Deductive System によるCプログラムのポインタ解析

千代 英一郎†

本論文では C プログラムに対する flow/context-insensitive，inclusion-based，field-sensitive なポインタ解析の新しい定式化および実現方法について述べる．inclusion-based，field-sensitive なポインタ解析は多くのコンパイラ最適化を可能とする高い解析精度を持つが，その解析効率（scalability）の点で実用性に疑問が持たれている．C におけるポインタ解析の効率向上については，これまで多くの手法が提案されているが，その大半は field-sensitive な解析を対象としておらず，構造体メンバ参照式等のない単純な言語における field-independent な解析に対するものである．C に存在する複雑な構文要素を扱う field-sensitive な解析にそれらの手法を適用することは容易ではなく，解析効率を高めるために複雑な実現を必要とし，解析処理系の品質確保を難しくする．本論文では，解析を式の構造に基づき deductive system により定式化することで，実用的な解析処理系に望まれる，高い解析効率と実現の容易性の両立が可能となることを示す．実現方法の特徴は，定式化された解析規則を関係代数演算として書き換える点にあり，これにより関係 database の基本的な最適化手法の利用が可能となる．

Pointer Analysis for C Programs Using a Deductive System

Eiichiro Chishiro†

We present a new formalization and implementation of flow/context-insensitive, inclusion-based, field-sensitive pointer analysis for C programs. Although inclusion-based, field-sensitive pointer analysis has high analysis precision which enables many compiler optimizations, whether its practically-efficient implementation is possible or not is still an open question. There exist many optimizations of pointer analysis for C, but most of them are designed for field-independent analysis on a simple language, namely, without struct member expression. It is not trivial to apply these optimizations to field-sensitive analysis on a complex language like full C. Typical implementations become complicated to get high efficiency and this makes it difficult to confirm their correctness. In this paper, we show a formalization of analysis using a deductive system based on a structure of expressions, and its implementation which is simple and highly efficient, both are desirable property for a practical analyzer. The point of our implementation method is the step of rewriting analysis rules to relational algebraic operations, which enables us to use basic optimizations in a relational database.

1. はじめに

ポインタ解析とは，プログラム中のポインタの指示先を静的に求めるプログラム解析である．ポインタの指示先に関する情報はメモリ参照式間の依存関係を正確に求めるのに必要であり，特に C プログラムにおいてはコンパイラの最適化の効果を大きく左右するため，これまで多くの方法が提案されてきている[15]．

以下にポインタ解析方法を分類する代表的な観点をあげる．

flow-sensitivity プログラムの制御フローを考慮するかどうか．一般に flow-sensitive な解析ではプログラム点ごとにポインタ指示先情報（解）を求めるのに対し，flow-insensitive な解析ではプログラム全体に対して共通する 1 つの解を求める．

context-sensitivity 関数呼び出し時の異なる文脈を区別するかどうか．context-sensitive な解析では，関数を呼び出したときに生じる作用を，その呼び出し元（callsite）ごとに文脈（引数の値等）を区別して求めるのに対し，context-insensitive な解析ではすべての文脈に対して共通する作用を求める．

assignment direction 代入の作用を包含関係として扱う（inclusion-based）か等値関係として扱う（equality-based）か．たとえば `p = q` という代入文の作用は，前者では「q の指示先はすべて p の指示先に含まれる」と見なされるのに対し，後

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

者では「p と q の指示先は等しくなる」と見なされる．

aggregate modeling 構造体内部の扱いに関して，各メンバを区別する（field-sensitive）か，区別しない（field-independent）か，メンバは区別するが同じ型の構造体変数どうしを区別しない（field-based）か．たとえば `struct S { int *p, *q; } s, t;` という構造体に対して，field-sensitive な解析では `s.p`，`s.q`，`t.p`，`t.q` を区別するが，field-independent な解析では `s.p`，`s.q` は `s`，`t.p`，`t.q` は `t` として扱い，個々のメンバを区別しない．また field-based な解析では逆に `s.p`，`t.p` は `S_p`，`s.q`，`t.q` は `S_q` として扱い，同じ型の構造体変数を区別しない．

