

静的単一代入形式を用いた最適化（導入編）

佐々 政孝 滝本 宗宏

コンパイラでは、機械語の目的コードを生成するだけでなく、実行させたときにその目的コードが効率良く実行できるように、様々な変換を行う。これを「最適化」という。最適化の方法としては、従来は中間表現の上で変数の定義と使用のデータの流れの解析を行う方法が使われていたが、最近では静的単一代入形式というものをを用いた最適化の方法が注目を浴びている。静的単一代入形式（SSA 形式）は、すべての変数の使用に対して、その値を定義（代入）している場所がテキスト上 1 箇所しかないように変数の名前替えをした中間表現の形式である。この性質を利用することにより、いろいろな最適化が見通し良く、容易に行えるようになる。本稿（導入編）では、SSA 形式のあらまし、通常形式から SSA 形式への変換、SSA 形式から通常形式への逆変換について、解説する。

In compilers, in addition to generating the object code in machine language, various transformations are made to improve the efficiency of the execution of the object code. This is called *optimization*. Previous methods for optimization generally use the data flow analysis of variable definitions and variable uses on the intermediate code. However, optimization methods using the *static single assignment form (SSA form)* attract compiler writers' recent attention. An SSA form is an intermediate representation, in which all variables are renamed so that for all uses of each variable, the point defining (assigning) its value is made unique in the program text. Various optimizations can be written clearly and easily utilizing this property. In this paper (introduction part) we explain the outline of SSA form, translation from normal form to SSA form, and back translation from SSA form to normal form.

1 はじめに

コンパイラでは、機械語の目的コードを生成するだけでなく、実行させたときにその目的コードが効率良く実行できるように、様々な変換を行う。これを最適化という。たとえば、ループの中で実行しなくても良い命令はループの外に出す、同じ計算は省略する、などの変換を行って目的コードの効率を向上させる手法がよく知られている。最適化は、現在のコンパイラ

で力が注がれている重要な技術の一つである。

最適化の方法としては、従来は変数の定義と使用に関してデータフロー解析（データの流れの解析）と呼ばれる方法が使われていたが、最近では静的単一代入形式というものをを用いた最適化の方法が注目を浴びている。

静的単一代入形式（Static Single Assignment Form, 以下 SSA 形式と呼ぶ）は、すべての変数の使用に対して、その値を定義（代入）している場所がプログラム上 1 箇所しかないように変数の名前替えをした中間表現の形式である。通常の間表現では変数の使用に対してその値を定義している場所が複数個あり得るのだが、SSA 形式では、各変数の定義がプログラム上で 1 箇所しかない。この性質を利用することにより、いろいろな最適化が見通し良く、容易にできるようになる。実際の例は 2 節や発展編で述べる。

Optimization in static single assignment form - introduction part

Masataka Sassa, 東京工業大学大学院情報理工学研究所, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

Munehiro Takimoto, 東京理科大学理工学部, Faculty of Science and Technology, Tokyo University of Science.

コンピュータソフトウェア, Vol.16, No.5 (2000), pp.78-83. [解説論文] 2000 年 8 月 3 日受付.

SSA 形式を利用した最適化を静的単一代入形式最適化 (SSA 最適化) という。SSA 最適化では最適化の処理効率もほとんどの場合に向上する。このためいくつかの最適化コンパイラ [10][11][12][19] が最適化パスの一部で SSA 形式を採用するようになってきている。もちろん最適化パスの別の箇所でも通常形式を用いた最適化を行うことも多い。

導入編では、SSA 形式、SSA 形式への変換と逆変換について、発展編では、SSA 形式上での最適化について、いずれも例を挙げながら述べる。

2 SSA 形式

2.1 通常形式と SSA 形式

SSA 形式は本来中間表現レベルのものであり、次のような機能をもつ中間言語が SSA 形式で扱える。

- 単純変数
- 手続き
- 代入文、入出力文、無条件および条件付き飛越し文、手続き呼び出し文、return 文

これはあくまでも中間言語での話であるので、ソースプログラムに立ち戻って考えると、ほとんどの制御文 (if, if...else, switch, while, do...while, break, continue, goto) が扱える。

ただし、配列変数については、その SSA 形式の研究はあるが [15]、一般的ではないので、ここでは扱わない。また、ポインタについては注意を要する。ポインタにより指される変数が一意に定めれば、そのような変数を SSA 形式にすることができるが、そうでない場合は、SSA 形式にするのは困難である。なお、ポインタそれ自身は SSA 形式にすることができる。

以下では、変数は単純変数だと仮定して説明する。

SSA 形式とは、プログラム上のすべての変数の使用に対して、その使用に対する定義が 1 箇所しかないように表現した中間表現形式である。SSA 形式では、変数の定義が字面上、つまりプログラムのテキスト上で唯一になる。この形式は静的に (つまりプログラムのテキスト上で) 単一代入なので、静的単一代入形式と呼ばれる。一方、SSA 形式でないふつうの形式を通常形式と呼ぶことにする。

