情報処理学会論文誌　プログラミング　Vol. 4　No. 2　92–103 (Mar. 2011)

静的単一代入形式を用いたポインタ解析アルゴリズム

田中雄一[†1]　大山口通夫[†1]

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．すなわち，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の両方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，限定されたプログラムのクラスにおいて，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．

A Pointer-analysis Algorithm Using Static Single Assignment Form

Yuichi Tanaka[†1] and Michio Oyamaguchi[†1]

Pointer analysis is prerequisite for many program analyses, and the pointer information is useful for program optimization and improvement of reliability. However, pointer analysis performed before program execution is impossible to compute a complete pointer information in general, so that a large variety of approximation algorithms taking a trade-off between the efficiency and precision into consideration have been proposed. Flow-sensitivity is one of several dimensions that affect this trade-off of pointer analysis, that is, flow-sensitive analysis provides precise information, but inefficient, whereas flow-insensitive analysis is more efficient, but less precise. Hasti and Horwitz (1998) have proposed an algorithm that combines a flow-insensitive pointer analysis with static single assignment (SSA) form and shown that it is useful for improving the imprecise result of flow-insensitive analysis. They also posed the question of whether the resulting pointer information is the same as that of a flow-sensitive pointer analysis as an open problem. Hardekopf and Lin (2009) have expected that the open problem may be negative, that is, the flow-insensitive approach is less precise than a flow-sensitive one and proposed an algorithm that combines a flow-sensitive pointer analysis with SSA form. In this paper we solve the open problem posed by Hasti and Horwitz, that is, show that the two algorithms for a restricted program class give the same analysis result to the contrary to the conjecture of Hardekopf and Lin. We also propose a new algorithm which improves Hasti-Horwitz's algorithm in efficiency.

1. 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．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[7]．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[6] は静的単一代入形式（Static Single Assignment Form，以下 SSA 形式）[2] 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[5] 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の両方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，限定したプログラムのクラスにおいて，Hardekopf ら

†1 三重大学大学院工学研究科
Graduate School of Engineering, Mie University

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．なおプログラムを限定したことは本質的な問題ではなく，本稿の後半で述べるように構文規則を拡張したクラスに対しても，2 つのアルゴリズムが同等の解析能力を有することを確認済みである．

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．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．これまでに提案された静的解析アルゴリズムは制御フロー（プログラムの文の実行順序）を考慮するか否かで大きく 2 つに分類される．実行される文の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，文の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[1),3),4),6),8)–11)]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[5)]．

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[5)]．

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* [1)] と *equality-based* [9)] 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[3),4)]．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 [9)]

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

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

図 1　通常のプログラム
Fig. 1　A program fragment.

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

図 2　SSA 形式への変換例
Fig. 2　An example of translation to SSA form.

表 1　図 1 のプログラムに対するポインタ解析の結果
Table 1　The results of flow-sensitive and flow-insensitive pointer analyses for the program of Fig.1.

	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)　$a \to w$ (3)　$a \to w$, $p \to a$ (4)　$a \to w$, $b \to x$, $p \to a$ (5)　$a \to w$, $b \to \{x, y\}$, $p \to a$ (6)　$a \to w$, $b \to \{x, y\}$, $p \to b$	$a \to w$ $b \to \{x, y\}$ $p \to \{a, b\}$

のアルゴリズムは $O(n\alpha(n, n))$[⋆2]である．

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．

図 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** に示す．ここで，$p \to a$ は p が a を指す，$p \to \{a, b\}$ 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

⋆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 以下と考えられる）．

て，不正確であることが分かる．

1.2　静的単一代入形式（SSA 形式）

SSA 形式[2] 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 ϕ 関数[⋆1]を挿入する．

SSA 形式の例を**図 2** に示す．同じ変数に対して，代入文で定義されるたび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，ϕ 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で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 形式の性質を利用してフロー非依存ポインタ解析の精度を改善した手法が Hasti ら[6] によって提案された．図 2 に対してこの手法を用いて得られる結果を次に示す．

$$a_1 \rightarrow w_0$$
$$b_1 \rightarrow x_0\ ,\ b_2 \rightarrow y_0\ ,\ b_3 \rightarrow \{x_0, y_0\}$$
$$p_1 \rightarrow a_1\ ,\ p_2 \rightarrow b_3$$

