修 士 論 文

別名解析に基づく静的単一代入形式変換アルゴリズムの実装と比較

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

西本 真介

2004 年 3 月

修 士 論 文

別名解析に基づく静的単一代入形式変換アルゴリズムの実装と比較

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　二木厚吉 教授
審査委員　大堀淳 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

210067 西本 真介

提出年月: 2004 年 2 月

概 要

静的単一代入形式 (Static Single Assignment Form) は, プログラム中の変数の使用に対して定義が一箇所だけになるように表された中間表現である. 静的単一代入形式を用いることで最適化の実現容易性と実行効率が向上するといわれている.

プログラムを静的単一代入形式に変換するためには変数の使用と定義を正確に把握する必要があり, そのためにポインタのような別名を扱うことは難しい. 実際に, 従来の静的単一代入形式変換のアルゴリズム [9, 11] ではポインタを使用することはできなかった. この問題を解決するべく, ポインタ解析の結果を利用して静的単一代入形式変換を行うアルゴリズムが提案されている. 多くはフロー依存の解析を元にするものであるが, 最近ではフロー非依存の解析を行うものもある. しかし, これらのアルゴリズムを実際に比較した研究はあまりなされていない.

本研究では, このようなポインタ解析に基づくアルゴリズムを比較するための評価環境の実装を行う. 同時にフロー依存の解析を行う Cytron らのアルゴリズム [18] とフロー非依存の解析を行う Hasti らのアルゴリズム [19] の実装を行うことで評価環境の有用性を確認し, 最終的にはこれらのアルゴリズムの実用上の比較を行う.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

現在, 最適化は必要不可欠な技術である. 多くのソフトウェアが最適化されることを前提に作られており, 今後マイクロプロセッサの高性能化, 計算機アーキテクチャの複雑化に伴い, その傾向は大きくなるだろう. 例えば並列計算機ではプログラムの最適化に加えて計算機の構成と処理の内容に応じた最適化が必要であるし, 組み込み機器などの分野では性能だけでなくコードサイズや消費電力についても最適化が必要とされている.

しかし, 最適化を行うコンパイラの開発では長い期間にわたって技術を積み重ねることが重要であり, そのための労力は計り知れない. コンパイラ開発を効率的に行うためには研究の基盤となる共通のコンパイラ・フレームワークを作る必要がある. コンパイラ・フレームワークは組み合わせ可能なコンパイラ部品から構成され, 研究者は比較的容易に新しい部品をコンパイラに追加できるため, コンパイラ開発の効率の向上が見込める. 実際のコンパイラ・フレームワークとしてはNCI Project[1] のSUIF[2] やGNU Project[5] のGCC[6], COINS Project[8] のCOINSなどがある.

近年, 最適化コンパイラの性能向上に向けた新しい手法として, 静的単一代入形式 (Static Single Assignment Form, 略してSSA形式) に基づく最適化が研究されている. SSA形式とは全ての変数への代入が一度しか行われないように表したプログラムの中間表現である. SSA形式ではデータフローが単純化されているため, 最適化に必要なデータフロー解析が簡略化できる. このため最適化の実現容易性と実行効率が向上する. SSA形式による最適化を行うためにはそれぞれのコンパイラで使用されている中間表現をSSA形式に変換する必要がある. 代表的なSSA形式の変換アルゴリズムとしてCytronらの方法 [9, 10] とSreedharらの方法 [11, 12] がある. SSA形式への変換では, そのデータフローを単純化するという性質のためにポインタの存在が問題となる. 実際に上で述べた従来の方法ではポインタを扱うことはできない. このような問題を解決するものとして, ポインタ解析の結果を利用するアルゴリズムが提案されている. 多くのアルゴリズムがフロー依存の解析を元に変換を行う. SSA形式の変換アルゴリズムを提案したCytronらによるポインタ解析の方法 [18] もその中の一つである. それに対し, 最近ではフロー非依存の解析を元にしたアルゴリズムも現れている. Hastiらの方法 [19] などがそれにあたる. 一般にフロー非依存の解析はフロー依存の解析に比べて得られる情報の精度が低いが, その分高速であるという特徴がある. これらのアルゴリズムはその提案者によって実装が行われているものはあるが, 実際に同じコンパイラ・フレームワーク上でその性能が比較されてはいない.

1.2 目的

本研究の目的はSSA形式の変換時にポインタ解析を行うアルゴリズムを効率良く比較するための評価環境として, XMLを利用したコンパイラ・フレームワークを作成することである. 実装した評価環境上でCytronらの方法[18]とHastiらの方法[19]の実装を行うことで評価環境の有用性を確認し, 最終的にはこれらのアルゴリズムの実用上の比較を行う.

この独自のコンパイラ・フレームワークの実装によって以下の成果を期待できる.

1. 必要最小限の構成にすることでアルゴリズムの実装と比較が容易になり, 問題点を明確にしやすくなる. また複数のアルゴリズムを比較的容易に実装でき, 平等な比較が行える.
2. XMLを用いることでSSA変換器, 最適化器の実装言語を自由に選択できる. また, XMLは関連ツールなどの資産が豊富であるためプログラムの負担を軽減できる.
3. 将来的には, 評価環境を拡張してXMLによるデータ中心の交換が可能となるようなコンパイラ・フレームワークを作成し, コンパイラ開発効率の向上と最適化手法の研究基盤を提供することを目指す.

SSA形式の変換時におけるポインタ解析のアルゴリズムのうち, Cytronらの方法[18]とHastiらの方法[19]を比較対象のアルゴリズムとしたことは以下の狙いに基づく.

1. ポインタ解析を用いるアルゴリズムの中で代表的なものであり, それぞれデータフロー依存, データフロー非依存のポインタ解析に基づいている点で対照的である.
2. どちらも同じ手順を繰り返し適用することで最終的に完全なSSA形式が得られるという方法であり, 論理的な性能 (最悪の場合) と実際の性能に差が表れやすい. これらを実装して比較することで実際の性能を調べる.
3. これらの結果はアルゴリズムを実用とする上での参考となる.

1.3 アプローチ

本研究ではアルゴリズムの評価環境として次のものを提供する.

- C言語のサブセットを対象とするコンパイラフロントエンド. (図1.1のA)
- 従来の方法[9]によるSSA変換器. (図1.1のB1)
- テスト用の簡単な最適化器. 無用命令削除と定数伝播. (図1.1のC)
- 従来の方法[9]によるSSA逆変換器. (図1.1のD)
- XMLを入力としてSPARC用アセンブリ言語を出力するコンパイラバックエンド. (図1.1のE)

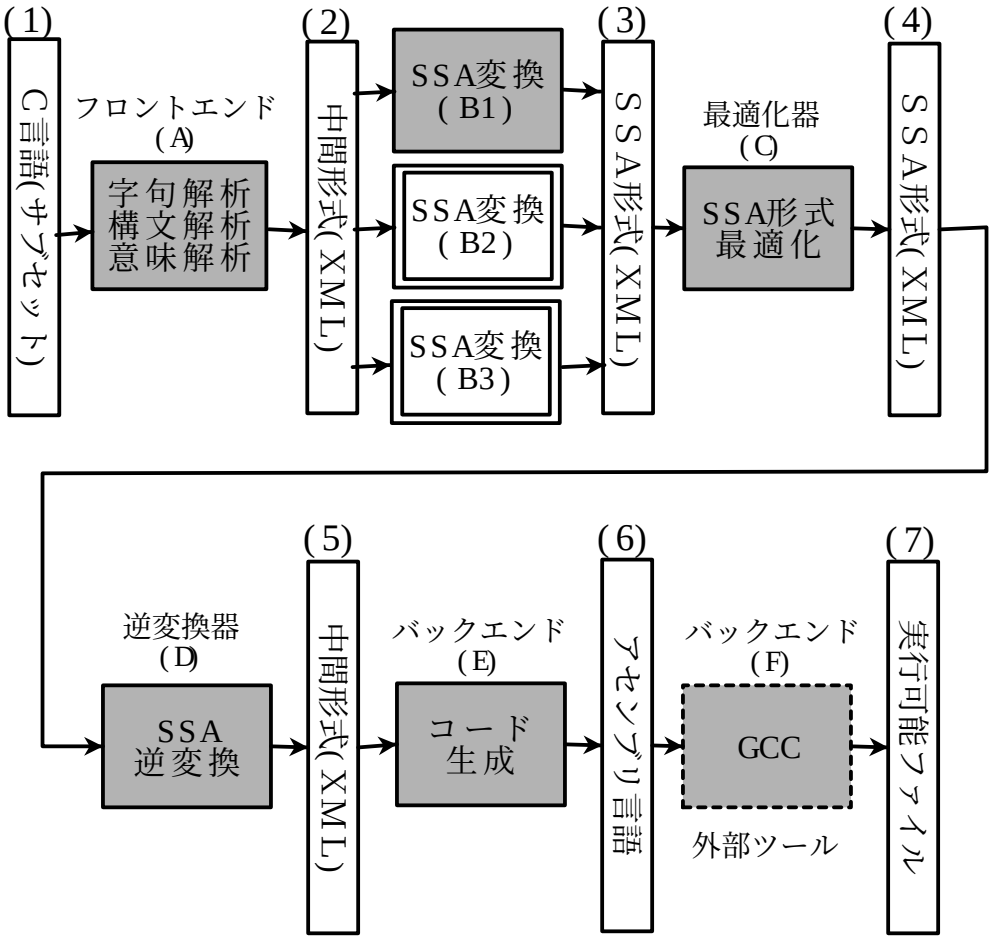


図 1.1: 評価環境の全体図

- XML で中間表現を表すための DTD. (図 1.1 の 2, 3, 4, 5)

比較対象アルゴリズムとして次のものを実装する.

- Cytron らによるポインタ解析を利用した SSA 変換アルゴリズム [18]. (図 1.1 の B2)
- Hasti らによるポインタ解析を利用した SSA 変換アルゴリズム [19]. (図 1.1 の B3)

最終的なコード生成には, 外部ツールとして GCC を利用する. (図 1.1 の F)

コンパイラ・フレームワークの性能評価は以下の点について行う.

- コンパイルの効率.
  - コンパイル (図 1.1 の 1 から 6) に掛かった時間を測定して比較する.
- ポインタ解析を利用した SSA 変換の効率.
  - SSA 変換 (図 1.1 の 2 から 3) に掛かった時間を測定して比較する.
- 最適化の効率.
  - 最適化を行った場合と行わなかった場合のプログラムの大きさを比較する. (図 1.1 の 6 の時点でのプログラムの行数を調べる)
  - 最適化を行った場合と行わなかった場合のプログラムの実行時間を比較する. (図 1.1 の 7 の時点での実行可能ファイルの実行時間を調べる)
  - SSA 形式での最適化を行った場合と GCC での最適化を行った場合を比較する.
- 評価用アルゴリズムと最適化器の実装に関する経験.
  - Cytron らのアルゴリズム [18] と Hasti らのアルゴリズム [19], 最適化器 (定数伝播, 無用命令削除) を実装した際の経験から議論する.
- コンパイラ・フレームワークの適用範囲.
  - ポインタや配列, 構造体などの対応について議論する.

最後にこれらの結果を考察する.

1.4 結論

本研究ではポインタ解析に基づく SSA 変換のアルゴリズムの比較を目的として, XML による中間表現を用いたコンパイラ・フレームワークを実装した. 実装は C 言語にて行い, XML 環境として libxml2 を用いた. 実験は Sun Blade 1500(SPARC), Solaris8 にて行った. 予備評価では次の利点を持つことを確認した.

- ポインタ解析に基づく SSA 変換のアルゴリズムを扱うのに十分な機能を持っている.
- XML による中間表現は実装コストを下げる.
- 実際にコンパイル, 実行を行って性能を測定できる.

ポインタ解析に基づく SSA 変換のアルゴリズムとしてはフロー依存の解析を用いる Cytron らの方法とフロー非依存の解析を用いる Hasti らの方法の実装を行い, これらが正しく動作することを確認した.

しかし, これらのアルゴリズムの比較はまだ十分ではなく, 本研究で行った実験はアルゴリズムの比較を行うための予備的な位置付けであると考えられる. 今後, 更に改良を重ねて詳細な実験を行えるようにする必要がある.

1.5 本論文の構成

第 1 章 はじめに 本研究の背景としてコンパイラ・フレームワーク, SSA 形式, ポインタの問題について述べた上で, アルゴリズム比較の意義と研究のアプローチ, そこから得られた結論について概説する.

第 2 章 静的単一代入形式 (SSA 形式) SSA 形式は最適化の実現を容易にする中間表現である. SSA 形式の概要とその利点について説明する.

第 3 章 SSA 変換とポインタ解析 SSA 変換時に起きるポインタの問題について述べ, ポインタ解析の必要性について説明する.

第 4 章 Cytron らのアルゴリズム

第 5 章 Hasti らのアルゴリズム ポインタ解析を用いた SSA 変換のアルゴリズムについて説明する.

第 6 章 実装と実験 本研究で実装した評価環境について説明した上で, 実験結果とその考察を行う.

第 7 章 関連研究 本研究に関連する研究について述べる.

第 8 章 おわりに 本研究の結果と将来の課題についてまとめる.

第2章　静的単一代入形式 (SSA形式)

2.1　SSA形式の概要

SSA形式(Static Single Assignment Form) とは, プログラム中の全ての変数の使用に対して, 対応する定義が1箇所しかないように表した中間表現である. SSA形式では変数の使用と定義の関係が明確になる. プログラムをSSA形式に変換するためには, 変数の名前の付け替えとφ関数の挿入が必要である. このとき元のプログラムの意味を変えないようにしなくてはならない. 例えば,

$a = 0;$
$a = a + 1;$
$b = a;$

というプログラムでは, 1行目と2行目でaを定義しているので, 3行目のaの使用に対して定義が2箇所存在している. これをSSA形式にすると次のように変換される.

$a_0 = 0;$
$a_1 = a_0 + 1;$
$b_0 = a_1;$

こうすると使用と定義の関係が一対一になっていることがわかる. ここにifやwhileなどの分岐が含まれる場合にはφ関数と呼ばれる仮想的な関数が必要となる.

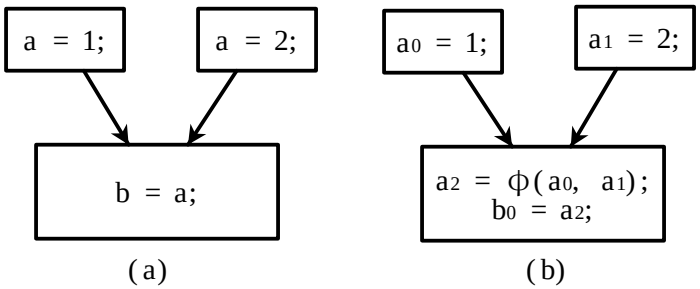


図 2.1: φ関数

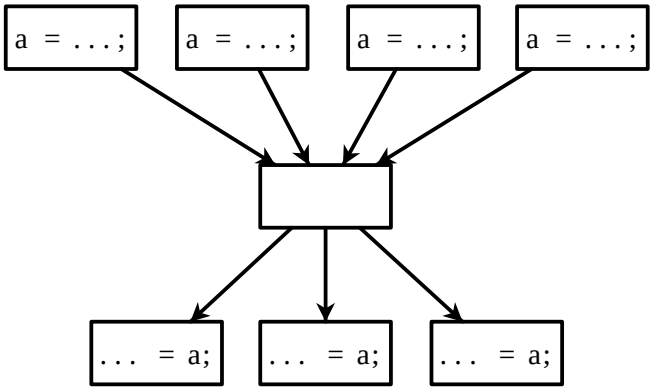


図 2.2: 分岐と合流のあるプログラム

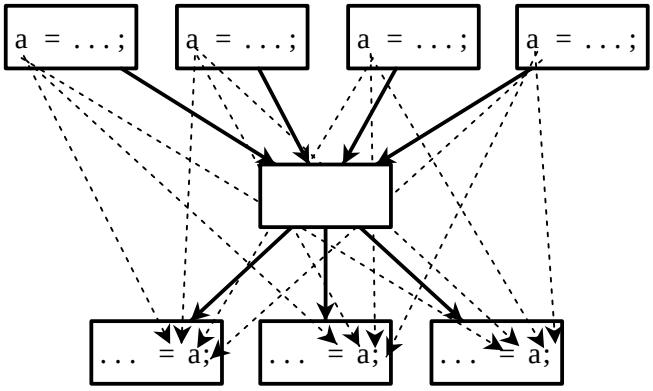


図 2.3: 使用と定義のリンク

図 2.1(a) のような制御フローを持つプログラムは変数の名前を付け替えるだけでは上手くいかない. そこで図 2.1(b) のように制御フローの合流地点にφ関数を挿入する. この場合, φ関数は左のブロックからきたときは a_0 を, 右のブロックからきたときは a_1 を返す.

