

general: プログラムの構造を if と while だけに限定したプログラム．大部分のプログラムは，このプログラムの制御フローグラフに近い形をしている [3].

do-while: do-while 文を入れ子にしたプログラム．

ladder: 制御フローグラフが梯子グラフになるようなプログラム．

実験環境は，CPU Athlon 650 MHz，メモリ 256 MB，OS linux 2.4.0 である．実装は Java 言語，Java の処理系は Sun Java2 SDK 1.3.0, HotSpot Client VM を用いた．実行時におけるヒープ領域獲得のコストを計測しないために，実行開始時にヒープ領域を 30MB 割り当てている．

図 5 と図 6 に実験結果を示す[1]．general の変換に

[1]図 5 と図 6 においてグラフが非連続なのは，ごみ集めの処理時間も計測しているためである．

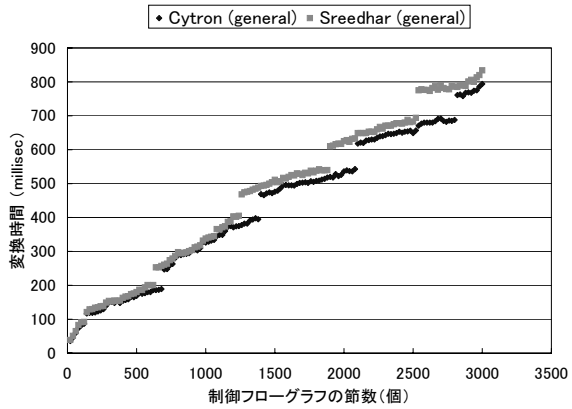

図 5: general の変換時間

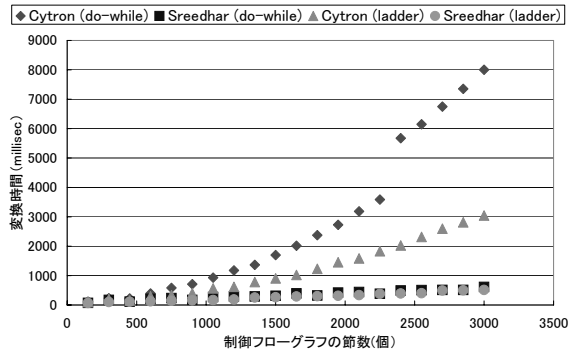

図 6: do-while と ladder の変換時間

3.2 SSA 形式最適化

SSA 形式を用いた最適化として，共通部分式除去，無用命令除去，定数伝播 [8] を実装した．

3.3 SSA 逆変換

SSA 形式に含まれる ϕ 関数は仮想的な関数であるので，目的コードを生成する前に消去して，通常形式に戻す必要がある．この作業を **SSA 逆変換**という．

我々が実装した SSA 逆変換アルゴリズムは [3] を参考にした．$v_0 = \phi(v_1, v_2, \ldots, v_n)$ という ϕ 関数を消去するために，各 i 番目の先行節の最後に $v_0 = v_i$ を挿入した（図 7）．ただし，ϕ 関数が存在する節とその先行節の間の辺がクリティカル辺 [4, 8] である場合は，その辺を分割している．

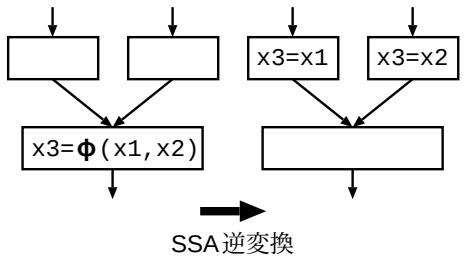

図 7: SSA 逆変換

おいて，Cytron らの方法が Sreedhar らの方法よりもわずかに速いことが分かった．この結果は Sreedhar らの主張と異なっている．また，do-while 文を入れ子にしたプログラムや制御フローグラフが梯子グラフになるプログラムにおいて，Cytron らの方法は 2 次の計算時間がかかるが，Sreedhar らの方法では線形時間で SSA 変換できることが，実際に確認された．

Cytron らの [3] では，do-while 文を入れ子にしたプログラムや制御フローグラフが梯子グラフになるプログラムにおいて，変数名の変更には，プログラムの大きさに対して 3 乗の計算時間がかかるといわれている．しかし，実験より線形時間で変数名の変更が可能であることが分かった．

Cytron らの方法は，ϕ 関数の挿入の前に支配辺境を求める必要があるが，その計算にかかる時間は，general で制御フローグラフの節数が 3000 個の場合でも 26 ms であった．支配辺境を求めるアルゴリズムは，最悪時に 2 次の振舞いをするが，かなり効率がよいことも分かった．

また，general の変換において，変数名の変更が変換時間の 60 〜 70 % を占めており，SSA 変換の効率を考える上で ϕ 関数の挿入の効率だけを比べることは重要ではないと考えられる．

4 今後の方針

今回，我々が実装したプロトタイプシステムは，サブセット C 言語を対象とし，行なう最適化も単純なものに限定していたため，本格的な実装の前に検討すべき多くの課題がある．

