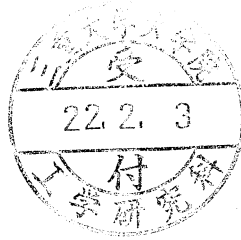


修 士 論 文

プログラムの参照先解析に関する研究



平成21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 友幸

目次

はじめに	1
第 1 章 参照先解析とポインタ解析	2
1.1 フロー依存性 (Flow-sensitivity)	3
1.2 文脈依存性 (Context-sensitivity)	3
第 2 章 従来手法	4
2.1 Andersen-style の解析	4
2.1.1 Java プログラムに対する Andersen-style の解析	5
2.1.2 Andersen-style の解析の計算量	5
2.2 オブジェクト依存解析	6
2.2.1 オブジェクト依存解析の計算量	6
2.3 Light Context-Sensitive 解析 (LightSens)	6
2.3.1 オブジェクトグラフ	7
2.3.2 オブジェクトグラフの生成	7
2.3.3 インターセクション	8
2.3.4 LightSens の例	9
2.3.5 LightSens の計算量	10
第 3 章 LightSens の拡張	11
3.1 ローカル変数の複製について	11
3.2 インターセクションについて	11
3.3 拡張処理アルゴリズム	11
3.4 拡張手法の例	12
3.5 拡張手法の計算量	13
3.5.1 処理対象を限定した場合	13
3.6 考察と今後の展望	13
3.6.1 本手法と SObjectSens の正確さの比較	14
3.6.2 計算量の削減について	14
3.6.3 正確さの向上について	14
第 4 章 SSA 形式と Java プログラムの参照先解析	15
4.1 SSA 形式の概要	15

4.2 SSA 形式を利用した Java プログラムの参照先解析	15
4.2.1 フィールドに対する処理	17
第 5 章 C プログラムの Andersen の解析アルゴリズムの改良	18
5.1 従来手法について	18
5.2 提案手法	18
5.2.1 入力プログラム	19
5.2.2 アルゴリズム	19
5.2.3 計算量について	20
おわりに	21
謝辞	22
参考文献	23

はじめに

現在ソフトウェアはあらゆる業界で利用され、その利用域は今も拡大し続けている。そのためソフトウェアの実行効率や信頼性、安全性の向上が常に求められ、それらを実現するためにプログラム解析に関する研究が行われている。参照先解析 (Points-to Analysis)、ポインタ解析 (pointer Analysis) はその根幹を担い、得られる情報は実行効率を高めるためのメソッド呼び出しの解決、セキュリティ問題の原因となるバッファオーバーランの検出などに幅広く利用され、結果的にプログラムの最適化や信頼性、安全性向上に役立っている。そのため解析コストや得られる情報の正確さが持つ影響力は無視できるものではなく、より低コストでより正確な解析手法を求められ、これまで様々な手法が提案され盛んに研究されている。

参照先解析はプログラム中で使用される参照型の変数の取り得る値を求める静的解析ことを、ポインタ解析はポインタが指す内容、つまりポインタ変数の取り得る値を計算する静的解析を指し、両者を同様の解析として扱う。

本研究では 3 つの新しい解析手法を提案する。

1 つ目は 2007 年 Milanova によって提案された、Java プログラム¹を対象とする解析手法 [Mil07] を拡張した手法である。従来手法の結果では不正確な値を含んでいたローカル変数に対して、属するオブジェクトごとに細分化したうえで、そのオブジェクトのアクセス可能な値に絞り込むことで、不正確な値を減少させることに成功した。またこの手法では計算量のオーダーが従来に比べ増加しない。

2 つ目は最適化などでよく利用される SSA 形式を Java プログラム¹に適用することで、参照先解析の正確さを高めた手法である。SSA 形式とは静的単一代入 (Static Single Assignment) 形式の略で変数に代入される値を 1 つに限定する。プログラムの流れを考慮しない参照先解析に利用することで、流れを考慮した結果に近い正確さを得ることができる。なかでもフィールドアクセスに対する SSA 変換は単純には不可能だが、参照先解析の結果を利用しフィールドを一意に区別することで SSA 変換が可能になり、結果的に解析の正確さが向上した。

3 つ目は C プログラムのポインタ解析において、従来手法である Andersen の解析 [And94] を行う、推移閉包に基づいたアルゴリズムを提案する。この手法は従来手法に比べ実行効率を高めることができる。

本稿において Java プログラムを対象とする場合、変数は主に参照型のフィールドやローカル変数のことを指す表現とする。フィールドとはクラス定義の内部で定義された変数のことを指し、スタティックフィールドとインスタンスフィールドを含む。またローカル変数はメソッド内 (またはメソッド内のブロック内) で宣言された変数のことを指し、仮引数を含めた表現とする。

¹今回は Java プログラムについて述べているが、オブジェクト指向型のプログラムであれば同等の解析が可能である。

第1章 参照先解析とポインタ解析

参照先解析とはコンピュータ上で実行可能なプログラムを受け取り、そのなかで使用される参照型の変数の取り得る値 (オブジェクト) を計算する解析である。そしてその情報は、変数とオブジェクトを表現する頂点と、それらの参照関係を表現する辺によって構成される参照先グラフで表現する。

ポインタ解析はポインタ変数の取り得る値を計算する。ポインタが参照する値は変数のアドレス、もしくはヒープ領域のアドレスである。

Java プログラムを対象とする場合 解析で扱うオブジェクトは生成文 (new 文) ごとに名前をつけて表現¹する。ローカル変数は、それを一意に区別するため、変数名とそれが属するメソッド名、クラス名、パッケージ名と組み合わせて表現する。フィールドに関しては次の3つの表現方法がある。1つ目はフィールド名とそれが属するオブジェクト名の組で表現する方法で、この方法が最も実行時の状態に近い。2つ目はフィールド名のみで表現し、どのオブジェクトに属しているかは考慮されない。3つ目はCプログラムのポインタ解析でよく使われる方法で、フィールド名を無視して属するオブジェクトで区別される。つまり、特定のオブジェクトに属する全てのフィールドを同じものとして扱う。正確さの観点からJavaプログラムに対しては1つ目の表現を使用した解析手法が多く、本稿で紹介する解析手法もすべてこの方法を使用している。

C プログラムを対象とする場合 解析で求められる参照先は、変数のアドレス、もしくはヒープ領域のアドレスであり、変数名や領域確保ごとに一意に名前を付けて表現¹する。ローカル変数は関数名と組合せ一意に区別する。

この解析はプログラムの実行前に行うので、完全な解を求めるのは不可能である。また、数万から数十万行を越えるような大規模なプログラムを対象とする場合が多く、その場合でも現実的な時間で解析を可能にする手法が求められ、様々な近似的手法が研究されてきた。その近似では参照する可能性があるものは計算結果に含めるため、実際には参照しない誤った参照先が解析結果に現れる。その数が少ないほど正確な解析が行われたと言える。従来手法は次の2つの性質、フロー依存性 (Flow-sensitivity) と文脈依存性 (Context-sensitivity) により大きく4つに分類できる。