SSA 形式の特徴は使用と定義の関係が単純になることである. 図 2.2 のようなプログラムで合流の数を m, 分岐の数を n とすると, そのままでは mn 通りの使用と定義のリンクが考えられる (図 2.3). SSA 形式では図 2.4 のように $m + n$ 通りとなり普通の場合に比べて使用と定義の関係が疎なグラフを構成している.

2.2 SSA 形式の定義

SSA 形式の定義はいくつか存在するが (Choi ら [14] や Wegman ら [24] が定義している), ここでは Wegman らの定義 [24] を述べる.

定義 もとのプログラムの全ての変数 v に対して, 次の三つの条件を満たすとき, そのプロ

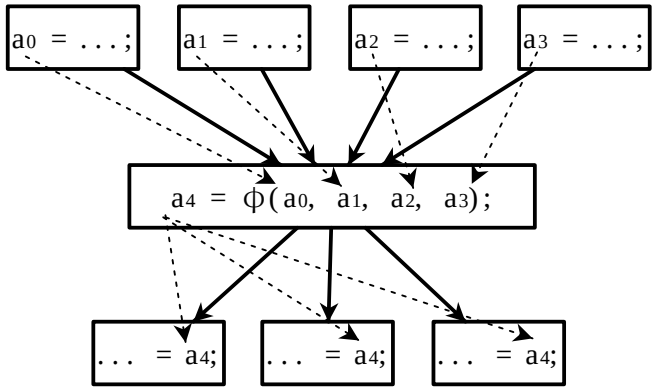


図 2.4: SSA 形式による使用と定義のリンク

グラムは SSA 形式である.

1. もし制御フローグラフのノード Z が, v の定義を含む二つのノード X と Y について, 空でない路 $X \to Z$ と $Y \to Z$ の最初の共通ノードであるなら, v に関するφ関数が Z に挿入されている.
2. v に対するそれぞれの新しい名前 v_i はプログラムテキスト上で唯一の定義を持つ.
3. 任意の制御フロー上の路において, v に対する新しい名前 v_i の使用と, もとのプログラムでそれに対応する v の使用を考えたとき, v と v_i は同じ値を持つ.

上記の条件を満たし, かつφ関数の数が最小な SSA 形式のことを**最小 SSA** (minimal SSA) という. ここで最小とは定義を満たす上で必要な最小であり, 実際に実行するためには必要ではないφ関数も存在し得る.

本論文では SSA 形式に変換される前のプログラムを**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**, SSA 形式から通常形式への変換のことを**正規化**(normalization), または**SSA 逆変換**と呼ぶことにする.

2.3 SSA 形式の例

図 2.5 に SSA 形式の例を示す. 点線で囲まれた部分は制御フローグラフでの基本ブロックを表し, 太字は SSA 形式で挿入されたφ関数を表す.

2.4 SSA 形式の利点

SSA 形式の利点は最適化の実現が容易になることである. 例えば, コピー伝播 (copy propagation), 定数伝播 (constant propagation), 共通部分式削除 (common subexpression elimination) などの最適化がデータフロー解析を行わずに実現できる.

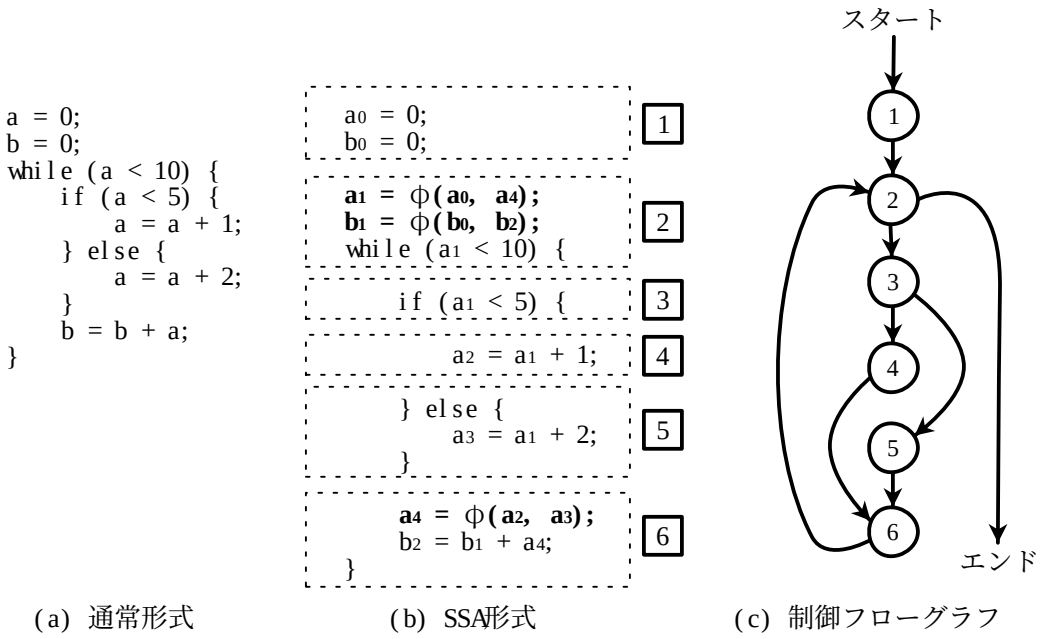


図 2.5: SSA 形式の例

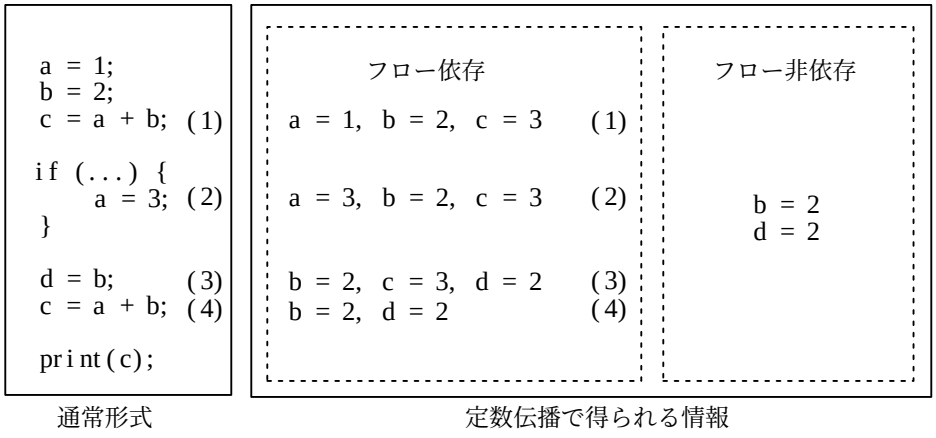


図 2.6: 通常形式における定数伝播

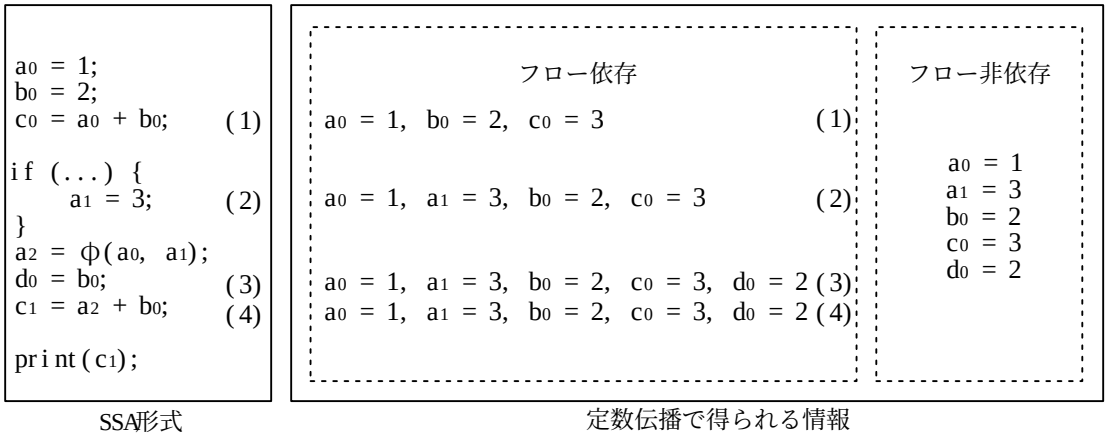


図 2.7: SSA 形式における定数伝播

もう 1 つの利点は, SSA 形式では変数の定義と使用が疎なグラフを構成しているため, それを表現するためのメモリ使用量が少なくてすむことである. また情報を辿るための演算量も少なくなる.

図 2.6 に通常形式における定数伝播, 図 2.7 に SSA 形式における定数伝播の例を示す. それぞれの図でフロー依存の情報はプログラムのある一時点での変数の値を表し, フロー非依存の情報はプログラム全体での変数の値を表す. 通常形式ではフロー非依存よりもフロー依存の情報の方が詳しくなっているが, SSA 形式ではどちらも同じだけの情報を持っている. 従って, SSA 形式では定数伝播を行うにあたってデータフロー解析が簡略化できる.

第3章　SSA変換とポインタ解析

3.1　SSA変換の概要

代表的なSSA変換のアルゴリズムとしてCytronらの方法[9, 10]とSreedharらの方法[11, 12]がある．これら二つの方法はφ関数を挿入する位置を決定するアルゴリズムが異なっている．Cytronらの方法では支配辺境(dominance frontier)というデータ構造を用いてφ関数を挿入する．Sreedharらは，さらにDJ-graphというデータ構造を用いて，不必要な支配辺境の計算を省くことで効率を高めている．Cytronらの方法は最悪の場合には2次の計算時間がかかるのに対して，Sreedharらの方法は線形時間しかかからない．しかし，これらのアルゴリズムの実装と比較を実際に行った研究[25]では特殊な場合を除けばこの二つのアルゴリズムの実行時間には大きな差はないことがわかっている．

以下ではCytronらの提案する方法に沿ったSSA変換のアルゴリズムを説明する．SSA変換の大まかな流れは次の通りである．

1. プログラムの制御フローグラフから支配木を求める．
2. プログラムの制御フローグラフと支配木から支配辺境を求める．
3. 支配辺境を用いてφ関数を挿入する．
4. 変数名を付け替える．

3.2　用語

これからの議論に必要な用語について説明する．

3.2.1　制御フローグラフ

制御フローグラフ(control flow graph)とは，プログラムの制御の流れをグラフで表したものである．プログラムのうち，合流も分岐もない連続した部分(これは**基本ブロック**と呼ばれる)をノードとし，それらの間を分岐や合流を表す有向辺で結んだ有向グラフである．制御フローグラフ上でブロックXからブロックYへの有向辺が存在するとき，XはYの**先行ブロック**(predecessor)，YはXの**後続ブロック**(successor)という．ブロックXの先行ブロックの集合を$\mathbf{pred(X)}$，後続ブロックの集合を$\mathbf{succ(X)}$と書く．9ページの図2.5(c)の制御フローグラフを例にとると，$pred(3) = \{2\}$, $succ(3) = \{4, 5\}$と書ける．

3.2.2 支配木

X と Y を制御フローグラフのノードとしたとき, プログラムの入口から Y に達するどの路も必ず X を通るとき X は Y を**支配する**(dominate), または X は Y の**支配ブロック**(dominant) である, という. ブロック X の支配ブロックの集合を $\mathbf{DOM(X)}$ と書く. 支配関係は反射的かつ推移的である. 従ってブロック X は自分自身を支配する.

X が Y を支配し, かつ $X \neq Y$ のとき, X は Y を**厳密に支配する**(strictly dominate) という. X が Y を厳密に支配し, かつ X から Y への路に X 以外に Y を厳密に支配するノードがないとき, X は Y を**直接支配する**(immediately dominate) といい, $\mathbf{X = IDOM(Y)}$ と書く.

ノード間の直接支配の関係を辺で表した木構造を**支配木**(dominator tree) という. 支配木の根 (root) は入口ノードであり, X の親を $\mathbf{parent(X)}$, 子の集合を $\mathbf{child(X)}$ と書く. 例えば $X = IDOM(Y)$ であれば, $parent(Y) = X$, $child(X) \ni Y$ と書ける.

3.2.3 支配辺境

X の後続ノードを辿っていき, 最初に見つかった X が直接支配していないノードの集合を**支配辺境**(dominance frontier) という. X の支配辺境の集合は $\mathbf{DF(X)}$ と書き, 厳密には次のように定義される.

定義 $DF(X) = \{Y | U \in pred(Y)$ が存在し, X は U を支配し, X は Y を厳密な支配はしない $\}$

これは, これまでに登場した式を用いて

$$DF(X) = \{Y \in succ(X) | IDOM(X) \neq X\} \cup \bigcup_{Z \in child(X)} \{Y \in DF(Z) | IDOM(Y) \neq X\}$$

と書ける.

3.3 SSA 変換のアルゴリズム

3.3.1 支配木を求めるアルゴリズム

支配木を求めるアルゴリズムには Harel の方法 [23] や Lengauer らの方法 [22] がある. これらは効率を高めるため複雑なアルゴリズムとなっている. ここでは, 効率は悪いがわかりやすいアルゴリズムを説明する.

まずは制御フローグラフを深さ優先で辿り, 各ノードが支配するノードの集合を求める. 次に各ノードが支配するノードの集合から直接支配しているノードの集合を求める.

Entry 入口ノード.

visited(X) ブロック X を訪問したかどうかのフラグ.

rdom(X) ブロック X が支配するノードの集合. 但し, X 自身を含まない.

parent(X) ブロック X の支配木での親. IDOM(X) と同じ.

child(X) ブロック X の支配木での子の集合.

idom_flag 直接支配しているかどうかのフラグ.

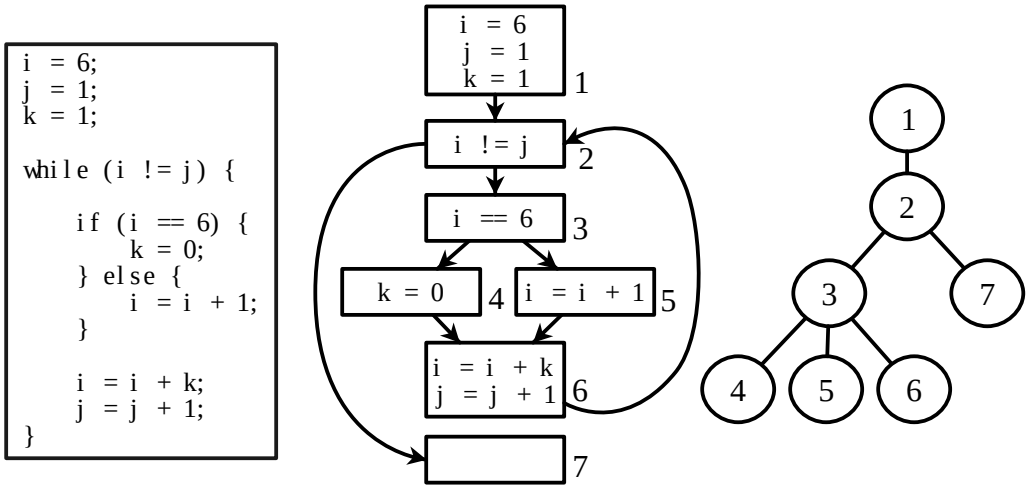
アルゴリズム

```
/* 支配木を求めるアルゴリズム */
for each 全てのブロック A do
        rdom(A) ← 空集合
        call COMPUTE_RDOM(A)
end for
call COMPUTE_DOM_TREE(Entry)

/* X が支配するノードの集合を求める */
COMPUTE_RDOM(X) :
        for each X を除く全てのブロック B do
                visited(B) ← 0
        end for
        visited(X) ← 1
        call VISIT_NODE(Entry)
        for each 全てのブロック C do
                if visited(C) = 0 then
                        rdom(X) に C を加える
                fi
        end for

/* 制御フローグラフ上で入口ノードから順に探索する */
VISIT_NODE(Y) :
        if visited(Y) = 1 then return
        visited(Y) ← 1
        for each D ∈ succ(Y) do
                call VISIT_NODE(D)
        end for

/* 支配木を求める */
COMPUTE_DOM_TREE(Z) :
        for each E ∈ rdom(Z) do
```



(a) 通常形式　　(b) 制御フローグラフ　　(c) 支配木

図 3.1: 支配木の例

$idom\_flag \leftarrow 1$
for each $F \in rdom(Z)$ do
　　if $E \in rdom(F)$ then $idom\_flag \leftarrow 0$
end for
if $idom\_flag$ then
　　$parent(E) \leftarrow Z$
　　$child(Z)$ に E を加える
fi
end for

図 3.1 に支配木の例を示す. アルゴリズムの結果は図 3.2 の通りである.