一般に各項目の解析精度と解析効率（解析に要する時間やメモリ使用量）は対立関係にあり，ポインタ解析を設計する際には目的に応じてこれらの扱いを決める必要がある．

本論文では，flow/context-insensitive，inclusion-based，field-sensitive な解析の定式化および実現方法について述べる．大規模なプログラムを解析対象とする場合，現在のところ flow/context-insensitive な解析が妥当であることは多くの研究者の間で一般的な見解である[15]．一方で inclusion-based，field-sensitive な解析については，その実用性（解析効率の scalability）について意見が分かれている[8]．特に field-sensitive な解析については，その重要性は指摘されている[15),30)]が，大規模な C プログラムを効率的に解析できるような方法はまだ示されていない．これまでにポインタ解析の効率を向上させるための様々な手法が提案されてきているが，その多くは field-independent/based な解析に対するもので，それらの field-sensitive な解析への適用方法については明らかでない[11),14),28)]．最近では Pearce らにより field-sensitive な解析の効率向上方法が提案されているが，そこでの結果は否定的なもので，大規模なプログラムに対しては実用的でないと結論づけられている[21]．これに対して本論文では inclusion-based，field-sensitive な解析が十分に実用的な解析効率を持ちうることを示す．

実用性という観点を考慮すると，解析の定式化方法についてはその実現の容易性も重要な要素である．特にポインタ解析のテストやデバッグは難しいため，定式化された解析（仕様）と実現の間に密接な対応があり，実現が仕様の素直な反映になっていることが望ましい．

だが解析効率を向上させようとしたとき，そのための各種最適化はこれを妨げる要因の 1 つとなりうる．特に field-sensitive な解析では，定式化された解析そのものが複雑になるため，最適化の正しさを確認するのは容易ではない．

従来の定式化方法は，そのほとんどがグラフと集合に基づく実現を前提としており，そのような実現に対して，グラフの閉路を動的に検出したり[1]，あるノードから到達可能なノード集合の計算結果を cache 化したりする[14),30)] 等の最適化が提案されている．だが，多くの論文でその手法の提示に用いられている単純な言語を，複雑な構文要素を持つ実際の C やコンパイラ中間語に拡張した場合，これらの最適化の組合せを正しく実現するのは容易ではない．

これらの問題に対して，本論文では，解析を式の構造に基づき deductive system により定式化することで，実用的な解析処理系に望まれる，高い解析効率と実現の容易性の両立が可能となることを示す．実現方法の特徴は，定式化された解析規則を関係代数演算として書き換える点にあり，グラフと集合でなく table に基づく実現を行うことで，関係 database の基本的な最適化手法の利用が可能となる．

本論文の構成は以下のとおりである．2 章では従来の代表的なポインタ解析の定式化方法について概観する．3 章では本論文で提案する定式化方法について述べる．4 章では，3 章で定式化した解析を関係代数に基づき実現する方法について述べる．5 章では，提案手法に基づくポインタ解析の効率をベンチマークプログラムにより評価した結果を示す．6 章では関連する研究について述べる．7 章は結論である．

2. 従来の定式化

本章では，提案する定式化方法を示すための準備として，従来の代表的な定式化方法である type system および deductive system による定式化について概観し，その問題点を考察する．

2.1 type system による定式化

ポインタ解析の type system による定式化として代表的なのは Andersen[3] および Steensgaard[26] によるものである．両者は 1994〜1996 年頃に発表され，その後これらを基にした多くの解析が提案されてきている[1),8),11),13),23),25),28)]．

