修 士 論 文

# 静的単一代入形式を用いたポインタ解析アルゴリズム


平成 22 年度 修了
三重大学大学院　工学研究科
博士前期課程　情報工学専攻
計算機ソフトウェア研究室

**田中　雄一**

## 概 要

ポインタ解析は多くのプログラム解析にとって必須であり，その解析情報はプログラムの最適化や信頼性の向上に役立つ．しかし，プログラムの実行前に行うポインタ解析では，完全な解析情報を求めることは一般に不可能である．そのため近似的な解析情報を求めるアルゴリズムの研究がさかんに行われ，これまでに計算量と正確さのトレードオフを考慮した様々な近似アルゴリズムが報告されている．このトレードオフに影響を与える1つの指針としてフロー依存性がある．フロー依存解析は正確だが実行効率が悪いのに対して，フロー非依存解析は実行効率は良いが正確さで劣る．

このフロー非依存ポインタ解析の不正確さを改善するために，Hasti ら (1998) は静的単一代入形式（Static Single Assignment Form, 以下 SSA 形式）を用いたフロー非依存ポインタ解析アルゴリズムを提案し，その有用性を示した．また,「そのアルゴリズムがフロー依存ポインタ解析アルゴリズムと同等の解析能力を有するか否か」を未解決問題として提起した．Hardekopf ら (2009) は両者の解析能力が同等ではないと予想し，SSA 形式と通常のフロー依存解析を組み合わせたアルゴリズムを提案した．

本稿ではこの未解決問題について考察し，Hardekopf らの予想に反して，2つのアルゴリズムが同等の解析能力を有することを示すとともに，Hasti らのアルゴリズムを改善した新しいアルゴリズムを提案する．


三重大学大学院　工学研究科

# 目次

# 第1章　はじめに

現在ソフトウェアはあらゆる分野で使用され，またその複雑さと規模はますます増大し続けている．そのためソフトウェアの実行効率や信頼性の向上が強く求められ，それらを実現するためのプログラム解析に関する研究がさかんに行われている．ポインタ解析は多くのプログラム解析にとって必須であり，その解析情報はプログラムの最適化や信頼性の向上に役立つ．しかし，プログラムの実行前に行うポインタ解析では，完全な解析情報を求めることは一般に不可能である [Ram94]．そのため近似的な解析情報を求めるアルゴリズムの研究がさかんに行われ，これまでに計算量と正確さのトレードオフを考慮した様々な近似アルゴリズムが報告されている [Hin01]．このトレードオフに影響を与える1つの指針としてフロー依存性がある．一般に，文の実行順序を考慮するフロー依存解析は文の実行順序を考慮しないフロー非依存解析より正確である．しかしフロー依存解析はフロー非依存解析と比べて実行効率が悪い．

フロー非依存ポインタ解析の不正確さを改善するために，Hasti ら [HH98] は静的単一代入形式 (Static Single Assignment Form，以下 SSA 形式)[CFR+91, AH00] を用いたフロー非依存ポインタ解析アルゴリズムを提案し，その有用性を示した．また，「そのアルゴリズムがフロー依存ポインタ解析アルゴリズムと同等の解析能力を有するか否か」を未解決問題として提起した．Hardekopf ら [HL09] は両者の解析能力が同等ではないと予想し，SSA 形式と通常のフロー依存解析を組み合わせたアルゴリズムを提案した．

本稿ではこの未解決問題を考察し，Hardekopf らの予想に反して，2つのアルゴリズムが同等の解析能力を有することを示すとともに，Hasti らのアルゴリズムを改善した新しいアルゴリズムを提案する．

## 1.1　ポインタ解析

ポインタの存在するプログラムを解析するためには，まず最初にプログラムの各点におけるポインタ変数の指しうる対象の集合を求めるポインタ解析が行われなければならない．そして，ポインタ解析の結果の正確さは，到達定義解析のようなポインタ解析の結果を必要とする解析の正確さと効率に影響を与える [SH97b, HP00]．

ポインタ解析には動的な解析と静的な解析があり，前者はプログラムの実行中に行うものであり，後者はプログラムの実行に先立って行うものである．特に，後者はプログラムの実行中のオーバーヘッドが生じないという利点があり，これまでさかんに研究が行われてきた．ポインタ解析には，その解析の正確さと計算量のトレードオフに影響を与えるいくつかの指針があり，フロー依存性と文脈依存性によって大きく分類される．

### 1.1.1　フロー依存性 (Flow-sensitivity)

フロー依存性とはプログラムの文の実行順序を考慮するか否かである．文の実行順序を考慮するフロー依存 (Flow-sensitive) 解析は得られる解析情報が正確であるが，実行効率が悪い．一方，

```
(1) a = &w;
(2) p = &a;
(3) b = &x;
    if(···)
(4)     b = &y;

(5) p = &b;
(6)
```

図 1.1: 通常のプログラム

```
a1 = &w0;
p1 = &a1;
b1 = &x0;
if(···)
   b2 = &y0;
b3 = φ(b2,b1);
p2 = &b3;
```

図 1.2: **図 1.1** を SSA 形式に変換した例

文の実行順序を考慮しないフロー非依存 (Flow-insensitive) 解析は実行効率は良いが，正確さで劣る．従来は計算量やメモリ使用量の観点からフロー非依存の手法 [And94, HH98, HL07a, HL07b, HBCC99, SH97a, Ste96, 田中 10] について多く研究されてきたが，近年ではメモリ使用量や計算量を抑えたフロー依存の手法も提案されている [EGH94, HL09, HBCC99, NL09].

フロー依存解析は実行される文の順序を考慮するため，文単位で解析結果を求める．従来のフロー依存ポインタ解析はデータフロー解析の枠組みを使用しているが，近年では最適化の手法を導入することでスケーラビリティを改良した新しいフロー依存ポインタ解析も提案されている [HL09].

フロー非依存解析では，実行される文の順序を考慮しないため，プログラム全体で 1 つの解析結果を求める．フロー非依存ポインタ解析には，*inclusion-based*[And94] と *equality-based*[Ste96] の 2 つの考え方[1]がある．inclusion-based ポインタ解析はプログラムから制約を作成し，制約グラフを構築する．ポインタ情報はこのグラフの推移閉包を求めることで得られるため，時間計算量は $O(n^3)$ である．またこの inclusion-based ポインタ解析に対して，解析結果に影響を与えることなしに入力サイズ n を減らすことで，スケーラビリティを向上させる研究もなされている [HL07b, HL07a]. equality-based ポインタ解析は型推論を用い，推論規則から変数に割り当てられた型を union-find データ構造を使って統合していく．そのため Steensgaard[Ste96] のアルゴリズムは $O(n\alpha(n,n))$[2] である．

**図 1.1** に対するフロー依存ポインタ解析とフロー非依存ポインタ解析の結果を**表 1.1** に示す．ここで，p → a は p が a を指す，p → a,b は p が a か b を指すことを表す．フロー依存ポインタ解析で得られるポインタ情報はその行番号の直前までに得られているものであり，フロー非依存ポインタ解析ではプログラム全体の情報となる．このようにフロー非依存ポインタ解析で得られるポインタ情報は，フロー依存ポインタ解析の各行における情報と比べて，不正確であることが分かる．

### 1.1.2 文脈依存性 (Context-sensitivity)

文脈依存性とは関数呼び出しを個別に扱うか否かである．関数呼び出しを区別して扱う文脈依存解析は，呼び出す際の引数と返却値の関係を考慮した解析である [EGH94]．一方，文脈非依存

[1] inclusion-based ポインタ解析は *Andersen-style* ポインタ解析，equality-based ポインタ解析は *Steensgaard-style* ポインタ解析とも呼ばれる．

[2] $\alpha(n,n)$ はアッカーマン関数の逆関数であり，非常にゆっくり増加する関数である (現実的なサイズ n では定数 4 以下と見なせる)．

表 1.1: **図 1.1** のプログラムに対するポインタ解析の結果

| | フロー依存ポインタ解析 | フロー非依存ポインタ解析 |
|---|---|---|
| ポインタ情報 | (2) $a \rightarrow w$<br>(3) $a \rightarrow w$ , $p \rightarrow a$<br>(4) $a \rightarrow w$ , $b \rightarrow x$ , $p \rightarrow a$<br>(5) $a \rightarrow w$ , $b \rightarrow \{x, y\}$ , $p \rightarrow a$<br>(6) $a \rightarrow w$ , $b \rightarrow \{x, y\}$ , $p \rightarrow b$ | $a \rightarrow w$<br>$b \rightarrow \{x, y\}$<br>$p \rightarrow \{a, b\}$ |

解析は同じ関数の呼び出し式をまとめて扱う，すなわち関連する関数呼び出しのすべての引数の値はまとめられ，その結果として得られる返却値がすべての呼び出し元に返される [SH97a, HL09].

### 1.1.3 may ポインタ解析，must ポインタ解析

ポインタ解析には *may* と *must* の 2 種類がある．may ポインタ解析はプログラムのある実行中に生じる結果を求める．一方，must ポインタ解析はプログラムのすべての実行の中に生じる結果を求める．本稿において，ポインタ解析といえば may ポインタ解析を表す．

## 1.2 静的単一代入形式 (SSA 形式)

SSA 形式 [CFR+91, AH00] とは，コンパイラ最適化などに使われる中間表現の 1 つで以下のような特徴がある．

i. 各変数は 1 度だけ定義されるように添字が付けられる

ii. 1 つの変数に対する定義が複数到達する箇所に $\phi$ 関数[3]を挿入する

SSA 形式の例として，**図 1.1** を SSA 形式に変換したものを**図 1.2** に示す．同じ変数に対して，代入文で定義される度に新しい添字が付けられているのが分かる．また if 文の後には変数 b に対する定義が複数 ($b_1$ と $b_2$) 到達するので，$\phi$ 関数が挿入されている．

SSA 形式はその特徴からプログラムのフローを考慮した形となっている．そのためデータフロー解析で得ていた情報は添字付き変数によって容易に得ることができる．この点から近年，定数伝播や共通部分式の除去といったコンパイラの最適化フェーズや，大規模なソフトウェアの解析に広く用いられるようになっている．

## 1.3 ポインタ解析と SSA 形式の組合せ

SSA 形式はその特徴から，フロー非依存ポインタ解析の精度を改善させることができる．たとえば，**図 1.2** に対してフロー非依存ポインタ解析を行った結果は**表 1.2** のようになる．この結果は**図 1.1** に対するフロー依存ポインタ解析の結果と同等であり，フロー非依存ポインタ解析の結果を改善したものとなっている．

[3]制御の流れによって，右辺のどれかの変数を返す仮想的な関数

表 1.2: **図 1.2** に対するフロー非依存ポインタ解析の結果

| | |
|---|---|
| ポインタ情報 | $a_1 \rightarrow w_0$<br>$b_1 \rightarrow x_0$ , $b_2 \rightarrow y_0$ , $b_3 \rightarrow \{x_0, y_0\}$<br>$p_1 \rightarrow a_1$ , $p_2 \rightarrow b_3$ |

| | | |
|---|---|---|
| (1) | a = &w; | $a_1$ = &$w_0$; |
| (2) | p = &a; | $p_1$ = &$a_1$; |
| (3) | a = &x; | $a_2$ = &$x_0$; |
| (4) | c = *p; | $c_1$ = *$p_1$; |
| (5) | *p = &y; | *$p_1$ = &$y_0$; |
| (6) | d = a; | $d_1$ = $a_2$; |

(a) 通常のプログラム　　(b) SSA 形式プログラム

図 1.3: デリファレンスを含むプログラムの SSA 形式への変換例

しかし，ポインタ解析と SSA 形式は単純には組み合わせられず，いくつかの問題点がある．その問題とは，プログラム中にアドレス演算子 (&x) やデリファレンス (*p) が現れる場合に，うまく SSA 形式に変換できないということである．その例を**図 1.3** に示す．

この**図 1.3** の例では問題点が 2 つある．まず**図 1.3(b)** の (4) に問題がある．$*p_1$ の値は (2) より $a_1$ である．しかし実際には，(3) で定義された $a_2$ でなければならない．これは a の添字付きの変数のアドレスを値として持ったため，その後 (3) で a が再定義されたことを反映されていないために起こっている．2 つ目は (6) の $a_2$ についてである．(5) は実際には a の定義文であるので，$a_3$ を=の左辺に持つ定義文にする必要がある．したがって (6) の a の添字を 3 にする必要がある．これはデリファレンス ($*p_1$) により，プログラム中に a が現れていないため，添字の更新が行われないことが原因である．