3.3.2　支配辺境を求めるアルゴリズム

支配辺境を求めるためには, 前述した

$$DF(X) = \{Y \in succ(X) | IDOM(X) \neq X\} \cup \bigcup_{Z \in child(X)} \{Y \in DF(Z) | IDOM(Y) \neq X\}$$

を用いる.

支配木を下から上に辿りながら上記の式で支配辺境を求めていく.

Entry　入口ノード.

ノードX	rdom(X)	parent(X)	child(X)
1	{ 2, 3, 4, 5, 6, 7 }		{ 2 }
2	{ 3, 4, 5, 6, 7 }	1	{ 3, 7 }
3	{ 4, 5, 6 }	2	{ 4, 5, 6 }
4	{ }	3	{ }
5	{ }	3	{ }
6	{ }	3	{ }
7	{ }	2	{ }

図 3.2: 支配木を求める過程で各ノードに付与された情報

DF(X) ブロック X の支配辺境の集合.

IDOM(X) 支配木を求めるアルゴリズムで計算した parent(X) を用いる.

アルゴリズム

/* 支配辺境を求めるアルゴリズム */
call $VISIT\_NODE\_FROM\_CHILD(Entry)$

/* 支配木を下から上に辿る */
$VISIT\_NODE\_FROM\_CHILD(X)$:
    for each $A \in child(X)$ do
        call $VISIT\_NODE\_FROM\_CHILD(A)$
    end for
    call $COMPUTE\_DF(X)$

/* 支配木を求める */
$COMPUTE\_DF(Y)$:
    $DF(Y) \leftarrow$ 空集合
    for each $B \in succ(Y)$ do
        if $IDOM(B) \neq Y$ then $DF(Y)$ に B を追加する
    end for
    for each $C \in child(Y)$ do
        for each $D \in DF(C)$ do
            if $IDOM(D) \neq Y$ then $DF(Y)$ に D を追加する

end for
end for

図 3.1 の支配木を例にとると, アルゴリズムの結果は以下の様になる.

$DF(7) = \{\}$
$DF(6) = \{2\}$
$DF(5) = \{6\}$
$DF(4) = \{6\}$
$DF(3) = \{2\}$
$DF(2) = \{2\}$
$DF(1) = \{\}$

3.3.3 φ関数を挿入するアルゴリズム

φ関数の挿入では入口ブロックに全ての変数の定義があると仮定する. まず, ブロック X に変数 V の定義が存在するならば, X の支配辺境 Y にも V を定義するφ関数を挿入する. すると Y にも変数 V の定義があることになるので, Y の支配辺境にも V を定義するφ関数を挿入する. これをφ関数を挿入する必要がなくなるまで繰り返す.

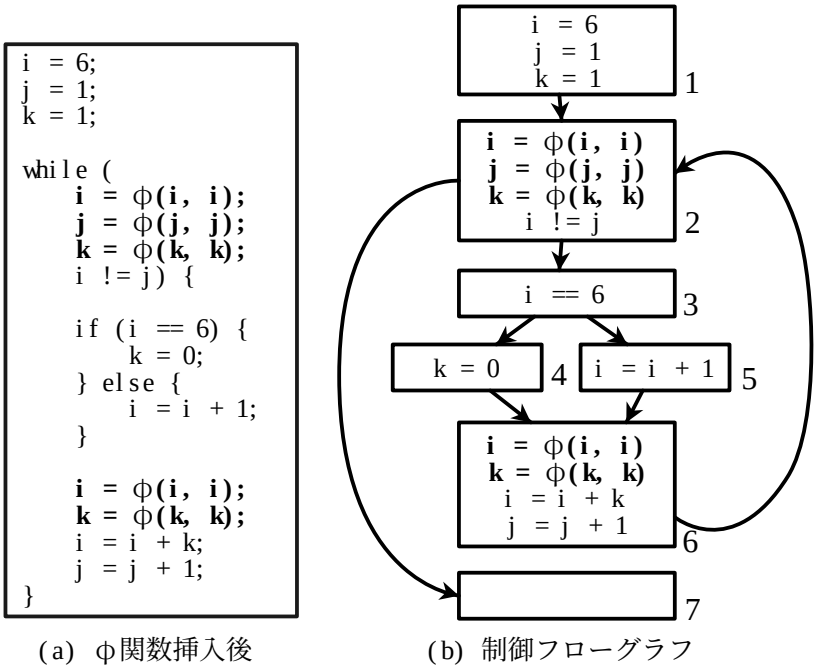
inserted(X) 変数名の集合. 変数 V を定義するφ関数をブロック X に挿入したことを示す.

work(X) 変数名の集合. 変数 V の定義があるブロック X を W に追加したことを示す.

W 未処理のブロックの集合.

アルゴリズム

/* φ関数を挿入するアルゴリズム */
for each 全てのブロック A do
    $inserted(A)$ = 空集合
    $work(A)$ = 空集合
end for
W = 空集合
for each 全ての変数 V do
    for each V の定義があるブロック B do
        W に B を加える
        $work(B)$ に V を加える
    end for



(a) φ関数挿入後　　(b) 制御フローグラフ

図 3.3: φ関数挿入後のプログラム

while W が空集合でない do
　W からあるブロック X を取り除き, その X について
　for each $C \in DF(X)$ do
　　if $V \notin inserted(C)$ then
　　C に V を定義するφ関数を挿入する
　　$inserted(C)$ に V を加える
　　if $V \notin work(C)$ then
　　W に C を加える
　　$work(C)$ に V を加える
　　fi
　　fi
　end for
end while
end for

図 3.1 のプログラムにφ関数を挿入したものを図 3.3 に示す.

3.3.4 変数名を付け替えるアルゴリズム

変数名の付け替えでは支配木を上から下へ辿り, 新しく変数が定義される度に番号を増やしながら, 変数に番号を振っていく.

Entry 入口ノード.

C(V) 変数 V の現在の添え字の値.

S(V) 変数 V の添え字を保存しているスタック.

Top(S(V)) スタック S(V) の現在指している値を返す.

Pop(S(V)) スタック S(V) から値をポップする.

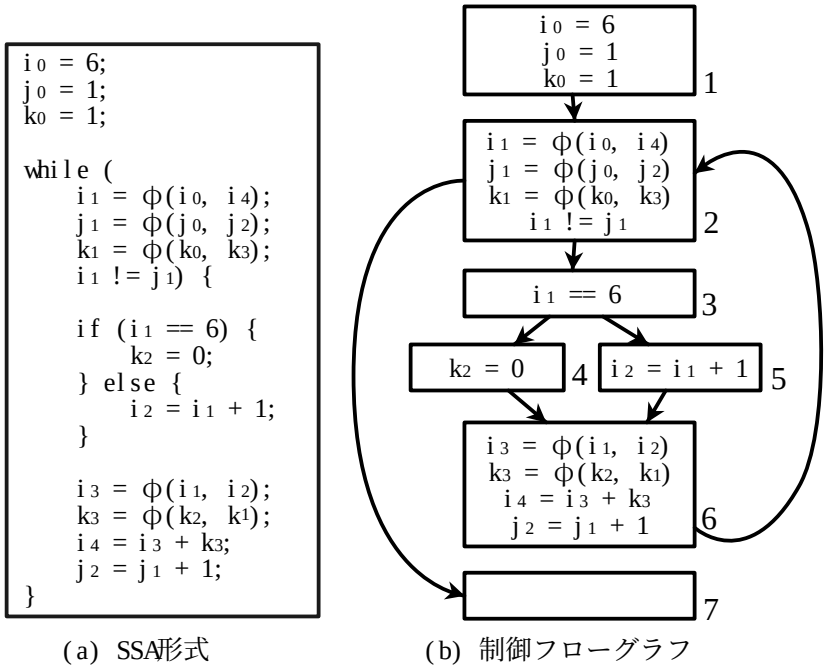
Push(S(V), i) スタック S(V) に値 i をプッシュする.

pred#(Y, X) ブロック X がブロック Y の何番目の先行ブロックかを返す.

アルゴリズム

/* 変数名を付け替えるアルゴリズム */
for each 全ての変数 V do
    $C(V) \leftarrow 0$
    $S(V) \leftarrow$ 空スタック
end for
call $SEARCH(Entry)$

/* ブロック毎の処理 */
$SEARCH(X)$:
    for each 文 A in ブロック X do /* ブロックの先頭から順に */
        if 文 A の右辺がφ関数でない then
            for each 変数 V in 文 A の右辺 do
                右辺の V を V_i に置き換える
                    (但し, $i = Top(S(V))$)
            end for
        fi
        for each 変数 V in 文 A の左辺 do
            $i \leftarrow C(V)$
            左辺の V を V_i で置き換える
            $Push(S(V), i)$
            $C(V) \leftarrow i + 1$
        end for
    end for
    for each $Y \in succ(X)$ do

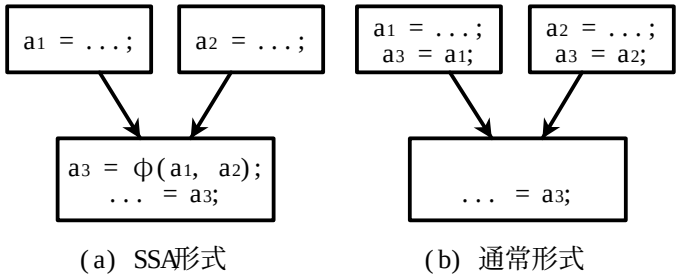


(a) SSA形式　　(b) 制御フローグラフ

図 3.4: 変数名の付け替え後のプログラム

$j \leftarrow pred\#(Y, X)$
for each φ関数 F in Y do
F の j 番目の引数の V を V_i で置き換える
(但し, $i = Top(S(V))$)
end for
end for
for each $Y \in child(X)$ do
call $SEARCH(Y)$
end for
for each 文 A in ブロック X do
for each A の左辺にあった V do
$Pop(S(V))$
end for
end for

変数名の付け替え後を図 3.3 に示す. これで SSA 形式となっている.



(a) SSA形式　　(b) 通常形式

図 3.5: 単純な方法による SSA 逆変換

3.3.5 SSA 逆変換

SSA 形式から通常形式に戻すためにはφ関数を取り除く必要がある. 図 3.5 の (a) を (b) のように変更することでφ関数を取り除くことができる. しかし, 最適化の内容によってはこのような単純な方法で SSA 逆変換が行えない場合がある. そのような状況を解決した代表的な SSA 逆変換のアルゴリズムに Briggs らの方法 [15] と Shreedhar らの方法 [17] がある.

図 3.6 は単純な方法で SSA 逆変換を行った場合の結果である.

3.4 SSA 変換時のポインタの問題

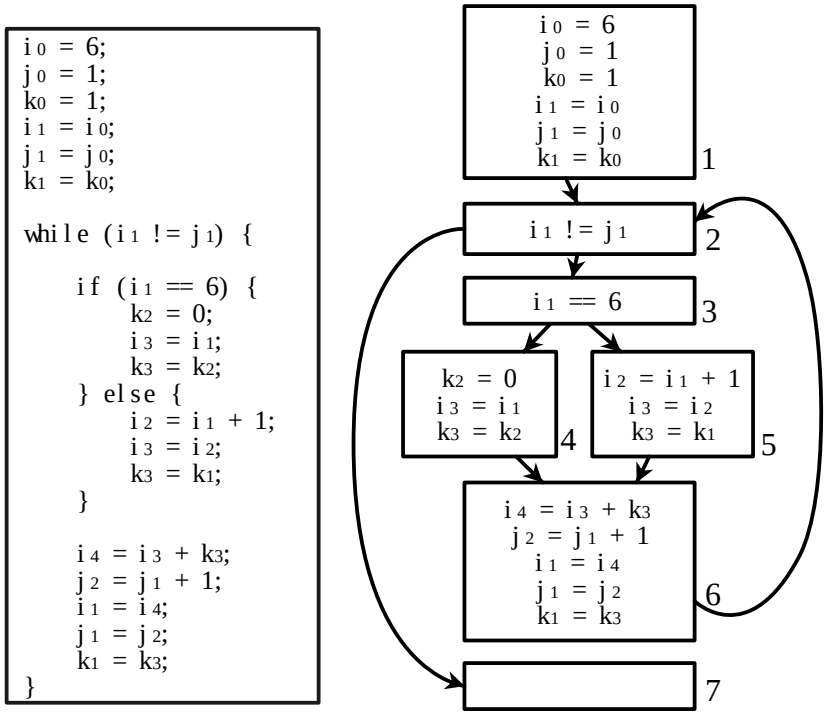
もしポインタを含むプログラムをこれまでに述べた方法で SSA 変換したなら, 正しく SSA 変換することができない. 以下に簡単な例を用いてこれを説明する.

3.4.1 ポインタ参照の問題

ポインタの第一の問題はポインタ参照による定義である. ポインタ参照による定義が行われたとき, ポインタが指している先がわからないと定義された変数が何であるかわからないため, SSA 変換に失敗する. 図 3.7 がこの例である.

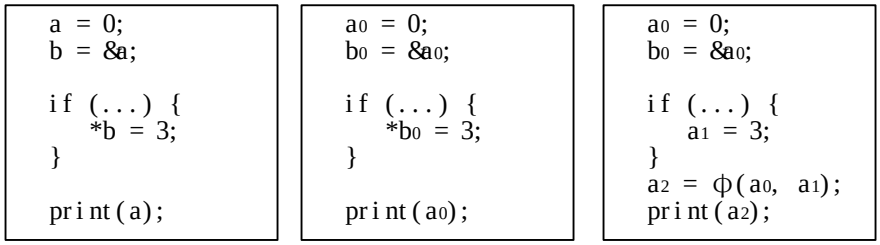
3.4.2 アドレス演算の問題

ポインタのアドレスが計算されるとき, その変数の格納領域は普通は一箇所だけでなくてはならない. しかし SSA 形式では, $\&V$ が V_0 のアドレスを指すのか, V_1 のアドレスを指すのか知ることができない. 図 3.8 がこの例である.



(a) 通常形式　　(b) 制御フローグラフ

図 3.6: SSA 逆変換後のプログラム

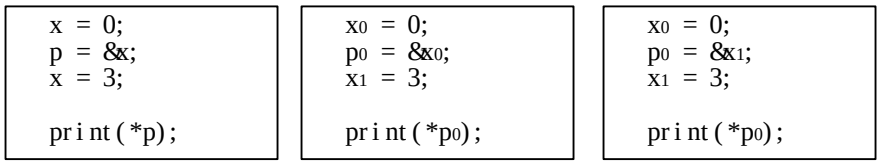


(a) 通常形式
最後の出力は3

(b) 間違ったSSA形式
最後の出力は0

(c) 正しいSSA形式
最後の出力は3

図 3.7: SSA 変換時のポインタの問題 1



(a) 通常形式
最後の出力は3

(b) 間違ったSSA形式
最後の出力は0

(c) 正しいSSA形式
最後の出力は3

図 3.8: SSA 変換時のポインタの問題 2

3.4.3 ポインタ解析の必要性

以上のように, ポインタを含むプログラムを SSA 形式に変換するためには, まずポインタ解析を行い, ポインタの情報を取得した上で SSA 変換時に上手く利用する必要がある. ここでのポインタ解析とはポインタ変数が指している変数の集合を調べることである. これにはポインタが必ず指している変数の集合を調べる must-alias 解析と, ポインタが指す可能性のある変数の集合を調べる may-alias 解析がある. また制御フローを考慮する方法と考慮しない方法があり, それぞれ使い分ける必要がある.