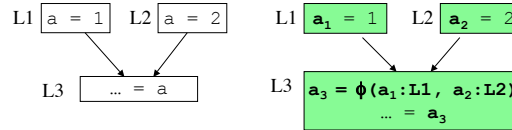
```
a = x + y
a = a + 3
b = x + y
```

(a) 通常形式

```
a1 = x0 + y0
a2 = a1 + 3
b1 = x0 + y0
```

(b) SSA形式

図 1 SSA 形式での変数名の付け方



(a) 通常形式

(b) SSA形式

図 2 SSA 形式での ϕ 関数

SSA 形式のポイントは 2 点ある。1 つ目は定義される変数に唯一の名前をつけること、2 つ目は ϕ 関数と呼ばれるものである。

1 つ目のポイントは、すべての変数の使用に対して、その値を定義 (代入や読み込みなど) している場所が 1 箇所しかないようにするために変数に唯一の名前をつけることである。たとえば、図 1(a) のような通常形式のプログラムがあったとする。

図 1(a) に対する SSA 形式は図 1(b) のようになる (SSA 形式は本来は中間表現形式であるが、わかりやすさのため、以後ソースプログラムの形式で表す)。代入があるごとに代入の左辺に現れる変数に新しく唯一な変数名を付け、 a_1 , a_2 のように区別する。唯一な変数名をつける際は、慣例として、バージョン (version) 番号と呼ばれる 1, 2 などの添字をつけることが多いが、 a_{-1} , a_{-2} とか a_1 , a_2 などとしてもよい。代入文の右辺で変数を使用する場合は、その変数がどの代入文で定義されたものであるかを探して、対応する代入文の左辺の (唯一の) 変数名を記す。たとえば、図 1(b) の 2 行目の右辺で使用している a は図 1(a) の 1 行目の代入文で定義した a なので、 a_1 とする。

SSA 形式での 2 つ目のポイントは、 ϕ 関数 (ファイカンすう, phi function) と呼ばれるものである。これは、制御の合流点 (後述) に、通常形式で同じ変数

であった変数の異なるバージョンが到達する（使用可能となる）ときに用いられる．例を図 2 に示す． ϕ 関数を用いる理由は，図 2(a) のプログラムを SSA 形式に変換しようとするとき，基本ブロック L3 で使用している a に対応する定義を一意に決めることができないからである．

ちなみに，基本ブロックとは，リーダーという概念を用いて以下で定義されるものである [1] [2] [14] [16]．プログラムの最初の文，無条件あるいは条件付き飛越しの行き先の文，無条件あるいは条件付き飛越しの直後の文をリーダーという．

- リーダーからはじまり，次のリーダーの 1 つ前まであるいはプログラムの最後までの一連の文を基本ブロックという．

さて，前述の問題点を解決するため，SSA 形式では図 2(b) のような ϕ 関数を導入する「 $\phi(a_1:L1, a_2:L2)$ 」は，基本ブロック L1 から来たときは a_1 の値を返し，基本ブロック L2 から来たときは a_2 の値を返す（仮想的な）関数を表す．これにより，図 2(b) の L3 の最後の行の a_3 が， ϕ 関数で定義された唯一の a_3 を参照するようになる．

なお， ϕ 関数の記述の煩雑さを避けるため，「 $a_3 = \phi(a_1:L1, a_2:L2)$ 」を単に「 $a_3 = \phi(a_1, a_2)$ 」と略記することも多い．この記法は， ϕ の第 1 オペランドが図での左上から来ること，第 2 オペランドが図での右上から来ること，を仮定している．

SSA 形式について，一つ注意を述べる．SSA 形式での各変数は，字面上あるいはプログラムのテキスト上で定義が唯一であるが，「値」が唯一である訳ではない．たとえば図 1(b) がループの中にある場合を考えると， a_1, a_2, b_1 などの「値」はループを回るとに変わりうる．前述のように，静的単一代入形式の「静的」とは「プログラムのテキスト上で」という意味で，動的（実行時）に値が唯一である訳ではない．図 2(b) の例でも， a_3 の「値」が実行時に唯一になる訳ではないことは明らかであろう．

2.2 SSA 形式の利点

SSA 形式を用いると，プログラムの各変数の使用に対応する定義が 1 箇所だけになるので，変数の使用

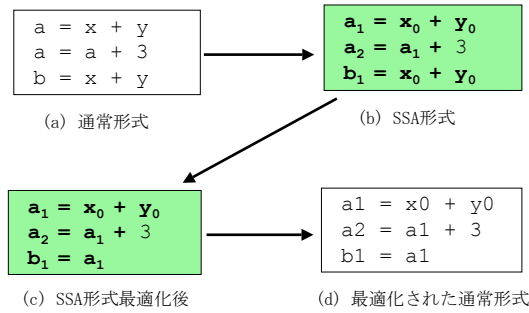


図 3 SSA 形式での最適化（共通部分式除去）

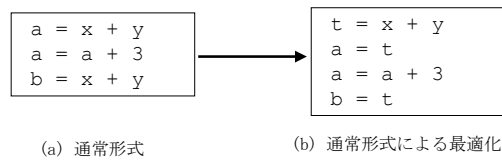


図 4 通常形式による最適化（共通部分式除去）

と定義の関係が明確になる．通常形式におけるような変数の定義と使用の関係の解析が不要となるので，SSA 形式では最適化の実現が容易となる．また最適化の処理効率が向上する．その例を図 3 に示す．

図 3(a) の通常形式から (b) の SSA 形式への変換は前述の通りである．図 3(b) を見ると，3 行目の「 $x_0 + y_0$ 」が 1 行目の右辺と同じであることがわかる．つまり，この 2 つは共通部分式である．そこで，3 行目の「 $x_0 + y_0$ 」を 1 行目の左辺の「 a_1 」で置き換えると，図 3(c) が得られ，共通部分式除去がなされる．これを後述の SSA 逆変換により通常形式に戻すと，図 3(d) が得られる．