2.1.1 Andersen による定式化

Andersen の解析[3] は flow/context-insensitive，inclusion-based なポインタ解析の代表的存在であり，この種類の解析はしばしば「Andersen 風のポインタ解析」と呼ばれている．

$$\frac{}{\tilde{S} \vdash_{pexp} v : \{v\}}\ (var) \qquad \frac{\tilde{S} \vdash_{pexp} e_1 : O_1}{\tilde{S} \vdash_{pexp} *e_1 : \bigcup_{o \in O_1} \tilde{S}(o)}\ (indr) \qquad \frac{\tilde{S} \vdash_{pexp} e_1 : O_1 \quad \tilde{S}(l) \supseteq O_1}{\tilde{S} \vdash_{pexp} \&^l e_1 : \{l\}}\ (address)$$

$$\frac{\tilde{S} \vdash_{pexp} e_1 : O_1 \quad \tilde{S} \vdash_{pexp} e_2 : O_2 \quad \forall o \in O_1 . \tilde{S}(o) \supseteq \bigcup_{o_2 \in O_2} \tilde{S}(o_2)}{\tilde{S} \vdash_{pexp} e_1 \ = \ e_2 : O_2}\ (assign)$$

図 1 Andersen の型規則（抜粋）
Fig. 1 Typing rules of Andersen (excerpt).

$$\frac{A \vdash x : \mathtt{ref}(\alpha_1) \quad A \vdash y : \mathtt{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash welltyped(x = y)}\ (1) \qquad \frac{A \vdash x : \mathtt{ref}(\tau \times \_) \quad A \vdash y : \tau}{A \vdash welltyped(x = \&y)}\ (2)$$

$$\frac{A \vdash x : \mathtt{ref}(\alpha_1) \quad A \vdash y : \mathtt{ref}(\mathtt{ref}(\alpha_2) \times \_) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash welltyped(x = *y)}\ (3)$$

$$\frac{A \vdash x : \mathtt{ref}(\mathtt{ref}(\alpha_1) \times \_) \quad A \vdash y : \mathtt{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash welltyped(*x = y)}\ (4)$$

図 2 Steensgaard の型規則（抜粋）
Fig. 2 Typing rules of Steensgaard (excerpt).

Andersen の解析では，変数等のメモリ領域のアドレスを抽象 location（$ALoc$）と呼ぶ記号表現によって扱う．ポインタ解析の解は，抽象 location から，それが示す領域の持つアドレス値（抽象 location）の集合への関数 $\tilde{S} : ALoc \to \wp(ALoc)$ である．たとえば，変数 `p`，`x` の抽象 location を $\alpha(\mathtt{p})$，$\alpha(\mathtt{x})$ とすると，`p` が `x` を指しうるとき，正しい解析の解は $\tilde{S}(\alpha(\mathtt{p})) \supseteq \{\alpha(\mathtt{x})\}$ を満たす．すなわち $\tilde{S}$ はメモリ領域を抽象 location によりモデル化したものになっている．

解析は型規則として定式化される．ここでの型は通常の type system における `int` や `float` 等ではなく，抽象 location の集合 O である．型規則は，式に対してそれが表す抽象 location の集合を型として割り当てる．また正しい解を得るために必要な条件を型に関する制約の形で規則の前提条件として定める．

Andersen の解析の型規則の一部を**図 1** に示す．代入に対する型規則 ($assign$) の前提にある包含制約

$$\forall o \in O_1 . \tilde{S}(o) \supseteq \bigcup_{o_2 \in O_2} \tilde{S}(o_2)$$

は，「左辺の表す各抽象 location の指示先が右辺の表すすべての抽象 location の指示先を含む」というポインタ解析の解が満たすべき基本的な条件を定めている．

この定式化によりポインタ解析は対象プログラムに対する型推論問題となるが，その実質は型規則適用時に前提として要請されるすべての制約を満たす解を探す制約解消問題に帰着される．

Andersen の解析は，言語の構文構造に基づいて構成的に定式化されており，使用されている制約も言語の semantics に即した明瞭なもので，その意味は直感的に理解しやすい．Andersen は構造体メンバ参照式を field-based な方法で取り扱っているが，これを field-sensitive な方法に拡張するのは容易である．