¹文がループ内に存在している場合は1つの参照先と考える。

1.1 フロー依存性 (Flow-sensitivity)

フローつまりプログラムの流れを考慮するかどうか問われる。流れを考慮するフロー依存解析はプログラムの文 (処理) ごとに変数の参照先を計算する²。一方、流れを考慮しないフロー非依存解析は、すべての文を計算した結果をまとめたものが解となる。フロー依存解析から得られる情報は正確だが、計算コストが大きい。逆にフロー非依存解析の計算コストは小さいが正確さは低下する。

計算量やメモリ使用量の観点から、従来手法の多くがフロー非依存に基づいているが、最近では計算コストを抑えながらフローを考慮する手法 [HL09] も提案されてきている。

1.2 文脈依存性 (Context-sensitivity)

手続きはその呼び出しごとにその振る舞いが異なり、ローカル変数の持つ値も変わってくる。その違いを解析に反映させようと考えられたのが文脈依存である。文脈非依存解析ではすべての呼び出しについて得られる情報をまとめるため、不正確な情報が生まれる。例えば複数の呼び出しにおいて、仮引数は複数の実引数の値を持つことになる。一方、文脈依存解析では手続きを文脈ごとに区別して考えられるため、手続き内部の引数やローカル変数の参照先の正確さが向上する。

文脈として最も単純なものは呼び出し文で、その文の位置や呼び出しごとの名前を用いることで手続きを区別する。文脈は「ある文脈 s_1 で呼び出される手続き m_1 中の文脈 s_2 で呼び出される手続き m_2 」というように入れ子構造になり、 m_2 の文脈は (s_1, s_2) と表現できる [VLSU07]。この文脈の長さは再帰呼び出しなどで無限になるため近似が必要になる。文脈依存解析では、文脈を何で表現するか、文脈の長さをどう近似するかで正確さや計算コストが変わってくる。

本稿では小さなメソッドを多く利用する Java プログラム向けのフロー非依存の文脈依存解析の手法を 2 つ紹介し、うち 1 つの手法を拡張した新しい手法を提案する。

²コンパイラなどで利用されるデータフロー解析のようなものである。

第2章 従来手法

従来手法として、フロー非依存解析の代表的な手法である Andersen-style の解析, Java プログラム向けの解析として、文脈にオブジェクトを利用した文脈依存であるオブジェクト依存解析, さらにオブジェクトのアクセス関係を利用することで、計算量を抑えながら文脈依存を実現した Light Context-Sensitive 解析を紹介する。

3 章では計算量を抑えながら Light Context-Sensitive 解析を拡張し、正確さを向上させた手法を提案する。さらに 5 章では Andersen-style の解析を求める、推移閉包に基づくアルゴリズムを提案する。

2.1 Andersen-style の解析

Andersen は自身の博士論文 [And94] の中で C プログラムのポインタ解析 (Pointer Analysis) について述べている。今までこの解析に関して、この解析の実装を含む Java プログラムの参照先解析に関する実験ツールの開発 [LH03], 実行におけるこの解析の計算量について [SC09] など数多くの研究が今まで行われ、現在でも注目されている。本稿では Andersen の解析に基づく、主に以下の 3 つの特徴をもつ解析のことを Andersen-style の解析と呼ぶ。

- フロー非依存
- 部分集合に基づく (subset-based)

プログラム上の代入文において、代入する側が参照するオブジェクト集合は代入される側が参照する集合の部分集合であると考え、つまり代入文「 $X = Y$ 」は「 $X \supseteq Y$ 」と解釈する。

対照的な手法としては等価性に基づいた (equality-based) 手法 [Ste96] があり、代入の両辺は同じものを参照するものとして扱う。そうすることで正確さは犠牲になるが、ほぼ線形時間で解が得られる。

- 文脈非依存

Andersen は論文の中で文脈非依存解析にあたる手続き内 (intra procedural) 解析について述べている。さらにその拡張として手続き間 (inter procedural) 解析、ここで言う文脈依存解析にも言及しているが、今回は前者の解析に限定して述べる。

2.1.1 Java プログラムに対する Andersen-style の解析

Java プログラムに対する Andersen-style の解析は次の 4 種類の文に対して参照先グラフに辺を追加する処理に集約できる。

文	追加する辺
(1) $s_i : l = \text{new} C$	$l \rightarrow O_i$
(2) $l = r$	$l \rightarrow O_i \in Pt(r)$
(3) $l.f = r$	$\langle O_i \in Pt(l), f \rangle \rightarrow O_i \in Pt(r)$
(4) $l = r.f$	$l \rightarrow Pt(\langle O_i \in Pt(r), f \rangle)$

(1) オブジェクト生成文において、 s_i の i は生成文に対する通し番号で、生成されるオブジェクトは O_i と名付けられる。この文では変数 l から O_i への辺を追加する。

(2) 代入文については、左辺の変数 l から、右辺の変数 r が参照するオブジェクト集合 $Pt(r)$ の要素すべてに辺を追加する。

(3) フィールドへの書き込みに対しては、 $\langle l$ が参照するオブジェクトと f との組 \rangle で表現されたフィールドから、 r が参照するオブジェクトへ辺を追加する。

(4) フィールドから読み出しでは、 l から、 $\langle r$ が参照するオブジェクトと f との組 \rangle で表現されたフィールドへの辺を追加する。

またメソッドの呼び出し文を考える場合には、対応する実引数から仮引数への代入文、返り値からメソッドが代入される変数への代入文を追加することで実現できる。

以上のように各文を処理するが、各変数の参照先は処理するごとに変化するため、各文を 1 回ずつ処理するだけでは不十分である。これを効率良く解く方法アルゴリズムについては [LH03, SC09] を参照されたい。

2.1.2 Andersen-style の解析の計算量

Andersen は自身の博士論文 [And94] の中で、解析の計算時間はプログラムサイズの多項式時間と述べるに留まっている。ここで述べるのは、より計算量が小さいと考えられている手法である。

Andersen-style の解析は、変数と参照先を頂点、代入文を有向辺と考えることでグラフの推移閉包を求める問題と考えられる¹。ただしポインタの Dereference により推移閉包から求められない新たな辺が加わることがある。その場合、その辺に対する推移閉包を再び求める必要がある。このような枠組の問題は動的推移閉包 (Dynamic Transitive Closure) 問題と呼ばれている [SC09]。動的推移閉包問題は、辺の追加だけを考慮するもの、辺の削除だけを考慮するもの、またはその両方を考慮するものに分けられる。Andersen-style の解析は辺の追加のみを考慮すれば良く、この場合の計算量は、 n を頂点数として $O(n^3)$ であることが証明されている [IK83]。

¹このとき頂点を変数のみと考え、推移閉包を求めた後に各変数に直接代入される参照先の和集合を求めることもできる。変数のみの場合、要素数がより少ないため推移閉包を求める計算量は減少する。

以上より Andersen-style の解析は $O(n^3)$ で計算できることがわかる。 n はグラフの推移閉包を考える場合の頂点数だが、Andersen-style の解析に置き換えれば変数と参照先の数の和であり、プログラムサイズに比例する。