参考として，通常形式での共通部分式除去の最適化の例を図 4 に示す．SSA 形式を使わずに，図 4(a) の通常形式のまま共通部分式除去を行おうとすると，図 4(a) の 1 行目で代入された a の値が 2 行目で上書きされているので，簡単な処理では済まなくなる．ふつうは，図 4(a) の 1~3 行目間で x, y の値が書き換えられないことを確認してから，一時変数 t を導入し，図 4(b) のような最適化がなされる．この



図5 SSA形式を用いた最適化の流れ

ように通常形式でも同様な最適化を行うことはできるが、SSA形式で行った方が最適化の処理が容易となり、最適化アルゴリズムや実装の誤りも少なく、またその結果、最適化処理の効率も一般に向上する。

一方、SSA形式を用いると、通常形式では容易には行えなかった「条件分岐を考慮した定数伝播」が行えたり、通常形式では処理が複雑になる「値番号付けを含む共通部分式除去」が簡単に行える。詳しくは「発展編」[22]で述べる。

ただし、前述のとおり、SSA形式最適化には不得手な分野もある。たとえば、配列の扱いやポインタによる別名の処理、などSSA形式最適化の技法がまだ確立されていない分野もある。SSA形式にするよりはソースプログラムに近い中間コード上で行ったほうが良いループ展開などの最適化もある。

以上の例で示したように、SSA形式を用いた最適化はふつう最適化のパスで図5のような流れで利用される。

3 SSA変換

3.1 SSA変換

SSA変換とは、通常形式からSSA形式に変換することである。たとえば、図1(a)をSSA変換すると図1(b)になり、図2(a)をSSA変換すると図2(b)になる。

図1や図2は簡単な例であったが、プログラムの流れが複雑になると、どこに ϕ 関数を挿入すればよいか、変数にどのようにバージョン番号(添字)をつければよいか、などのアルゴリズムが必要となる。

少し準備をする。前提として、SSA形式では、未定義変数が存在するときの扱いの複雑さを避けるために、プログラムの入口にすべての変数の初期化の文

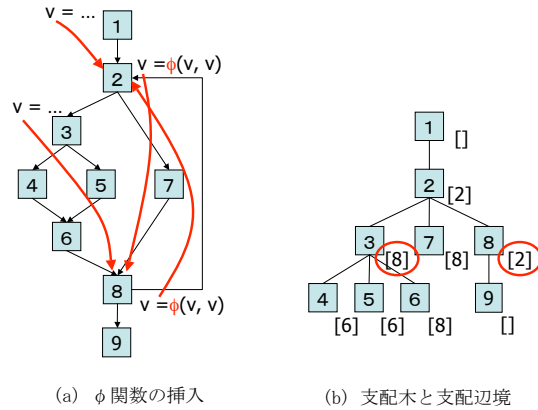


図6 SSA変換と支配境界

を追加して考えることになっている。

また、制御フローグラフというものを定義する。これは、基本ブロック(ブロックともいう)を頂点とし、ブロック間の制御の流れの関係を有向辺で表したものである。詳しくは、ブロック B_1 からブロック B_2 に有向辺が引かれるのは、次の場合である[1][16]。

1. B_1 の最後の文から B_2 の最初の文へ無条件あるいは条件付き飛越しがある。
2. B_1 の最後が無条件飛越し文以外の文で終わっていて、プログラムの字面上で B_1 の直後に B_2 が来る。

このとき、 B_1 を B_2 の先行ブロックという。

図6(a)は、制御フローグラフの例である。

2つ以上の先行ブロックのあるブロックを制御の合流点という。たとえば、図6(a)のブロック2, 6, 8は制御の合流点である。

ここで、 ϕ 関数の挿入について、少し複雑な例で見てみることにする。例を図6に示す。図6(a)は、 ϕ 関数の挿入場所を示す。ブロック3に v の定義(代入文)があり、かつブロック7を通ってくる v の定義がある(少なくともプログラムの入口に v の初期化の定義がある)ので、その2つの定義がブロック8で合流する。よって、ブロック8に v に対する ϕ 関数を挿入する必要がある。ブロック8に v に対する ϕ 関数が挿入されたので、さらにこのブロック8の v の定義とブロック1の v の定義がブロック2で合流

することとなり、その結果、ブロック 2 にも ϕ 関数を挿入する必要が生じる。こうして、図 6(a) のように ϕ 関数が挿入される。

上で述べた ϕ 関数の挿入場所は、支配境界というブロックを計算することによって求めることができる。支配境界について詳しく述べるには、まず支配関係というものの定義が必要である。プログラムの入口からブロック B_2 に達するどの路も必ずブロック B_1 を通るとき、ブロック B_1 はブロック B_2 を支配する (dominate) という。ブロック B は自分自身を支配することになっている。

また、支配関係は反射的かつ推移的であり、木の形で書くことができることが知られている。この木では、親のブロックが子孫のブロックを支配する。この木を支配木 (dominator tree) という。たとえば、図 6(a) の支配木を図で表すと図 6(b) のようになる。たとえば、ブロック 3 は、ブロック 3, 4, 5, 6 を支配し、ブロック 2 は、ブロック 2, 3, 4, 5, 6, 7, 8, 9 を支配する。

ブロック i の支配境界 (dominance frontier) とは、制御フローグラフ上で、初めて i の支配関係から「はずれた」ブロック j のことである。