次章ではこれらの情報を使って SSA 変換を行う方法を説明する.

第4章　Cytronらのアルゴリズム

4.1　基本方針

Cytron らのアルゴリズム [18] では始めに従来の SSA 変換を行い, その結果にポインタ解析の情報を少しずつ反映させていくという手順を踏む. 従って最初に得られる SSA 形式 (これを SSA_0 とする) は正しくない状態にある. 例えば, SSA_0 が以下のようになっていたとする.

$a_0 = 1;$
$\vdots$
$*p_0 = 3$
$\vdots$
$b_0 = a_0;$

このときポインタ p_0 の may-alias 解析の結果に a_0 が含まれるならば, $IsAlias$ 関数を挿入する.

$a_0 = 1;$
$\vdots$
$*p_0 = 3$
$a_1 = IsAlias(p_0, \&a_0);$
$\vdots$
$b_0 = a_1;$

$IsAlias$ 関数の戻り値はポインタ解析の結果によって三つの場合に分けられる. まず must-alias 解析の結果からポインタ p_0 が a_0 を必ず指すことが分かっている場合には $IsAlias(p_0, \&a_0)$ は $*p_0$ を返す. 次に may-alias 解析の結果からポインタ p_0 が a_0 を絶対に指さないことが分かっている場合には $IsAlias(p_0, \&a_0)$ は a_0 を返す. どちらともいえない場合には, p_0 と $\&a_0$ の値を比較して等しいときだけ p_0 が a_0 を指していると判断する.

$IsAlias$ 関数は次のように書くことができる.

$IsAlias(w, x) : value$
　　if w = x then

$ans_0 \leftarrow *w$

else

$ans_1 \leftarrow *x$

fi

$ans_2 \leftarrow \phi\ (ans_0, ans_1)$

return ans_2

プログラムに分岐と合流がある場合には, $IsAlias$ 関数を挿入する度に必要であればφ関数も同時に挿入しなくてはならない. 図 4.1 はこのような場合の例を示している. 図 4.1 では太字で表記した a_0 の使用と定義について $IsAlias$ 関数を挿入している. 図中のポインタ p_0 は a_0 を指している可能性があるものとする.

このようにポインタ解析の情報を少しずつ用いながら使用と定義を分断していき, 最終的に注目すべき変数の使用が無くなった時に終了する ($SSA_0 \rightarrow SSA_1 \rightarrow \ldots \rightarrow SSA_i = SSA$). 次に注目する変数の使用を見つけるために定数伝播を利用している.

アルゴリズムの大まかな流れは以下の通りである.

1. 従来の SSA 変換で SSA_0 を求める.
2. ポインタ解析を行い, プログラム中のポインタ参照毎に may-alias 情報と must-alias 情報を調べる.
3. 注目すべき変数の使用の集合を求める (定数伝播を用いる). もし集合が空なら終了する.
4. ポインタ解析の情報を用いて $IsAlias$ 関数とφ関数を挿入する.
5. 3. に戻る.

詳しいアルゴリズムは 4.3 で述べる.

4.2 ポインタの問題

$IsAlias$ 関数を導入することで 3.4 で述べたポインタ参照の問題は解決される. しかし, この方法ではアドレス演算の問題を解決することはできない. SSA_0 自体がアドレス演算の問題を含んでいるかもしれないからである. 図 3.8(b) の間違った SSA 形式に少し変更を加えた例を考える.

$x_0 = 0;$
$p_0 = \&x_0$
$x_1 = 3;$
$\vdots$
$*p_0 = 5;$

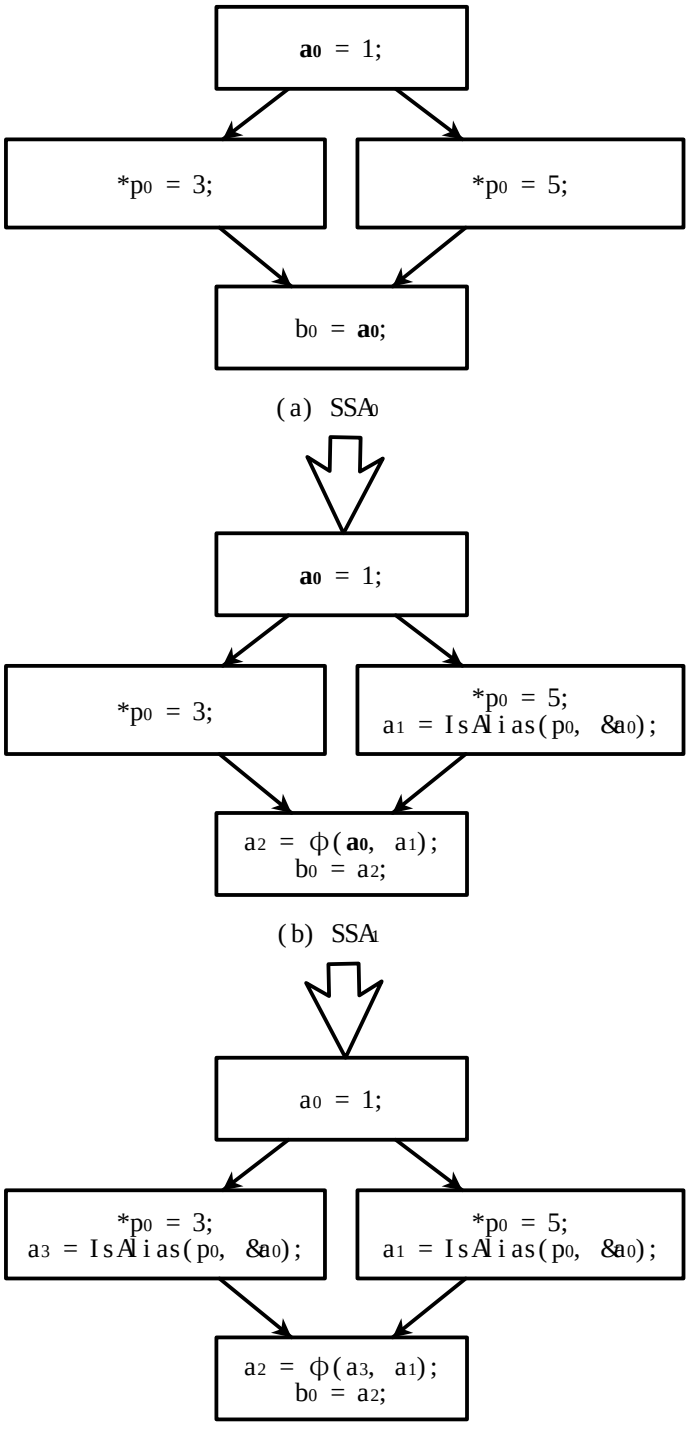


図 4.1: IsAlias 関数の挿入の例

$$
\begin{array}{l}
\quad\vdots \\
b_0 = x_1;
\end{array}
$$

これを x_1 の使用に注目して $IsAlias$ 関数を挿入すると次のようになる.

$$
\begin{array}{l}
x_0 = 0; \\
p_0 = \&x_0 \\
x_1 = 3; \\
\quad\vdots \\
*p_0 = 5; \\
x_2 = IsAlias(p_0, \&x_1); \\
\quad\vdots \\
b_0 = x_2;
\end{array}
$$

p_0 と $\&x_1$ は等しくないため ($p_0 = \&x_0$ であるから), 意図した通りには動かない.

これに対して第 5 章で述べる Hasti らのアルゴリズムではアドレス演算の問題も解決できる.

4.3 アルゴリズム

Cytron らのアルゴリズムは以下の通りである.

DefSites プログラム中の全ての定義の集合.

SSA$_0$ プログラムを従来の方法で SSA 変換したもの.

Visited(d) 定義 d を処理したときの VisitStamp の値.

VisitStamp アルゴリズムの繰り返し回数.

Form 変換途中の SSA 形式 (SSA_i).

DList 定数伝播の対象となる定義の集合.

IncCProp(Form, DList) 定数伝播.

Soln 定数伝播の結果.

UList 次に対象とする変数の使用の集合.

Symbol(r) 変数の参照 r のシンボル (a や $*p$ など. 参照は変数の使用か定義のどちらかである).

Rdef(u) 変数の使用 u の SSA 形式における唯一の定義. アルゴリズムの進行とともに更新される.

Node(r) 変数の参照 r を含む制御フローグラフの基本ブロック.

FirstDef(X) 基本ブロック X における最初の定義.

LastDef(X) 基本ブロック X における最後の定義.

DomDef(r) 変数の参照 r のすぐ前の定義. 変数の参照 r が基本ブロック X の最初にある場合には, X を直接支配するブロック Y の最後の定義.

MayAlias(d) 変数の定義 d が指している可能性のある変数のシンボルの集合.

CreateIsAlias(d, Rdef) 定義 d のすぐ後に $d^1 = IsAlias(u^1, u^2)$ を挿入する. このとき次の条件を満たす (r は d^1, u^1, u^2 などの上付き数字のある参照).

$$Symbol(d^1) = Symbol(Rdef)$$
$$Symbol(u^1) = Symbol(d)$$
$$Symbol(u^2) = Symbol(Rdef)$$
$$Rdef(u^1) = d$$
$$Rdef(u^2) = Rdef$$
$$DomDef(r) = d, r \neq u^2$$
$$DomDef(u^2) = DomDef(d)$$
$$Node(r) = Node(d)$$

CreatePhi(d, Rdef, m, IDef) Node(d) の先頭に $d^1 = \phi\ (u^1, \ldots, u^k)$ を挿入する. このとき次の条件を満たす.

$$DomDef(d) = d^1$$
$$Symbol(r) = Symbol(Rdef)$$
$$Node(d^1) = Node(d)$$
$Node(u^i) = Node(d)$ の i 番目の先行ノード
$$DomDef(u^i) = LastDef(Node(u^i))$$
$Rdef(u^i) = Rdef$ (但し, $i \neq m$)
$Rdef(u^i) = IDef$ (但し, $i = m$)

アルゴリズム

/* Cytron らのアルゴリズム */
for each $d \in DefSites$ do
    $Visited(d) \leftarrow 0$
end for
$VisitStamp \leftarrow 0$
$Form \leftarrow SSA_0$
$DList \leftarrow DefSites$
$[Soln, UList] \leftarrow IncCProp(Form, DList)$
while $UList$ が空集合でない do
    $VisitStamp \leftarrow VisitStamp + 1$
    $[Form, DList] \leftarrow Update(Form, UList)$

```
        [Soln, UList] ← IncCProp(Form, DList)
end while

/* DList の更新 */
Update(Form, UList): [Form, DList]
        DList ← 空集合
        for each u ∈ UList do
                if (∃d|Symbol(u) ∈ MayAlias(d)) then
                        NewDef ← Snoop(DomDef(u), Rdef(u), Rdef(u))
                        if NewDef ≠⊥ then
                        /* ⊥ は値が存在しないことを表す */
                                Rdef(u) ← NewDef
                        fi
                fi
        end for

/* IsAlias 関数と φ関数の挿入 */
Snoop(StartDef, EndDef, Rdef): Def
        d ← StartDef
        while d ≠ StopDef do
                Visited(d) ← VisitStamp
                if Symbol(d) = Symbol(Rdef) then
                        DList ← DList ∪ {d}
                        return d
                fi
                if Symbol(Rdef) ∈ MayAlias(d) then
                        NewDef ← CreateIsAlias(d, Rdef)
                        DList ← DList ∪ {NewDef}
                        return NewDef
                fi
                if d = FirstDef(Node(d)) then
                        IDef ←⊥
                        for each m ∈ Node(d) の先行ノードの集合 do
                                if Visited(LastDef(m)) ≠ VisitStamp
                                then
                                        IDef ← Snoop(LastDef(m),
                                                LastDef(idom(Node(d))),
                                                Rdef)
```

```
                    fi
                    if IDef ≠⊥ then
                        NewDef ← CreatePhi(d, Rdef, m, IDef)
                        DList ← DList ∪ {NewDef}
                        return NewDef
                    fi
                end for
            fi
            d ← DomDef(d)
        end while
        return ⊥
```

図 4.2 にアルゴリズムの実行例を示す. Snoop 関数は引数の StartDef と EndDef の間を再帰的に辿り, $IsAlias$ 関数が必要な部分を探す処理である. 各図で太字で表記した部分がそのときの $StartDef$ と $EndDef$ である. これらは一致する場合もあり, そのときは何もせずに Snoop 関数は終了する.

(b) の SSA_0 に $NULL = \ldots$ という代入が存在するが, これは定義が存在しないブロックが無くなるように便宜上挿入された意味のない定義である. これによって $FirstDef$ や $LastDef$ が必ず値を持つことを保証する.

このプログラムでは注目すべき変数の使用は p_0 が 2 つ, a_0 が 2 つ存在するが, $IsAlias$ 関数の挿入が行われるのは a_0 のときだけである. 例ではブロック 4 における a_0 の使用に対する Snoop 関数の呼び出しを追っている. 1 回目, 2 回目, 3 回目の Snoop 関数では何もしておらず, 4 回目で初めてブロック 3 に到達して $IsAlias$ 関数を挿入する. そして 1 つ Snoop 関数を戻ったところでφ関数を挿入して, 注目している使用の変数名を正しく置き換える.

Snoop 関数の終了時でブロック 2 の $a_0 < 10$ の部分は $a_2 < 10$ となるべきである. この部分はブロック 2 の a_0 について Snoop を実行したときに正しく置き換わる.

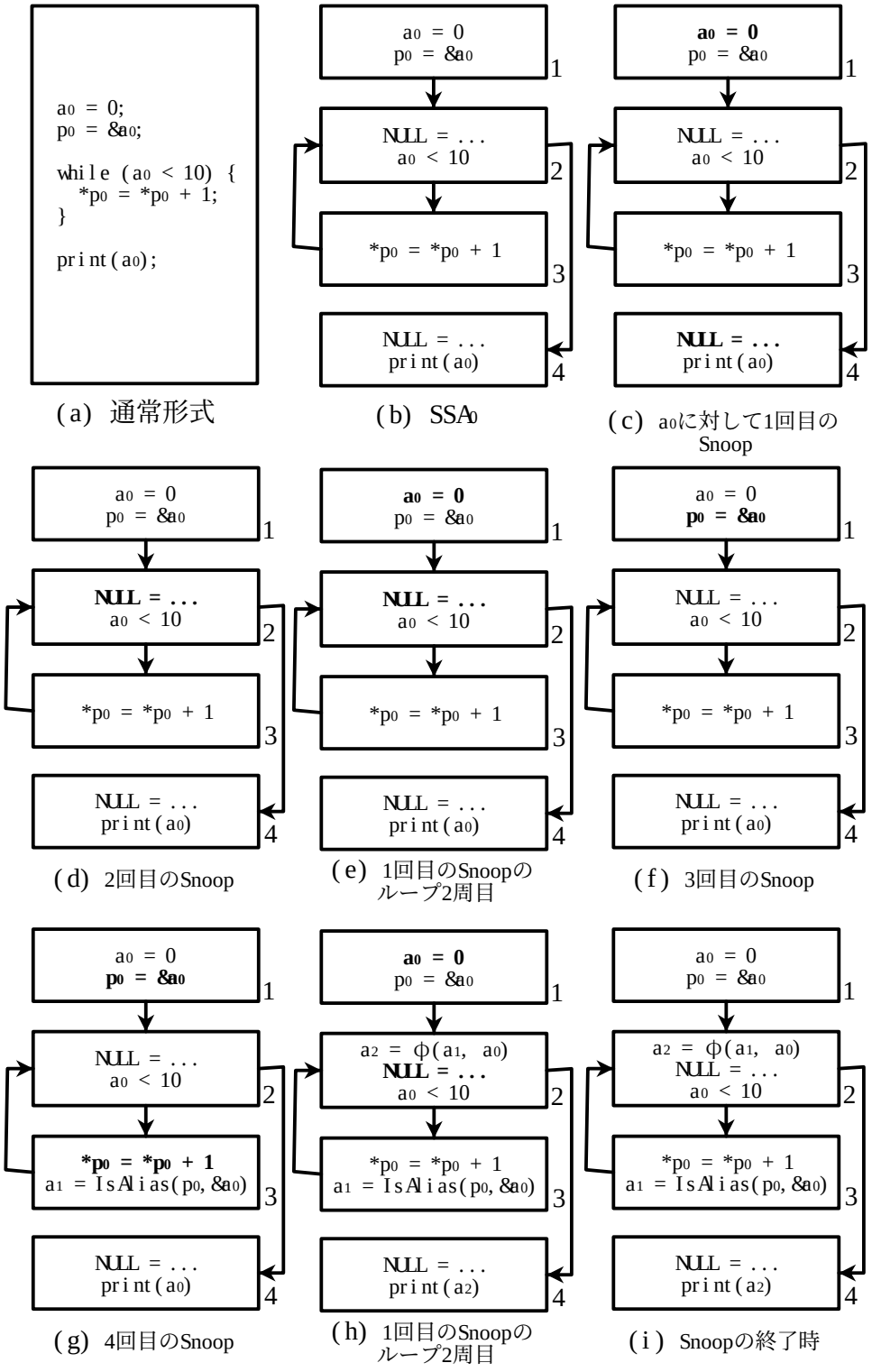


図 4.2: Cytron らのアルゴリズムの例

第5章　Hastiらのアルゴリズム

5.1　基本方針

Hasti らのアルゴリズム [19] では, SSA 変換する前にフロー非依存のポインタ解析を行い, その結果を用いてプログラムからポインタ参照を取り除いた中間形式にする (これを IM と呼ぶ). ポインタ参照を取り除くためには, そのポインタが指している可能性のある変数すべてについて分岐を行うようにすればよい. 図 5.1 に例を示す.