一方，Andersen の定式化における大きな問題は，型が集合であり，かつ制約においてその要素を参照しているため，unification を用いた通常の型推論アルゴリズムのような効率的な実現が難しいことである．この点については 2.1.3 項で考察する．

2.1.2 Steensgaard による定式化

Steensgaard の解析[26] は，大規模な実用プログラムの解析を主目的として設計されており，解析精度を犠牲にすることでほぼ線形時間の解析効率を達成している．Andersen の解析と同様に flow/context-insensitive な解析であるが，代入の扱いが異なり，代入における制約を包含関係でなく等値関係によって定めている（equality-based）点に特徴がある．**図 2** に Steensgaard の解析の型規則の一部を示す．Andersen の場合と異なり Steensgaard の解析対象言語は式の形が単純なものに制限されているため，文に対する型規則が式に対する型規則を取り込む形で与えられている．

解析における型は次のように定義されている．

$$\alpha ::= \tau \times \lambda$$
$$\tau ::= \bot \mid \mathtt{ref}(\alpha)$$
$$\lambda ::= \bot \mid \mathtt{lam}(\alpha_1 ... \alpha_n)(\alpha_{n+1} ... \alpha_{n+m})$$

Steensgaard の解析では，変数と関数のアドレスを区別しており，型 α は変数に対応する型 τ と関数に対応する型 λ の組となっているが，この区別は本論文での議論に影響しないため，ここでは変数に対応する型 τ だけを考える．$\perp$ はアドレス以外の値を表す型で，$\mathtt{ref}(\alpha)$ は型 α によって表される領域のアドレスを表している．型規則の前提に使用されている関係 $\trianglelefteq$ は条件付きの等値関係で，

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$$
$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4)$$

と定義されている．したがって，通常の（型変換のない）ポインタ代入だけを考える場合，$t_1 \trianglelefteq t_2$ は $t_1 = t_2$ という等値関係と同じ意味になる．

たとえば単純な変数間の代入に対する型規則 (1) の制約 $\alpha_2 \trianglelefteq \alpha_1$ は「左辺の指示先と右辺の指示先は等しい」ということを，解が満たすべき条件として定めている．すなわち代入 `x = y` と `y = x` は区別されず，同じ効果を持つ．

代入の効果を等値制約として扱うことにより型が集合でなく atomic なものとなるため，Steensgaard の解析はよく知られている union/find アルゴリズムによる効率的な実現が可能である．一方，代入の効果を包含制約によって扱う Andersen の解析に比べると一般に解析精度は大きく低下する．

2.1.3 type system による定式化の利点と欠点

これまで見てきたとおり，type system による解析の定式化においては，プログラム中の各構文要素に対して何らかの解析値を型として対応付け，求める解析結果が満たすべき条件を型の間の制約関係として表現する．

type system による定式化には次のような利点が存在する．

- プログラム中の各構文要素に対して体系的・構成的に値を割り当てることができる．
- プログラミング言語理論の分野でこれまで蓄積されてきた type theory の結果を利用できる．たとえば解析の正しさはその type system の健全性の問題に帰着させることができる．
- 型が atomic なものであれば，通常の型推論アルゴリズムのように効率的な実現が可能である．

一方で，式に対して値の集合を型として割り当てるという方法は，Andersen の解析のように代入を両辺の間の包含制約として扱う場合に，解析効率の点で問題が生じる．具体的にいえば，Andersen の型規則（*assign*）の前提にある制約 $\forall o \in O_1.\tilde{S}(o) \supseteq \bigcup_{o_2 \in O_2} \tilde{S}(o_2)$ に現れている o や o_2 は型ではない．このことは型推論系や汎用的な制約解消系による効率的な実現を困難にする．一方 Steensgaard のように代入を equality-based に扱えばこの問題は解消するが，解析精度は大きく低下する．

Foster，Fähndrich，Aiken らは，型を共変成分と反変成分の対として定義することで，代入における制約を型の間の包含制約として表現できるようにした定式化方法を提案している[1),13)]．ただし，6 章で触れるように，その解析を効率的に実現するためには制約解消系における様々の複雑な最適化が必要であり，まだ十分な解析効率は得られていない．また構造体内部は field-independent な方法で取り扱っており，それを field-sensitive な方法に拡張する方法は明らかでない．