この結果は図 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．しかし SSA 形式とポインタ解析を組み合わせるには問題点がある．その問題とはデリファレンス（*p）が存在するプログラムではうまく SSA 変換できない場合があるということである．その例を**図 3** に示す．

この図 3 の例では問題点が 2 つある．まず 1 つ目は図 3 (b) の (4) である．$*p_1$ 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．以上より，SSA 形式に正しく変換するためにはデリファレンスを考慮した変換を考える必要がある．

⋆1 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

	(a) 通常のプログラム	(b) SSA 形式プログラム
(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;

図 3　デリファレンスを含む SSA 形式への変換例
Fig. 3　An example with pointer dereferences and their translation to SSA form.

(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
repeat
(3)　　CFG から中間形式（IM）を作る
(4)　　IM を SSA 形式に変換する （IM_{ssa}）
(5)　　IM_{ssa} に対してフロー非依存ポインタ解析を行う
(6)　　IM_{ssa} と解析結果を使って CFG の注釈を更新する
until 注釈に変化がない

図 4　Hasti らのアルゴリズム
Fig. 4　Hasti-Horwitz's algorithm.

Hasti らはこのデリファレンスに関する問題を克服するために，**図 4** に示したアルゴリズムを提案した．図 4 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，$p \to \{a, b\}$ 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，最初にフロー非依存ポインタ解析を行い (1)，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 (2)．そして繰返しの中では，まず CFG のデリファレンスを注釈を用いて置き換えている (3)．これにより，プログラムにデリファレンスがなくなり図 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 (4)．次に 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 (5)，(6)．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．図 4 のアルゴリズムの実行例を付録 A.1 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[6]．

2. 準　　備

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]．

$$n \text{ 重ポインタ変数} = n \text{ 重ポインタ変数}; \quad (1)$$
$$n \text{ 重ポインタ変数} = \&(n-1 \text{ 重ポインタ変数}); \quad (2)$$
$$n \text{ 重ポインタ変数} = *(n+1 \text{ 重ポインタ変数}); \quad (3)$$
$$*(n+1 \text{ 重ポインタ変数}) = n \text{ 重ポインタ変数}; \quad (4)$$

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．

⋆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 (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 (4) の代入文の形に変形可能である．

2.2 デリファレンスの置き換え

文献 6) 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 ϕ 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 { ⋯} を含む代入文を本稿での choice 文の記法とする．

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，その代入文が x = e’; の形を持つときであるか，または d:*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’ ≡ & a
(2) e’ ≡ 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’ ≡ *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．

$$d\text{”} : g = g\text{’}; (l(d\text{”}) \equiv \&y\ ,\ r(d\text{”}) \equiv \&a)$$
$$\vdots$$
$$d\text{’} : f = f\text{’}; (l(d\text{’}) \equiv \&z\ ,\ r(d\text{’}) \equiv \&y)$$
$$\vdots$$
$$d : e = *z;$$

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 ≡ y

(2) e ≡ *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以下のように定義する．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’∈D，x, y∈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．

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

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．

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

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，ϕ 関数を含む代入文，通常のプログラムにおける到達可能性の定義を SSA 形式に適用することで得られる到達可能性を定義する．

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∈D を SSA 形式に変換したものを $\mathrm{d_{ssa}}$ とする．そして，その SSA 形式の代入文の集合を $\mathrm{D_{ssa}}$ とする．

定義 7（ϕ 関数の扱い）. ϕ 関数を右辺に含む代入文 $\mathrm{x_i} = \phi(...)$; の左辺値は$\&\mathrm{x_i}$，右辺値は ϕ 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 $\mathrm{d'_{ssa}}$:$\mathrm{x_i} = \mathrm{e}$; が代入文 d_{ssa} に到達可能であるとは，$\mathrm{d'_{ssa}}$ から $\mathrm{d_{ssa}}$ に至るある（長さが正の）路が存在して，その途中に$\&\mathrm{x_j}$（j≠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，$\mathrm{d_{ssa}}$ における変数 $\mathrm{x_i}$ 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）. $\mathrm{d_{ssa}}$, $\mathrm{d'_{ssa}} \in \mathrm{D_{ssa}}$ が与えられたとき，代入文 $\mathrm{d_{ssa}}$ における変数 $\mathrm{x_i}$ のとりうる値の集合を次のように定義する．