2.2 オブジェクト依存解析

Milanova らは 2002 年にメソッドが属するオブジェクト²を文脈として扱う文脈依存解析をオブジェクト依存 (Object Sensitive) 解析として提案している [MRR02, MRR05](以下、この手法を標準的なオブジェクト依存と言う意味で SObjectSens と呼ぶ)。

メソッドを属するオブジェクトごとに区別して考えることで、文脈を考慮しない場合に生じる、オブジェクト指向型プログラム特有のカプセル化や継承による不正確さを改善することができる。

またこの手法はパラメータを与える事で、正確さや計算コストを調節できるように設計されている。パラメータとしてオブジェクトの抽象化の度合い (オブジェクトを生成文で表現するか、生成文と文脈の組で表現するか) や文脈で区別する変数を選ぶことができる。

この手法の有用性は他の論文 [LH06] でも認められていて、無駄な文脈³が比較的少なく、仮想メソッド呼出しの解決、キャスト安全性の解析などで効果的であるとされている。

2.2.1 オブジェクト依存解析の計算量

SObjectSens では解析にパラメータを与える事によって、解析の正確さと計算量を調節できる枠組みになっている。そのため計算量の考察は容易ではないが、ここでは最も単純な場合について考察する。

文脈依存解析はメソッドを文脈で区別するため、その文脈の数だけメソッドを複製した後、文脈非依存解析すると考える事ができる。SObjectSens の文脈はオブジェクトであり、最も単純な場合、オブジェクトは生成文で抽象化され、その総数は $O(n)$ である⁴。最悪の場合はプログラム中の全メソッドが、全オブジェクトに属している場合で、プログラムサイズは元のサイズの $O(n)$ 倍の $O(n^2)$ になる。

SObjectSens は文脈で区別する以外は Andersen-style の手法を踏襲しているので、サイズ $O(n^2)$ のプログラムに対して 3 乗の解析を行うことになり、結果的に $O(n^6)$ の計算量が必要となる。

2.3 Light Context-Sensitive 解析 (LightSens)

SObjectSens より計算コスト抑えた手法が Milanova によって 2007 年に提案されている [Mil07] (以下 LightSens と呼ぶ)。LightSens ではまず Andersen-style の解析を行い、その結果となる参照

²メソッド呼び出しに利用される変数の参照先であるオブジェクトが、そのメソッドの属するオブジェクトである。

³考慮して (ローカル変数を区別して) も正確さの向上がない文脈のこと。

⁴もっと詳細化するならば、ループの展開や文脈依存によって区別された文によって抽象化することができる。そうすることでより実行時に近いオブジェクトの抽象化が可能になり、解析の正確さも向上するが、オブジェクト総数は増加し、計算量にも影響を与える。

先グラフを作成すると共に、オブジェクト間のアクセス関係を近似したオブジェクトグラフを作成し、その2つのグラフのインターセクション(共通部分を取り出すこと)により正確さを高める。

オブジェクトグラフを利用することで、アクセス不可能なオブジェクトへの参照先を、参照先グラフから取り除くことができる。この手法で得られる情報の正確さは SObjectSens に劣るが、文脈を考慮しない Andersen-style の解析の正確さを改善することができ、時間計算量については文脈依存解析のなかで最も効率の良い部類に属する。

2.3.1 オブジェクトグラフ

オブジェクトグラフの頂点はオブジェクトと root である。root とはプログラムの開始地点となる main メソッドのことを表す。そして以下で述べる頂点間のアクセス関係を辺として表現する。あるオブジェクト O1 が別のオブジェクト O2 にアクセス可能とは以下の場合である。

- O1 に属するフィールドが O2 を参照している
- O1 に属するメソッドのローカル変数が O2 を参照している
- O1 に属するメソッドから呼び出しされるスタティックメソッド(またそのスタティックメソッドから呼び出されるスタティックメソッド…のように1回以上のスタティックな呼び出しで呼び出されるスタティックメソッド)のローカル変数が O2 を参照している

さらに、main メソッド、または main メソッドから1回以上スタティックに呼び出されるスタティックメソッドのローカル変数が O1 を参照している場合、root が O1 に対しアクセス可能とする。

2.3.2 オブジェクトグラフの生成

オブジェクトグラフは、Andersen-style の解析から得られる参照先グラフ、到達するメソッドの集合を利用し、入力プログラムの到達可能な(メソッドの)文に対して、以下のように辺を追加することで生成される。扱う文は 2.1.1 節で述べた4種類の文(オブジェクト生成文、代入文、フィールドの読み出し、フィールドへの書き込み)にメソッド呼び出し文($l = r.m(p)$)を加えた5種類の文に絞って述べる。return 文は、返り値が代入される特別な変数を扱うことで実現できる。

文	グラフに追加する辺
$s_i : l = \text{new } C(\dots)$	$(1) o \rightarrow o_i \text{ s.t. } o \in RC_m$
$l = r.n(\dots) \text{ s.t. } r \neq \text{this},$ $l = r.f \text{ s.t. } r \neq \text{this}$	$(2) o \rightarrow o_i \text{ s.t. } o \in RC_m \wedge o_i \in Pt(l)$
$l = \text{new } C(r),$ $l.n(r) \text{ s.t. } l \neq \text{this},$ $l.f = r \text{ s.t. } l \neq \text{this}$	$(3) o_i \rightarrow o_j \text{ s.t. } o_i \in Pt(l) \wedge o_j \in Pt(r)$
$l = \text{this},$ $\text{this}.n(\text{this}),$ $\text{this}.f = \text{this}$	$(4) o_i \rightarrow o_i \text{ s.t. } o_i \in Pt(\text{this})$

ここで RC_m はメソッド m のレシーバオブジェクト、つまり m が属しているオブジェクトの集合である。 m がインスタンスメソッドなら RC_m は $Pt(m.this)$ ⁵、つまり m の暗黙の引数が参照するオブジェクト集合に等しい。 m がスタティックメソッドならば、 RC_m は、 m から 1 回以上のスタティックな呼び出しを逆にたどって到達可能なすべてのインスタンスメソッドの属するオブジェクトの和集合に等しい。 言い替えれば、それらのインスタンスメソッドからは、 1 回以上のスタティックな呼び出しによって m に到達することができる。 もし main メソッドから 1 回以上のスタティックな呼び出しによって m に到達する場合、 RC_m は root を含む。

(1) で追加される辺はメソッドの中で新しいオブジェクトが生成されるなら、そのメソッドの属するオブジェクトは新しいオブジェクトにアクセス可能という意味を表す。

(2) の O_i は、 r が参照するオブジェクトを通してこのメソッドに入ってくるオブジェクトのことを表し、ローカル変数 l に代入されることで、 l が宣言されている (属している) メソッドの属するオブジェクトからアクセス可能になる。 $l = this.f$ の場合、そのフィールドへの代入文 (つまり $l.f = r$ 、もしくは $this.f = this$) で辺が追加される ((3)、もしくは (4) の場合に相当する) ので、この文で追加する必要はない。 $l = this.n(...)$ の場合、 n の内部で返り値となる変数への代入文 ($l = r.n(...)$ 、 $l = r.f$ 、 $l = new C(r)$ 、 $l = this$) で辺が追加されるので、ここで辺の追加する必要はない⁶。