この状態でも正しい SSA 形式になっているが, ポインタ解析がフロー非依存のため情報の精度は低い. 2.4 で述べたように SSA 形式はそれ自体にデータフロー解析の結果を反映しているので, 上述の方法で得られた SSA 形式に再びフロー非依存のポインタ解析を行うと, 始めに得られた情報よりも高い精度の情報が得られる (図 5.2).

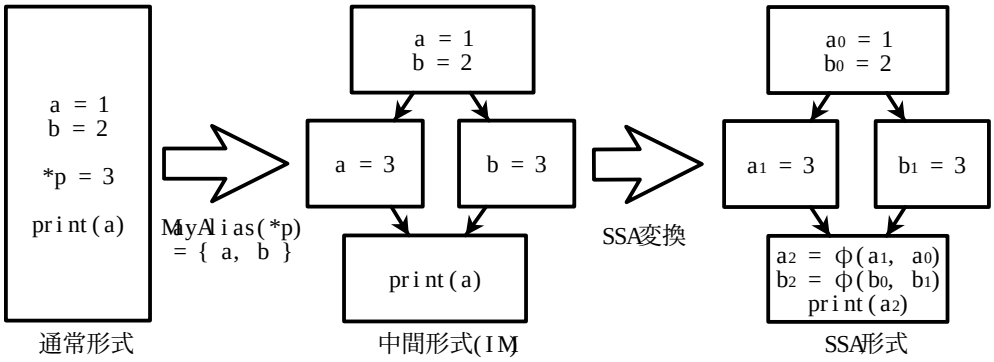
新しく得られたポインタ解析の情報を用いて, 元のプログラムを再びポインタ参照の無い中間形式に変換することで最初に得られた SSA 形式よりもポインタ解析の情報の精度が高まる. これを繰り返すことで, 最終的にはフロー依存のポインタ解析を行った場合と同じだけの情報を持った SSA 形式にすることができる.

アルゴリズムの大まかな流れは以下の通りである.

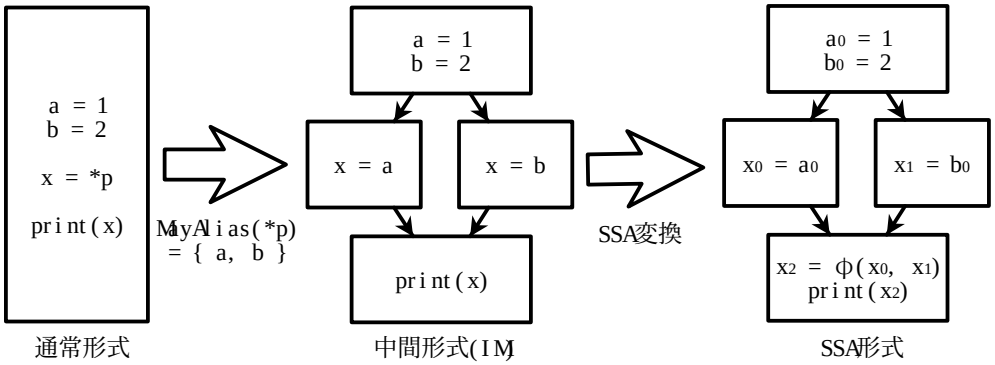
1. 元のプログラムにフロー非依存のポインタ解析を行い, 各ポインタ参照についてポインタが指す可能性のある集合を求める.
2. ポインタ解析の情報を用いてポインタ参照を分岐に置き換えて中間形式 (IM) を作成する.
3. ポインタ参照が無い中間形式は従来の方法で SSA 変換できるので, SSA 変換を行う.
4. 得られた SSA 形式に対してフロー非依存のポインタ解析を行う. ここでのポインタ解析の結果と既にあるポインタ解析の情報を比較して, それらが等しければ終了する. そうでなければ, 新しい情報を古い情報に反映させる.
5. 2. に戻る.

5.2　ポインタの問題

Hasti らのアルゴリズムでは, そもそもプログラムをポインタ参照の無い形式に変換してしまうため, 3.4 で述べたポインタ参照の問題は起こらない. また, 第 4 章で述べた Cytron



(a) ポインタ参照による定義を分岐に置き換える



(b) ポインタ参照による使用を分岐に置き換える

図 5.1: ポインタ参照を分岐で置き換える例

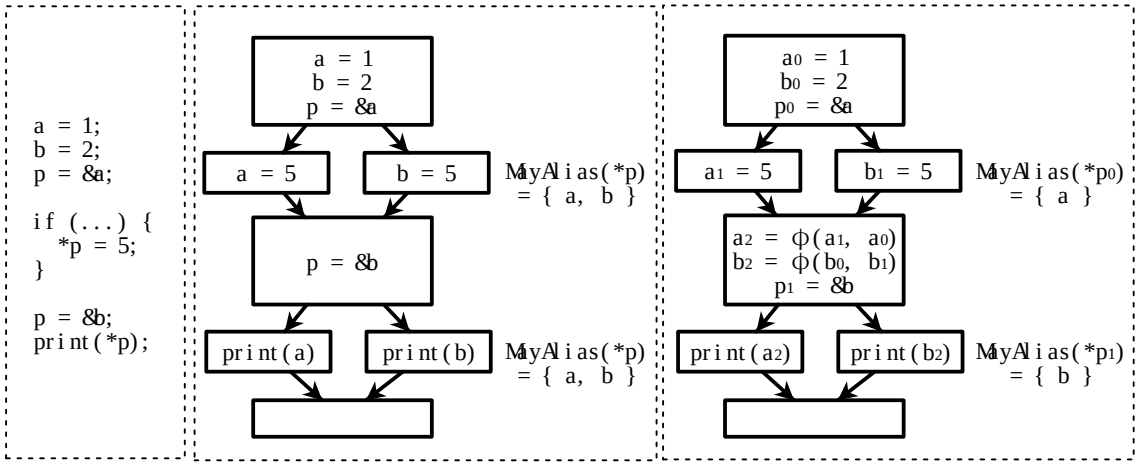


図 5.2: SSA 形式に対するフロー非依存のポインタ解析の例

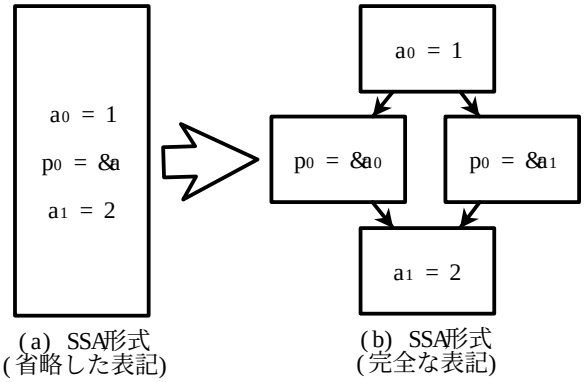


図 5.3: アドレス演算を分岐で表現する例

らのアルゴリズムとは異なり, アドレス演算の問題も解決している. 5.1 では述べていないが Hasti らのアルゴリズムでは SSA 変換後にアドレス演算に対しても分岐を導入する. 図 5.3(a) の SSA 形式は (b) のようなアドレス演算の分岐を省略した書き方である (5.1 では省略した表記を使っている. 以降も特に断りが無い限り, アドレス演算の分岐は省略した表記を用いる). 図 5.3 の例では a_0 と a_1 について分岐しているが, 一般的には $\&a$ が存在するときには SSA 形式に含まれている全ての変数 a_i について分岐する.

5.3 アルゴリズム

Hasti らのアルゴリズムは以下の通りである.

G 元のプログラムの制御フローグラフ.

S(p) プログラム中の全てのポインタ p について, p が指す可能性のある変数の集合.

IM 元のプログラムからポインタ参照を取り除いた中間形式.

$\mathbf{IM}_{ssa}$ 中間形式 IM を SSA 変換したもの.

AliasP(G) フロー非依存のポインタ解析. S(p) を返す.

NewS(p) 新しい S(p) を一時的に保持する.

アルゴリズム

/* Hasti らのアルゴリズム */
$S(p) \leftarrow AliasP(G)$
while $S(p)$ が変化した do
　　$S(p)$ の情報を用いて G から IM を作成する.
　　/* ポインタ参照を分岐で置き換える */

IM から IM_{ssa} を作成する. /* SSA 変換 */
$NewS(p) \leftarrow AliasP(IM_{ssa})$
$S(p)$ に $NewS(p)$ の内容を反映する.
end while

図 5.4 にアルゴリズムの実行例を示す.

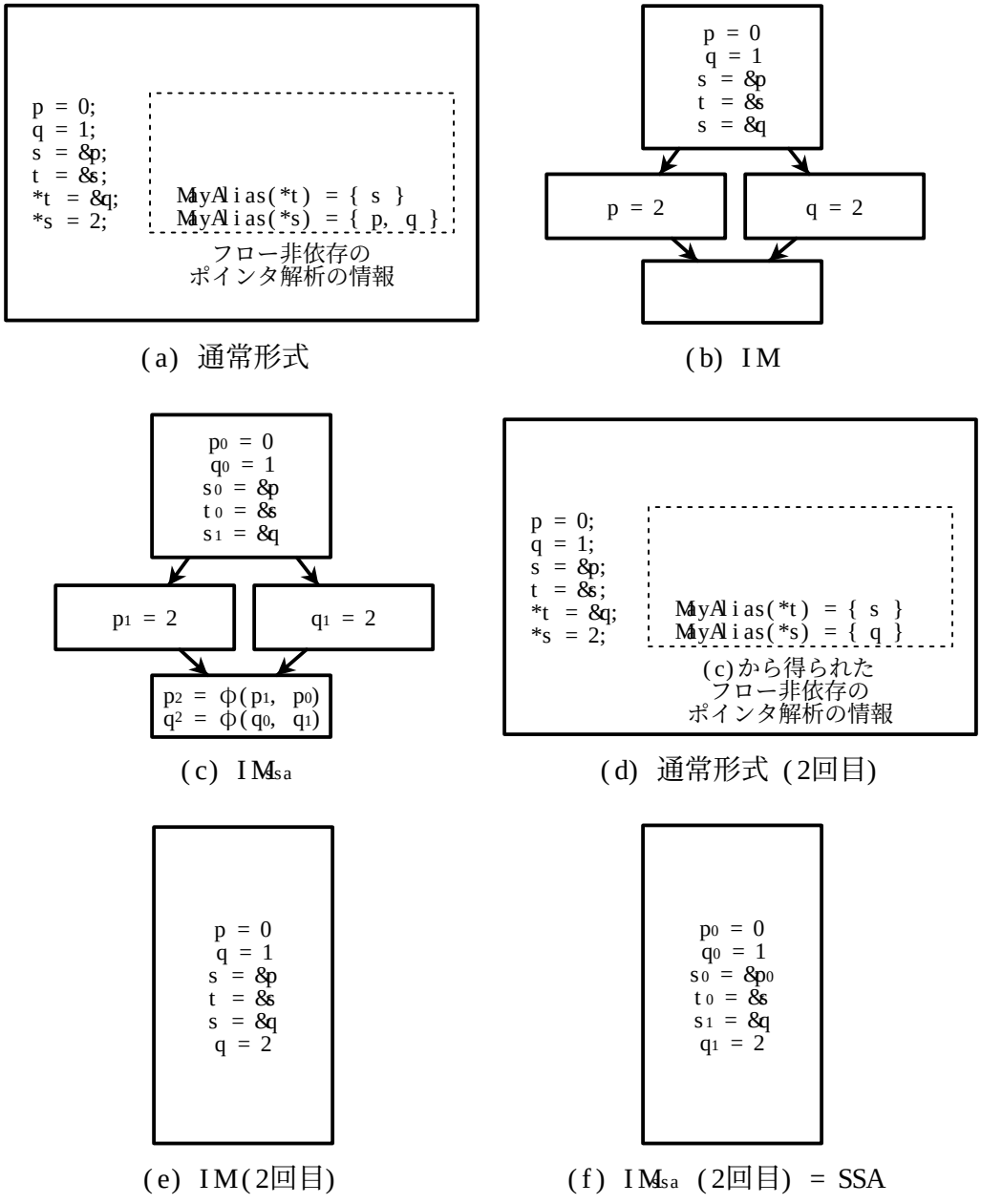


図 5.4: Hasti らのアルゴリズムの例

第6章　実装と実験

6.1　コンパイラ・フレームワークの実装

ここでは本研究で実装するコンパイラ・フレームワークの詳細な実装について述べる.

6.1.1　コンパイラ・フレームワークの概要

コンパイラ・フレームワークとは, コンパイラの再利用可能な部分を部品として提供することで, 部品を必要に応じて組み替えることができるようにしたソフトウェアである. コンパイラを拡張するために必要な部分だけを実装して追加することができるため, コンパイラ開発の効率が向上する. 一般的なコンパイラ・フレームワークは字句解析器, 意味解析器などといった基本的な機能の他にプログラムの中間表現とそれにアクセスするためのプログラムインターフェースを持つ.

本研究のコンパイラ・フレームワークはポインタ解析に基づく SSA 変換のアルゴリズムを評価するための環境であり, 可能なかぎり簡単な構成を保つように考えられている. そのために以下のような特徴を持つ.

1. 中間表現はコンパイラ・フレームワークから独立しており, その中にプログラムとして必要な情報が全て含まれている.
2. 中間表現はファイルに保存することができる.
3. 中間表現にアクセスするためのプログラムインターフェースを提供しない.

まず 1 と 2 の特徴によって中間表現のコンパイラ・フレームワークへの依存を少なくする. 一度中間表現に変換すれば, 中間表現の意味を壊さない限り, いくらでも処理を追加することができる. また 3 の特徴によって特定のプログラム言語への依存を少なくする. つまり, 複数の言語で書かれた最適化の処理を順に実行するといったことが可能になる.

これらの特徴を実現するために, 本研究では XML を採用している. XML は単純さと汎用性を兼ね備えており, 上述の特徴とは相性が良い. XML は単純なテキストであり, なおかつプログラムを容易にする利用可能な資産が豊富であるため, 特に 2 と 3 の特徴を実現するのに適している.