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

なお，SSA 形式のプログラムにおいては，$\mathrm{V_{ssa}}(\mathrm{d_{ssa}},\mathrm{x_i}) \neq \emptyset$ 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，ϕ 関数を右辺に含む代入文を ϕ 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 ∈ D とすべての変数 x ∈ Var に対して，ある添字 i が存在して，$\mathrm{V}(\mathrm{d},\mathrm{x}) = \mathrm{V_{ssa}}(\mathrm{d_{ssa}}, \mathrm{x_i})$ を示すことと等しいといえる．

3. 未解決問題の証明

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 1. 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 $\overline{\mathrm{D_{ssa}}}$ とする．

補題 2. $\mathrm{d} \in \mathrm{D}$，$\mathrm{x} \in \mathrm{Var}$（x は最大多重度の変数）とし，$\mathrm{d_{ssa}} \in \overline{D}_{ssa}$，$\mathrm{x_i}$ 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，$\mathrm{V(d,x)} = \mathrm{V}_{ssa}(\mathrm{d_{ssa}}, \mathrm{x_i})$ が成立する．

証明 2. 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 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) d_i は 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．

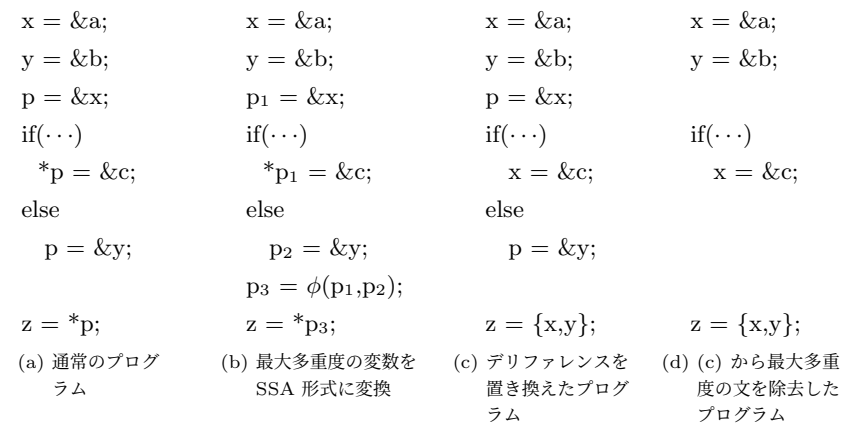
$\mathrm{d}_0 : \mathrm{z}_{j_0}^{(0)} =$ &y;
$\mathrm{d}_1 : \mathrm{z}_{j_1}^{(1)} = \mathrm{z}_{j_0}^{(0)}$; or $\mathrm{z}_{j_1}^{(1)} = \phi(\cdots, \mathrm{z}_{j_0}^{(0)}, \cdots)$;
$\vdots$
$\mathrm{d}_n : \mathrm{z}_{j_n}^{(n)} = \mathrm{z}_{j_{n-1}}^{(n-1)}$; or $\mathrm{z}_{j_n}^{(n)} = \phi(\cdots, \mathrm{z}_{j_{n-1}}^{(n-1)}, \cdots)$;
d_{ssa} :

この代入文の列において，各 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，ϕ 関数代入文 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 d_k:z = z; を挿入する．代入文 d_k:z = 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 d_i（$0 \leq i \leq n$）の右辺値として&y を含む．すなわち&y $\in \mathrm{V(d,x)}$ が成立する．

(ii) V（d,x）$\subseteq \mathrm{V}_{ssa}$（$\mathrm{d}_{ssa}$, 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 $\mathrm{d'}_{ssa}$ について，$\mathrm{d'}_{ssa}$ と d_{ssa} の間には複数個の $\mathrm{x}_j = \phi(\cdots)$; の形の ϕ 関数代入文が存在する．これらの ϕ 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 $\in \mathrm{V}_{ssa}$（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 5** に示す．P を図 5 (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 5 (c) から p の定義文を取り除いたプログラム図 5 (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．



(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 5　デリファレンスの置き換え
Fig. 5　Replacement of each pointer dereference with its values.