### 1.3.1 Hasti らのアルゴリズム

Hasti ら [HH98] は，このデリファレンスの問題を克服し，フロー非依存ポインタ解析と SSA 形式を組み合わせたアルゴリズムを提案した (**Algorithm 1**)．**Algorithm 1** において，CFG は制御フローグラフ，注釈はデリファレンス (*p) の変数 p がとりうる値の集合，中間形式はデリファレンスを注釈を用いて置き換えて得られるプログラムである．デリファレンスの置き換えは，デリファレンスの変数 p のとりうる値の集合の要素数によって以下の 2 通りがある．

i. 変数 p が値&a のみをとるときは`*p = b;` を `a = b;` とする．

ii. 変数 p が値を複数とるときは元の代入文を分岐に置き換え，各分岐先で各要素に対して (i) を行う．たとえば，$p \rightarrow \{a, b\}$ のとき，代入文`*p = x;` は 2 つの分岐に置き換えられ，それぞれの分岐に代入文 `a = x;` と `b = x;` をおく．

このアルゴリズムでは，まず 1-2 でフロー非依存ポインタ解析を行い，それによって得られるポインタ情報を各デリファレンスに与える．そして 3-8 の繰返しでは，4 でプログラム中のデリ

**Algorithm 1** Hastiら [HH98] のアルゴリズム

**Require:** 制御フローグラフ CFG

1: CFG に対してフロー非依存ポインタ解析を行う
2: CFG の中のデリファレンスに注釈を付ける
3: **repeat**
4: CFG から中間形式 (IM) を作る
5: IM を SSA 形式に変換する ($IM_{ssa}$)
6: $IM_{ssa}$ に対してフロー非依存ポインタ解析を行う
7: $IM_{ssa}$ と解析結果を用いて CFG の注釈を更新する
8: **until** 注釈に変化がない

ファレンスを注釈を用いて置き換える．これにより，プログラムにデリファレンスがなくなり**図 1.3(b)** のような問題点を生ずることなく SSA 形式に変換できる (5)．6-7 では，SSA 形式に変換したプログラムに対してフロー非依存ポインタ解析を行い，その解析結果を注釈に反映させる．このとき注釈に変化があれば，この一連の処理を繰り返す．このアルゴリズムが不動点に達したとき，つまり注釈に変化が生じなくなったとき，その時点で得られているポインタ情報を最終的な結果とする．またこのアルゴリズムは，不動点に達する前に繰返しを止めることで，解析精度が低くなるが解析コストを抑えることが可能である．**Algorithm 1** のアルゴリズムの実行例を付録 A に示す．

Hasti らは，この最終的な結果が元のプログラムに対するフロー依存ポインタ解析によって得られる結果と等しいかどうか，つまりこの Hasti らのアルゴリズム (または Hasti らが提案したポインタ解析手法を用いたあるアルゴリズム) がフロー依存ポインタ解析のアルゴリズムと同等の能力を有するか否かを未解決問題として提起した．

本稿の構成は次のようになっている．**第 2 章**では本稿で使用する語句の定義を示す．**第 3 章**では，未解決問題について 2 つのアルゴリズムが同等の解析能力を有することを証明するとともに，その結果として得られる新しい SSA 形式を用いたフロー非依存ポインタ解析のアルゴリズムを提案する．**第 4 章**ではその提案手法について考察を行う．**第 5 章**では，2 つのアルゴリズムが関数呼び出しを考慮しても同等の解析結果をもたらすことを示し，提案手法を手続き間解析に拡張する．**第 6 章**では，本研究の提案手法と Hasti らの手法の評価を行う．**第 7 章**では，関連研究を紹介し，最後に**第 8 章**で本研究の結論と今後の課題を述べる．なお，付録 A として，Hasti らの手法の実行例，付録 B として，Hasti らのインクリメンタルな手法に対応するように変更した提案手法，付録 C に，再帰関数に対する supergraph の例，そして付録 D として supergraph を用いた解析の実行例をそれぞれ示した．

# 第2章　準備

この章では，本研究で使用する構文規則と定義を示す．

## 2.1　対象のプログラミング言語

対象とするプログラミング言語は if 文，while 文，choice 文 (2.2 節)，そして以下の形を持つ代入文から構成されるものとする[1]．

$$n\text{重ポインタ変数} = n\text{重ポインタ変数}; \tag{2.1}$$
$$n\text{重ポインタ変数} = \&(n-1\text{重ポインタ変数}); \tag{2.2}$$
$$n\text{重ポインタ変数} = *(n+1\text{重ポインタ変数}); \tag{2.3}$$
$$*(n+1\text{重ポインタ変数}) = n\text{重ポインタ変数}; \tag{2.4}$$

$n$ 重ポインタ変数とは，型の*の数が $n$ であるポインタ変数のことである．たとえば `int **p` ならば p は 2 重ポインタ変数である．またこの*の数を多重度と呼び，p の多重度は 2 である．プログラム内の変数の多重度の最大値が $m$ であるとき，そのプログラムを最大多重度 $m$ のプログラムという．

## 2.2　デリファレンスの置き換え

文献 [HH98] で述べられている，デリファレンス (*x) を x の値で置き換える処理は，x のとりうる値の個数によって次のようになされる．

- x のとりうる値が 1 つに定まる場合

  置き換えたものは通常の代入文として処理する．

- x のとりうる値が複数個である場合

  それらのうちのどれかをとることを表す choice 文を挿入する．挿入した choice 文は以下のように取り扱う．

  - 元の代入文を d とすると，それを置き換えてできた choice 文も d として扱う．
  - SSA 形式に変換されるとき，choice 文の後ろには $\phi$ 関数が挿入される．

  たとえば，`y = *x;` で x のとりうる値の集合が `{&a,&b}` であるならば，`y = {a,b};` である．この例のように $\{\cdots\}$ を含む代入文を本稿での choice 文の記法とする．

[1] 代入文には `x = **p;` のような形も存在するが，これは一時的な変数 (`tmp`) を導入することにより，`tmp = *p; x = *tmp;` のように変形することができる．したがって，このような代入文の形も (2.3) の代入文の形に変形することが可能である．代入文`**x = p;` のような形も同様に (2.4) の代入文の形に変形可能である．一般的な C プログラムへの適用については 4.2 節で述べる．

## 2.3 諸定義

変数 x の左辺値を&x と書く．これ以降，代入文と等式を混同しやすい箇所では，それらを明確に区別するために，代入には"="，等式には"≡"を用いる．

**定義 1 (変数の定義文)**. ある代入文 $d$ が変数 $x$ の定義文であるとは，その代入文 $d$ が $x = e'$; の形を持つときであるか，または $*p = e'$; であり，$p$ が $d$ の直前で $\&x$ を値として持つときである．

代入文 d が choice 文であるとき，たとえば {x,y}= e'; のとき，d は変数 x および y の定義文と考える．

**定義 2 (代入文の到達可能)**. 代入文 $d'$ が代入文 $d$ に到達可能とは，$d'$ から $d$ に至るある (長さが正の) 路が存在して，その途中に $d'$ と同じ変数の定義文が存在しないときである．

代入文の左辺値がその代入文の左辺式の左辺値を表すとする．また同様に，代入文の右辺値がその代入文の右辺式の右辺値を表すとする．

**定義 3 (代入文の右辺値)**. 代入文 $d$ : $e = e'$; の右辺値 $r(d)$ が $\&a$ をとりうるとは，次の (1)，(2)，(3) のうちのどれかを満たすときである．

(1) $e' \equiv \&a$

(2) $e' \equiv y$ かつ ある代入文 $d'$ が存在して，$\&y$ を $d'$ の左辺値，$\&a$ を $d'$ の右辺値として持ち，$d'$ が $d$ に到達可能

(3) $e' \equiv *z$ かつ ある代入文 $d'$ が存在して，$\&z$ を $d'$ の左辺値，$\&y$ を $d'$ の右辺値として持ち，$d'$ が $d$ に到達可能 かつ ある代入文 $d$" が存在して，$\&y$ を $d$" の左辺値，$\&a$ を $d$" の右辺値として持ち，$d$" が $d$ に到達可能 ．すなわち，次のような代入文の列が存在する．

$$
\begin{aligned}
d\text{"} &: g = g'; (l(d\text{"}) \equiv \&y, r(d\text{"}) \equiv \&a) \\
&\vdots \\
d' &: f = f'; (l(d') \equiv \&z, r(d') \equiv \&y) \\
&\vdots \\
d &: e = *z;
\end{aligned}
$$

**定義 4 (代入文の左辺値)**. 代入文 $d$ : $e = e'$; の左辺値 $l(d)$ が $\&y$ をとりうるとは，次の (1)，(2) のうちのどれかを満たすときである．

(1) $e \equiv y$

(2) $e \equiv *x$ かつ ある代入文 $d'$ が存在して，$\&x$ を $d'$ の左辺値，$\&y$ を $d'$ の右辺値として持ち，$d$ に到達可能

定義 3，4 は choice 文を含むプログラムに容易に拡張できる．すなわち，choice 文 d : {x,y} = {a,b}; のときは，d : x = a;，d : x = b;，d : y = a;，d : y = b; のいずれかが実行されると考えることにより，同様に定義可能である．

代入文 d における変数 x のとりうる値の集合を次のように定義する．ここで代入文の集合を D，変数の集合を Var とする．

**定義 5 (変数のとりうる値の集合)**. $d$, $d'$ $\in D$, $x, y \in Var$ が与えられたとき，代入文 $d$ における変数 $x$ のとりうる値の集合を次のように定義する.

$$V(d,x) = \{\&y \mid \text{ある代入文 } d' \text{が存在して, } d' \text{が } d \text{ に到達可能かつ } l(d') \equiv \&x,\ r(d') \equiv \&y\}$$

なお，V(d,x) は d の実行直前にとりうる x の値の集合である．また変数 x が最大多重度のポインタ変数であるとき，V(d,x) の定義は以下のようになる.

$$V(d,x) = \{\&y \mid \text{ある代入文 } d' \colon x = e'; \text{が存在して, } d' \text{が } d \text{ に到達可能かつ } r(d') \equiv \&y\}$$

ポインタ解析は各代入文 d∈D に対し，d の実行直前における各変数 x の指し先の集合を求めるものである．したがって，フロー依存ポインタ解析アルゴリズムは V(d,x) を正しく求めるアルゴリズムであるということができる.

SSA 形式上での変数のとりうる値の集合を定義するために，まず通常のプログラムと SSA 形式のプログラムの対応関係，$\phi$ 関数を含む代入文，通常のプログラムにおける到達可能の定義を SSA 形式に適用することで得られる到達可能を定義する.

**定義 6 (SSA 形式の代入文の集合)**. 通常のプログラムの代入文 $d \in D$ を *SSA* 形式に変換したものを $d_{ssa}$ とする．そして，その *SSA* 形式の代入文の集合を $D_{ssa}$ とする.

**定義 7 ($\phi$ 関数の扱い)**. $\phi$ 関数を右辺に含む代入文 $x_i = \phi(...);$ の左辺値は$\&x_i$，右辺値は $\phi$ 関数の引数の右辺値の集合である.

**定義 8 (SSA 形式における代入文の到達可能)**. *SSA* 形式において代入文 $d'_{ssa} : x_i = e;$ が代入文 $d_{ssa}$ に到達可能であるとは，$d'_{ssa}$ から $d_{ssa}$ に至るある *(*長さが正の*)* 路が存在して，その途中に $\&x_j\,(j \neq i)$ を左辺値として持つ代入文が存在しないときである.

定義 6，7，8 を用いて，$d_{ssa}$ における変数 $x_i$ のとりうる値の集合を以下のように定義する.