1.3 の図 1.1(3 ページ) の全体図で, 評価環境はいくつかの部分に分けることができる. 1 つはコンパイラ・フレームワークであり, ANSI C 言語をサブセットとするコンパイラ (XCC) とその中間表現としての XML(XC-XML) から構成される. そして残りは比較対象

のアルゴリズムと最適化器であり, どちらも中間表現を入出力とする.
以下ではコンパイラ (XCC) とその中間表現 (XC-XML) について説明する.

6.1.2 XCC

XCC(eXperimental C Compiler) は北陸先端科学技術大学院大学のソフトウェア環境特論 (i425) でコンパイラ実装の課題として出題される, ANSI C 言語のサブセット (XC 言語) を対象としたコンパイラである. 実装は C 言語, FLEX 2.5.4, BISON 1.28 にて行った. 本研究では, この XCC を拡張したものをコンパイラとして利用する.
XCC は以下の制限を持つ.

- 型に関する制限
  - 原始型は int, char, void のみである. 以下は利用できない.
    - 型指定子 (long, short, unsigned, signed)
    - 型修飾子 (const, volatile)
    - 記憶クラス (static, extern, register, auto, typedef)
  - 派生型は関数, ポインタ, 配列のみである. 以下は利用できない.
    - 構造体 (struct)
    - 共用体 (union)
    - 列挙型 (enum)
- 構文に関する制限
  - 1 つの宣言文で宣言できる変数は 1 つだけである.
  - 制御文は if, if-else, while, goto, return のみである. 以下は利用できない.
    - for, do-while, switch, break continue
- その他の制限
  - 比較演算子は $<$ と $==$ のみである.
  - インクリメント演算子 ($++$) とデクリメント演算子 ($--$) は使用できない.
  - sizeof 演算子は使用できない.

制御文や式に関しては XC 言語の持つもので ANSI C 言語の大部分を実現できる. データ型については 6.3 でも議論するが本研究の目的には十分である. つまり, XC 言語は ANSI C 言語のサブセットではあるが, 本研究の目的であるコンパイラ・フレームワーク用の言語として, ほぼ十分な記述力を持つ. XC 言語の問題点は, 既存のプログラムをそのままコンパイルすることができず, 制御文などを XC 言語用に変換する必要があることである.

しかし, XC 言語は構文解析を難しくする要素を出来るだけ省くことで実装を容易にし, なおかつ目的に十分な表現力をもっているため, 本研究で利用するメリットは大きい.

6.1.3 XC-XML

XC-XML は XCC の中間表現を XML を用いて表したものである. XC-XML は SSA 変換を行うことを前提に考えられており, 制御フロー解析や変数名の付け替えが容易になるようになっている.

XC-XML は次の特徴を持つ.

1. 制御文は全て分岐とラベルによって表す. 従って, この点では C 言語の抽象構文木よりもアセンブラに近い構造を持つ.
2. 変数とそのスコープを保持する. 従って, この点ではアセンブラよりも C 言語の抽象構文木に近い構造を持つ.

1 の特徴により制御フローグラフを構成するための基本ブロックの抽出が単純になり, 2 の特徴によって変数の追加と削除, 変数名の変更が容易になる.

XC-XML の概要

XC-XML は 2 つの基本的な要素を持つ. 1 つは func 要素であり, もう 1 つは sym_tab 要素である. トップレベルの xc-xml 要素は func 要素と sym_tab 要素をそれぞれ子要素として複数個持つ (図 6.1). func 要素は関数定義を表し, sym_tab 要素は関数に含まれる変数スコープ毎のシンボルテーブルを表している.

func 要素は変数スコープを表す scope 要素を 1 つだけ持ち, scope 要素は自分自身かあるいは命令文 (%instr;) の列を持つ. また scope 要素は対応する sym_tab 要素を ID 属性で参照している.(図 6.2)

sym_tab 要素は XML の構造上では全て xc-xml の子要素であり, 並列になっているが, ID 属性の参照による構造を持っている. グローバル sym_tab を除く全ての sym_tab が func あるいは scope から ID 属性で参照される。(図 6.2)

sym_tab 要素はシンボルを表す sym 要素か文字リテラルを表す str 要素を持つ. sym 要素は 1 つだけ type 要素を持ち, これは型を表す. また sym 要素は関数に含まれる変数の使用を表す var 要素から ID 属性で参照される. 同様に str 要素は定数を表す const 要素から参照される.

XC-XML は以下の命令 (%instr;) 及び型 (type) を持つ.

- 命令の種類
  - 二項演算 (binary_op)
    - 比較演算 (eq, lt)

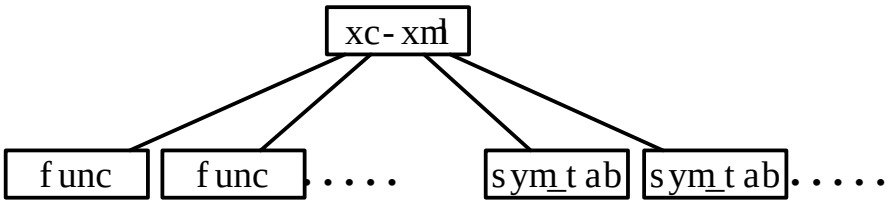


図 6.1: xc-xml 要素

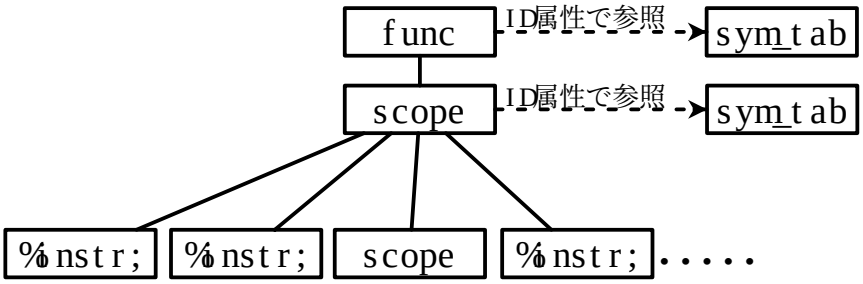


図 6.2: func 要素

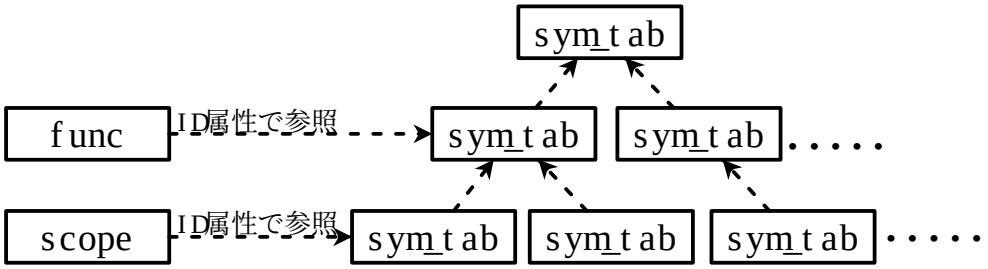


図 6.3: sym_tab 要素 (1)

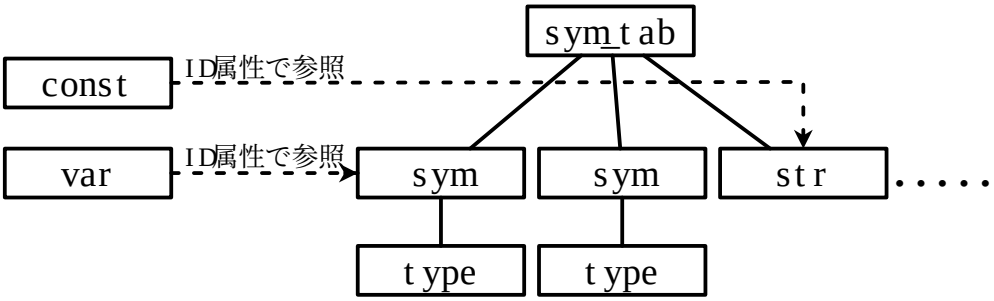


図 6.4: sym_tab 要素 (2)

        * 論理演算 (and, or)
        * 四則演算 (add, sub, mul, div)
    - 単項演算 (unary_op)
        * アドレス演算 (addr)
        * ポインタ参照 (ptr)
        * 正負 (plus, minus)
        * 否定 (not)
    - 代入 (assign)
    - フロー制御
        * 条件付き分岐 (branch_z)
        * 無条件分岐 (branch)
        * ラベル (label)
    - 配列 (array)
    - 関数呼び出し (call, ret)
    - 変数 (var), 定数 (const)
    - 括弧 (paren)
- 型の種類
    - 基本型 (prim)
        * int, char, void
    - 関数 (func)
    - ポインタ (ptr)
    - 配列 (array)

付録 1(58 ページ) に XC-XML の DTD を示す実際.

6.2 実験

実装したコンパイラ・フレームワーク上でプログラムをコンパイルし, 正しく動くことを確認する. また同時にコンパイル時間, 実行時間などの性能に関わる値を調べる.

実験の目的は次の点を明らかにすることである.

1. コンパイラ・フレームワークが正しく動作すること.
2. SSA 変換が正しく動作すること.

3. ポインタを含むプログラムの SSA 変換が正しく動作すること.
4. 最適化が正しく動作すること.
5. これらの動作の実際の性能を表す値を測定する.

ここでの実験は最終的なアルゴリズム比較のための予備実験である. 評価環境の性能測定が主な目的であり, 第 4 章, 第 5 章で述べたポインタ解析を用いたアルゴリズムが正しく動くことの確認のみを行っている. 実験環境は Sun Blade 1500(SPARC), Solaris8 である. 実験はいくつかの小さなテスト用プログラムを使用して行っているが, これらのアルゴリズムの実用性を比較するための本実験では, 実際に利用されている大きなプログラムについて同様の実験を行う必要があるだろう. この点については 6.3 で議論する.

6.2.1 実験手順

まず実験を行うためのテスト用のプログラムとして次のプログラムを準備した. これらは XC 言語によって記述されている.

(a) フィボナッチ数列の第 n 項を求めるプログラム.
(b) 動的計画法 (ナップザック問題) のプログラム.
(c) 素数を求めるプログラム (エラトステネスのふるい).
(d) 素因数分解のプログラム.
(e) 3.4 で述べたポインタ参照の問題を含むプログラム.
(f) 3.4 で述べたアドレス演算の問題を含むプログラム.

(a) から (d) のプログラムについては, それぞれポインタを含むものと含まないものを準備した. 以降, ポインタを含むものは (a'), (b') のように表記する.

これらのプログラムを用いて以下の手順で実験を行った.

1. 全てのプログラムを XCC によって XC-XML に変換し, そのまま何もせずに XCC によってアセンブラを生成する. これによってコンパイラが正しく動作していることを確認する.
2. 全てのポインタを含まないプログラム (a) から (d) について従来の方法とポインタ解析に基づく方法による SSA 変換をそれぞれ行い, そのまま何もせずに SSA 逆変換によって戻す. どちらの方法でも正しい結果が得られることを確認する.
3. 全てのポインタを含むプログラム (a') から (d'), (e), (f) について従来の方法とポインタ解析に基づく方法による SSA 変換をそれぞれ行い, そのまま何もせずに SSA 逆変換によって戻す. 従来の方法では正しい結果が得られないこと, ポインタ解析に基づく方法では正しい結果が得られることを確認する.
4. 全てのポインタを含まないプログラム (a) から (d) を従来の方法によって SSA 変換を行い, 定数伝播と無用命令削除の最適化を施す. これによって最適化が正しく動作し

ていることを確認する.

5. 全てのポインタを含むプログラム (a') から (d'), (e), (f) をポインタ解析に基づく方法で SSA 変換を行い, 定数伝播と無用命令削除の最適化を施す. これによってポインタを含むプログラムの最適化が正しく動作していることを確認する.

コンパイラ・フレームワークの性能を評価するための指針として以下の値を測定する.

- コンパイラの効率
  - 手順 1 においてコンパイルに要した時間を測定する.
- 最適化の効率
  - 手順 1 で得た実行ファイルと手順 4, 5 で得た実行ファイルの実行時間を測定する. (最適化ありの場合と最適化なしの場合の実行時間を比較する)
  - 手順 1 で得たアセンブラと手順 4, 5 で得たアセンブラのコードの行数を調べる. (最適化ありの場合と最適化なしの場合のコードの行数を比較する)
  - 手順 4, 5 でコンパイルに要した時間のうち, 最適化にかかった時間の割合を調べる.
- SSA 変換の効率
  - 手順 3 でコンパイルに要した時間のうち, SSA 変換にかかった時間の割合を調べる.

6.2.2 結果

表 6.1 から表 6.5 に動作確認の結果を, 表 6.6 から表 6.10 には性能測定の結果を示す.

まず SSA 変換や最適化を全く行わず, プログラムを XCC で XC-XML に変換し, そのまま XCC でアセンブラに変換した場合の動作確認が表 6.1 である. 全てのプログラムが正しく動作している. XCC はポインタに対応しているため, ここではポインタは問題とはならない.

表 6.2 と表 6.3 では SSA 変換の動作確認を行っている. プログラムは XCC で XC-XML に変換された後, それぞれの SSA 変換器によって SSA 変換される. そして最適化は行わずに SSA 逆変換を行い, 最後に XCC でアセンブラに変換する. ポインタが存在しない場合にはどの SSA 変換を用いてもプログラムは正しく動作する. しかし, ポインタが存在する場合には従来の SSA 変換では問題が起きている. 表 6.3 で, △マークはプログラムの実行結果は正しかったが, SSA 変換された結果が正しい SSA 形式になっていない場合である. ここでは最適化を行っていないために, SSA 変換が間違っていてもプログラムは動くことが多い. (e) と (f) のプログラムは意図的にポインタの問題が起こるように書かれており, これらのプログラムは正しく動作しないことが確認できた. また 4.2 で述べたように,

Cytron らのアルゴリズムではポインタ演算の問題を含む (f) のプログラムは正しく動作しない.

表 6.4 と表 6.5 では最適化の動作確認を行っている. ここで行った最適化は定数伝播と無用命令削除であり, 最適化としての効果はあまり高くはない. Cytron らの方法を行った後の最適化で (f) のプログラムが×マークになっているのは最適化が正しく動かないからではなく, SSA 変換自体が間違っているためである.

表 6.6 ではコンパイルにかかった時間を XCC と GCC で比較している. XCC はアセンブラまでの変換しか行わないため, GCC はオプション-S を付けている. 結果はおよそ 2 倍から 6 倍にかけて XCC のコンパイルの方が遅かった. これは XCC が一度 XC-XML をファイルとして出力し, 次に XC-XML を入力としてアセンブラを出力するという 2 段階の処理に分かれていることが大きな原因であると予測される.

表 6.7 と表 6.8, 表 6.9 では最適化の効率を調べている. 表 6.7 ではプログラムの実行時間を測定しているが, 最適化なしと最適化ありではほとんで有意の差は見られない. これはテスト用のプログラムが小さいものばかりであることと, 最適化が効果の薄いものであることが原因だろう. 時間の計測には UNIX の gettimeofday 関数を使用しておりマイクロ秒までの測定が可能であるが, 実際の精度はミリ秒程度である. 表 6.8 ではアセンブラの行数の比較を行っている. 最適化の結果, ある程度プログラムが小さくなっていることがわかる. 表 6.9 では最適化にかかった時間を計測しているが, 実行時間の場合と同じように値の信頼度は低い. この問題については 6.3 でも言及する.

表 6.10 では SSA 変換にかかった時間を比較している. Cytron らのアルゴリズムの方が若干速いようにも見えるが, 確実に速いとは到底言い切れない. アルゴリズムの比較を正確に行うためには, 別のテスト用プログラムを準備した本実験を行う必要がある. これについては 6.3 で述べる.

実験の結果をまとめると以下のようになる.

- ここで実装したコンパイラ・フレームワークが XC 言語に対して正常に動作することが確認できた.
- コンパイラ・フレームワークに SSA 変換器及び最適化器を追加して, それぞれが正しく動作することが確認できた.
- コンパイラ・フレームワークの性能は実用のコンパイラ (GCC) に比べて 2 倍から 6 倍程度悪いことが分かった.
- コンパイラフレームワークに追加した SSA 変換器及び最適化器の性能に関してほとんどの計測結果が誤差の範囲にあると思われるため, はっきりとしたことはわからなかった. これについては, 更に実験を行う必要がある.