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 $\mathrm{d}_0 \in$D に対して，d_0 が d∈D に到達可能であるならば，P' の対応する代入文 d_0'∈D' は P' において d'∈D' に到達可能である．逆もまた成立する．

証明 3. 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 ⇒ 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 ∈ D，x ∈Var（x：最大多重度 −1 の変数），d' ∈ D' であるとする．このとき，$V(d,x) = V'(d',x)$ である．

証明 4. 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 ∈ 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 ∈ 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 ≡ x または e ≡ $*q^{(0)}$ の場合と e' ≡ &y または e' ≡ $p^{(1)}$ または e' ≡ $*q^{(1)}$ の場合がある．

(a) size = 1，すなわち e ≡ x かつ e' ≡ &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 ∈ 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 ≡ $*q^{(0)}$ かつ e' ≡ &y，すなわち d_0: $*q^{(0)}$ = &y; のとき，&x ∈ $V(d_0, q^{(0)})$ が成立する．帰納法の仮定より，&x ∈ 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 ∈ V'(d', x) が成り立つ．

(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．帰納法の仮定より，&y ∈ 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 ∈ V'(d', x) である．

(3) e ≡ x かつ e' ≡ $*q^{(1)}$，すなわち d_0: x = $*q^{(1)}$; のとき，&z ∈ $V(d_0, q^{(1)})$ かつ&y ∈ $V(d_0, z)$ が成立する．帰納法の仮定より，&z ∈ V'(d_0', $q^{(1)}$)，&y ∈ 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 ∈ V'(d', x) が成立する．

(4) e ≡ $*q^{(0)}$ かつ e' ≡ $p^{(1)}$，すなわち d_0: $*q^{(0)}$ = $p^{(1)}$; のとき，&x ∈ $V(d_0, q^{(0)})$ かつ&y ∈ $V(d_0, p^{(1)})$ が成立する．帰納法の仮定より，&x ∈ V'(d_0', $q^{(0)}$)，&y ∈ 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 ∈ 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 ∈ 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 $l(d_0') \equiv$ &x，$r(d_0') \equiv$ &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

```
for（n = max ; n > 0 ; n−−）{
   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
}
```

図 6　提案手法
Fig. 6　An overview of our algorithm.

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 ∈ D，x ∈ Var（x：最大多重度 −1 以下の変数），d'∈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 5. 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いえる．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**図 6** に示す．ここで max とはプログラムの最大多重度を表す．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．

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図 6 のアルゴリズムの結果は等しい．

証明 6. 最大多重度 n に関する帰納法で証明を行う．また**図 7** に証明の概略を示す．

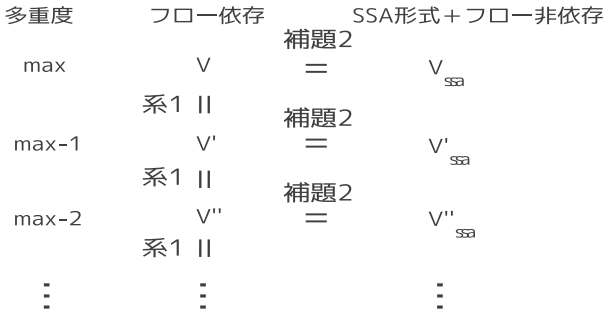


図 7　定理 1 の証明の概略
Fig. 7　An outline of the proof of Theorem 1.