**定義 9 (SSA 形式における変数のとりうる値の集合)**. $d_{ssa}$, $d'_{ssa} \in D_{ssa}$ が与えられたとき，代入文 $d_{ssa}$ における変数 $x_i$ のとりうる値の集合を次のように定義する.

$$V_{ssa}(d_{ssa}, x_i) = \{\&y \mid \text{ある代入文 } d'_{ssa} \text{が存在して, } d'_{ssa} \text{が } d_{ssa} \text{に到達可能かつ } l(d'_{ssa}) \equiv \&x_i, r(d'_{ssa}) \equiv \&y\}$$

なお，SSA 形式のプログラムにおいては，$V_{ssa}(d_{ssa}, x_i) \neq \emptyset$ となる変数 x の添字 i はたかだか 1 つのみ存在する．以降，$\phi$ 関数を右辺に含む代入文を $\phi$ 関数代入文，それ以外を通常の代入文と呼ぶ.

以上より，未解決問題「フロー依存ポインタ解析と SSA 形式上のフロー非依存ポインタ解析が同等の解析能力を有する」を示すことは，すべての代入文 $d \in D$ とすべての変数 $x \in Var$ に対して，ある添字 i が存在して，$V(d,x) = V_{ssa}(d_{ssa}, x_i)$ を示すことと等しいといえる.

# 第3章　未解決問題の証明と提案手法

この章では2.1節で定義した構文規則を満たすプログラムについて，フロー依存ポインタ解析とSSA形式上のフロー非依存ポインタ解析が同等の解析能力を有することを示す．

**補題 1.** 最大多重度のポインタ変数への代入文は次の1，2の形式を持つ．

**1.** $n$ 重ポインタ変数 $=$ $n$ 重ポインタ変数；

**2.** $n$ 重ポインタ変数 $=$ &($n-1$ 重ポインタ変数)；

**証明 .** 最大多重度を $n$ とすると，最大多重度が $n+1$ のポインタ変数は存在しないので明らか．□

補題1から最大多重度の変数はそれより下の多重度の変数の影響を受けないことが分かる．この性質を用いて，最大多重度の変数に対して，両者の解析結果が等しいことを示す．以下では，最大多重度の変数のみをSSA形式に変換したと考え，その代入文の集合を $\overline{\mathrm{D}}_{\mathrm{ssa}}$ とする．

**補題 2.** $d \in D$, $x \in Var$(*x*は最大多重度の変数) とし，$d_{\mathrm{ssa}} \in \overline{D}_{ssa}$，$x_{\mathrm{i}}$ は $d_{ssa}$ に到達する $x$ の添字付き変数であるとする．このとき，$V(d,x) = V_{ssa}(d_{\mathrm{ssa}}\ ,\ x_{\mathrm{i}})$ が成立する．

**証明 .** 次のi，iiを示す．

i. $\mathrm{V(d,x)} \supseteq \mathrm{V}_{ssa}(\mathrm{d}_{ssa}\ ,\ \mathrm{x}_i)$

&y $\in \mathrm{V}_{ssa}(\mathrm{d}_{ssa}\ ,\ \mathrm{x}_i)$ とする．このとき，ある $n+1$ 個の最大多重度の変数 $\mathrm{z}_{j_0}^{(0)}$，$\mathrm{z}_{j_1}^{(1)}$，$\cdots$，$\mathrm{z}_{j_n}^{(n)} \equiv \mathrm{x}_i$ と代入文 $\mathrm{d}_i{:}\mathrm{z}_{j_i}^{(i)} = \mathrm{e}_i;$ が存在して，次の条件 (1)，(2)，(3) を満たす．

(1) $\mathrm{e}_0 \equiv$ &y

(2) $\mathrm{e}_i \equiv \mathrm{z}_{j_{i-1}}^{(i-1)}$ or $\mathrm{e}_i \equiv \phi(\cdots, \mathrm{z}_{j_{i-1}}^{(i-1)}, \cdots)$ , $1 \leq i \leq n$

(3) $\mathrm{d}_i$ は $\mathrm{d}_{i+1}$ に到達可能，$1 \leq i \leq n+1$(ただし，$\mathrm{d}_{n+1} \equiv \mathrm{d}_{ssa}$)

これを図示すると次のようになる．

$$
\begin{aligned}
d_0 \quad & : z_{j_0}^{(0)} = \&y; \\
d_1 \quad & : z_{j_1}^{(1)} = z_{j_0}^{(0)};\ or\ z_{j_1}^{(1)} = \phi(\cdots, z_{j_0}^{(0)}, \cdots); \\
& \ \ \vdots \\
d_n \quad & : z_{j_n}^{(n)} = z_{j_{n-1}}^{(n-1)};\ or\ z_{j_n}^{(n)} = \phi(\cdots, z_{j_{n-1}}^{(n-1)}, \cdots); \\
d_{ssa} \quad & :
\end{aligned}
$$

この代入文の列において，各 $\mathrm{d}_i(0 \leq i \leq n)$ の右辺値は&yを含む．元のプログラムにおいて，$\phi$ 関数代入文 $\mathrm{d}_k{:}\mathrm{z}_{j_k}^{(k)} = \phi(\cdots, \mathrm{z}_{j_{k-1}}^{(k-1)}, \cdots);$ に対応するプログラムの位置に代入文 $\mathrm{d}_k{:}\mathrm{z} = \mathrm{z};$ を挿入する．代入文 $\mathrm{d}_k{:}\mathrm{z} = \mathrm{z};$ を挿入しても，元のプログラムと同じ計算が行われることから，各代入文 $\mathrm{d}_i(0 \leq i \leq n)$ の右辺値として&yを含む．すなわち&y $\in$ V(d,x) が成立する．

| | | | |
|---|---|---|---|
| x = &a; | x = &a; | x = &a; | x = &a; |
| y = &b; | y = &b; | y = &b; | y = &b; |
| p = &x; | $p_1$ = &x; | p = &x; | |
| if(···) | if(···) | if(···) | if(···) |
| *p = &c; | *$p_1$ = &c; | x = &c; | x = &c; |
| else | else | else | |
| p = &y; | $p_2$ = &y; | p = &y; | |
| | $p_3 = \phi(p_1,p_2)$; | | |
| z = *p; | z = *$p_3$; | z = {x,y}; | z = {x,y}; |
| (a) 通常のプログラム | (b) 最大多重度の変数をSSA形式に変換 | (c) デリファレンスを置き換えたプログラム | (d) (c) から最大多重度の文を除去したプログラム |

図 3.1: デリファレンスの置き換え

ii. $V(d,x) \subseteq V_{ssa}(d_{ssa}, x_i)$

&y $\in V(d,x)$ とすると，$V(d,x)$ の定義より d に到達可能なある代入文 d':e = e'; が存在して，l(d') ≡ &x，r(d') ≡ &y を満たす．x は最大多重度の変数であることから，e ≡ x が成立する．d と d' を SSA 形式に変換した $d_{ssa}$ と $d'_{ssa}$ について，$d'_{ssa}$ と $d_{ssa}$ の間には複数個の $x_j = \phi(\cdots)$; の形の $\phi$ 関数代入文が存在する．これらの $\phi$ 関数代入文はすべて定義 7 より右辺値に&y を含む．したがって，&y $\in V_{ssa}(d_{ssa}, x_i)$ である．

□

元のプログラムを P，最大多重度の変数のデリファレンスを補題 2 で得られる結果を用いて置き換えたプログラムを P' とする．プログラム P' に対する定義 5 の関数 V を V'，代入文の集合を D' と書くことにする．また代入文 d' ∈ D' は P における代入文 d と対応している．デリファレンスの置き換えによるプログラムの変化の例を**図 3.1** に示す．P を**図 3.1(a)** とすると，最大多重度の変数（ここでは変数 p）を置き換えたプログラム P' は**図 3.1(c)** から p の定義文を取り除いたプログラム**図 3.1(d)** となる．また P' は最大多重度の変数 p の定義文を消去したものであるので，最大多重度-1 を最大多重度としたプログラムと考えることができる．新しい最大多重度は補題 1 が成立する．

この P と P' における代入文の到達可能について，次の補題が成立する．

**補題 3.** 最大多重度 $-1$ の変数 $x$ の定義文 $d_0 \in D$ に対して，$d_0$ が $d \in D$ に到達可能であるならば，$P'$ の対応する代入文 $d'_0 \in D'$ は $P'$ において $d' \in D'$ に到達可能である．逆もまた成立する．

**証明．** 次の i，ii を示す．

i. $d_0$ が d に到達可能 ⇒ $d_0$' は d' に到達可能

定義より $d_0$ と d の間に変数 x の定義文が存在しない．そして定義文の定義より，その間にある代入文*p = e; の左辺のデリファレンス (*p) は&x 以外の値になっている．したがって，デリファレンスの置き換え後のプログラム P' における代入文 $d_0$' も d' に到達可能である．

ii. $d_0$' が d' に到達可能 $\Rightarrow$ $d_0$ は d に到達可能

$d_0$ が d に到達可能でないとする．そのとき $d_0$ から d へ至るすべての路に x の定義文が存在する．このことは $d_0$' が d' に到達可能であることに矛盾する．したがって，$d_0$' が d' に到達可能であれば，$d_0$ は d に到達可能である．

□

最大多重度 $-1$ の変数について，次の補題が成立する．

**補題 4.** $d \in D$，$x \in Var(x$ : 最大多重度 $-1$ の変数$)$，$d' \in D'$ であるとする．このとき，*V(d,x) = V'(d',x)* である．

**証明．** 最大多重度 $-1$ の変数について，次の i，ii を示す．

i. V(d,x) $\subseteq$ V'(d',x)
$p^{(0)}$，$p^{(1)}$ は最大多重度 $-1$ の変数，$q^{(0)}$，$q^{(1)}$ は最大多重度の変数であるとし，size(&y $\in$ V(d,x)) を代入文 d の直前で，変数 x に&y を代入するために実際に行われた代入文の回数とする．この size に関する帰納法で証明する．

&y $\in$ V(d,x) とするとき，すなわちある代入文 $d_0$: e = e'; のとき，$d_0$ が d に到達可能かつ $l(d_0) \equiv$ &x, $r(d_0) \equiv$ &y である．このとき e $\equiv$ x または e $\equiv$ $*q^{(0)}$ の場合と e' $\equiv$ &y または e' $\equiv$ $p^{(1)}$ または e' $\equiv$ $*q^{(1)}$ の場合がある．

(a) size $= 1$，すなわち e $\equiv$ x かつ e' $\equiv$ &y のとき，定義より代入文 $d_0$: x = &y; が存在し，$d_0$ が d に到達可能である．このとき補題 3 より P' においても $d_0$' が d' に到達可能である．よって&y $\in$ V'(d',x) である．

(b) size $= k$ のとき成り立つと仮定し，size $= k+1$ を証明する．次の 4 つの場合に分けて考える．

(1) e $\equiv$ $*q^{(0)}$ かつ e' $\equiv$ &y，すなわち $d_0$: $*q^{(0)}$ = &y; のとき，&x $\in$ V($d_0$,$q^{(0)}$) が成立する．帰納法の仮定より，&x $\in$ V'($d_0$',$q^{(0)}$)，そして補題 3 より $d_0$' が d' に到達可能なので&y $\in$ V'(d',x) が成り立つ．

(2) e $\equiv$ x かつ e' $\equiv$ $p^{(1)}$，すなわち $d_0$: x = $p^{(1)}$; のとき，&y $\in$ V($d_0$,$p^{(1)}$) が成立する．帰納法の仮定より，&y $\in$ V'($d_0$',$p^{(1)}$)，そして補題 3 より，$d_0$' が d' に到達可能なので&y $\in$ V'(d',x) である．

(3) e $\equiv$ x かつ e' $\equiv$ $*q^{(1)}$，すなわち $d_0$: x = $*q^{(1)}$; のとき，&z $\in$ V($d_0$,$q^{(1)}$) かつ&y $\in$ V($d_0$,z) が成立する．帰納法の仮定より，&z $\in$ V'($d_0$',$q^{(1)}$)，&y $\in$ V'($d_0$',z) がいえる．そして $d_0$ が d に到達可能であるため，補題 3 より，&y $\in$ V'(d',x) が成立する．

(4) e $\equiv$ $*q^{(0)}$ かつ e' $\equiv$ $p^{(1)}$，すなわち $d_0$: $*q^{(0)}$ = $p^{(1)}$; のとき，&x $\in$ V($d_0$,$q^{(0)}$) かつ&y $\in$ V($d_0$,$p^{(1)}$) が成立する．帰納法の仮定より，&x $\in$ V'($d_0$',$q^{(0)}$)，&y $\in$ V'($d_0$',$p^{(1)}$) がいえる．そして $d_0$ が d に到達可能であるので，&y $\in$ V'(d',x) が成立する．