2.2 deductive system による定式化

Heintze らは，解析の基本的な方針（flow/context-insensitive，inclusion-based）に関して Andersen の解析を踏襲しながらも，McAllester が行った deductive system による dataflow 解析の定式化[19)] に倣い，ポインタ解析を同様の deductive reachability system として定式化している[14)]．

彼らの解析対象言語は以下に示すきわめて単純なものである．

$$\begin{array}{llll} Exp & e & ::= & x \mid *x \mid \&x \\ Stmt & s & ::= & e_1 = e_2 \qquad where\ e_1 \not\equiv \&x \\ Prog & p & ::= & s^* \end{array}$$

構造体メンバ参照式はスカラ変数に変換して取り扱う．変換方法は field-based もしくは field-independent な方法による．

解析対象プログラム P に対するポインタ解析は，**図 3** の規則によって構成される．各規則は P 上で成り立つポインタ指示関係の導出規則を定めており，ポインタ解析の解は，これらの規則の閉包（規則から導出されるすべてのポインタ指示関係の集合）となる．これまで示した定式化手法に比べると，図 3 の定式化はきわめて簡潔である．type system による定式化との比較については 3 章で述べる．

2.2.1 Heintze らの定式化の利点と欠点

その一方で，この定式化は文と解析値（ポインタ指示関係）を直接結び付けている点で，古典的な dataflow 方程式による定式化と共通の問題を含んでいる．すなわち，複雑な構文要素を構成的に取り扱えるようには

$$\frac{x \to \&y}{y \to e}\ (if\ *x = e\ in\ P) \qquad \frac{x \to \&y}{e \to y}\ (if\ e = *x\ in\ P) \qquad \frac{}{e_1 \to e_2}\ (if\ e_1 = e_2\ in\ P)$$

$$\frac{e_1 \to e_2 \quad e_2 \to e_3}{e_1 \to e_3}$$

図 3 Heintze らの導出規則
Fig. 3 Deduction rules of Heintze, et al.

$$\frac{x \Rightarrow new_y}{new_y.f \Rightarrow e}\ (if\ x.f = e\ in\ P) \qquad \frac{x \Rightarrow new_y}{e \Rightarrow new_y.f}\ (if\ e = x.f\ in\ P) \qquad \frac{}{e_1 \Rightarrow e_2}\ (if\ e_1 = e_2\ in\ P)$$

$$\frac{e_1 \Rightarrow e_2 \quad e_2 \Rightarrow e_3}{e_1 \Rightarrow e_3}$$

図 4 Whaley らの導出規則
Fig. 4 Deduction rules of Whaley, et al.

なっていない.

実際，文献 14) では，構造体内部の扱いに関しては field-based および field-independent な解析を比較するにとどまり，field-sensitive な解析の実現については今後の課題とされている．field-based/independent な解析はともにメモリ参照式の持つ構文構造の扱いを必要とせず，それが図 3 に示すような簡潔な解析の構成につながっている．

一方で field-based な解析は，解析対象プログラムが ANSI/ISO で定める C の言語仕様[2] に strictly conforming な場合でも解析結果が安全でない場合がある☆．また field-based/independent な解析は，field-sensitive な解析に比べて解析精度が大きく低下することが知られている[30]．

Whaley らは，Heintze らの定式化に基づき，Java プログラムに対して field-sensitive なポインタ解析を定式化している[30]．ただし，Whaley の手法においても，文と解析値を直接結び付けるアプローチは変わっておらず，それで問題が生じていないのは，Java の場合 C に比べてポインタ値に影響を及ぼす構文パターンがきわめて限定されるため，問題が顕現するほどの複雑さが存在しないという事情による．