(3) では l が参照しているオブジェクトの中へ、 r の参照しているオブジェクトが入ることでアクセス関係が生まれ、それを表現する辺を追加する。 l が $this$ の場合は、 $Pt(this)$ から $Pt(r)$ への辺が追加されることになるが、それはローカル変数 r にオブジェクトが代入されるときに追加されるので、ここで辺を追加する必要がない。

(4) は自分自身にアクセス可能と言う意味の辺を追加する。ここで $Pt(this)$ と RC_m は等しい、なぜなら $this$ を使用している段階でインスタンスメソッドだからである。

上記にない代入文 $l = r$ について、 r への代入文において辺が追加されるために不要。また、 $l.n()$ については直接的なオブジェクトの移動はない。

2.3.3 インターセクション

この手法のインターセクションはプログラム開始地点から到達可能なメソッド m 内のローカル変数 l に関する各種代入文によって以下の 2 つに場合に分かれる。

- $l = r, l = this.f, l = this.n(\dots)$ をローカル変数 l に対する内的 (internal) 代入とし、この場合 $NewPt(l) = Pt(l) \cap Ag(m)$
- $l = r.f, l = r.n(\dots)$ をローカル変数 l に対する外的 (external) 代入とし、この場合 $NewPt(l) = Pt(l) \cap Ag(m) \cap Ag(r)$

ここで $Pt(l)$ は Andersen-style の解析から得られる l の参照可能なオブジェクト集合であり、 $NewPt(l)$ は LightSens で得られる l の参照先集合である。 $Ag(m)$ はメソッド m が属するオブ

⁵ Pt の値は Andersen-style の解析から得られているオブジェクト集合。

⁶ 入力プログラムとして文の種類を限定している場合。もし返り値が変数に代入されずに、オブジェクト生成文が直接返り値になる場合は (2) と同様に辺を追加する必要がある。

ジェクト集合 RC_m がアクセス可能なオブジェクト集合で $Ag(m) = \bigcup_{o \in RC_m} Ag(o)$ と計算される。 $Ag(o)$ はオブジェクト o がアクセス可能なオブジェクト集合 (オブジェクトグラフから得られる)。 $Ag(r)$ はローカル変数 r が参照するオブジェクト集合がアクセス可能なオブジェクトの集合で $Ag(r) = \bigcup_{o \in Pt(r)} Ag(o)$ と計算される。

内的に代入されるオブジェクトは、その代入文を含むメソッドがアクセス可能なオブジェクトに限定する。 外的に代入されるオブジェクトは、さらに右辺のフィールドやメソッドが属するオブジェクトに限定することで正確さを向上させることが出来る。

2.3.4 LightSens の例

LightSens が Andersen-style の解析の正確さを改善する例を挙げる。

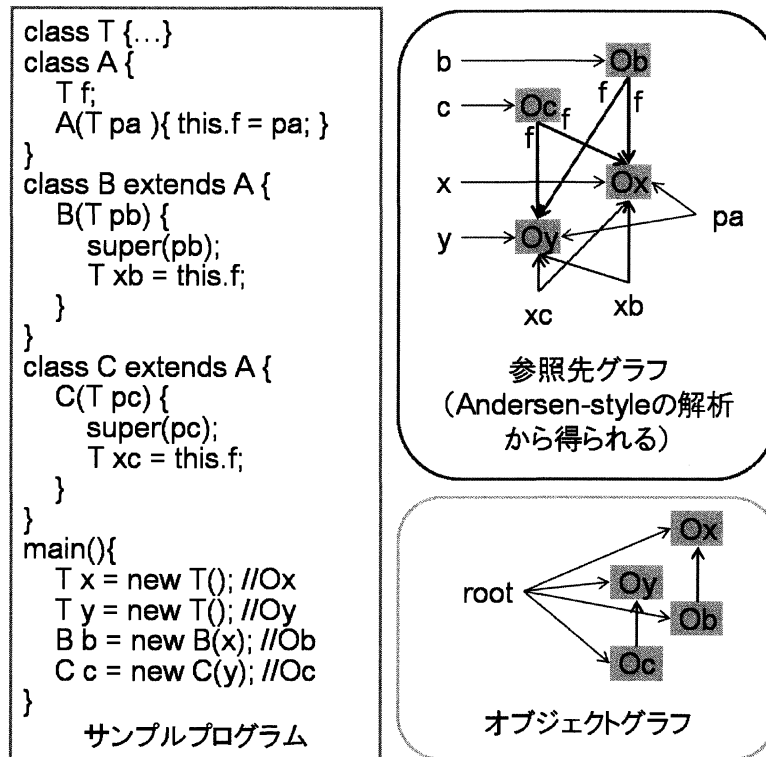


図 2.1: 参照グラフとオブジェクトグラフの例

図 2.1 はサンプルプログラムと、そのプログラムを入力として与えた場合の Andersen-style の解析から得られる参照先グラフ、LightSens で生成されるオブジェクトグラフを示している。

プログラム下部の main メソッドの中では 4 つのオブジェクト Ox , Oy , Ob , Oc が順に生成され、それぞれ変数 x , y , b , c に代入される。オブジェクト Ob , Oc が生成時には、 Ox , Oy がそれぞれコンストラクタに実引数として渡され、両方ともスーパークラス A のコンストラクタ A に渡る。このとき文脈を考慮しない Andersen-style の解析では、 A の仮引数の pa に Ox , Oy の両方を参照することになり、 Ob と Oc のフィールド f も、 Ox , Oy の両方を参照することになる。そ

して、その f によって代入されるローカル変数 xb と xc は Ox , Oy の両方を参照することになる。これら情報が図 2.1 に参照先グラフに反映されている。しかし、実際には xb は Ox , xc は Oy しか参照しない。

オブジェクトグラフの root から辺が伸びているオブジェクトは main メソッド内で参照可能なオブジェクトである。図 2.1 では Ox , Oy , Ob , Oc が main メソッドで生成されるために、root から辺でつながれている。 Ob から Ox , Oc から Oy への辺は、 Ob , Oc が生成されるときには、 Ox , Oy がそれぞれ実引数として渡されていることから作られる。

この例で改善されるのは xb , xc の参照先である。 xb は Ob , xc は Oc のみに属していて、 Ob がアクセス可能なのは Ox , Oc がアクセス可能なのは Oy であることがオブジェクトグラフから得られるためインターセクションによって改善される。式で表現すると次のようになる。 $NewPt(xb) = Pt(xb) \cap Ag(Ob) = \{Ox, Oy\} \cap \{Ox\} = \{Ox\}$, $NewPt(xc) = Pt(xc) \cap Ag(Oc) = \{Ox, Oy\} \cap \{Oy\} = \{Oy\}$ 。

2.3.5 LightSens の計算量

LightSens は Andersen-style の解析、オブジェクトグラフの生成、インターセクションを順次行うので計算量はそれらのうち最大のものに依存する。ここでプログラムサイズを n とする。そうすると解析で扱うオブジェクトの数は生成文の数⁷となるため $O(n)$ である。