テスト用プログラム	動作チェック
(a) フィボナッチ数列	○
(a') フィボナッチ数列 (ポインタあり)	○
(b) ナップザック問題	○
(b') ナップザック問題 (ポインタあり)	○
(c) エラトステネスのふるい	○
(c') エラトステネスのふるい (ポインタあり)	○
(d) 素因数分解	○
(d') 素因数分解 (ポインタあり)	○
(e) ポインタ参照の問題 (ポインタあり)	○
(f) アドレス演算の問題 (ポインタあり)	○

表 6.1: 手順 1. コンパイラ・フレームワークの動作チェック

テスト用プログラム	動作チェック		
	従来の SSA 変換	Cytron らの方法	Hasti らの方法
(a) フィボナッチ数列	○	○	○
(b) ナップザック問題	○	○	○
(c) エラトステネスのふるい	○	○	○
(d) 素因数分解	○	○	○

表 6.2: 手順 2. SSA 変換の動作チェック (ポインタなし)

テスト用プログラム	動作チェック		
	従来の SSA 変換	Cytron らの方法	Hasti らの方法
(a') フィボナッチ数列	△	○	○
(b') ナップザック問題	△	○	○
(c') エラトステネスのふるい	△	○	○
(d') 素因数分解	△	○	○
(e) ポインタ参照の問題	×	○	○
(f) アドレス演算の問題	×	×	○

表 6.3: 手順 3. SSA 変換の動作チェック (ポインタあり)

テスト用プログラム	動作チェック
(a) フィボナッチ数列	○
(b) ナップザック問題	○
(c) エラトステネスのふるい	○
(d) 素因数分解	○

表 6.4: 手順 4. 最適化の動作チェック (ポインタなし, SSA 変換は従来の方法)

テスト用プログラム	動作チェック	
	Cytron らの方法	Hasti らの方法
(a') フィボナッチ数列	○	○
(b') ナップザック問題	○	○
(c') エラトステネスのふるい	○	○
(d') 素因数分解	○	○
(e) ポインタ参照の問題	○	○
(f) アドレス演算の問題	×	○

表 6.5: 手順 5. 最適化の動作チェック (ポインタあり)

テスト用プログラム	コンパイル時間 [秒]	
	XCC	GCC (gcc -S)
(a) フィボナッチ数列	0.119	0.053
(a') フィボナッチ数列 (ポインタあり)	0.176	0.054
(b) ナップザック問題	0.221	0.055
(b') ナップザック問題 (ポインタあり)	0.308	0.057
(c) エラトステネスのふるい	0.081	0.053
(c') エラトステネスのふるい (ポインタあり)	0.096	0.053
(d) 素因数分解	0.097	0.054
(d') 素因数分解 (ポインタあり)	0.144	0.055
(e) ポインタ参照の問題 (ポインタあり)	0.035	0.053
(f) アドレス演算の問題 (ポインタあり)	0.033	0.053

表 6.6: コンパイル時間の比較 (SSA 変換なし)

テスト用プログラム	実行時間 [秒]		
	最適化なし	最適化あり	GCC による最適化 (-O3)
(a) フィボナッチ数列	0.004	0.004	0.004
(a') フィボナッチ数列 (ポインタあり)	0.004	0.004	0.004
(b) ナップザック問題	0.005	0.005	0.004
(b') ナップザック問題 (ポインタあり)	0.005	0.005	0.004
(c) エラトステネスのふるい	0.145	0.135	0.092
(c') エラトステネスのふるい (ポインタあり)	0.136	0.112	0.108
(d) 素因数分解	221.359	201.895	33.063
(d') 素因数分解 (ポインタあり)	165.299	164.304	164.276
(e) ポインタ参照の問題 (ポインタあり)	0.004	0.004	0.004
(f) アドレス演算の問題 (ポインタあり)	0.004	0.004	0.004

表 6.7: 最適化なしと最適化ありの実行時間の比較 (SSA 変換は従来の方法)

テスト用プログラム	アセンブラの行数 [行]	
	最適化なし	最適化あり
(a) フィボナッチ数列	389	297
(a') フィボナッチ数列 (ポインタあり)	441	321
(b) ナップザック問題	647	556
(b') ナップザック問題 (ポインタあり)	640	565
(c) エラトステネスのふるい	268	226
(c') エラトステネスのふるい (ポインタあり)	269	227
(d) 素因数分解	381	369
(d') 素因数分解 (ポインタあり)	375	363
(e) ポインタ参照の問題 (ポインタあり)	77	73
(f) アドレス演算の問題 (ポインタあり)	73	73

表 6.8: 最適化なしと最適化ありのコード量の比較 (SSA 変換は従来の方法)

テスト用プログラム	最適化時間 [秒] (括弧内は最適化時間÷コンパイル時間 [%])
(a) フィボナッチ数列	0.187 (28.8%)
(a') フィボナッチ数列 (ポインタあり)	0.247 (27.1%)
(b) ナップザック問題	0.295 (29.7%)
(b') ナップザック問題 (ポインタあり)	0.273 (28.5%)
(c) エラトステネスのふるい	0.083 (25.1%)
(c') エラトステネスのふるい (ポインタあり)	0.091 (26.2%)
(d) 素因数分解	0.152 (28.1%)
(d') 素因数分解 (ポインタあり)	0.152 (28.7%)
(e) ポインタ参照の問題 (ポインタあり)	0.028 (25.5%)
(f) アドレス演算の問題 (ポインタあり)	0.025 (24.0%)

表 6.9: 最適化時間の比較 (SSA 変換は従来の方法)

テスト用プログラム	SSA 変換時間 [秒] (括弧内は SSA 変換時間÷コンパイル時間 [%])		
	従来の SSA 変換	Cytron らの方法	Hasti らの方法
(a') フィボナッチ数列	0.345 (42.0%)	1.111 (60.1%)	0.841 (63.7%)
(b') ナップザック問題	0.315 (38.1%)	0.488 (48.7%)	1.032 (63.0%)
(c') エラトステネスのふるい	0.111 (37.1%)	0.158 (45.6%)	0.265 (55.2%)
(d') 素因数分解	0.143 (36.5%)	0.233 (48.4%)	0.277 (43.4%)
(e) ポインタ参照の問題	0.036 (42.8%)	0.027 (34.3%)	0.041 (43.4%)
(f) アドレス演算の問題	0.031 (41.1%)	0.027 (37.6%)	0.039 (44.0%)

表 6.10: SSA 変換時間の比較 (コンパイル時間は最適化を行わない場合のもの)

6.3 議論

6.3.1 コンパイラ・フレームワークの有用性

6.2 ではポインタ解析に基づく SSA 変換アルゴリズムを比較するための評価環境としてコンパイラ・フレームワークの実装をし, その実験を行った. ここではいくつかの点から, このコンパイラ・フレームワークがアルゴリズム評価のための環境として有用であることを述べる.

適用範囲

コンパイラ・フレームワーク上でポインタ解析に基づく SSA 変換を比較するためには, ポインタ等の機能にコンパイラ・フレームワークが対応していなくてはならない. 図 6.11 に, 今回実装したコンパイラ・フレームワーク, SSA 変換器, 最適化器が対応する機能をまとめた.

図 6.11 の記号 (○, △, ×) の意味は次の通りである. まずコンパイラ・フレームワークは入力言語にその機能を含むことが出来る場合には○, 出来ない場合は×とする. SSA 変換器と最適化器ではその機能を含んでいるときに SSA 変換あるいは最適化が出来る場合は○, 出来ない場合は×である. △はそれ以外のケースである. 例えば, SSA 変換できない変数は SSA 変換を行わないでおくことができる場合がある. 配列とグローバル変数がそれにあたる. この 2 つは SSA 変換を行うことができないがプログラム中に存在する分には問題ない. なぜなら, SSA 変換は常に全ての変数に対して行う必要はないからである (但し, SSA 変換を行わなかった変数は SSA 変換による最適化はできない). 同様に最適化器でも配列とグローバル変数は最適化を行わないだけで存在しても良い. また関数呼び出しも存在する分にはいくらあってもよいが, SSA 変換の対象とできるのは 1 つの関数のみである. goto 文には変数のスコープの内側から外側へのジャンプはできないという制限がある.

これらの制限は変数スコープと制御フローグラフの扱いに関係がある. 従来の SSA 変換では対象とする全ての変数がプログラムの入口の基本ブロックで定義されていることを仮定している. もしもプログラムが複数の変数スコープを持っていた場合, 入口のブロックではアクセスできない変数が存在する可能性がある. 従って, 異なるスコープに属する変数を SSA 変換するためにはそのスコープ毎の部分的な制御フローグラフを考慮する必要がある. Cytron らの方法と Hasti らの方法でも従来の SSA 変換を部分的に利用しているために同様の制限を持っている.

図 6.11 が示す通り, コンパイラ・フレームワークは最も多くの機能に対応している. Cytron らの方法や Hasti らの方法ははじめから構造体, 共用体, 配列, 関数呼び出し等に対応していないため, これらのアルゴリズムを比較する分には十分な機能を持っているといえる.

	コンパイラ・フレームワーク	SSA 変換器			最適化器
		従来の方法	Cytron らの方法	Hasti らの方法	
構造体, 共用体	×	×	×	×	×
ヒープ (malloc)	○	×	×	×	×
配列	○	△	△	△	△
グローバル変数	○	△	△	△	△
ポインタ参照	○	×	○	○	○
アドレス演算	○	×	×	○	○
関数呼び出し	○	△	×	×	×
goto 文	○	△	△	△	○

表 6.11: 実装したコンパイラ・フレームワーク, SSA 変換器, 最適化器の適用範囲

XML による中間表現の利点

本研究のコンパイラ・フレームワークは中間表現に XML を用いたことで, 次のような利点を持つ.

- コンパイラ・フレームワークに追加するプログラムの実装言語を自由に選択できる.
- コンパイラ・フレームワークに追加するプログラムは XC-XML を入力として XC-XML を出力とするため, 各プログラム間での依存が少なくなる.
- XC-XML はプログラムの情報としてはそれのみで完結しているため, XC-XML を本来の目的以外の所で利用できる. (例えば, プログラム理解など)
- XML の関連技術を利用できる. XML には既に利用可能なツールが沢山あるので, 自分で XML をパースするプログラムを書く必要もないし, ツリーの検索には XPath, 変換には XSLT なども利用できる.

XML の最大の欠点は性能である. XML はテキストであるのでバイナリを使用する場合に比べてファイルサイズは大きくなる. また, XML を用いた多くの処理が C 言語などを用いて書いた専門のプログラムに比べて遅くなるだろう. しかし本研究ではアルゴリズムの比較を目的としているため, 性能が多少実用的ではなくてもアルゴリズムを容易に実装, 比較できることを重要視している.

今回のアルゴリズムの実装では, 全て C 言語を使用したために実装言語に依存しない利点は活かされなかったが, より高級な言語やスクリプト言語を用いることで実装時間はさらに短縮することができると考えている. 図 6.12 に SSA 変換器, 最適化器などの実際の実装期間と実装言語を示す.

	実装にかかった期間	実装に用いた言語	コードの行数
XCC	2ヶ月	C 言語 + FLEX + BISON	約 6000 行
従来の SSA 変換	1 週間	C 言語 + libxml2	約 3000 行
Cytron らの方法	5 日	C 言語 + libxml2	従来の SSA 変換 + 約 1500 行
Hasti らの方法	3 日	C 言語 + libxml2	従来の SSA 変換 + 約 1000 行
定数伝播	1 日	C 言語 + libxml2	約 300 行
無用命令削除	1 日	C 言語 + libxml2	約 300 行

表 6.12: 実装期間, 実装言語, コードの行数

他のコンパイラ・フレームワークとの違い

6.1.1 で述べたように, 本研究では評価環境を実装するにあたり, コンパイラ・フレームワークから独立し, かつプログラム言語に依存しない中間表現を基本としていた. しかし, 現在存在する多くのコンパイラ・フレームワークは中間表現にアクセスするためのプログラムインターフェースが中心となっている. 我々が実装したコンパイラ・フレームワークは性能や機能において明らかに他のものに劣っているが, それらにはない利点も同時に持っている. それは以下の通りである.

- プログラムインターフェースを提供しないことでプログラム言語の自由を確保している. アルゴリズムの実装と比較を目的としているので, 性能が高くても手間がかかる言語を強制されるよりも, その時その時で生産性の高い言語を利用できることの方がメリットがある.
- コンパイラ・フレームワーク自体が小さい. 多くの機能を持たない代わりに必要最小限の実装を行っているため, 小さくて分かり易いものとなる. また XC-XML の仕様も単純であるので, すべて理解した上で平等な比較が行える.

まとめと問題点

これまでに主張してきた我々のコンパイラ・フレームワークの有用性をまとめると, 次のようになる.

- ポインタ解析に基づく SSA 変換のアルゴリズムを扱うのに十分な機能を持っている.
- プログラム言語を自由に選択できるため, 実装の生産性が向上する.
- プログラムインターフェースがない不便を XML 関連技術が補っている.

- 中間表現が簡単な構造を持ち, コンパイラ・フレームワーク自体も小さいため理解がしやすい.
- 実際にアルゴリズムを実装しコンパイラ・フレームワーク上で動かして確認した.

しかし, 解決できていない問題も存在する. 確かにアルゴリズムを実装して動作を確認することはできたが, それらのアルゴリズムを実際の性能を比較するためには, まだ不十分な点が残っている. これについては次の 6.3.2 で述べる.

6.3.2 アルゴリズムの比較

本研究の最終的な目標は, コンパイラ・フレームワーク上にアルゴリズムを実装して, それらを実際に動かして比較することである. その対象として Cytron らのアルゴリズムと Hasti らのアルゴリズムを採用し, 実装を行った. しかし現段階ではまだ, これらの比較は行うことは難しい. 現段階では正確な比較が行えない大きな理由としては入力となるプログラムが C 言語のサブセットであることが挙げられる. 今回の実験では入力プログラムを全て自前で準備したため, 一般的なプログラムでの実験は行っていない. それに加えて, アルゴリズムの実用上の比較を行うためには, アルゴリズムの最悪の場合を考えてそれがどの程度現実に表れるかを考慮する必要がある.

第7章　関連研究

7.1　コンパイラ・フレームワークの関連研究

コンパイラ・フレームワークの関連研究として, NCI(National Compiler Infrastructure) や COINS(COmpiler INfraStructure), また複数言語を共通の中間表現に変換するという点から GCC(GNU Compiler Collection) などが挙げられる.

7.1.1　NCI

NCI(National Compiler Infrastructure)[1] は米国のコンパイラ基盤作成プロジェクトであり, すでに多くのコンパイラの開発や研究が行われている. 複数の言語, 機種に対応する共通の中間表現である SUIF(Stanford University Intermediate Format)[2] をはじめとして, オブジェクト指向言語の最適化に向いた OSUIF[3] や機械語に近いレベルの最適化に向いた MachSUIF(Machine SUIF) [4] などのプロジェクトが存在する. 特に MachSUIF では SSA 形式による最適化も研究されている.

7.1.2　COINS

COINS(COmpiler INfraStructure)[8] は日本の文部科学省科学技術振興調整費に基づくプロジェクトであり, 新しいコンパイラ方式を容易に実験, 評価できるような共通インフラストラクチャの開発を目的としている. COINS は複数の言語, 機種に対応した高水準中間表現と低水準中間表現を持ち, それぞれのレベルで最適化を行う. SSA 形式による最適化の研究も行われている.

7.1.3　GCC

GCC(GNU Compiler Collection)[6] は GNU プロジェクト [5] によるコンパイラである. GCC は複数の言語, 機種に対応しており, 現在広く利用されているコンパイラの 1 つである. GCC の中間表現は RTL(Register Transfer Language) と呼ばれる機械語のレベルに近いものである. SSA 形式による最適化の実装は行われているようだ [7].