よって，V(d,x) $\subseteq$ V'(d',x) が成立する．

ii. V(d,x) $\supseteq$ V'(d',x)
$p^{(0)}$，$p^{(1)}$ は最大多重度 $-1$ の変数，$q^{(0)}$，$q^{(1)}$ は最大多重度の変数であるとし，size(&y $\in$

V'(d',x)) を代入文 d' の直前で，変数 x に&y を代入するために実際に行われた代入文の回数とする．(i) と同様に，この size に関する帰納法で証明する．

&y ∈ V'(d',x) とするとき，定義よりある代入文 $d_0$': e = e'; が存在して，$d_0$' が d' に到達可能かつ l($d_0$') ≡ &x，r($d_0$') ≡ &y である．このとき e について，e ≡ x，e ≡ {···,x,···} の場合があり，また e' について，e' ≡ &y，e' ≡ $p^{(1)}$，e' ≡ {···,$p^{(1)}$,···} の場合がある．

(a) size = 1

(1) e ≡ x かつ e' ≡ &y のとき，すなわち代入文 $d_0$': x = &y; のとき，$d_0$' が d' に到達可能である．補題 3 より $d_0$' に対応する代入文 $d_0$ が d に到達可能であることがいえる．&y ∈ V(d,x) である．

(2) e ≡ {···,x,···} かつ e' ≡ &y，すなわち $d_0$': {···,x,···} = &y; のとき，代入文 $d_0$: $*q^{(0)}$ = &y; である． 補題 2 より，&x ∈ V($d_0$,$q^{(0)}$) が成立する．そして $d_0$' が d' に到達可能であるから，補題 3 より $d_0$ が d に到達可能である．よって&y ∈ V(d,x) が成り立つ．

(b) size = $k$ のとき成り立つと仮定し，size = $k+1$ を証明する．次の 3 つの場合に分けて考える．

(1) e ≡ x かつ e' ≡ $p^{(1)}$，すなわち $d_0$': x = $p^{(1)}$; のとき，&y ∈ V'($d_0$',$p^{(1)}$) が成立する．帰納法の仮定より，&y ∈ V($d_0$,$p^{(1)}$)，そして $d_0$' が d' に到達可能であるから，$d_0$ が d に到達可能なので&y ∈ V(d,x) が成り立つ．またこのとき $d_0$: x = $*q^{(1)}$; の場合があるが，補題 2 より V($d_0$,$q^{(1)}$) = {&$p^{(1)}$} がいえるため成立する．

(2) e ≡ x かつ e' ≡ {···,$p^{(1)}$,···}，すなわち $d_0$': x = {···,$p^{(1)}$,···}; のとき，&y ∈ V'($d_0$',$p^{(1)}$) が成立する．e' が choice 文なので，代入文 $d_0$ は x = $*q^{(1)}$; であり，補題 2 より&$p^{(1)}$ ∈ V($d_0$,$q^{(1)}$) である．また帰納法の仮定より，&y ∈ V($d_0$,$p^{(1)}$)，そして $d_0$' が d' に到達可能であるから，$d_0$ が d に到達可能なので，&y ∈ V(d,x) が成り立つ．

(3) e ≡ {···,x,···} かつ e' ≡ $p^{(1)}$，すなわち $d_0$': {···,x,···} = $p^{(1)}$; のとき，&y ∈ V'($d_0$',$p^{(1)}$) が成立する．また代入文 $d_0$: $*q^{(0)}$ = $p^{(1)}$; と補題 2 より，&x ∈ V($d_0$,$q^{(0)}$) が成立する．帰納法の仮定 (&y ∈ V'($d_0$',$p^{(1)}$)) より，&y ∈ V($d_0$,$p^{(1)}$)，そして $d_0$' が d' に到達可能であるから，$d_0$ が d に到達可能なので&y ∈ V(d,x) が成り立つ．

よって V(d,x) ⊇ V'(d',x).

以上より，V(d,x) = V'(d',x) である． □

**系 1.** $d \in D$，$x \in Var$($x$：最大多重度 − 1 以下の変数)，$d' \in D'$ は $P$'において $d$に対応する代入文であるとする．このとき，*V(d,x) = V'(d',x)* である．

**証明．** 補題 4 より最大多重度 −1 の変数に対して，各変数がとりうる値が等しいことが言える．また最大多重度 −1 を新しく最大多重度と考えると，補題 4 を用いることで，1 つ下の多重度についても同じ集合を得ることができる．したがって，元の最大多重度 −2 の変数に対しても V(d,x) = V'(d',x) がいえる．以上のことを繰り返し用いることで，最大多重度-1 以下のすべての変数に対して，V(d,x) = V'(d',x) がいえる． □

補題 1，2，系 1 を考慮した SSA 形式上のフロー非依存解析アルゴリズムを **Algorithm 2** に示す．このアルゴリズムの基礎となる考えは最大多重度の変数はそれより下の多重度の変数の影響を受けないということである．

**Algorithm 2** 提案手法

**Require:** 制御フローグラフ CFG
  **for** n = 最大多重度 to 1 **do**
    CFG の n 重ポインタ変数を SSA 形式に変換する ($SSA_n$)
    $SSA_n$ に対してフロー非依存ポインタ解析を行う
    $SSA_n$ の n 重ポインタ変数のデリファレンスを解析結果を用いて置き換える
  **end for**

| 多重度 | フロー依存ポインタ解析 | | SSA 形式+フロー非依存ポインタ解析 |
|---|---|---|---|
| | | 補題 2 | |
| max | V | = | $V_{ssa}$ |
| | 系 1 ‖ | 補題 2 | |
| max-1 | V' | = | $V'_{ssa}$ |
| | 系 1 ‖ | 補題 2 | |
| max-2 | V" | = | $V"_{ssa}$ |
| | 系 1 ‖ | | |
| ⋮ | ⋮ | | ⋮ |

図 3.2: 定理 1 の証明の概略

このアルゴリズムを用いて，以下の定理を示す．

**定理 1.** 2.1 節で定義した構文規則を満たすプログラムにおいて，$V$を求めるフロー依存ポインタ解析と ***Algorithm 2*** のアルゴリズムの結果は等しい．

**証明 .** 最大多重度 $n$ に関する帰納法で証明を行う．また**図 3.2** に証明の概略を示す．

i. $n = 1$ のとき 補題 2 より多重度 1 の変数 x について，$V(d,x) = V_{ssa}(d_{ssa},x_i)$ である．

ii. $n = k$ のとき成り立つとし，$n = k+1$ を考える．多重度 $k+1$ の変数 x に対して補題 2 より $V(d,x) = V_{ssa}(d_{ssa},x_i)$ である．また補題 4 より，多重度 $k$ の変数 x' について $V(d,x') = V'(d',x')$ であり，仮定から最大多重度が $k$ の場合は成り立つ．以上より最大多重度が $k+1$ のときも成り立つ．

したがって，すべての多重度の変数に対して，$V(d,x) = V_{ssa}(d_{ssa}, x_i)$ がいえる． □

提案手法と Hasti らの手法の解析能力が等しいことは**第 4 章**で述べる．したがって，この同等性より次の定理を得る．

**定理 2.** Hasti らのアルゴリズムおよび提案手法がフロー依存ポインタ解析と同等の解析能力を有する．

定理 1，2 より，本稿において Hasti らの提起した未解決問題を 2.1 節で定義した構文規則を満たすプログラムに対して解決した．

# 第4章　提案手法の考察

この章では，**第3章**で示した提案手法について述べ，次に1.3節で説明したHastiらの手法との比較を行う．そして最後に本手法を2.1節の構文規則の範囲に入らないような構文要素を含む一般的なプログラムに対して，どのように適用するかを示す．

提案手法は多重度の概念を用いて，最大多重度の変数から順にSSA形式への変換，フロー非依存ポインタ解析，デリファレンスの置き換えを行う (**Algorithm 2**)．また定理1より，その解析結果はフロー依存ポインタ解析と同等のものとなる．提案手法の実行例を**図4.1**に，得られるポインタ情報を**表4.1**に示す．フロー依存解析で得られる情報はその文の直前までに得られるものである．

| | (a) | (b) | (c) | (d) | (e) |
|---|---|---|---|---|---|
| | a = 0; | a = 0; | a = 0; | a = 0; | a = 0; |
| | b = 1; | b = 1; | b = 1; | b = 1; | b = 1; |
| | p = &a; | p = &a; | p = &a; | $p_1$ = &a; | $p_1$ = &a; |
| (1) | q = &b; | q = &b; | q = &b; | $q_1$ = &b; | $q_1$ = &b; |
| (2) | t = &p; | $t_1$ = &p; | $t_1$ = &p; | $t_1$ = &p; | $t_1$ = &p; |
| | if(···) | if(···) | if(···) | if(···) | if(···) |
| (3) | *t = &b; | *$t_1$ = &b; | p = &b; | $p_2$ = &b; | $p_2$ = &b; |
| | | | | $p_3 = \phi(p_2,p_1)$; | $p_3 = \phi(p_2,p_1)$; |
| (4) | *p = 2; | *p = 2; | *p = 2; | *$p_3$ = 2; | {a,b} = 2; |
| | t = &q; | $t_2$ = &q; | $t_2$ = &q; | $t_2$ = &q; | $t_2$ = &q; |
| (5) | | | | | |

図 4.1: 提案手法の実行例

表 4.1: **図4.1**のプログラムに対する各ポインタ解析の結果

| | フロー依存ポインタ解析 | SSA形式+フロー非依存ポインタ解析 |
|---|---|---|
| ポインタ情報 | (1) $p \rightarrow a$<br>(2) $p \rightarrow a$ , $q \rightarrow b$<br>(3) $p \rightarrow a$ , $q \rightarrow b$ , $t \rightarrow p$<br>(4) $p \rightarrow \{a,b\}$ , $q \rightarrow b$ , $t \rightarrow p$<br>(5) $p \rightarrow \{a,b\}$ , $q \rightarrow b$ , $t \rightarrow q$ | $t_1 \rightarrow p$ , $t_2 \rightarrow q$<br>$p_1 \rightarrow a$ , $p_2 \rightarrow b$ , $p_3 \rightarrow \{a,b\}$<br>$q_1 \rightarrow b$ |

**図4.1(a)**の最大多重度の変数はtであるため，まずtをSSA形式に変換する (**図4.1(b)**)．次

に**図 4.1(b)** の最大多重度の変数に対してフロー非依存ポインタ解析を行う．そして解析結果の $t_1 \rightarrow p$，$t_2 \rightarrow q$ を用いて $*t_1$ を指し先 p で置き換える (**図 4.1(c)**)．これで最大多重度に関する処理が終了する．同様のことを最大多重度 $-1$ の変数に対して行う (**図 4.1(d)**，**図 4.1(e)**)．**図 4.1(e)** の代入文 {a,b} = 2; が choice 文である．

## 4.1 Hasti らの手法との比較

### 4.1.1 同等性

提案手法と Hasti らの手法の同等性については，次に述べる性質より容易に証明可能である．最大多重度はそれ以下の多重度の変数の影響を受けないため，すべての変数に対して繰り返し解析を行う Hasti らの手法と，最大多重度の変数のみを解析する提案手法は最大多重度の変数に対して等しい解析結果を得る．これは，Hasti らの手法では無駄な計算をしているが，最大多重度の変数に対しては無害であることを示している．このことを最大多重度 $-1$，最大多重度 $-2$，$\cdots$ と順に考えることで同等性がいえる．

### 4.1.2 解析の効率

Hasti らのアルゴリズム [HH98] と本研究の提案手法 (**Algorithm 2**) の実行効率を比較する．Hasti らのアルゴリズムは多重度を考慮していないため，プログラム全体を不動点に到達するまで繰り返し解析を行っている．つまり各繰返しの中で $\phi$ 関数の挿入と変数の名前付けをすべての変数に対して行っている．一方，提案手法では多重度を考慮したアルゴリズムとなっており，その結果各変数について 1 度だけ $\phi$ 関数の挿入と名前付けを行えばよい．その点から無駄な計算を除いた提案手法は Hasti らの手法より実行効率の良いものとなっている．なお，本研究の結果より Hasti らのアルゴリズムは，最悪の場合，最大多重度 $+1$ 回の繰返しで不動点を求められることがわかっている．