Andersen-style の解析は前述の通り $O(n^3)$ である。

オブジェクトグラフは、プログラムの各文に対して辺を追加していくことによって生成される。その追加する辺の数を考える。最悪の場合、代入文においての右辺と左辺に現れる 2 つの変数が参照し得る $O(n)$ のオブジェクトの直積が辺となる場合があるので $O(n^2)$ 。これを $O(n)$ の文について繰り返すために $O(n^3)$ となる。

インターセクションも各文について処理を行う。その処理のなかで変数の参照するオブジェクト集合がアクセス可能なオブジェクト集合 $Ag(r)$ を求める必要がある。この計算は $O(n)$ のオブジェクト集合に対して $O(n)$ の演算するため $O(n^2)$ 。これを $O(n)$ の文について繰り返すために $O(n^3)$ となる。

以上により Andersen-style の解析、オブジェクトグラフの生成、インターセクション全てが高々 $O(n^3)$ で計算可能なために、LightSens 全体としても $O(n^3)$ の計算量が必要になる。

⁷文がループ内に存在している場合でも生成される参照先は 1 つと考える。

第3章 LightSensの拡張

本稿では LightSens を拡張した手法を提案する。この拡張では計算量のオーダーを増加させず、解析で得られる情報の正確さを向上させることが可能である。本手法では LightSens で生成される参照先グラフとオブジェクトグラフを利用するため、LightSens を行った後に処理を付け加える。付け加える処理はローカル変数の複製とインターセクションである。実際には複製しつつインターセクションを計算する。

3.1 ローカル変数の複製について

LightSens ではローカル変数はプログラム中に書かれた名前で区別される。しかし、それでは複数のメソッド呼び出しによって異なる値が代入される場合、解析結果として「複数の参照先を持つ可能性がある」としか言えない。これが参照先解析における不正確さとなる。そこで文脈依存解析では、さらにローカル変数を文脈によって細分化する事で正確さを向上させる。特に SObjectSens ではローカル変数をその属する (メソッドが属する) オブジェクトで区別する。その考えを本拡張にも取り入れる。SObjectSens のような従来の文脈依存解析ではローカル変数を文脈に区別したのち代入される参照値の伝播を計算していたが、本拡張ではローカル変数を区別したのちにインターセクションを行う。その違いが計算量に影響を与える。

3.2 インターセクションについて

LightSens でもインターセクションを行うが、内容は少し異なる。LightSens では代入文の種類によって処理を分けていたが、今回は文を参照する必要はなく、変数とその変数が属するオブジェクト、そのオブジェクトがアクセス可能なオブジェクト集合が計算できれば良い。

3.3 拡張処理アルゴリズム

図 3.1 は LightSens に付け加える処理のアルゴリズムである。1 行目の Reachable Methods とはプログラム開始地点から到達可能なメソッドの集合¹である。2 行目では各メソッドの属するオブジェクト集合 RC_m の数が 1 より大きいかどうかを調べている。1 以下であればローカル変数を複製しても参照先を限定できないので、それ以降の処理は不要になる。

¹Andersen-style の解析で得られる。

$Pt(v)$ は既に行っている LightSens から得られているローカル変数 v の参照可能なオブジェクト集合であり, $NewPt(v^o)$ はこの拡張手法で新しく得られるオブジェクト集合で, オブジェクト o に属するローカル変数 v の参照先である.

```

for (each Method  $m \in$  Reachable Methods)
  if ( $|RC_m| > 1$ )
    for (each Local Variable  $v$  in  $m$ )
      for (each  $o \in RC_m$ )
         $NewPt(v^o) = Pt(v) \cap Ag(o)$ ;

```

図 3.1: 拡張処理アルゴリズム

以上では複数のオブジェクトに属している全ローカル変数について, 複製とインターセクションを行っているが, インターセクションにより結果が向上されないものについては参照先を複製する必要はなく, 「向上しない」つまり「もとの参照先 (Pt) と同じ」ということを記録しておけばメモリの消費量は抑えられる.

またもとの参照先 (Pt) の数や属するオブジェクト (RC_m) の数が多い場合は, 複製とインターセクションにより有用な結果が得られる可能性は低いと考えられる. そのため複製とインターセクションを行う条件に「 $Pt(v)$, RC_m がそれぞれ一定数以下」を加えれば, 計算量やメモリ消費量を節約できる.

3.4 拡張手法の例

拡張手法が LightSens の正確さを改善する例を挙げる.

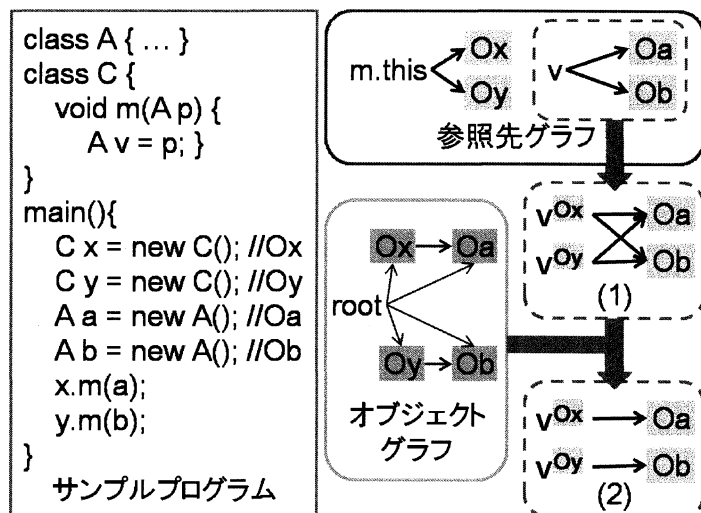


図 3.2: 拡張手法の改善例

図 3.2 はサンプルプログラムと、そのプログラムを入力として与えた場合 LightSens から得られる参照先グラフ、オブジェクトグラフ、さらに参照先グラフの一部を複製したグラフ (1)、(1) に対してインターセクションを行い誤った参照先が取り除かれたグラフ (2) を示している。

LightSens から得られる参照先グラフとオブジェクトグラフは 2.3.4 節の例と同様に生成される。参照先グラフ中の $m.this$ はメソッド m の暗黙のパラメータでメソッド m が属するオブジェクトを参照する。

参照先グラフの点線で囲まれた部分は「変数 v が O_a , O_b の両方を参照する可能性がある」ことを示しているが、LightSens ではこれ以上正確な情報を得られない。しかし、この変数 v を O_x に属する v^{O_x} と、 O_y に属する v^{O_y} に複製する (図 3.2 の (1)) ことで、「 O_x が O_a , O_y が O_b のみアクセス可能」という情報を持つオブジェクトグラフとインターセクションをとることでより正確な参照先が得られる (図 3.2 の (2))。

3.5 拡張手法の計算量

今回提案した手法は先述のように LightSens を行った後にローカル変数を属するオブジェクトごと複製してインターセクションを行う。ここでプログラムサイズを n とする。そうすると解析で扱うオブジェクトの数は生成文の数²となるため $O(n)$ である。