4.1 SSA 変換

Cytron らの方法や Sreedhar らの方法で求めた SSA 形式は，最小 SSA 形式 [3] と呼ばれる．最小 SSA 形式では，生存していない変数に対する ϕ 関数が挿入される可能性がある．このような無駄な ϕ 関数がない SSA 形式のことを pruned SSA 形式 [2] という．Pruned SSA 形式は最小 SSA 形式に比べて，SSA 形式での各種最適化の時間的および空間的効率が優れている．しかし，pruned SSA 形式を作るには，SSA 変換の前に生存変数解析を行なう必要があるため，生存変数解析によるコストが大きい．

両者の中間的な SSA 形式として semi-pruned SSA 形式 [1] も提案されている．Semi-pruned SSA 形式は，1 つの基本ブロックの中だけで定義と使用がなされている変数には ϕ 関数を作らないという SSA 形式である．Semi-pruned SSA 形式は，アルゴリズム

が簡単なため，求める際のコストが小さく，また無駄な ϕ 関数を減らす効果もかなりある．

3.1 節の最後で述べたように，変数名の書き換えのコストは大きい．しかし，無駄な ϕ 関数を挿入しなければ，書き換える変数の数が減るため，SSA 変換に必要な時間を短縮できると考えられる．

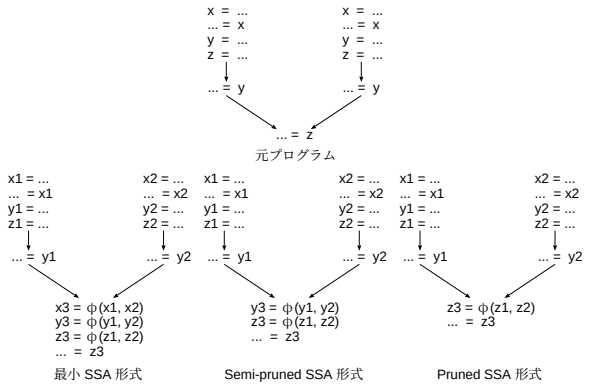

図 8: pruned SSA 形式と semi-pruned SSA 形式

今後の実装では，上記の 3 種類の SSA 形式を求められるようにする予定である．

プロトタイプではデータ型を整数型に限定していたが，より精密な SSA 形式最適化を行なうには，別名を扱う必要がある．今後の実装における SSA 形式上の別名情報の表現方法は，まだ検討段階である．

4.2 SSA 逆変換

SSA 形式上でコピー伝播などの最適化を行なった場合，3.3 節で述べた方法では，正しく ϕ 関数を除去することができない．たとえば，図 9 のような例が考えられる．図 9 の（a）を SSA 変換し，コピー

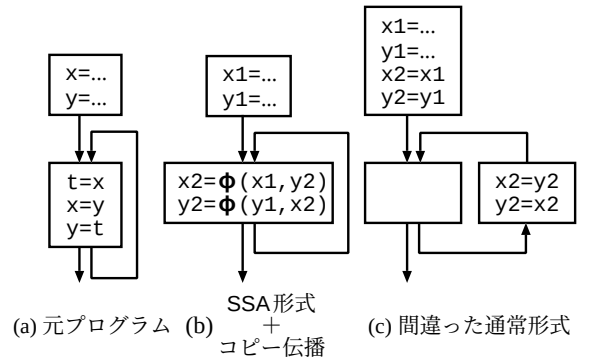

図 9: SSA 逆変換の危うい例

文を伝播させると（b）のような SSA 形式になる．SSA 形式において，同一の基本ブロック内に含まれる ϕ 関数は並列に実行されるとみなすので，（b）は正しい．ここで，（b）を 3.3 節で述べた方法で逆変換すると（c）のようになるが，プログラムの意味が変わってしまっている．このような例を正しく逆変換する方法はいくつか存在する [1, 4, 7]．検討の結果，それぞれに一長一短があると分かったので，今後は Sreedhar らによる逆変換法に拡張を加えたものを実装することにした．

5 おわりに

本稿では，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，および今後の実装方針について概説した．

今後は未決定の検討事項を熟考し，実装を進める予定である．

なお，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」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. *Softw. Pract. Exper.*, 28(8):859–881, July 1998.

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

[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 Trans. Prog. Lang. Syst.*, 13(4):461–486, October 1991.

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

[5] 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.

[6] V. C. Sreedhar and G. R. Gao. Computing ϕ-nodes in linear time using DJ graphs. *J. Prog. Lang.*, 3:191–213, 1995.

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

[8] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[9] 佐々木, 佐伯, 奥平, 廣田, 市川, 佐々. SSA 形式を中間言語とするコンパイラの属性文法による定式化と開発. 情報処理学会プログラミング研究会, 第 18–24 巻, pp. 177–182, March 1998.

[10] 中谷俊晴. 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 学士論文研究, 2001. Available from http://www.is.titech.ac.jp/˜nakaya8/papers/.