### 4.1.3 インクリメンタルな手法

Hasti らの手法はインクリメンタルな手法，すなわち繰返しを途中で止めることで，解析精度は低くなるが，解析コストを抑えることが可能である．本手法はその点で異なり，すべての変数を解析するまで繰返しを止めることはできない．しかし，次に示すように同等の結果を得られるアルゴリズムは本手法を用いても実現可能である．

Hasti らの手法を $k$ 回繰り返したときに得られる結果の説明のために，次の定義を必要とする．

**定義 10** (極大なポインタ変数)．多重度 $n$ のポインタ変数 $x$ が極大であるとは，$x$ に $\&a$ を与える代入文の列[1]があり，それらの代入文の両辺に多重度 $n+1$ 以上のポインタ変数が出現しないときである．

Hasti らの手法を $k$ 回繰り返して得られる結果のうち，フロー依存ポインタ解析と同等の結果となる変数は次のものとなる．

- 極大なポインタ変数

[1]たとえば，z = &a; y = z; x = y; のような代入文の列である．

```
struct S{
 int i;
 int j;
};
```

(a) 構造体

```
struct S s, t, *p;
s = t;
p = &s;
```

(b) プログラム

```
struct S s, t, *p;
int s_i, s_j, t_i, t_j, *p_i, *p_j;
s_i = t_i;
s_j = t_j;
p_i = &s_i;
p_j = &s_j;
```

(c) 変換例

図 4.2: 構造体型変数の基本型の変数への置き換え例

- 繰返しによって新たに得られる極大なポインタ変数

これは多重度 $n(n_{max} - k < n \leq n_{max})$ の変数を含む．これ以外の変数についてはフロー依存ポインタ解析の結果と同等であることは保証されない．すなわち，フロー依存ポインタ解析を部分的に行い，かつフロー非依存ポインタ解析を行った近似的な値となる．以上のことを考慮することにより，本手法を用いた同等の解析手法が得られる (付録 B 参照)．

## 4.2 提案手法の拡張

本手法は，C 言語で記述された一般的なプログラムのポインタに関係する文を抜き出して，それが本稿で定義した構文規則を満たしていれば適用することができる．また構文規則を満たしていない場合でも，**x を含むような代入文においては一時変数を導入すれば，規則を満たす代入文に変形することが可能である．さらに本稿で定義した構文規則の範囲内には入らない構成要素に関しては，次のように扱うことで適用できる．

- 配列

  配列の要素への参照はその配列全体への参照として扱う．たとえば，配列 a に関係した代入文 p = &a[i]; や p = a+i; を単に代入文 p = &a; と近似することによって解析可能になる．

- 構造体，共用体

  構造体名_メンバ名で 1 つの変数と見なすことで対応できる．構造体のコピー文はそれぞれに対応するメンバの代入文として扱う．例を**図 4.2** に示す．ただし，構造体のメンバに構造体へのポインタを含む場合に，同様の変換を行うためには，さらなる解析を必要とする．共用体についても同様に扱う．

- 動的メモリ確保 (malloc など)

  x = malloc(···); のような代入文は，動的メモリ確保関数を変数として考え，そのアドレスを x に代入する文であるとする．これらの関数は呼び出しごとに異なる変数であるとする．繰返し中は同一の変数を表すとする．

関数呼び出しについては**第 5 章**で述べる．このように構文要素を扱うことで，本稿で定義した構文規則に基づいた本手法を一般的なプログラムに適用できる．

# 第5章　手続き間解析への拡張

**第3章**で証明した未解決問題は手続き内解析，つまり関数呼び出しの無いプログラムに対するものであった．この章では，本稿で定義した構文規則に関数呼び出しを加えたとしても，フロー依存ポインタ解析とSSA形式を用いたフロー非依存ポインタ解析の結果が等しいことを示す．まず，手続き間解析の実現方法の1つであるsupergraphについて説明する．次に，supergraphを用いた手続き間フロー依存ポインタ解析アルゴリズムを示し，そのアルゴリズムとSSA形式を用いたフロー非依存ポインタ解析の解析結果が等しいことを証明する．そして，提案手法 (**Algorithm 2**) を関数呼び出しを考慮したアルゴリズムに拡張する．

## 5.1　supergraph

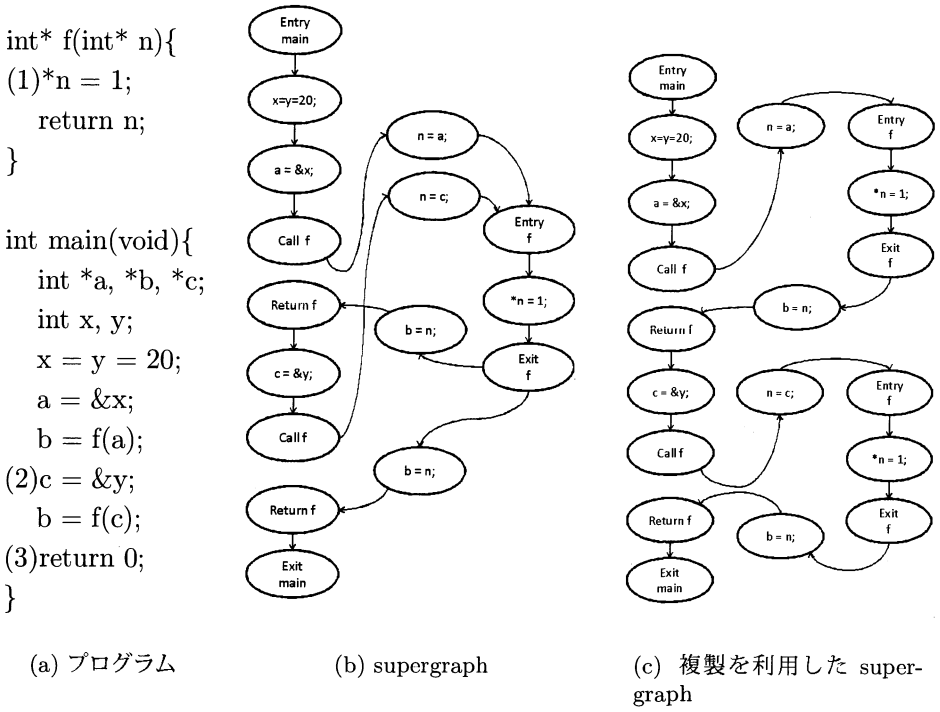
手続き間解析では，呼び出される関数に，実引数とグローバル変数，そしてそれらの変数から参照可能な変数の値が渡される．そして，その値を用いて関数内を解析した結果が呼び出しもとに返される．*supergraph* とは各関数に対する制御フローグラフに次のようなノードと辺を加えることで，関数間をつないだフローグラフである [ALSU07, HH98, LR91, Mye81][1]．

- 各呼び出し点 (関数呼び出しの文) に対して *call* ノードと *return* ノードを作る．
- call ノードに入る辺は呼び出し点に入る辺であり，return ノードから出る辺は呼び出し点から出る辺である．
- call ノードから呼び出される関数の entry ノードへの辺と，呼び出される関数の exit ノードから return ノードへの辺を追加する．

さらに，引数がある場合は，仮引数に実引数の値を代入するための文をまとめた新しいノードをcall ノードと entry ノードの間に加え，もし x = f(···); であれば，返却値を受け取る変数 x への代入文を持つ新しいノードを exit ノードと return ノードの間に加える．これにより，引数の値の伝播が実現される．このsupergraphは1つの制御フローグラフと見なせるので，supergraphに対して手続き内解析を行うことで，手続き間解析の結果を得られる．supergraphの例を**図5.1**に示す．

supergraphを用いた場合の文脈依存性を考えたとき，**図5.1(b)**のように単純に関数間に辺を追加した場合は，呼び出される関数への各入力がまとめられ，その出力がすべての呼び出し点に戻されることになるので，文脈非依存の手続き間解析となる．supergraphで文脈依存解析を実現する方法の1つとしては，関数の複製がある (**図5.1(c)**)．すなわち，関数 f を呼び出す各点から関数 f の制御フローグラフへ辺をつなぐのではなく，各呼び出し点に対して関数 f の制御フローグラフを複製し，その複製に対して辺の追加を行う．こうすることで，呼び出し点と関数が1対1に対応するので，呼び出しを区別した文脈依存の解析が行える．ただし再帰がある場合には，複製が無限に行われるので，この方法を用いることができない．再帰を構成する関数に対しては複製を行わず，従来の方法で呼び出し点と呼び出される関数を結ぶ．そして再帰に関する結果は一般

[1] Landi ら [LR91] は，*interprocedural control flow graph(ICFG)* という言葉を使っている．




(a) プログラム　(b) supergraph　(c) 複製を利用した supergraph

図 5.1: (a) のプログラムに対する supergraph

に不動点を求めることによって得られる [EGH94, ALSU07]．再帰を含む supergraph の例を付録 C に示す．

表 5.1: **図 5.1(a)** に対するポインタ解析の結果

| | 文脈非依存 (**図 5.1(b)**) | 文脈依存 (**図 5.1(c)**) |
|---|---|---|
| ポインタ情報 | (1) n → {x,y}, a → x, b → {x,y}, c → {NULL,y} | (1) n → x, a → x |
| | | (1)' n → y, a → x, b → x, c → y |
| | (2) n → {x,y}, a → x, b → {x,y}, c → {NULL,y} | (2) n → x, a → x, b → x |
| | (3) n → {x,y}, a → x, b → {x,y}, c → {NULL,y} | (3) n → y, a → x, b → y, c → y |

**図 5.1(a)** のプログラムに対して，supergraph を用いた文脈非依存フロー依存ポインタ解析と文脈依存フロー依存ポインタ解析の結果を**表 5.1** に示す．文脈依存解析において関数 f は区別されるので，1 回目の呼び出しに対する結果を (1)，2 回目の呼び出しに対する結果を (1)' に示す．ポインタ変数が指す変数が不定であることを NULL を用いて表す．

```
void (*fp)();

int main(void){
    ⋮
  if(···){
    fp = f;
  }
  else{
    fp = g;
  }
  fp();    ···(*)
    ⋮
}

void f(){
    ⋮
  if(···){
    fp();    ···(1)
  }
    ⋮
}

void g(){
    ⋮
  if(···){
    fp();    ···(2)
  }
    ⋮
}
```

図 5.2: 関数ポインタの例

### 5.1.1 関数ポインタ

関数ポインタが存在する場合，関数ポインタのとりうる値が分からなければ，supergraph を構築することができない．したがって，supergraph を構築するためには，まず関数ポインタに関するポインタ解析を行い，それらの値を得る必要がある．

また，グローバルな関数ポインタによって呼び出される関数 `f` を解析するとき，関数 `f` 内でその関数ポインタの初期値が `f` であると考えなければならない [EGH94]．これについて，**図 5.2** を用いて説明する．(*) で関数ポインタによる関数呼び出しが行われる．この点において，グローバル変数 `fp` のポインタ情報は `fp` → {`f`,`g`} である．値 `f` を用いて関数呼び出しを行うとき，呼び出される関数 `f` の中では `fp` の値は `f` であり，決して (1) の点において，関数 `g` を呼び出すことはない．同様に関数 `g` においても関数 `f` を呼び出すことはない．これは，呼び出す関数に対する引数の代入文を加えたノードに代入文 (この例では，`fp = f;` や `fp = g;`) を追加することで，supergraph 上で実現できる．

## 5.2 手続き間解析を行う提案手法

### 5.2.1 構文の拡張

関数呼び出しとして，以下の構文を考える．

$$f(\cdots);$$
$$x = f(\cdots);$$

f は関数名または関数ポインタであり[2]，x は $n$ 重ポインタ変数か*($n+1$ 重ポインタ変数) である．

### 5.2.2 supergraph を用いた解析

supergraph を用いた手続き間フロー依存ポインタ解析は **Algorithm 3** によって行われる．まず，1 において通常の関数呼び出しに対して supergraph を構築する．このとき，関数ポインタを用

[2]C 言語では fp を関数ポインタとした場合，fp(···) と (*fp)(···) のどちらでも記述できるので，ここでは区別しない．

**Algorithm 3** supergraphを用いたフロー依存ポインタ解析アルゴリズム

```
1: supergraph Sを構築する
2: repeat
3:   repeat
4:     for all s ∈ Sの代入文の集合 do
5:       データフロー方程式を計算する
6:     end for
7:   until 解析結果に変化がない
8:   計算された値をもとにSを更新する
9: until Sに変化がない
```