インターセクションは $O(n)$ の要素を持つオブジェクト集合に対する集合演算なので 1 回につき $O(n)$ の計算量が必要。本手法ではインターセクションはローカル変数の数だけ行う。ここでローカル変数の数はもともと $O(n)$ だが、複製することによって $O(n^2)$ に増加する。よって結果的に $O(n) \times O(n^2) = O(n^3)$ の計算量が必要になる。

LightSens は前述の通り $O(n^3)$ 、追加する処理も $O(n^3)$ なので拡張しても同じ $O(n^3)$ に抑えられる。

3.5.1 処理対象を限定した場合

以上では複数のオブジェクトに属している全ローカル変数について、複製とインターセクションを行った場合の計算量について述べたが、処理対象となるローカル変数を限定してもよい。LightSens ですでに得られている参照先の数、属するオブジェクトの数がそれぞれ一定数以下のローカル変数に対してのみ処理を行うのであれば、インターセクションの対象となる要素の数と変数の複製数が定数になり、 $O(c_1) \times O(n \times c_2) = O(c_1 \times c_2 \times n) = O(n)$ となる。(c_1 はインターセクション前の参照数の上限、 c_2 は属するオブジェクト数の上限、つまり複製する数。)

3.6 考察と今後の展望

本拡張により、文脈を考慮しない Andersen-style の解析を計算量のオーダーを増加させずに、文脈を考慮する SObjectSens の正確さに近づけることができた。本手法が SObjectSens 比で計算量

²文がループ内に存在している場合でも生成される参照先は 1 つと考える。

が大幅に小さいのは、SObjectSens が文脈を考慮したのちに各変数の値を伝播させる必要があったことにある。本手法では文脈を考慮 (変数を複製) した後に、伝播させることなく変数ごとに独立して計算 (インターセクション) を行っている。また、独立して計算するということで、特定の変数のみの正確さを向上させることが可能ともいえる。

3.6.1 本手法と SObjectSens の正確さの比較

本手法は解析の正確さにおいて SObjectSens に劣っている。それはオブジェクトグラフの正確さの議論も必要だが、それを差し置いたとしても、SObjectSens が、変数だけでなくオブジェクトに対しても、文脈を考慮して細分化³できることから言える。(もちろん、その分だけ計算量は増加するのだが。) 本手法ではオブジェクトを生成文ごとに区別して扱う Andersen-style の解析、LightSens の結果を利用するために、オブジェクトを細分化するならば少なくともオブジェクトグラフの再生成が必要になり、計算量の増加を防ぐの困難だと考えられる。

3.6.2 計算量の削減について

入力プログラムがある特定の条件に当てはまる場合、Andersen-style の解析の計算量は $O(n^2)$ であるという研究 [SC09] がある。その条件に基づいた場合、本手法も計算量が $O(n^2)$ と言えるかもしれない。たとえ現状の手法では言えなくても、改良を加えることで言える可能性は十分ある。

3.6.3 正確さの向上について

本手法でインターセクションを行う際に使用しているオブジェクトグラフは、Andersen-style の解析結果に基づいて生成されている。LightSens でもそうだが、本手法でも解析結果を向上させているので、その結果を用いてオブジェクトグラフを再生成した後、インターセクションを行うことで、さらに正確さを向上できる可能性がある。うまく差分を計算することで、解析全体のオーダーを増加させずに、オブジェクトグラフを再生成できると考えられる。

³もっと細かく近似すること、より実行時の状態に近づく。

第4章 SSA形式とJavaプログラムの参照先解析

この節では SSA 形式について、またそれを Java プログラムの参照先解析に利用する手法について述べる。

4.1 SSA形式の概要

SSA 形式とは静的単一代入 (Static Single Assignment) 形式の略で、各変数への代入を 1 回だけに限定したプログラムの表現形式のことで、数多くの最適化に有効な形式である。

すべてのプログラムは図 4.1 のように変数の代入が 1 回になるように名前の付け替えを行うことで SSA 形式に変換できる。分岐のあるプログラムを SSA 変換する際は、SSA 形式の整合性を保証するため、図 4.2 のように ϕ 関数を挿入する。 ϕ 関数は引数のどれかを返すという意味である。SSA 変換における ϕ 関数の追加に関する詳細は他の文献 [中田 99] に譲る。

$x = 1;$ $x_1 = 1;$
 $y = 2;$ SSA 変換 $y_1 = 2;$
 $y = x;$ $y_2 = x_1;$

図 4.1: SSA 変換の例

if (...)
 $x_1 = ...;$
else
 $x_2 = ...;$
 $x_3 = \phi(x_1, x_2);$

図 4.2: ϕ 関数の挿入例

参照先解析に関して、SSA 変換後のプログラムにフロー非依存解析を行って得られた情報は、変換前のプログラムにフロー依存解析を行って得られる情報に近い正確さが得られると考えられる。

4.2 SSA形式を利用したJavaプログラムの参照先解析

C プログラムにおけるポインタ解析では SSA 形式を用いた解析手法 [HH98] が提案されている。本稿で提案する手法はその手法とは異なり、Java プログラムの参照先解析に適するように SSA 形式を利用した手法である。簡単化のため、入力プログラムが許す文は 2.1.1 節で述べた 4 種類の文 (オブジェクト生成文、代入文、フィールドの読み出し、フィールドへの書き込み) に加えてメソッド呼び出し文 ($l = r.m(p)$) の合計 5 種類の文に絞って述べる。

提案手法の手順を次に示す。

1. ローカル変数に対してのみ SSA 変換を行う。

2. 参照先解析を行う (各変数の参照先が得られる).
3. フィールド参照に使用される変数を参照するオブジェクトに置き換え, オブジェクトとフィールドの組で 1 つの名前として名前の付け替えを行う.
4. 参照先解析を行う (フィールドアクセスに関する代入について正確さが向上する).
5. 4 の結果により正確さが向上する可能性があるため, さらに 3, 4 を繰り返す. これを正確さが向上しなくなるまで繰り返す. (または一定回数繰り返す.)

SSA 変換は変数の定義 (値の代入) の実行順に基づいて行われる. ローカル変数は, その変数に対する代入文のみによって値が書き換えられるため, そのまま SSA 変換できる. しかしフィールドに関しては, フィールドアクセス式からフィールドが一意に判別できないこと, さらに異なるメソッド内でアクセス可能なため (メソッドの副作用によって値が書き換えられる可能性があり) 単純に SSA 変換できない.

今回の解析において, フィールドは属するオブジェクトごとに参照先を計算する. フィールドアクセス式は, ローカル変数とフィールドの組で表現されるが, そのローカル変数の参照先が求まっていなければ, フィールドを一意に判別できない.

例えば, $a.f = x$, $b.f = y$, $z = a.f$ という文の並びがあった場合, 一見 z には x の値が代入されるように思えるが, それは a と b が異なるオブジェクトを参照している場合に成立する. もし a と b が同じオブジェクトを参照していた場合, z には y の値が代入されることになる. また a , b はそれぞれ複数のオブジェクトを参照する可能性があるため, z には x , y のどちらかが代入されるとしか言えない場合もある. この処理の詳細は 4.2.1 節で述べる.