正確には以下のとおりである。支配境界の計算では、ブロック X 内のコードが、ブロック Y の入口を支配するかどうかが問題になる。ここで、 $X=Y$ の場合を考えると、幾つかの辺を通らずに X 内のコードが Y の入口を支配することはない。そこで、支配関係から反射律を除いた厳密な支配という概念を用いると便利である。ブロック X がブロック Y を支配し、 $X \neq Y$ であるとき、 X は Y を厳密に支配するという。

ブロック X の支配境界 $DF(X)$ は次の式で定義される。

$$DF(X) = \{ Y \mid Y \text{ の先行ブロック } U \text{ が存在し、} \\ X \text{ は } U \text{ を支配し、} X \text{ は } Y \text{ を厳密に支配はしない} \}$$

たとえば、図 6(a) のブロック 3 は、ブロック 3, 4, 5, 6 を支配しているが、その後続ブロックのブロック 8 は厳密に支配していないので、ブロック 3 の支配境界はブロック 8 となる。図 6(b) では、各ブロックについて、そのブロックの支配境界を [] で囲んで示した。たとえば、ブロック 3 の支配境界はブロック 8 で

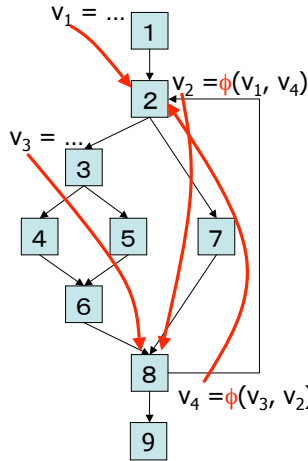


図 7 変数の名前替えの結果

ある。

さて、話を ϕ 関数の挿入場所に戻すと、 ϕ 関数を挿入すべきブロックは、各定義のあるブロックの支配境界であることが知られている。

これをインフォーマルに説明すると、支配境界の直前までは今考えている変数の定義が支配しており、その範囲では、その定義の値は不変であるが、支配境界では、別の道を通ってきた同じ変数の定義が必ず到達するので、 ϕ 関数を挿入するのである (詳しく言うと、SSA 形式では、プログラムの入口に各変数の初期化の文を追加して考えることになっているので、他に同じ変数の定義がなくてもその初期化の定義が支配境界に到達する)

また、 ϕ 関数を新たに挿入した場合は、新たに ϕ 関数を挿入したブロックの支配境界にも ϕ 関数を挿入することになる。

たとえば、図 6(a) では、 v の定義があるブロック 1 の支配境界はないので、これについての ϕ 関数は挿入しない。 v の定義があるブロック 3 の支配境界はブロック 8 なので、ブロック 8 に ϕ 関数を挿入する。さらにこの結果、ブロック 8 の支配境界であるブロック 2 にも ϕ 関数を挿入する。

ここまでで、 $v = \phi(v, v)$ の形の ϕ 関数が挿入された。なお、ここで述べた方法は後述の最小 SSA 形式というものを作成する方法である。

最後に、挿入した ϕ 関数の左辺や右辺に現れる変数 v にバージョン番号をつける (添字づけ) 作業を行う。これを変数の名前替えという。図 6 の例に変数の名前替えを施すと図 7 となる。この処理は紙面の関係で省略するが、詳しくは、[2][6][7] ([7] はわかりやすい) [14]などを参照されたい。

3.2 SSA 変換の他の方法

SSA 変換には、他の方法もある。支配木に J-edge という辺を追加した DJ グラフというデータ構造を用いる Sreedhar らの方法 [20] はその一つである。Das らの方法 [9] はそれにさらに改良を加えたもので、やや複雑ではあるが、実験によると、本稿で紹介した方法よりも変換時間が SPEC2000 ベンチマークで 2, 3 割短い。

一方、Bilardi らは、Sreedhar らの方法を含む種々の SSA 変換のアルゴリズムを merge 関係という概念を用いて整理し、それぞれの位置づけを示した [3]。一部のアルゴリズムについては、実験結果も示している。

3.3 SSA 形式が満たす重要な性質

前節では SSA 形式への変換法を示したが、SSA 形式が満たす重要な性質は次のとおりである [8]。SSA 形式への変換はもとのプログラムを同じ制御フローグラフをもった新しいプログラムに置き換える。ここでは、もとのプログラムの各変数 v に対して以下の条件が成り立つ (ここでは、制御フローグラフとして、各文がひとつのブロックをなすものを考える。このブロックをノードと呼ぶ。)

1. もし 2 つの空でないパス (制御フローグラフのノードからノードをたどって別のノードへ至る道のこと) $X \rightarrow Z$ と $Y \rightarrow Z$ がノード Z で合流し、ノード X と Y が元のプログラムで v の定義を含むとき、 $v_{i_0} = \phi(v_{i_1}, \dots, v_{i_n})$ が、変換後のプログラムで Z に挿入されている。
2. 元のプログラムや挿入された ϕ 関数での v の参照は、SSA 形式での新変数 v_i の参照に置き換えられている。
3. 制御フローの任意のパスに沿って、元のプログ

ラムの変数 v の任意の使用と変換されたプログラムでの対応する v_i の使用を考えると、 v と v_i は同じ値をもつ。

3.4 3 種類の SSA 形式

SSA 形式を細かく分類すると、最小 (minimal) SSA 形式、半ば刈り込んだ (semi-pruned) SSA 形式、刈り込んだ (pruned) SSA 形式、の 3 種類に分けられる [4]。この 3 種類の例を図 8 に示す。