いた関数呼び出しは，その関数ポインタの値が不定であるのでそのままプログラムに残る．3-5ではプログラムの各点での各変数のとりうる値を計算する．そして6で，関数ポインタを用いた関数呼び出しに対して，その値に応じて辺を追加しsupergraphを更新する．この処理は，supergraphに更新がなくなるまで繰り返す(2-7)．**Algorithm 3**を適用した例を付録Dに示す．

このsupergraphを用いた手続き間解析について，次の定理が成り立つ．

**補題 5.** 2.1節の構文規則を満たす代入文と5.2.1節の関数呼び出しの列からなる関数を考える．これらの関数から複製を利用して構築された*supergraph*は2.1節の構文規則を満たしている．

証明. supergraphは関数呼び出しをその関数の本体で置き換えたものである．また関数ポインタの値によって，複数の呼び出し先を持つ可能性があるが，そのときはif文として見ることができる．よって得られたsupergraphは2.1節の構文規則を満たしている． □

**定理 3.** 2.1節と5.2.1節で定義した構文規則を満たすプログラムにおいて，**Algorithm 3**と**Algorithm 3**のデータフロー方程式を*SSA*形式上のフロー非依存ポインタ解析に置き換えたアルゴリズムの解析結果は等しい．

証明. 定理2より，データフロー方程式とSSA形式を用いたフロー非依存ポインタ解析の結果は等しくなる． □

supergraphを用いた解析において，全ての変数の値を正しく求めるためには，supergraphに更新が無い状態，つまり関数呼び出しが適切に呼び出し先の関数とつながっている状態である必要がある．また，関数ポインタについては，その値が通常のポインタ変数の値に依存しないことから，関数ポインタだけを事前に解析することができる．これを考慮したアルゴリズムを手続き間解析を行う提案手法として，**Algorithm 4**に示す．さらに，このアルゴリズムについて，次の定理が得られる．

**定理 4.** 2.1節と5.2.1節で定義した構文規則を満たすプログラムにおいて，**Algorithm 3**と**Algorithm 4**は同等の解析能力を有する．

証明. 定理3より，両者の解析において不動点に達したときに得られているsupergraphは等しい．そして，補題5と定理2より，そのsupergraphに対するそれぞれの解析は等しい解析結果を得る．したがって，**Algorithm 3**と関数ポインタを事前に解析する**Algorithm 4**は同等の解析能力を有する． □

**Algorithm 4** 関数呼び出しを考慮したポインタ解析アルゴリズム

**Require:** 制御フローグラフ (CFG) のリスト L

通常の関数呼び出しを考慮した supergraph S を構築する

**repeat**

S の中の関数ポインタ変数を SSA 形式に変換する ($SSA_f$)

$SSA_f$ に対してフロー非依存ポインタ解析を行う

解析結果を用いて S を更新する

**until** S に変化がない

**for** $n =$ 最大多重度 to 1 **do**

S の中の n 重ポインタ変数を SSA 形式に変換する ($SSA_n$)

$SSA_n$ に対してフロー非依存ポインタ解析を行う

$SSA_n$ のデリファレンスを解析結果を用いて置き換える

**end for**

# 第6章　実験評価

この章では，本研究の提案手法と Hasti らの手法 [HH98] を用いた解析結果を示し，2 つの手法の実行効率 (解析にかかる時間) について考察を行う．実験には**表 6.1** の 10 個の C プログラムを用いた．プログラムの行数 (LOC) は (ライブラリ関数を除く).c ファイルと.h ファイルを wc コマンドを用いて得られた値で，コメントも含む．代入文は解析対象の代入文の数を示し，多重度 1 の変数に関する代入文の数を多重度 1 として表している．

表 6.1: ベンチマークの特徴

| | | 代入文 | | | |
|---|---|---|---|---|---|
| | LOC | 多重度 1 | 多重度 2 | 多重度 3 | 関数呼び出し |
| allroots | 205 | 11 | 0 | 0 | 37 |
| queens | 363 | 13 | 1 | 0 | 28 |
| fixoutput | 456 | 7 | 0 | 0 | 94 |
| genetic | 509 | 28 | 0 | 0 | 43 |
| anagram | 647 | 39 | 14 | 10 | 44 |
| compress | 1497 | 51 | 9 | 0 | 83 |
| mway | 690 | 54 | 0 | 0 | 110 |
| ansitape | 1744 | 97 | 2 | 0 | 153 |
| compiler | 2368 | 49 | 2 | 0 | 377 |
| football | 2261 | 225 | 1 | 0 | 328 |

## 6.1　実装

本研究では 4 つの手法，すなわち提案手法 (手続き内解析 **Algorithm 2**，手続き間解析 **Algorithm 4**) と Hasti らの手法 (手続き内解析 **Algorithm 1** と手続き間解析 [HH98]) を実装した．これらの解析は文脈非依存の解析を行う．各手法で用いられるフロー非依存ポインタ解析には [And94] を，制御フローグラフの支配木を求めるアルゴリズムには [LT79] を，SSA 形式への変換アルゴリズムには [CFR+91] をそれぞれ用いている．

それらの解析手法の実装は COINS[Pro] 内に組み込む形で行った．COINS は入力プログラムに近い表現形式の高水準中間表現 (HIR) と機械語に近い表現形式の低水準中間表現 (LIR) を核として，それらを扱う様々なモジュールからなっている．本研究では，入力プログラムに近い形式であり，かつポインタの型情報などが保持されている HIR 上で実装を行った．つまり，COINS によって生成される HIR の制御フローグラフを解析手法の入力として受け取り，それに対して解析を行う形となる．

表 6.2: 各解析手法の解析時間 (ms)．比率は"Hasti らの解析時間 / 提案手法の解析時間"で求めたものである．

| | 手続き間 | | | 手続き内 | | |
|---|---|---|---|---|---|---|
| | 提案手法 | Hasti ら | 比率 | 提案手法 | Hasti ら | 比率 |
| allroots | 9.6 | 17.2 | 1.79 | 7.0 | 12.4 | 1.77 |
| queens | 15.6 | 20.8 | 1.33 | 9.2 | 10.6 | 1.15 |
| fixoutput | 13.0 | 23.8 | 1.83 | 8.8 | 14.8 | 1.68 |
| genetic | 32.2 | 55.2 | 1.71 | 13.2 | 36.4 | 2.76 |
| anagram | 71.6 | 181.8 | 2.54 | 65.0 | 157.6 | 2.42 |
| compress | 95.2 | 173.8 | 1.83 | 64.0 | 117.2 | 1.83 |
| mway | 93.8 | 180.0 | 1.92 | 59.0 | 143.6 | 2.43 |
| ansitape | 117.2 | 357.8 | 3.05 | 99.2 | 313.2 | 3.16 |
| compiler | 250.4 | 529.4 | 2.11 | 92.2 | 159.0 | 1.72 |
| football | 242.8 | 610.2 | 2.51 | 172.4 | 267.6 | 1.55 |

## 6.2 実験結果と考察

実験結果を**表 6.2** に，その結果を手続き間解析と手続き内解析に分けて図示したものを**図 6.1** と**図 6.2** に示す．手続き間解析の解析時間は，制御フローグラフのリストを受け取ってから最終的なポインタ情報を得るまでの時間であり，手続き内解析の解析時間は supergraph を前もって構築し，それを入力として受け取ってからポインタ情報を得るまでの時間である．各々の解析時間は 5 回の平均によるもので，比率は"Hasti らの解析時間 / 提案手法の解析時間"で計算したものである．

**表 6.2** より，手続き間，手続き内のそれぞれの解析において，提案手法は Hasti らの手法より解析時間が短いことが分かる．よって，提案手法は Hasti らの手法と比べて効率的であるといえる．また，この解析時間の違いには Hasti らの手法の繰り返し回数が大きく関係していると考えられる．**図 6.3**，**図 6.4** は**表 6.2** の比率と Hasti らの手法の繰り返し回数をまとめたものである．これらの図を見ると，若干の誤差は見られるが，比率と繰り返し回数のグラフが似た形になっていることが分かる．このことは繰り返しの回数だけ，解析時間が増えていることを表している．実際，提案手法は"プログラム全体を SSA 形式に変換し，ポインタ解析を行う"ことを 1 度だけ行うのに対し，Hasti らの手法は"プログラム全体を SSA 形式に変換し，ポインタ解析を行う"ことを何度も繰り返す．よって，この結果は両者のアルゴリズムの違いを表していると考えられる．

Hasti らの手法の不動点を求めるための繰り返し回数について考える．本研究ではその繰り返し回数が，最悪の場合，最大多重度+1 になると示した．実験結果を見ると，繰り返し回数は 2 か 3 であり，最大多重度+1 に収まっている．しかし，多重度は大きくても 3 であり，1 重ポインタ変数までしか含まないプログラムも多数見られた．よって，少数の実験結果ではあるが，一般的なプログラムに対する不動点計算の繰り返し回数は，最悪の場合は最大多重度の数+1 となるが，実際にはそれほど大きくならない (2，3 回程度) と推測できる．ただし，この繰り返し回数は関数ポインタを含まないプログラムに対してのものである．関数ポインタがある場合には，さらに回数が増えると予想される．