(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, 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次節で述べる．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限定的なプログラムのクラスに対して解決した．

4. 提 案 手 法

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．提案手法の実行例を**図 8** に，得られるポインタ情報を**表 2** に示す．

図 8 (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 8 (b)）．次に図 8 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 → p，t_2 → q を用いて`*t`$_1$ を指し先 `p` で置き換える（図 8 (c)）．これで最大多重度に関す

	(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 = 2;			$p_3 = \phi(p_2,p_1)$;	$p_3 = \phi(p_2,p_1)$;
(4)	t = &q;	*p = 2;	*p = 2;	*p_3 = 2;	{a,b} = 2;
		t_2 = &q;	t_2 = &q;	t_2 = &q;	t_2 = &q;
(5)					

図 8　提案手法の実行例
Fig. 8　An example of pointer analysis using the algorithm of Fig. 6.

表 2　図 8 のプログラムに対する各ポインタ解析の結果
Table 2　The results of flow-sensitive and flow-insensitive pointer analyses using SSA form for the program of Fig. 8.

	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)　$p \to a$	
	(2)　$p \to a$, $q \to b$	$t_1 \to p$, $t_2 \to q$
	(3)　$p \to a$, $q \to b$, $t \to p$	$p_1 \to a$, $p_2 \to b$, $p_3 \to \{a, b\}$
	(4)　$p \to \{a, b\}$, $q \to b$, $t \to p$	$q_1 \to b$
	(5)　$p \to \{a, b\}$, $q \to b$, $t \to q$	

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 8 (d)，図 8 (e)）．図 8 (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

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．すなわち，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．すなわち，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 と順に考えることで同等性がいえる．

Hasti らのアルゴリズム[6] と本研究の提案手法（図 6）を効率について比較する．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 ϕ 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 ϕ 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．

Hasti らの手法は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．ただし，この極大なポインタ変数は同じ変数でも定義位置によって区別されるものとする．

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
- 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 A.2 参照）．