以上の理由から, まずローカル変数に対して SSA 変換を適用した後, フローを考慮しない参照先解析を行う. この時点で既に, フロー非依存解析から得られる情報より正確な結果が得られる. さらにこの結果を用いれば, フィールドに対しても名前の付け替え (SSA 変換) が可能になり, その後で参照先解析を行うことで, さらに解析結果の正確さを高めることが可能になる. この時点で解析を終了してもよい. ただし, より正確な結果が得られれば, より正確なフィールド処理 (SSA 変換) が可能であり, さらに正確な参照先が得られる場合がある. そのため, 上で述べたアルゴリズムでは, フィールド処理と参照先解析を繰り返している. 参照先解析の結果が更新される限り繰り返す価値はあるが, 繰り返す度に更新される正確さ (繰り返す価値) は減少していくと考えられる. そういう意味で, 繰り返す時間と正確さのトレードオフを考慮して, 繰り返しをある一定数に制限することも現実的である.

本手法で利用する参照先解析は, フロー非依存である以外は特に指定しないが, SSA 変換による変数の増加によるメモリ消費量, フィールド処理における計算量を考慮すると, 高コストな解析は現実的ではない. ただし, 同じ参照先を持つ変数をまとめて表現する, 同じ処理の繰り返しを省略するなど, 様々な工夫が考えられる.

4.2.1 フィールドに対する処理

フィールドを一意に区別するにはローカル変数名とフィールド名の組では不十分であることは上で述べた。そこでフィールドアクセスに使用されるローカル変数を、その参照するオブジェクトで置き換えた、オブジェクト名とフィールド名の組ならば一意に区別できる。その後、オブジェクト名とフィールド名の組への代入文ごとに名前の付け替えを行う。

プログラム上に $x.f = y_k$ というフィールドへの代入文¹があった場合、 x の参照先が必要になる。 x が $O1$ を参照する場合は $O1.f_i = y_k$ と置き換えることが可能。ここで i は $O1.f$ に対する代入回数に相当する。また x が $O1, O2$ という2つのオブジェクトを参照する可能性があるならば、条件分岐を扱うのと同様に次の4つの文に書き換える。 $O1.f_i = y_k, O2.f_j = y_k, O1.f_{i+1} = \phi(O1.f_{i-1}, O1.f_i), O2.f_{j+1} = \phi(O2.f_{j-1}, O2.f_j)$ 。前者の2文はそれぞれが if 文と else 文の中身のように、どちらかが必ず実行される。ここで後者の2文で ϕ 関数は、前者の2文による2通りの実行結果に対応するために付け加える。 x の参照先が2つ以上ならばその数だけ代入文と ϕ 関数が必要になる。

プログラム上に $y_i = x.f$ というフィールド値の代入文¹があった場合、 x が $O1, O2$ という2つのオブジェクトを参照する可能性があるならば、次の3つの文に書き換えられる。 $y_i = O1.f_j, y_{i+1} = O2.f_k, y_{i+2} = \phi(y_i, y_i - 1)$

さらにメソッド呼び出しの副作用に対応するために、各フィールドに対してメソッドの開始地点とメソッド呼び出しの後に $Oi.f_j = Oi.f$ 、メソッドの終了地点と呼び出しの前に $Oi.f = Oi.f_j$ を挿入する必要がある。この添字なしのフィールドは、SSA 変換により同じ名前のフィールドに添字を付けた複数のフィールドから値が代入され、メソッド呼び出しや呼び出しから戻る際の橋渡し役となる。この文の存在により厳密には SSA 形式と呼べないかもしれないが、解析の正当性を保つためには必要になる。この文の追加は各メソッド内でアクセスされるフィールドに限定できる。

¹ローカル変数の SSA 変換後のプログラムを想定しているが、 x の添字は省略している。

第5章 CプログラムのAndersenの解析アルゴリズムの改良

この章では、入力となるCプログラムに対して少し制限があるものの、Andersenの解析と同等の解析を行う手法として、推移閉包の計算に基づいたアルゴリズムを提案する。

5.1 従来手法について

Andersenの解析の概要は2.1節で述べたように代入文「 $X = Y$ 」を「 $X \supseteq Y$ 」と解釈し、右辺の値を左辺の値に含めるように計算していく。Andersenの論文[And94]におけるアルゴリズムは、プログラム中の各文に対して集合演算を単純に繰り返すものだったが、推移閉包の考え方を採り入れればより効率的に計算できる。Andersenの解析の計算量について述べた論文[SC09]では動的推移閉包の問題であると述べている。この場合推移閉包を一度求めたのちにDereferenceに起因する辺の追加による処理が必要になる。

例えば、 $x = *p$ という文あった場合、 p の参照先が必要になる。そこで一度推移閉包を求める。そこから得られた参照先について x の参照先だけでなく、 x に依存する変数の参照先の計算が必要になる。この推移閉包計算後、辺の追加に必要な計算量は高々 $O(n^3)$ である[IK83]。

5.2 提案手法

今回提案する手法の鍵となる考え方は、解析をポインタの多重度¹ごとに分割して計算することである。そうすることで全体の推移閉包を計算する問題を、それぞれ排反である部分ごとに推移閉包を計算する問題になる。また、従来手法で処理が必要とされていた辺の追加処理は、多重度の高いものから順に推移閉包を求め、その結果をより低い多重度のポインタ変数に対する推移閉包を計算する前に行うことで、追加は定数時間の処理ですむ。ここで最大多重のポインタ変数は参照されないで、辺の追加がないことは明らかである。

例えば文の中に $*p$ がある場合、 p はそれより高い多重度を持つため参照先が既に求められていて、 p が x を参照しているなら x で置き換えればいいのである。そうすることで単に推移閉包を求めるだけで解が得られる。これを繰り返すことで、従来の推移閉包を求める手法に基づく計算量に依存して解析を行うことができる。

¹ポインタの多重度とは宣言時の「*」の数のことを指す。例えば「`int **p;`」という宣言があった場合、 p の多重度は2となる。

5.2.1 入力プログラム

この手法はポインタの多重度ごとに行うため、代入文において両辺の型 (多重度) が等しいプログラムしか正しく解析できない。またポインタの演算がある場合には、演算結果が別の参照先を表すがどうか判定できないために正しく解析ができない。実際は同じものを参照している場合でも、異なった参照先と扱ってしまう危険性がある。そこで代入文の両辺の型が等しく、ポインタ演算を行わないプログラムを入力と考える²。

入力となるプログラムで使用される文を簡単化のために以下の4種類に限定して考える。他の文に関しては単純な拡張によって得られる。

- (1) $x = \&y$ アドレス値の代入
- (2) $x = y$ ポインタ変数同士の代入
- (3) $*p = x$ Dereference された値への代入
- (4) $x = *p$ Dereference された値の代入