以上のことから，提案手法が Hasti らの手法と比べて効率的であり，かつ繰り返し回数が解析時間に大きく影響していることが分かる．

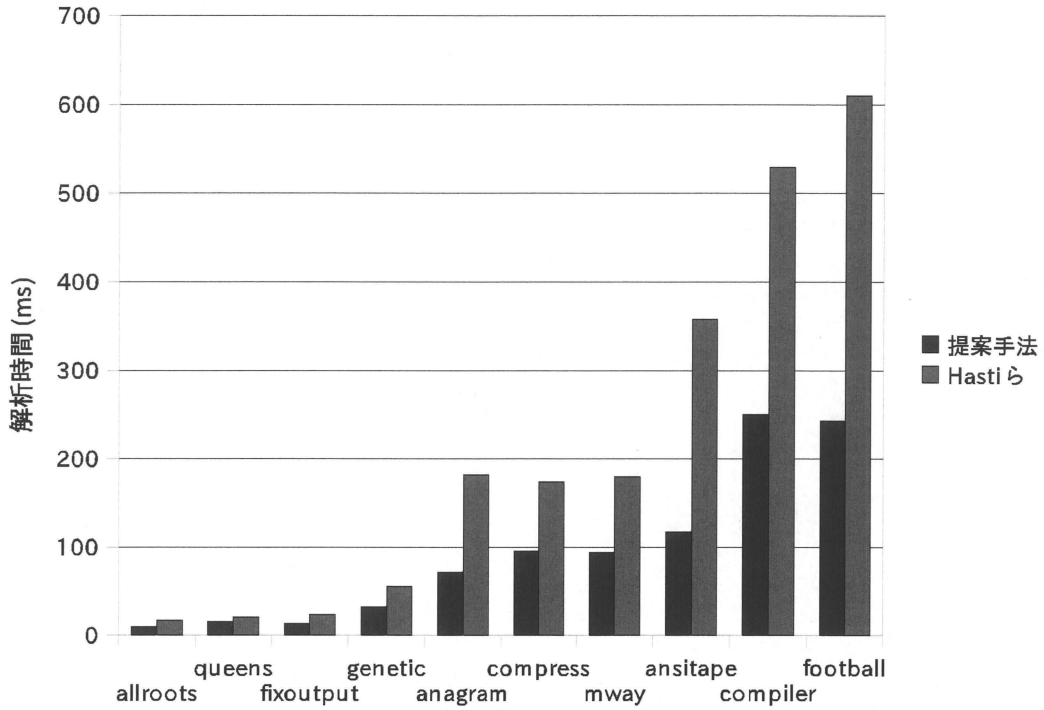



図 6.1: 手続き間解析の解析時間

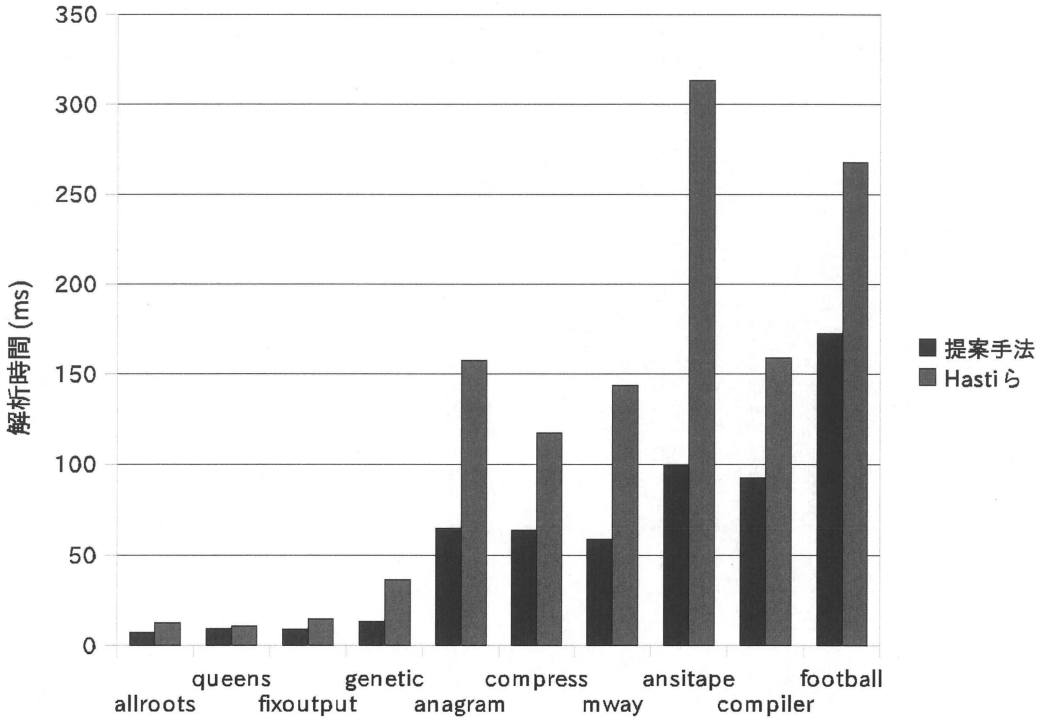



図 6.2: 手続き内解析の解析時間

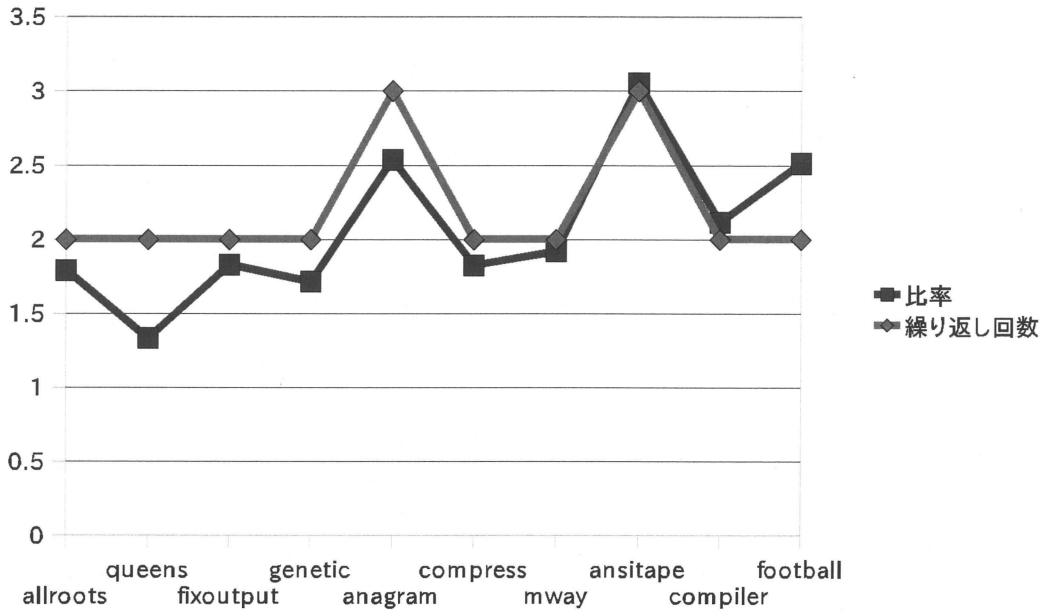



図 6.3: 手続き間解析における，提案手法に対する Hasti らの手法の解析時間の比率と Hasti らの手法の繰り返し回数

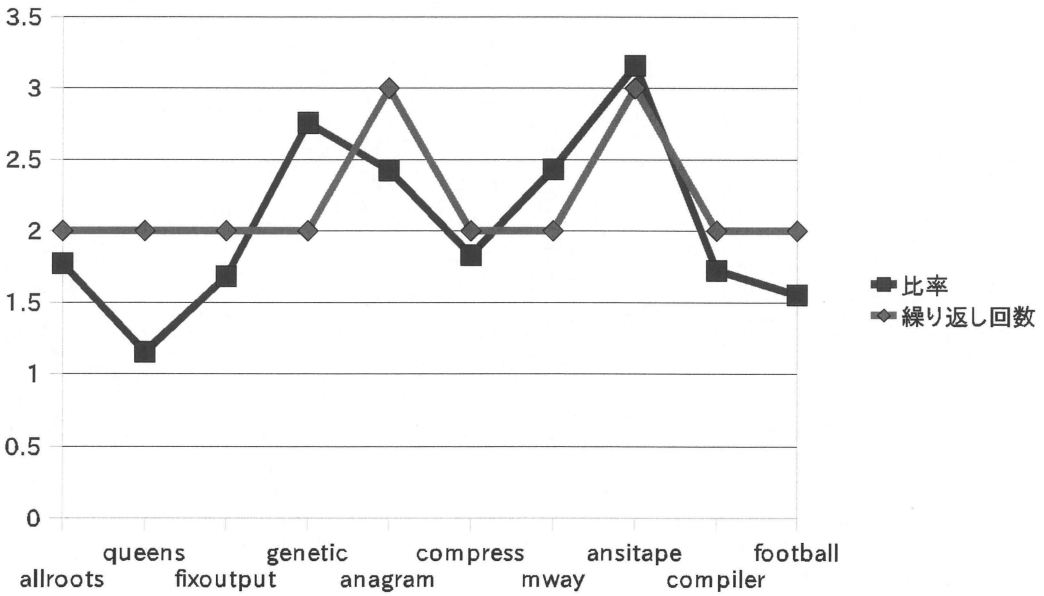



図 6.4: 手続き内解析における，提案手法に対する Hasti らの手法の解析時間の比率と Hasti らの手法の繰り返し回数

# 第7章　関連研究

Hastiら [HH98] は本研究の動機となった未解決問題,「フロー依存ポインタ解析とSSA形式を用いたフロー非依存ポインタ解析が同等の解析能力を有するか否か」を提起した．本研究はその未解決問題を考察し，解決した．そして，Hastiらの手法のように，全ての変数を繰り返し解析するのではなく，多重度の概念を用いて順に解析を行うことで，実行効率を改善できることを示した．

Lapkowskiら [LH98] はポインタを持つ言語に対するSSA形式の問題 (本稿でも述べたデリファレンスの問題など) について議論している．彼らは各ポインタ変数pに対して，pの値を表す添字と*pの値を表す添字という2つの添字を与えることで，問題の解決をはかった．しかし彼らの *Extend SSA Numbering* と呼ばれる方法によって変換されたプログラムは，$\phi$関数を持たないため，各変数の使用に対して一意に定義が定まるというSSA形式の性質を満たしていない形式となっている．

Hardekopfら [HL09] はSSA形式とフロー依存ポインタ解析を組み合わせた手法を提案した．彼らの手法は変数を2つのクラス，*top-level* 変数 (アドレス演算子&を用いてそのアドレスが他の変数に渡されない変数) と *address-taken* 変数 (アドレスが他の変数にわたり，ポインタによって間接的に参照される変数) に分類し，top-level変数をSSA形式に変換したあとフロー依存ポインタ解析を行う．top-level変数はSSA形式なので，それらのポインタ変数の指し先情報はプログラム全体で1つのみ保持すればいい．よって各文が保持するポインタ変数の指し先情報を減らすことができ，解析の効率の向上につながる．Hardekopfらの手法はtop-level変数に対する結果のみをグローバルに保持するのに対し，本手法は，繰返しの中でaddress-taken変数をtop-level変数の性質を持つように置き換え，最終的に全てのポインタ変数の指し先情報をプログラム全体で1つ保持するものである．Hardekopfらは，未解決問題に対して，両者の解析能力が同等ではないと予想していたが，本研究では予想に反して，同等の解析能力を有することを示した．

田中 [田中 10] はinclusion-basedポインタ解析における制約グラフの動的推移閉包の問題に対して，多重度の概念を導入することで，計算量を抑えたアルゴリズムを提案した．本研究ではその多重度の概念をSSA形式に適用した新しいアルゴリズムを提案し，従来手法より実行効率の良いことを確認した．

# 第8章　おわりに

本稿では，まずHastiら[HH98]が提起したSSA形式を用いたフロー非依存ポインタ解析に関する未解決問題を，限定的なプログラムにおいて解決した．すなわち本稿で定義した構文規則を満たすプログラムにおいては，Hastiらのアルゴリズムがフロー依存ポインタ解析と同等の解析能力を有することを示した．次に，構文規則を関数呼び出し，配列などを含むように拡張しても，同等の解析能力を有することも確認した．このことから，一般的なプログラムに対して，フロー依存ポインタ解析と同等の結果をSSA形式を用いたフロー非依存ポインタ解析でも得られると考えている．さらにHastiらのアルゴリズムと同等の解析能力を持ち，より実行効率の良い新しいアルゴリズムを提案し，実験によりその提案手法の有用性を示した．

今後の課題としては，手続き間解析と構造体の扱いについてが挙げられる．supergraphを用いた手続き間解析を行う提案手法は，確かにフロー依存ポインタ解析と同等の解析能力を持つ．しかし，複製を利用したsupergraphの構築は呼び出される関数のインライン展開(再帰を除く)と同様であると考えられ，プログラムのサイズが指数的に増加してしまうおそれがある．よって，手続き間解析を行う提案手法は，メモリ使用量や実行効率の点で，さらなる考察が必要となる．

構造体のメンバに構造体へのポインタがある場合は，それに対して**第4章**で示した適用方法を用いることができない．それを解決する1つの方法は手続き内解析の直前に構造体へのポインタ型の変数に対してポインタ解析を行うことである．その解析結果を利用し，構造体へのポインタ型の変数を置き換えることで，**第4章**の方法を適用できるようになると考えている．その構造体へのポインタ型の変数の値を求める解析には，多重度の概念を適用できないので，不動点を求める解析が必要になると考えている．

# 謝辞

本研究を述べるにあたり，日頃丁寧にご指導いただき，度々研究の議論の時間を設けていただいた大山口通夫教授に心より感謝いたします．また，講義等を通してご指導いただいた山田俊行講師，研究に関して様々なご指摘をいただいた三橋一郎助教，ならびに何かとお世話になりました落合美子事務職員に深く感謝いたします．

そして，研究や講義に関して，熱心に議論していただき，様々な指摘をしていただいた廣苅直人氏に深く感謝いたします．さらに，研究に関する情報提供や指摘をしてくださった研究室の学生諸氏に感謝いたします．

# 参考文献


[AH00] John Aycock and Nigel Horspool. Simple Generation of Static Single-Assignment Form. *Lecture Notes in Computer Science*, pp. 110–124, 2000.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools Second Edition.* Pearson/Addison-Wesley, 2007.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994.

[CFR+91] 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 Transaction on Programming Language and Systems*, Vol. 13, No. 4, pp. 451–490, October 1991.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 242–256. ACM, 1994.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 4, pp. 848–894, 1999.

[HH98] Rebecca Hasti and Susan Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 97–105, June 1998.

[Hin01] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54–61, 2001.

[HL07a] Ben Hardekopf and Calvin Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Static analysis: 14th international Symposium, SAS 2007*, pp. 265–280, 2007.

[HL07b] Ben Hardekopf and Calvin Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 290–299. ACM, 2007.

[HL09] Ben Hardekopf and Calvin Lin. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 226–238, January 2009.

[HP00] Michael Hind and Anthony Pioli. Which Pointer Analysis Should I Use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 113–123. ACM, 2000.