4.2　C 言語への適用

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，

⋆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{          struct S s, t, *p;    struct S s, t, *p;
 int i;            s = t;                int s_i, s_j, t_i, t_j, *p_i, *p_j;
 int j;            p = &s;               s_i = t_i;
};                                       s_j = t_j;
                                         p_i = &s_i;
                                         p_j = &s_j;
```

(a) 構造体　(b) プログラム　(c) 変換例

図 9　構造体型変数の基本型の変数への置き換え例
Fig. 9　An example of how to replace struct type variables with elementary type variables.

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．つまり，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 9** 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が詳細は割愛する．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近似する．

関数呼び出しを考慮しても，フロー依存ポインタ解析手法と SSA 形式を用いたフロー非依存ポインタ解析手法が同等の解析能力を持つことを確認している．

5. おわりに

本稿では Hasti ら[6] 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のクラスにおいて解決した．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．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た．なお構文規則を関数呼び出し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を確認している．このことから，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．この拡張した結果については，機会を改めて報告する予定である．

参 考 文 献

1) Andersen, L.O.: Program Analysis and Specialization for the C Programming Language, Ph.D. Thesis, University of Copenhagen (1994).
2) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
3) Hardekopf, B. and Lin, C.: Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis, *Static analysis: 14th international Symposium, SAS 2007*, pp.265–280 (2007).
4) Hardekopf, B. and Lin, C.: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, *Proc. 2007 ACM SIGPLAN Conference on Programming language design and implementation*, pp.290–299, ACM (2007).
5) Hardekopf, B. and Lin, C.: Semi-Sparse Flow-Sensitive Pointer Analysis, *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.226–238 (2009).
6) Hasti, R. and Horwitz, S.: Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp.97–105 (1998).
7) Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet?, *Proc. 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp.54–61 (2001).
8) Shapiro, M. and Horwitz, S.: Fast and Accurate Flow-Insensitive Points-To Analysis, *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.1–14 (1997).
9) Steensgaard, B.: Points-to Analysis in Almost Linear Time, *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.32–41 (1996).
10) 田中友幸：プログラムの参照先解析に関する研究，修士論文，三重大学大学院工学研

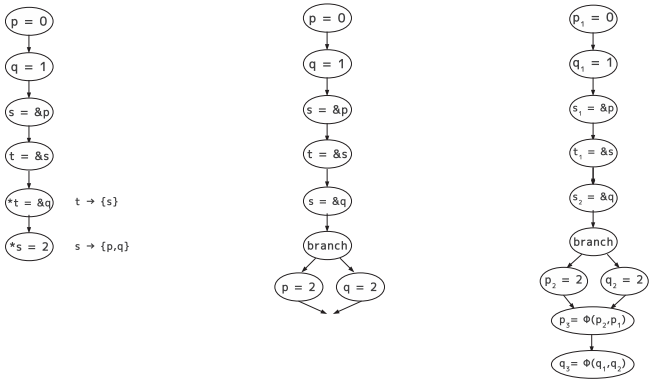
究科 (2010).
11) 田中友幸，大山口通夫：文脈を考慮した Java プログラムの参照先解析に関する一考察，数理解析研究所講究録，Vol.1691, pp.181–187 (2010).

付　　録

A.1　Hasti らの手法[6] の実行例

図 10，**図 11** に図 10 (a) のプログラムに対する Hasti らの手法の実行例を示す．

まず図 10 (a) に対してフロー非依存ポインタ解析を行い，結果として t → {s}，s → {p, q} を得る．この結果をデリファレンス（*t,*s）に付与する．この指し先の情報を用いて図 10 (b) の中間形式に変換する．このとき 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変わっている．図 10 (c) は図 10 (b) を SSA 形式に変換したものである．そして図 10 (c) に対してフロー非依存ポインタ解析を行う．それにより t_1 → {s}，s_1 → {p}，s_2 → {q} という結果を得る．この結果を用いて図 10 (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 $*t_1$ となっていたため，ポインタ情報 t_1 → {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 $*s_2$ となっているため，s_2 → {q} を用いる．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 11）．図 11 (c) に対してフロー非依存ポインタ解析を行うと先ほどと同様の結果が得られ，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．

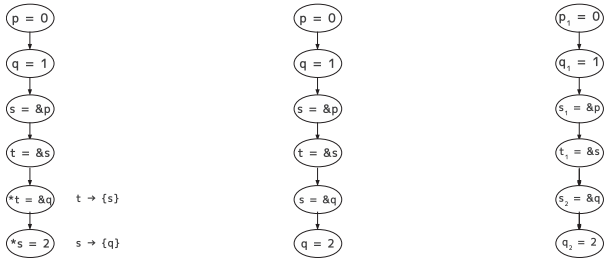


(a) CFG（注釈付き）　(b) 中間形式　(c) SSA 形式

図 10　Hasti ら[6] の実行例（1 回目）
Fig. 10　An example of pointer analysis using Hasti-Horwitz's algorithm (first iteration).

A.2　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**図 12** 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良く計算できる方法の 1 つである．図 12 の手法では，



(a) CFG（注釈付き）　(b) 中間形式　(c) SSA 形式

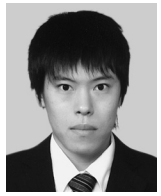
図 11　Hasti ら[6] の実行例（2 回目）
Fig. 11　An example of pointer analysis using Hasti-Horwitz's algorithm (second iteration).

for（n = max ; n > i ; n−−）{
　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
}
if（i > 0){
　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
}

図 12　Hasti らのインクリメンタルな手法に対応する提案手法
Fig. 12　An incremental algorithm using our technique corresponding to Hasti-Horwitz's incremental algorithm.

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が行われる．この点が大きな違いである．

(平成 22 年 9 月 28 日受付)
(平成 22 年 12 月 30 日採録)



田中　雄一（学生会員）

昭和 62 年生．平成 21 年三重大学工学部情報工学科卒業．同年同大学大学院工学研究科博士前期課程入学．ポインタ解析に関する研究に従事．

大山口通夫（正会員）

昭和 22 年生．昭和 52 年東北大学大学院博士課程修了．同年名古屋大学助手．昭和 53 年より三重大学工学部助教授．平成 2 年より同大学工学部教授．理論計算機科学，特に項書き換えシステム，オートマトン・形式言語理論，言語処理系，プログラム意味論，アルゴリズム等に関する研究に従事．工学博士．