図 8(b) は、最小 (minimal) SSA 形式と呼ばれ、定義された変数の合流点にすべて ϕ 関数を挿入する。前節で述べたアルゴリズムも、最小 SSA 形式への変換であった。最小 SSA 形式は作成は楽だが、以下で述べるように無駄な ϕ 関数が挿入されることが多い。

これに対し、図 8(d) は、刈り込んだ (pruned) SSA 形式と呼ばれ、生存している変数に対してしか ϕ 関数を挿入しない。

ここで、変数 v が制御フローグラフ上のブロック内のポイント (文と違ってよい) p で生存しているとは、 p から変数 v の使用に至り、そのパス上で v が再定義されていないような制御フローグラフ上のポイントを結ぶパスがあることである [2][1][7]。

たとえば、図 8(d) の一番下の基本ブロックでは、 x_3 や y_3 は生存していないので、これらに対する ϕ 関数を挿入しない。刈り込んだ SSA 形式では、無駄な ϕ 関数が現れないため、 ϕ 関数の数は 3 種類の SSA 形式のうちでもっとも少なく、このあとに最適化などの処理をするときの時間が少なくてすむ。

しかし、刈り込んだ SSA 形式を作るためには、生存変数の解析を行わなければならない、SSA 変換に多少手間がかかる。刈り込んだ SSA 形式を作る手間を少し減らしたのが、半ば刈り込んだ (semi-pruned) SSA 形式で、例を図 8(c) に示す。これは、生存区間が定義した基本ブロックを超えるような変数についてのみ ϕ 関数を挿入する。

ここで、変数の生存区間とは、その変数が生存し始めるポイントから生存しなくなるポイントまでを結ぶ区間をすべて集めたものである。その変数の定義や使用が複数あるときは、必ずしも一直線のものにはならず、合流や分岐のある区間となることがある。

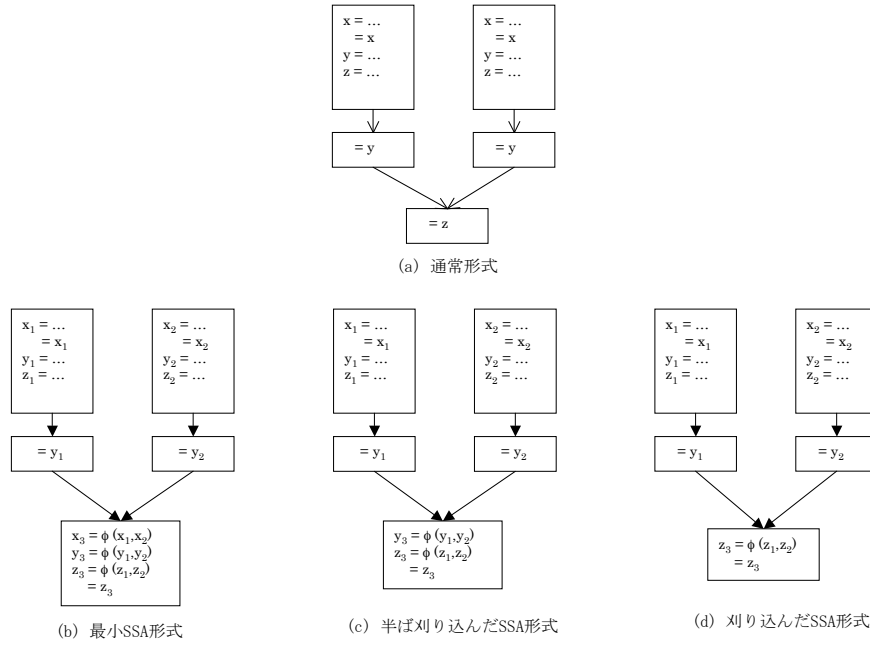


図 8 最小 SSA 形式，半ば刈り込んだ SSA 形式，刈り込んだ SSA 形式

図 8 に戻るが，たとえば，図 8(a) の左上の基本ブロックの y の生存区間は，文「 $y =$ 」からその下の基本ブロックの「 $= y$ 」までであり，生存区間は，定義された基本ブロックからその次の基本ブロックまで伸びている．そこで，半ば刈り込んだ SSA 形式では， y について ϕ 関数を挿入する．

半ば刈り込んだ SSA 形式は，その後の最適化などの処理が少ないときに，SSA 変換と合計した処理時間をほどよく少なくするのに有効である．

4 SSA 逆変換

SSA 逆変換とは，SSA 形式から通常形式に戻す変換のことである．一般に，目的機械には ϕ 関数に相当する命令はないので，SSA 形式から直接コードを生成することはできない．そこで，SSA 形式で最適化を行ったあとには，SSA 逆変換が必要になる．SSA 逆変換については，まとまって述べられている教科書やサーベイがほとんどないため，少し詳しく述べることにする．

主な SSA 逆変換法には，Briggs らの方法と Sreed-

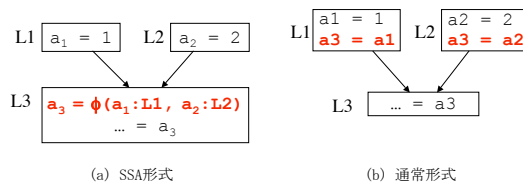


図 9 素朴法による SSA 逆変換

har らの方法があるが，その前に最初に提案された素朴な方法とその問題点を示す．

4.1 素朴な SSA 逆変換の方法とその問題点

Cytron らが最初に提案した SSA 逆変換の方法 [8] は，素朴な方法である．それを図 9 に示す．

素朴法では， ϕ 関数のあった基本ブロックの先行ブロックに， ϕ 関数に対応するコピー文を挿入し， ϕ 関数およびその左辺を消去する．以後， ϕ 関数およびその左辺のことを「 ϕ 命令」と呼ぶ．