[LH98] Christopher Lapkowski and Laurie J. Hendren. Extended SSA Numbering: Introducing SSA Properties to Languages with Multi-Level Pointers. *Lecture Notes in Computer Science*, Vol. 1383, pp. 128–143, 1998.

[LR91] William Landi and Barbara G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 93–103, 1991.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, pp. 121–141, 1979.

[Mye81] Eugene W. Myers. A Precise Inter-Procedural Data Flow Algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 219–230. ACM, 1981.

[NL09] Nomair A. Naeem and Ondřej Lhoták. Efficient Alias Set Analysis Using SSA Form. In *Proceedings of the 2009 International Symposium on Memory Management*, pp. 79–88, 2009.

[Pro] COINS Project. COINS Project homepage. http://www.coins-project.org/.

[Ram94] Ganesan Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1467–1471, 1994.

[SH97a] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–14, January 1997.

[SH97b] Marc Shapiro and Susan Horwitz. The Effects of the Precision of Pointer Analysis. *Lecture Notes in Computer Science*, Vol. 1302, pp. 16–34, 1997.

[Ste96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 32–41, January 1996.

[田中 10] 田中友幸. プログラムの参照先解析に関する研究. Master's thesis, 三重大学大学院工学研究科, March 2010.

# 付 録A　Hastiらの手法[HH98]の実行例

**図 A.1**, **図 A.2** に**図 A.1(a)** のプログラムに対する Hasti らの手法の実行例を示す.

**前処理**

**図 A.1(a)** に対してフロー非依存ポインタ解析を行い，解析結果として次の結果を得る.

$$\mathtt{t} \rightarrow \{\mathtt{s}\}, \mathtt{s} \rightarrow \{\mathtt{p}, \mathtt{q}\}$$

この結果をデリファレンス (`*t`,`*s`) に注釈として付与する.

**繰返し (1 回目)**

i. デリファレンスの注釈に基づいて，**図 A.1(b)** の中間形式に変換する．このとき変数 s は 2 つの指し先の候補を持っているため，branch ノードを用いて分岐の形に置き換える．そして，このプログラムを SSA 形式に変換する (**図 A.1(c)**).

ii. **図 A.1(c)** に対してフロー非依存ポインタ解析を行い，次のような解析結果を得る.

$$\mathtt{t}_1 \rightarrow \{\mathtt{s}\}, \mathtt{s}_1 \rightarrow \{\mathtt{p}\}, \mathtt{s}_2 \rightarrow \{\mathtt{q}\}$$

これを用いて**図 A.1(a)** の注釈を更新する．このとき，`*t` はもしそのまま現れていたとすると$*\mathtt{t}_1$ となっていたため，ポインタ情報 $\mathtt{t}_1 \rightarrow \{\mathtt{s}\}$ を用いて更新する．同様に`*s` はもし branch ノードに現れていたとすると$*\mathtt{s}_2$ となっているため，$\mathtt{s}_2 \rightarrow \{\mathtt{q}\}$ を用いる.

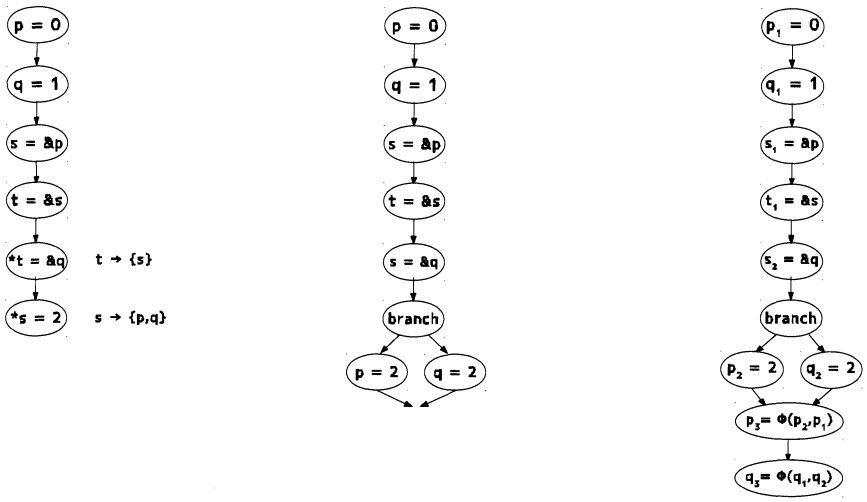
iii. この更新により`*s` の注釈に変化が生じたため，もう 1 度処理を繰り返す (**図 A.2**).

**繰返し (2 回目)**

1 回目の繰返しと同様の処理を行ったあと，**図 A.2(c)** に対してフロー非依存ポインタ解析を行うと，次の結果が得られる.

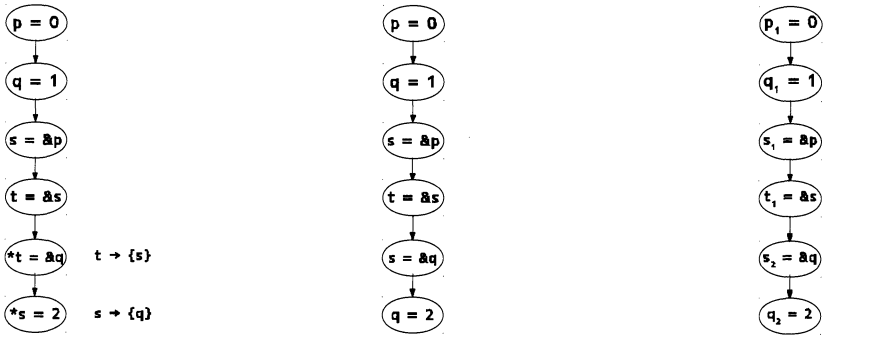
$$\mathtt{t}_1 \rightarrow \{\mathtt{s}\}, \mathtt{s}_1 \rightarrow \{\mathtt{p}\}, \mathtt{s}_2 \rightarrow \{\mathtt{q}\}$$

この解析結果は 1 回目の解析結果と同等であり，各デリファレンスの注釈も変化がないため，この結果が最終的なポインタ情報となる.




(a) CFG（注釈付き）　(b) 中間形式　(c) SSA 形式

図 A.1: Hasti ら [HH98] の実行例 ( 1 回目)




(a) CFG（注釈付き）　(b) 中間形式　(c) SSA 形式

図 A.2: Hasti ら [HH98] の実行例 ( 2 回目)

# 付 録B　提案手法のインクリメンタルな手法への変更

Hasti らの手法である途中で止めることで得られる解析結果と同等の結果を得る手法を **Algorithm 5** に示す．なお，極大なポインタ変数を求める方法としては，SSA 形式に変換して極大なポインタ変数を求めることが効率よく計算できる方法の 1 つである．**Algorithm 5** の手法では，極大なポインタ変数は 1 度だけ解析され，それ以降の解析では不必要となり除去されるのに対し，Hasti らの手法は何度も繰り返し解析を行われる．この点が大きな違いである．

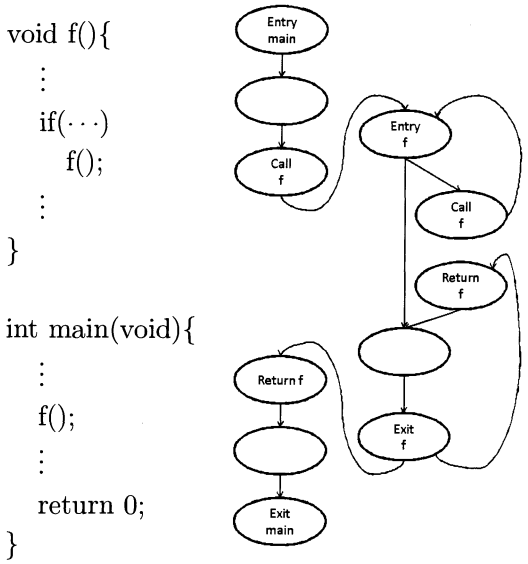
**Algorithm 5** 提案手法 (Hasti らのインクリメンタルな手法に対応)

**Require:** 制御フローグラフ CFG，多重度 i($1 \leq i \leq max$ )
  **for** n = max to i **do**
    極大なポインタ変数を SSA 形式に変換する ($SSA_n$)
    $SSA_n$ に対してフロー非依存ポインタ解析を行う
    $SSA_n$ の極大なポインタ変数のデリファレンスを解析結果を用いて置き換える
  **end for**

  **if** i > 0 **then**
    フロー非依存ポインタ解析を行う
    デリファレンスを置き換え，SSA 形式に変換する ($SSA_i$)
    $SSA_i$ に対してフロー非依存ポインタ解析を行う
  **end if**

# 付 録C　再帰呼び出しを含むsupergraph

再帰呼び出しを含むとき，supergraphは**図 C.1** のようになる．なお，このプログラムに対するsupergraphは，複製を用いるか否かに関わらず，**図 C.1(b)** の形となる．




(a) プログラム　　(b) supergraph

図 C.1: 再帰関数を含む supergraph

# 付 録D　supergraphの構築例

**図 D.1(a)** のプログラムに対する，**Algorithm 3** の適用例を示す．この例では複製を用いたsupergraphを構築する．

まず関数ポインタによらない関数呼び出しを考慮したsupergraphを構築する (**図 D.1(b)**)．このとき，2つの関数呼び出し `(*fp)();` は処理しない．

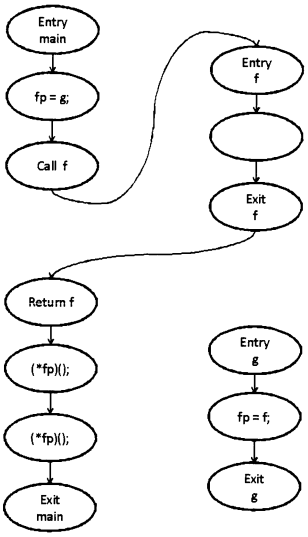
次にそのsupergraphに対してフロー依存ポインタ解析を行う．その結果，各 `(*fp)();` で fp → g というポインタ情報が得られる．これらの情報を用いてsupergraphを更新したものが**図 D.1(c)** となる．

supergraphに変化があったので，**図 D.1(c)** に対してフロー依存ポインタ解析を行う．この解析では，1つ目の `(*fp)();` には，fp → g，2つ目の `(*fp)();` には，fp → f というポインタ情報が与えられる．この情報から，supergraphは**図 D.1(d)** のように更新される．

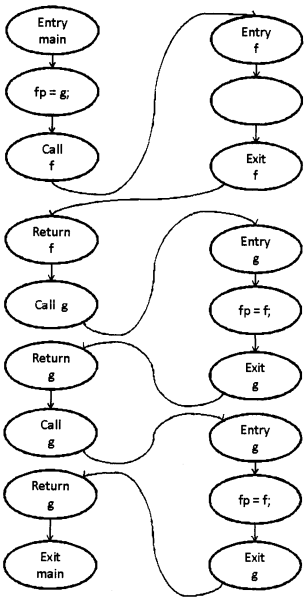
supergraphが更新されたので，さらにフロー依存解析を行う．この結果は前回の結果と変わらない．よってsupergraphの変化も無いため，解析が終了する．この例ではsupergraphの構築に焦点を当てたが，実際には，関数ポインタだけでなく，通常のポインタ変数に対する解析も同時に行われている．

```
void (*fp)();

void f(){
   ⋮
}

void g(){
   fp = f;
}

int main(void){
   fp = g;
   f();
   (*fp)();
   (*fp)();
   return 0;
}
```

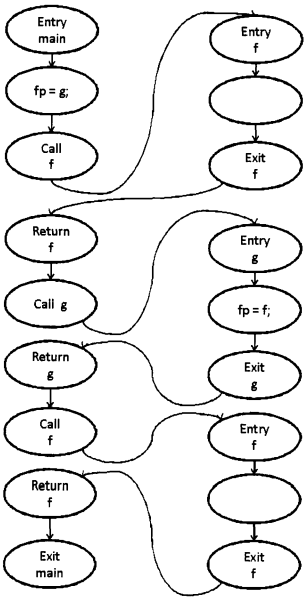
(a) プログラム




(b) 通常の関数呼び出しを考慮した supergraph




(c) 1 回目の繰返しで得られる supergraph




(d) 2 回目の繰返しで得られる supergraph

図 D.1: supergraph