5.2.2 アルゴリズム

1. プログラム中で使用されるポインタの最大の多重度を d とする
2. i を d とする
3. $i < 1$ ならば終了, それ以外は続ける
4. ここで入力文の集合を S , S_i は多重度 i 型の文の集合として, S_i に含まれる各文 $s(s \in S_i)$ について次の場合に分け, 空のグラフ G_i に辺を追加する
 - (a) $s: x = \&y$ ならば グラフ G_i に辺 $x \rightarrow \&y$ を追加
 - (b) $s: x = y$ ならば グラフ G_i に辺 $x \rightarrow y$ を追加
 - (c) $s: *p = x$ ならば 任意の変数 a について, グラフ G_{i+1} に辺 $p \rightarrow \&a$ が存在するならグラフ G_i に辺 $a \rightarrow x$ を追加
 - (d) $s: x = *p$ ならば 任意の変数 a について, グラフ G_{i+1} に辺 $p \rightarrow \&a$ が存在するならグラフ G_i に辺 $x \rightarrow a$ を追加
5. グラフ G_i の推移閉包を求める
6. i を $i - 1$ として手順3に戻る

アルゴリズムの4において, 最大の多重度 d の文集合 S_d には $s: *p = x$ と $s: x = *p$ が含まれない。

²ポインタ演算がある場合は考えられるすべての参照先を参照すると考えて解析を続けることもできるが, 誤った参照先が多く生まれることになる。

5.2.3 計算量について

プログラムサイズを n とする。そうすると、文、変数、参照先の数はそれぞれ $O(n)$ であり、変数と参照先を合わせた数も $O(n)$ である。

今回推移閉包はワーシャル法、対象となるグラフ³の頂点は変数と参照先と考えている。よって推移閉包の最悪の計算量は $O(n^3)$ である。

また、グラフの頂点を変数のみと考え、推移閉包を求めたのちに各変数の参照先の和集合を計算する方法も考えられる。この場合、頂点数が減少する分だけ、推移閉包の計算量が減少する。和集合については各変数が全変数に依存していたとしても、分割して和集合をとることで計算量は $O(n \log n)$ で、推移閉包の計算量を越えない。結果的に計算量は小さくなると考えられる。

提案アルゴリズムについて 任意のポインタ多重度の文 S_i の大きさを n_i とすると、 $n = \sum_{i=1}^d n_i$ 。任意の多重度の変数と参照先の数は $O(n_i)$ となる。アルゴリズムの手順4(辺の初期化)は、 $s : *p = x$ と $s : x = *p$ に関してそれぞれ $O(n_i)$ の辺が加わる可能性があり、文の数が $O(n_i)$ であることから、計算量は $O(n_i^2)$ である。ただし、最大多重度に関しては $s : x = \&y$ と $s : x = y$ のみを考えればよいので $O(n_d)$ 。合計 $O(n_d + \sum_{i=1}^{d-1} n_i^2)$ 。アルゴリズムの手順5の推移閉包の計算量は、グラフの頂点数に依存するため $O(n_i^3)$ である。以上より、この手法の計算量は $O(n_d + \sum_{i=1}^{d-1} n_i^2 + \sum_{i=1}^d n_i^3) = O(\sum_{i=1}^d n_i^3)$ である。

従来のアルゴリズムについて 動的に推移閉包を求める方法は、文ごとの辺の追加(初期化)に $O(n)$ 、推移閉包 $O(n^3)$ に加えて、推移閉包後の辺の追加に起因する処理 $O(n^3)$ が必要である。よって計算量は $O(n + n^3 + n^3) = O(n^3)$ である。

計算量の比較 従来の計算量 $O(n^3)$ と提案アルゴリズムの計算量 $O(\sum_{i=1}^d n_i^3)$ については、 $n = n_1 + n_2 + \dots + n_d$ ならば $n^3 > n_1^3 + n_2^3 + \dots + n_d^3$ が明らかなので $O(n^3) > O(\sum_{i=1}^d n_i^3)$ となり、今回提案する手法が、動的な辺の追加を考慮する手法より、計算量を削減できていることがわかる。

³行列と考えても問題ない。

おわりに

本稿では Java(オブジェクト指向型) プログラムと C プログラムを対象とする参照先解析の新しい手法を提案し、それらの有用性を明らかにした。

まず、文脈を考慮した Java プログラムの参照先解析について、従来手法である LightSens[Mil07] を拡張し、計算量のオーダーを増加させることなく、正確さを向上させることに成功した。この計算量のオーダーは文脈非依存である Andersen-style の解析とも同じなので、計算量のオーダーを増加させずに文脈を考慮した結果が得られる手法と言える。また、この拡張手法ではプログラムの特定部分のみの正確性を向上させることも可能である。それは各ローカル変数の参照先を、他の変数の参照先に依存せずに計算するからである。

加えて SSA 形式を利用して Java プログラムの参照先解析を行う手法を提案した。SSA 形式を利用することで、フロー非依存の参照先解析の正確さがフロー依存解析に近い正確さまで向上することを確認した。なかでもフィールドに対する SSA 変換は容易ではなく、フィールド参照に使用される変数を参照先で置き換えることで初めて SSA 変換が可能になる。またメソッドの副作用への対応が必要である。

さらに、Andersen の解析を C プログラムに対して行う際に、ポインタ変数の多重度ごとに解析を行うことで、排反な部分ごとに推移閉包を計算するアルゴリズムを提案した。そうすることでより少ない計算量で Andersen の解析を行うことができる。

謝辞

日頃から丁寧にご指導いただきました大山口通夫教授，山田俊行講師，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感謝いたします。また，不意の質問に対する返答や，熱心な議論をしていただいた研究室の学生諸氏に感謝いたします。

参考文献

- [And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language (Chapter 4 Pointer Analysis pp.111-152)*. Phd thesis, DIKU, University of Copenhagen, May 1994.
- [HH98] R. Hasti and S. Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 97–105. Citeseer, 1998.
- [HL09] Ben Hardekopf and Calvin Lin. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Symposium on Principles of Programming Languages (POPL)*, pp. 226–238, 2009.
- [IK83] T. Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. *Information Processing Letters*, Vol. 16, pp. 95–97, 1983.
- [LH03] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. *Lecture Notes in Computer Science*, pp. 153–169, 2003.
- [LH06] O. Lhotak and L. Hendren. Context-sensitive points-to analysis: Is it worth it? *Lecture Notes in Computer Science*, Vol. 3923, p. 47, 2006.
- [Mil07] Ana Milanova. Light Context-Sensitive Points-to Analysis for Java. In *the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007)*, June 2007.
- [MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java. In *the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.
- [MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 14, No. 1, pp. 1–41, 2005.
- [SC09] Manu Sridharan and Stephen J. Fink (IBM T.J. Watson Research Center). The Complexity of Andersen’s Analysis in Practice. *Lecture Notes In Computer Science*, Vol. 5673, pp. 205–221, 2009.

- [Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pp. 32–41, 1996.
- [VLSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, & Tools*, chapter 12, pp. 903–964. Pearson/Addison-Wesley, second edition, 2007.
- [中田 99] 中田育男. コンパイラの構成と最適化, 第 12 章 3 節 静的単一代入形式 (SSA 形式), pp. 320–358. 朝倉書店, 1999.