図 4 に Whaley による定式化を示す．Java には C と異なり，構造体（Object）の途中を指すポインタが存在しない，nest した構造体（Object）がない，共用体がない等の要因のおかげで，図 4 に示すようにメンバ参照の作用を文の解析規則に組み込んでしまうことが可能である．

3. 提 案 手 法

2 章では，代表的なポインタ解析の定式化方法について概観した．本章ではこれをふまえて，従来の欠点を克服するポインタ解析の定式化方法について述べる．

3.1 方針と特徴

type system による定式化では，各構文要素ごとに解析値（たとえば Andersen の方法では抽象 location の集合 O）を対応付け，解析値の間の制約関係によって求めるポインタ指示関係の条件を表していた．その重要な特徴は，式に対して対応付ける解析値が，その式がとりうるすべての値の集合を表していることである．Steensgaard の方法では集合は明示的に現れていないが，型変数 τ が意味するのはその式の値の集合であることに注意されたい．そのため，代入における解析値の間の制約関係を自然に表すことが難しく，型としての atomic な扱いを止めて集合を要素に分解したり（Andersen），包含関係を等値関係に近似することで解析精度を犠牲にしたり（Steensgaard），型に反変成分を導入することで複雑な定式化を行ったりする（Foster）必要があった☆☆．

これに対して deductive system による定式化では，規則から導出される個々の関係は基本的にとりうる値の 1 つに関する関係であり，集合間の関係ではない．type system の場合には解析値が集合を表すことで「とりうるすべての値」を考慮していたが，deductive system では，導出されるすべての関係の総体によってそれを考慮しているといえる．その一方で，Heintze らの方法では，解析結果として求めるポインタ指示関係を直接用いて定式化を行っており，Andersen の

☆ common initial sequence（同じ型のメンバの並び）を持つ構造体間のキャスト等.

☆☆ この点については文献 12) でも少し触れられている.

$$
\begin{array}{llll}
AExp & m & ::= & x \mid *m \mid m.k \qquad\qquad where\ k \in Nat \\
AStmt & s & ::= & m_1 = m_2 \mid m_1 = \&m_2 \\
AProg & p & ::= & s^*
\end{array}
$$

図 5 L_A の抽象構文
Fig. 5 Abstract syntax of L_A.

方法のように，式の解析値をその構成要素から組み立てていくようにはなっていない．Heintze らの定式化は McAllester の定式化[19] に基づいているが，そこでの McAllester の主目的は dataflow 解析を deductive system によって構成することであり，dataflow 方程式と同様に値の伝播を取り扱うために最適化された定式化となっている．その結果，そのままでは field-sensitive な方式のような構造体内部の扱いに関する精度の高い解析を定式化することが難しい．

本章では flow/context-insensitive，inclusion-based，field-sensitive な解析を deductive system により定式化する．ただし Heintze らの定式化と異なり，Andersen の type system による定式化と同様に，ポインタ指示関係を直接導出するのではなく，式とその左辺値，および左辺値の持つ右辺値という 2 種類の関係から間接的にポインタ指示関係を定義する．これにより，集合を扱わずにすむという deductive system の利点を保持すると同時に，式の構造に基づいて解析値を構成的に定めることができるという type system による定式化の利点を備えることができる．

3.2 解析言語

本手法で解析時に取り扱う（単純化された）言語 L_A の抽象構文の基本部分を**図 5** に示す．

メモリ参照式 *AExp* は，変数 x，間接参照式$*m$，メンバ参照式 $m.k$ からなる．構造体メンバは名前でなく整数値の byte offset によって参照する．本手法では配列要素の区別を行わないため，メモリ参照式に配列要素参照は含まれていない．文 *AStmt* は，ポインタ型のメモリ参照式間の代入文およびアドレス代入文からなる．本手法では flow-insensitive な解析のみを扱うため，制御構文は含まれていない．プログラム *AProg* は，文の列である．

Heintze らの解析言語に比べると，L_A のメモリ参照式は，任意個の間接参照およびメンバ参照を許す点で，Andersen の解析言語と同等の表現力を持っている．