この例では，L3 の

$$a_3 = \phi(a_1:L1, a_2:L2)$$

とは，制御が L1 から来たときは a_3 の値は a_1 ，L2 か

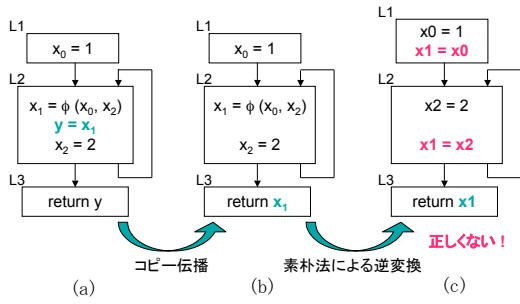


図 10 素朴法による SSA 逆変換の問題点 (lost copy 問題)

ら来たときは a_3 の値は a_2 , という意味だったので、それに相当する文を図 9(b) のそれぞれの先行ブロックに挿入し、 ϕ 命令を消去している。

しかし、素朴法には次のような問題点があることが知られている。

例として、図 10 を見られたい。図 10(a) はいくつかの最適化を施した後の正しい SSA 形式である。図 10(a) にコピー伝播を施すと、図 10(b) が得られる。これも正しい SSA 形式である。図 10(b) に素朴な SSA 逆変換を適用すると図 10(c) が得られるが、この結果は正しくない。なぜなら、図 10(a) や図 10(b) では、ブロック L3 での「return y」の値は、ループを 1 回しか回らなかったときに 1、2 回以上回ったときに 2 になるが、図 10(c) では、ブロック L3 での「return y」の値は、ループを回る回数に関わらず常に 2 になるからである。

4.2 Briggs らによる SSA 逆変換の方法

Briggs らの方法 (以下 Briggs 法) [4][6] は、素朴法の問題点を修正した方法の 1 つである。これは、素朴法と同様に、 ϕ 関数のあった基本ブロックの先行ブロックに、 ϕ 関数に対応するコピー文を挿入し、 ϕ 命令を消去する。ただし、そのままでは図 10 のような問題が起こるので、次のようにアルゴリズムを修正している。

Briggs 法による SSA 逆変換の例を図 11 に示す。図 11(a) は、図 10(b) と同じ SSA 形式である。

図 10(c) の素朴法による SSA 逆変換が正しくない

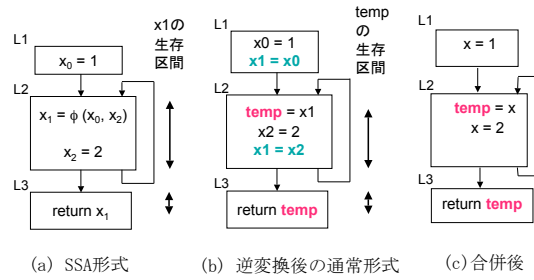


図 11 Briggs 法による SSA 逆変換

原因は、 x_1 の生存区間の中で x_1 への代入を行ったからである。わかりやすさのため、図 11(a) の右側に x_1 の生存区間を示した。

この問題を避けるため、Briggs 法では、 x_1 の値をいったん temp という一時変数に保存しておいてから、 x_1 の生存区間の中で ϕ 関数に対応するコピー文「 $x_1=x_2$ 」を挿入する。そして、もともと x_1 を使用していたブロック L3 の「return x_1 」では、 x_1 の代わりに、保存しておいた temp を使うようにする。これにより、図 11(b) は正しい通常形式となる。

これで問題は一応解決したが、図 11(b) にはコピー文が多数挿入され、効率が悪くなりそうである。Briggs らは、その後のレジスタ割り当てフェーズで合併 (coalescing) (コピー文で結ばれた変数を一つの変数にまとめ、コピー文を削除すること。[2][7][14] 参照) を行うことで、これらのコピー文は合併され、図 11(c) と同等なものが得られるので、これでかまわないと主張している。これは必ずしも正しくないのであるが、詳しい解析は[13]を見られたい。

なお、Briggs 法は、基本は素朴法であるので、図 9(a) に Briggs 法の逆変換を適用すると、素朴法と同じ図 9(b) が得られる。ここでも、コピー文が挿入されていることがわかる。

4.3 Sreedhar らによる SSA 逆変換の方法

Sreedhar らによる SSA 逆変換の方法 (以下 Sreedhar 法) [21] は、Briggs 法や素朴法とはまったく違った考え方に基づいている。

基本的な考え方は、「通常形式から SSA 形式に変換した直後の SSA 形式」を SSA 逆変換するには、

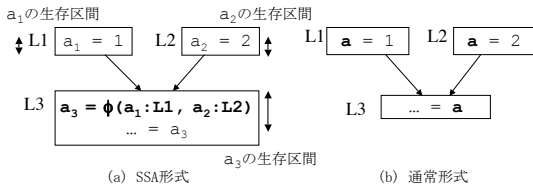


図 12 Sreedhar 法による SSA 逆変換の考え方

$a_{i_0} = \phi(a_{i_1}, a_{i_2}, \dots, a_{i_n})$ の形の ϕ 関数の引数にある変数 $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ とその ϕ 関数の左辺にある変数 a_{i_0} をすべて単一の変数に置き換え、 ϕ 命令を消去すればよい。たとえば、前述の図 2(b) と同じ図 12(a) は、SSA 形式に変換した直後の SSA 形式なので、 ϕ 関数とその左辺にある変数 a_3, a_1, a_2 をすべて単一の変数 a に置き換え、 ϕ 命令を消去すれば、図 12(b) の通常形式が得られることになる。図 12(b) は、前述の図 2(a) と同じものである。このように、SSA 変換直後の SSA 形式を Sreedhar 法により SSA 逆変換すると、もとの通常形式とまったく同じものが得られる。