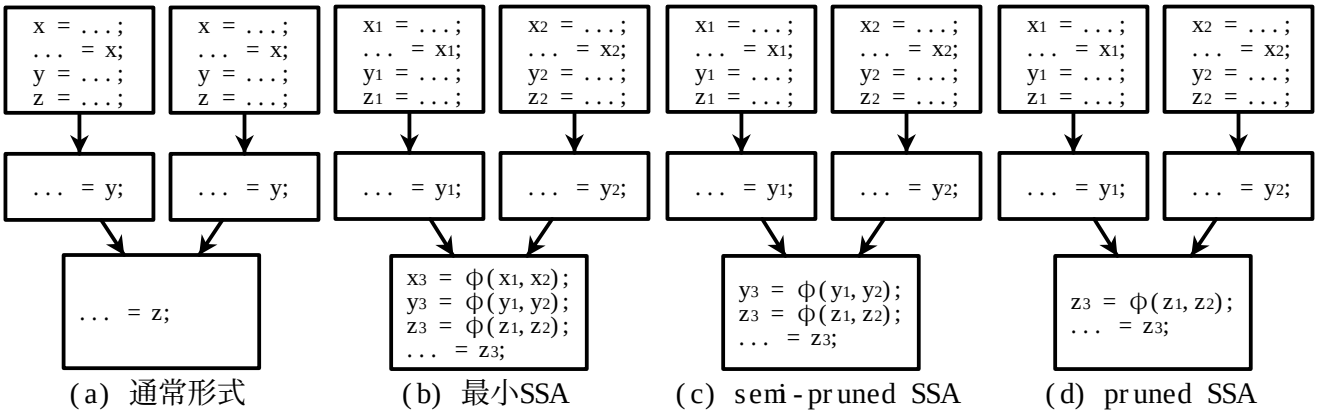


図 7.1: 3 種類の SSA 形式の違い

7.2 SSA 変換の関連研究

SSA 形式への変換の手法はいくつか知られているが, 第 3 章で述べたアルゴリズムで得られる SSA 形式は最小 SSA 形式である. 最小 SSA 形式とは SSA 形式の定義を満たすために必要な最小の SSA 形式であり, 実際には不要な φ 関数が挿入される可能性がある. そのような φ 関数が存在しない SSA 形式は pruned SSA[13] と呼ばれる. この形式では φ 関数が少なくなるので, その後の最適化の効率も良くなる. しかし, 不要な φ 関数を挿入しないためには SSA 変換時に変数の生死も解析する必要がある. また, 一見不要な φ 関数でも, それが存在することである種の最適化が可能になることがあるため, 中間的な SSA 形式として semi-pruned SSA [15] が提案されている. これは 1 つの基本ブロックの中だけで定義され使用される変数については φ 関数を作らないというものである. これら 3 種類の SSA 形式の違いを図 7.1 に示す.

第 3 章で述べた SSA 形式から通常形式への逆変換では最適化の内容によってはプログラムの意味を変えてしまう可能性がある. 例えば図 7.2(a) の SSA 形式にコピー伝播の最適化を行うと (c) のようになる. この状態で単純な SSA 逆変換を行うと (d) となりこれは元のプログラムとは違ってしまっている. この問題を解決したアルゴリズムとして, Briggs らの方法 [15] や Sreedhar らの方法 [17] などがある. Briggs らの方法はコピーが多くなってしまうため, グラフカラーリングによるレジスタアロケーション [16] を応用して変数の生存区間の合併を行うことで, その多くを削除できるとしている. Sreedhar らの方法は φ 関数内の変数間の生存区間の干渉を調べて, 干渉のないものを同じ変数で置き換えることで φ 関数を除去する.

第 4 章, 第 5 章で説明したアルゴリズムの他にも別名解析のアルゴリズムはいくつか存在する. 例えば, SSA の番号付けを拡張することで別名解析を行う Lapkowski らの方法 [20] や手続き間別名解析を行うために Sparse Evaluation Graph と呼ばれるデータ構造を用いる Choi らの方法 [21] などがある.

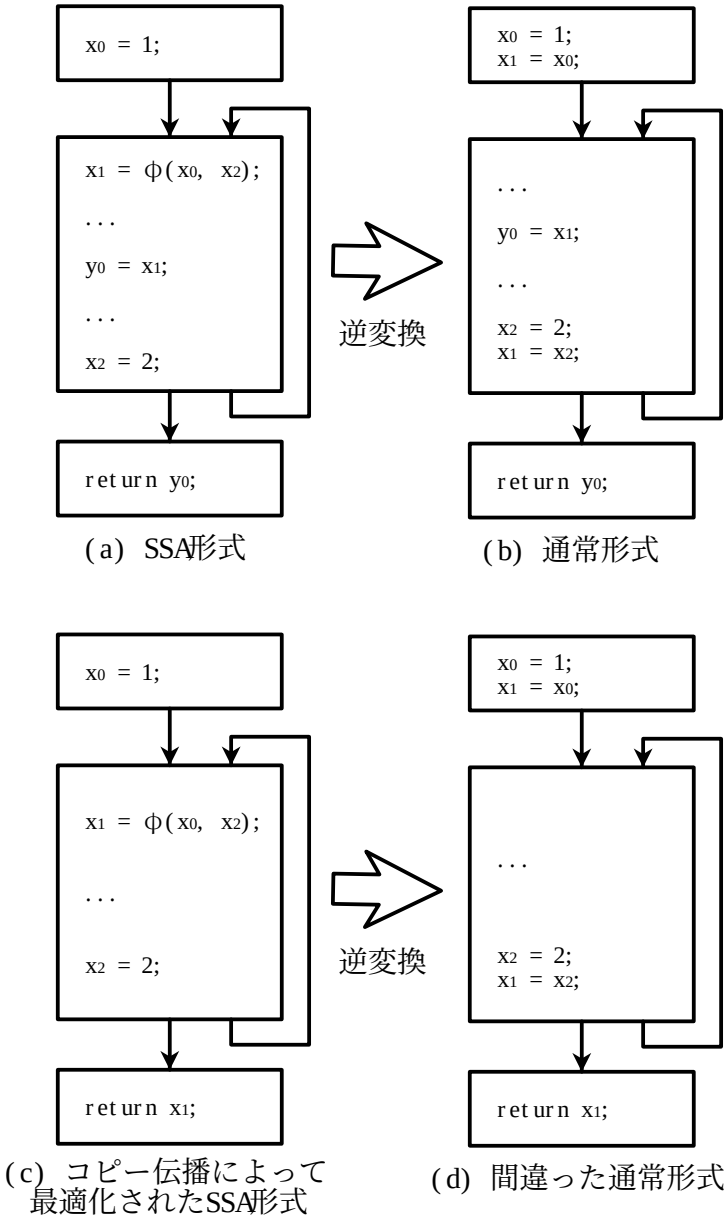


図 7.2: 単純な SSA 逆変換が上手くいかない例

第8章　おわりに

8.1　まとめ

本研究ではSSA変換のアルゴリズム, 特にその比較がまだあまり行われていないポインタ解析に基づくアルゴリズムを同一コンパイラ上に実装し, それらの実際の性能を評価したいという動機から, アルゴリズム評価のためのコンパイラ・フレームワークの実装を行った. 評価環境という特性上, 機能的に実用には足りなくても平等な比較を容易に行えることが重要であると考え, プログラム言語に依存せず, 各プログラムの依存関係を最小にするための工夫として, XMLを基にした中間表現を採用した. また, ポインタ解析に基づくアルゴリズムのうち, 対照的な性質を持つCytronらの方法とHastiらの方法を評価環境上に実装し, 正しく動くことを確認した.

コンパイラ・フレームワークがC言語のサブセットを対象としていることから実際のプログラムなどをアルゴリズムのテストに用いることができず, アルゴリズムの比較は十分に行う事ができなかったが, そのための予備的なものとして十分な結果を出すことができたと考える.

8.2　結論

本研究ではポインタ解析に基づくSSA変換のアルゴリズムの比較を目的として, XMLによる中間表現を用いたコンパイラ・フレームワークを実装した. 実装はC言語にて行い, XML環境としてlibxml2を用いた. 実験はSun Blade 1500(SPARC), Solaris8にて行った. 予備評価では次の利点を持つことを確認した.

- ポインタ解析に基づくSSA変換のアルゴリズムを扱うのに十分な機能を持っている.
- XMLによる中間表現は実装コストを下げる.
- 実際にコンパイル, 実行を行って性能を測定できる.

ポインタ解析に基づくSSA変換のアルゴリズムとしてはフロー依存の解析を用いるCytronらの方法とフロー非依存の解析を用いるHastiらの方法の実装を行い, これらが正しく動作することを確認した.

しかし, これらのアルゴリズムの比較はまだ十分ではなく, 本研究で行った実験はアルゴリズムの比較を行うための予備的な位置付けであると考えられる. 今後, 更に改良を重ねて詳細な実験を行えるようにする必要がある.

8.3 今後の課題

結論で述べたように, 現段階ではアルゴリズムの比較を正確に行うことはできない. 大きな理由はコンパイラ・フレームワークの入力となるプログラムが ANSI C 言語のサブセットであるということだ. 既存のプログラムを使うことができないため, 自分で作成したプログラムでしか試すことができない. 従って大きなプログラムのテストが行えない. この問題を解決することが当面の課題である.

問題を解決するためには以下のことを行う必要がある.

- XCC を拡張して, 入力言語を ANSI C 言語に近付ける. 特に制御文と関係演算子が必要最小限しか存在しないことがプログラムの負担を大きくしている.
- アルゴリズムが最悪の実行時間になるようなプログラムを見つける. これは必ずしも実際に使われているプログラムでテストを行う必要はないため, XCC を拡張しなくても行える. このようなプログラムがどの程度現実に表れるかを考える必要がある.

これらの問題が解決し, アルゴリズムの比較が行えるようになった後の今後の研究課題を次に示す.

- アルゴリズムの比較の結果を基にしてポインタ解析のアルゴリズムを改良する. 例えば, Cytron らの方法ではアドレス演算の問題が解決されないまま残っていたが, Hasti らの方法を参考にしてこれを解決できるかもしれない.
- ポインタを扱うその他のアルゴリズムの実装と比較を行う. 今回採り上げた, 2 つアルゴリズムはどちらも単純な変数のアドレスを持つポインタしか扱えない. しかし, それらの他にも配列やヒープを扱う方法なども提案されている.

謝辞

本論文を執筆するに当たり試行錯誤を繰り返してなかなか進まなかった私の研究を辛抱強くご指導して頂きました権藤克彦助教授, また研究の進め方から論文執筆に至るまで多岐にわたるアドバイスを頂きました片山卓也教授をはじめとするソフトウェア基礎講座の皆様に感謝を申し上げます.

付録1 XC-XMLのDTD

```
<!ENTITY % b_op  "eq|and|or|lt|add|sub|mul|div">
<!ENTITY % b_op2 "array|dot|arrow">
<!ENTITY % u_op  "addr|ptr|plus|minus|not">
<!ENTITY % op    "assign|binary_op|unary_op|%b_op2;">
<!ENTITY % flow  "branch_z|branch|label">
<!ENTITY % instr "%op;|%flow;|call|ret|paren|var|const">
<!ENTITY % l_val "unary_op|%b_op2;|paren|var">
<!ENTITY % r_val "%op;|call|paren|var|const">

<!ELEMENT xc-xml (func|sym_tab)*>

<!ELEMENT func (scope)>
<!ATTLIST func name       IDREF #REQUIRED
               frame_size CDATA #REQUIRED
               sym_tab    IDREF #REQUIRED>

<!ELEMENT scope (%instr;|scope)*>
<!ATTLIST scope sym_tab IDREF #REQUIRED>

<!ELEMENT assign ((%l_val;), (%r_val;))>

<!ELEMENT binary_op ((%r_val;), (%r_val;))>
<!ATTLIST binary_op kind (eq|and|or|lt|add|sub|mul|div) #REQUIRED>

<!ELEMENT array ((%r_val;), (%r_val;))>

<!ELEMENT dot ((%r_val;), (%r_val;))>

<!ELEMENT arrow ((%r_val;), (%r_val;))>
```

```
<!ELEMENT unary_op (%r_val;)>
<!ATTLIST unary_op kind (addr|ptr|plus|minus|not) #REQUIRED>

<!ELEMENT branch_z EMPTY>
<!ATTLIST branch_z label IDREF #REQUIRED>

<!ELEMENT branch EMPTY>
<!ATTLIST branch label IDREF #REQUIRED>

<!ELEMENT label EMPTY>
<!ATTLIST label id ID #REQUIRED>

<!ELEMENT call (%instr;)+>

<!ELEMENT ret (%instr;)?>

<!ELEMENT paren EMPTY>

<!ELEMENT var EMPTY>
<!ATTLIST var mem  (true|false) #REQUIRED
              sym  IDREF        #IMPLIED
              name NMTOKEN      #IMPLIED>

<!ELEMENT const EMPTY>
<!ATTLIST const kind  (int|char|str) #REQUIRED
                value CDATA          #IMPLIED
                str   IDREF          #IMPLIED>

<!ELEMENT sym_tab (sym|str)*>
<!ATTLIST sym_tab id ID #REQUIRED
                  super IDREF #IMPLIED>

<!ELEMENT sym (type)>
<!ATTLIST sym id     ID    #REQUIRED
              offset CDATA #IMPLIED>

<!ELEMENT str EMPTY>
<!ATTLIST str id    ID      #REQUIRED
```

```
              name  NMTOKEN #REQUIRED
              value CDATA   #REQUIRED>

<!ELEMENT type (type*)>
<!ATTLIST type kind       (prim|ptr|func|array|struct|union) #REQUIRED
               size       CDATA                              #IMPLIED
               array_size CDATA                              #IMPLIED
               name       NMTOKEN                            #IMPLIED
               sym_name   NMTOKEN                            #IMPLIED>
```

参考文献

[1] National Compiler Infrastructure Project Homepage. http://nci.pgroup.com/

[2] SUIF Homepage. http://www-suif.stanford.edu/

[3] OSUIF Homepage. http://www.cs.ucsb.edu/osuif/

[4] MachSUIF Homepage. http://www.eecs.harvard.edu/machsuif/

[5] GNU Project Homepage. http://www.gnu.org/

[6] GNU Compiler Collection Homepage. http://gcc.gnu.org/

[7] GNU Project. SSA for Trees. http://gcc.gnu.org/projects/tree-ssa/

[8] COINS Project Homepage. http://www.coins-project.org/

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 25-25, January 1989.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, Volume 13, No. 4, pages461-186, October 1991.

[11] Vugranam C. Sreedhar and Guang R. Gao. A Linear Time Algorithm for Placing ϕ-Nodes. Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 62-73, January 1995.

[12] Vugranam C. Sreedhar and Guang R. Gao. Computing ϕ-nodes in linear time using DJ graphs. Journal of Programming Languages, Volume 3, pages 191-213, 1995.

[13] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 55-66, January 1991.

[14] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the Efficient Engineering of Ambitious Program Analysis. IEEE Transactions on Software Engineering, 20(2), 1994.

[15] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Software - Practice and Experience, Volume 28, No. 8, pages 859-881, July 1998.

[16] G. J. Chaitin. Register allocation & spilling via graph coloring. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pages 98-101, 1982.

[17] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. Proceedings of the 6th International Symposium on Static Analysis, Volume 1694 of Lecture Notes in Comuter Science, pages 194-210. Springer-Verlag, 1999.

[18] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. Proceedings of the Conference on Programming Language Design and Implementation, pages 36-45, June 1993.

[19] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 97-105, 1998.

[20] Christopher Lapkowski and Laurie J. Hendren. Extended SSA Numbering: Introducing SSA properties to languages with multi-level pointers. CC '98, pages 128-143, 1998.

[21] Jong-Deok Choi, Michael Burke, Paul Carini Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects 20th ACM Symp. Principles of Programming Languages, 232-245, 1993.

[22] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 1, Issue 1, pages 121-141, July 1979.

[23] Dov Harel. A linear algorithm for finding dominators in flow graphs and related problems. Proceedings of the seventeenth annual ACM Symposium on Theory of Computing, pages 185-194, May 1985.

[24] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems, 13(2):181-210, April 1991.

[25] 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. http://www.is.titech.ac.jp/ nakaya8/papers/

[26] 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.

[27] 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, September 2001.

[28] 福岡岳穂, 高橋正人, 中谷俊晴, 佐々政孝. 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大会論文集, 第 19 回, September 2002.

[29] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.