さて、図 12 の例だけを見ると、Sreedhar 法は簡単に見えるが、実は、 ϕ 関数の引数とその ϕ 関数の左辺にある変数をすべて単一の変数に置き換えることができるためには、ある条件を満たしていないといけない。つまり、一般の SSA 形式は「通常形式から SSA 形式に変換した直後の形」にはなっていないので、上記の方法はそのままでは使えず、工夫が必要である。

そこで、まず用語の定義だが、 ϕ 関数の引数にある変数とその ϕ 関数の左辺にある変数の集合を phi congruence class という。1 つの変数が複数の ϕ 命令に含まれる場合は、phi congruence class はその和集合とする。各 phi congruence class に含まれるすべての変数を 1 つの代表変数に置き換え、 ϕ ノードを消去すると、それと等価な通常形式が得られるような形式を CSSA (Conventional SSA) 形式 [21] と呼ぶ。紙面の関係で深入りは避けることとするが、通常形式から SSA 形式に変換した直後の SSA 形式は CSSA 形式である。

CSSA 形式の特徴は、「 ϕ 関数とその左辺」に現れる変数の生存区間の間に重なりがないこと」である。

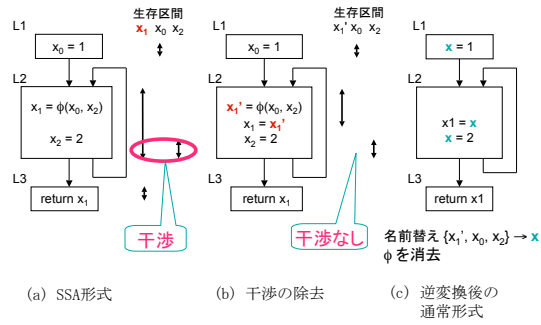


図 13 Sreedhar 法による SSA 逆変換

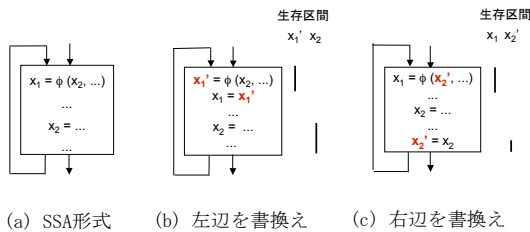
ただし、 ϕ 関数の変数の生存区間については次のように特別な定義をする。 ϕ 関数の左辺の変数については、その ϕ 関数が存在するブロックの先頭から生存区間が始まるとする。 ϕ 関数の引数については、その引数が由来する先行ブロックの最後までが生存区間だとする。

たとえば、図 12(a) で、 a_3, a_1, a_2 の生存区間を表すと、縦の実線のようになる。 a_2 の生存区間がブロック L2 の最後で終わっていること、 a_3 の生存区間がブロック L3 の先頭から始まることに注意されたい。

この図 12(a) では、 ϕ 関数 $a_3 = \phi(a_1, a_2)$ の引数 (ここでは左辺も引数に含める) である、 a_3, a_1, a_2 の生存区間の間に重なりがない。このような場合は、前述のとおり、 ϕ 関数の引数 a_3, a_1, a_2 を一つの変数 a に置き換え、 ϕ 命令を消去することで通常形式が得られる。

複数の変数の生存区間の間に重なりがあるとき、それらの変数の生存区間の間には干渉があるという。「 ϕ 関数とその左辺」に現れる変数の生存区間に干渉がある一般の SSA 形式を TSSA (Transformed SSA) 形式 [21] と呼ぶ。一般に、SSA 形式に変換したあとに最適化を施すと、CSSA 形式ではなくなり、TSSA 形式になることが多い。

Sreedhar 法の要点は、TSSA 形式で生じている ϕ 関数に関わる干渉を取り除き、CSSA 形式に変換する点にある。具体的には、CSSA 形式の条件が満たされないときは、いくつかの場合分けに沿って、 ϕ 関数を書き換えて新たなコピー文を挿入する、という操作を行う。これにより、CSSA 形式にすることができ



る．そこで、あとは ϕ 関数の左辺と引数を同一の変数に置き換えれば通常形式が得られる．

図 13 を例として説明する．図 13(a) では、 ϕ 関数 $x_1 = \phi(x_0, x_2)$ の引数（前述のとおりここでは左辺も引数に含める）である、 x_1, x_0, x_2 の間に生存区間の干渉がある．具体的に、この例では、 x_1 と x_2 の間に干渉がある．Sreedhar 法では、 ϕ 関数を書き換えることによって、 ϕ 関数の引数の間の干渉を取り除く．この例では、図 13(a) の「 $x_1 = \phi(x_0, x_2)$ 」を「 $x_1' = \phi(x_0, x_2); x_1 = x_1'$ 」に書き換えると、図 13(b) のようになり、新しい ϕ 関数の引数である、 x_1', x_0, x_2 の間の干渉がなくなる．そこで、 x_1', x_0, x_2 をすべて一つの変数 x に置き換えると図 13(c) の通常形式が得られる．

この図 13(c) には、Briggs 法にあったような無駄なコピー文がないことに留意されたい．Sreedhar 法は、Briggs 法におけるような合併を行わずとも、直接に、無駄なコピー文のない通常形式が得られることが利点である．

一般に ϕ 関数の左辺と引数の間に干渉がある場合、その ϕ 関数の書き換え方を図 14 に示す．図 14(a) において、左辺の変数が ϕ 関数の引数と干渉している場合は、図 14(b) のように左辺の変数を書き換える．図 14(a) において、 ϕ 関数の引数が左辺や他の引数と干渉している場合は、図 14(c) のように ϕ 関数の引数を書き換える．これらのうちのどの書換えを行えばよいかについては、 ϕ 関数の左辺と引数（正確には phi congruence class）についてそれぞれ生存区間の情報を使った 8 通りの場合分けをおこなう一部非決定的なアルゴリズムが与えられているが、詳しくは、[21][6] に記載されている．

5 おわりに

静的単一代入形式 (SSA 形式) について、SSA 形式のあらまし、通常形式から SSA 形式への変換、SSA 形式から通常形式への逆変換、について述べた．発展編では、SSA 形式最適化の例について述べる予定である．

なお、COINS コンパイラ・インフラストラクチャでの SSA 変換や SSA 逆変換の実際については、[17]、[18] に掲載されているので、実際の具体例で確かめてみたい場合は、そちらを参照されたい．

謝辞

コメントをいただいた COINS グループの各位と査読者に感謝する．

参考文献

- [1] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D.: *Compilers Principles, Techniques, & Tools*, second ed., Addison Wesley, 2007.
- [2] Appel, A.: *Modern Compiler Implementation in Java*, second ed., Cambridge University Press, 2002.
- [3] Bilardi, G. and Pingali, K.: Algorithms for Computing the Static Single Assignment Form, *J. ACM*, Vol. 50, No. 3, pp. 375-425, 2003.
- [4] Briggs, P., Cooper, K., Harvey, T. and Simpson, T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form, *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859-881, 1998.
- [5] COINS: 並列化コンパイラ向け共通インフラストラクチャ, <http://www.coins-project.org/>.
- [6] COINS: 静的単一代入形式に基づく最適化に関する研究, <http://www.is.titech.ac.jp/%7Esassa/coins-www-ssa/japanese/index.html>.
- [7] Cooper, K. and Torczon, L.: *Engineering a Compiler*, Morgan Kaufmann, 2003.
- [8] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 4, pp. 451-490 (1991).
- [9] Das, D. and Ramakrishna, U.: A Practical and Fast Iterative Algorithm for ϕ -Function Computation Using DJ Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol. 27, No. 3, pp. 426-440, 2005.
- [10] Fitzgerald, R., Knoblock, T. B., Ruf, E., Steensgaard, B. and Tarditi, D.: Marmot: An Optimizing Compiler for Java, *Software — Practice and Experience*, Vol. 30, No. 3, pp. 199-232 (2000).

- [11] GCC Homepage. <http://gcc.gnu.org/>.
- [12] IBM: Jikes Research Virtual Machine. <http://jikesrvm.sourceforge.net/>.
- [13] 伊藤陽, 小濱真樹, 佐々政孝: 静的単一代入形式からの逆変換アルゴリズムの比較と評価, 情報処理学会論文誌: プログラミング, Vol. 46, No. SIG 14 (PRO 27), pp. 30-42, 2005.
- [14] 中田育男: コンパイラの構成と最適化, 朝倉書店, 1999.
- [15] Rus, S., He, G., Alias, C., Rauchwerger, L., Region Array SSA, *In Proc. PACT '06*, pp. 43-52, 2006.
- [16] 佐々政孝: プログラミング言語処理系, 岩波書店, 1989.
- [17] 佐々政孝: コンパイラ・インフラストラクチャCOINSを用いた SSA 最適化 (その 1) および (その 2), 情報処理, Vol. 47, No. 8, pp. 907-913, (2006) および, 同, Vol. 47, No. 9, pp. 1032-1038, (2006).
- [18] 佐々政孝: デモの仕方. <http://www.coins-project.org/> より「静的単一代入最適化部」を辿り, さらに「21世紀のコンパイラ道しるべ - COINS をベースとして 情報処理学会誌の連載 - SSA 最適化部」を辿る.
- [19] Scale Compiler Group: Scale homepage. <http://www-ali.cs.umass.edu/Scale/>.
- [20] Sreedhar, V. C. and Gao, G. R.: A Linear Time Algorithm for Placing ϕ -Nodes, *In Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 62-73, 1995.
- [21] Sreedhar, V. C., Ju, R.D.-C., Gillies, D.M. and Santhanam, V.: Translating Out of Static Single Assignment Form, in Cortesi, A. and Filé, G. (Eds.) SAS'99, *Lec. Notes in Comp. Sci.*, Vol. 1694, pp. 194-210, 1999.
- [22] 滝本宗宏, 佐々政孝: 静的単一代入形式を用いた最適化 (発展編), コンピュータソフトウェア, Vol. ?, No. ?, pp. ?-?, 2008.