3.3 解析領域

次に L_A に対する解析を deductive system により定義する．deductive system は解析で取り扱う fact の集合（解析領域）および fact の導出規則から構成される．本解析における fact はすべて関係（relation）である．

最初に解析領域を定める各種の関係を定義する．

定義 1（ABlock） ポインタ解析の対象となる互いに重なり合わない（disjoint な）メモリ領域を抽象 block と呼ぶ．

$$b \in ABlock$$

□

プログラム中の各変数 `x`，`y`，.. の占めるメモリ領域に対応する抽象 block を x，y，.. によって表す．

構造体型変数のような，構造を持つ領域の内部は領域先頭からの offset によって区別する．すなわち解析ではメモリアドレスを領域と領域内の offset の組として取り扱う．

定義 2（ALoc） 抽象 block と，抽象 block が表す領域先頭からの相対 offset の組を抽象 location と呼ぶ．

$$l \in ALoc = ABlock \times Nat$$

□

たとえば変数 `x` の抽象 location は $(x, 0)$，構造体メンバ `s.4` の抽象 location は $(s, 4)$ と表される．

次に，解析領域における基本的な関係として左辺値関係 *AEnv* と右辺値関係 *AStore* の 2 つを定義し，これらを用いてポインタ指示関係を定義する．

定義 3（AEnv） プログラム中のメモリ参照式と，それが表すメモリ領域の抽象 location の対応を表す関係を左辺値関係 *AEnv* と呼ぶ．

$$aenv(m, l) \in AEnv = AExp \times ALoc$$

□

定義 4（AStore） 抽象 location と，それが表す領域の持つ値（抽象 location）の対応を表す関係を右辺値関係 *AStore* と呼ぶ．

$$astore(l_1, l_2) \in AStore = ALoc \times ALoc$$

□

定義 5（PointsTo） プログラム中のメモリ参照式と，それが指す領域（を表すメモリ参照式）との対応を表す関係をポインタ指示関係 *PointsTo* と呼ぶ．

$$\frac{x \in AExp_P}{x \underset{L}{\longrightarrow} (x,0)}\ (top) \qquad \frac{m \underset{L}{\longrightarrow} (b,o) \quad m.k \in AExp_P}{m.k \underset{L}{\longrightarrow} (b,o+k)}\ (field)$$

$$\frac{m \underset{L}{\longrightarrow} l_1 \quad l_1 \underset{R}{\longrightarrow} l_2 \quad *m \in AExp_P}{*m \underset{L}{\longrightarrow} l_2}\ (ind)$$

$$\frac{m_1 \underset{L}{\longrightarrow} l_1 \quad m_2 \underset{L}{\longrightarrow} l_2 \quad m_1 = \&m_2 \in AStmt_P}{l_1 \underset{R}{\longrightarrow} l_2}\ (assign\ address)$$

$$\frac{m_1 \underset{L}{\longrightarrow} l_1 \quad m_2 \underset{L}{\longrightarrow} l_2 \quad l_2 \underset{R}{\longrightarrow} l_3 \quad m_1 = m_2 \in AStmt_P}{l_1 \underset{R}{\longrightarrow} l_3}\ (assign\ simple)$$

図 6 ポインタ解析の導出規則
Fig. 6 Deduction rules for pointer analysis.

$$pointsto(m_1, m_2) \in PointsTo = AExp \times AExp$$

ポインタ指示関係は，*AEnv* と *AStore* から以下のように定める．

$$\frac{aenv(m_1, l_1) \quad aenv(m_2, l_2) \quad astore(l_1, l_2)}{pointsto(m_1, m_2)}$$

□

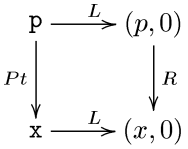
AEnv は，通常の semantics の定義における環境（変数からその location への写像）の概念を一般のメモリ参照式に拡張したものになっている．これらの関係が，type system における定式化のように集合を含んだ関係でないことに注意されたい．たとえば，type system による定式化に倣うなら，メンバ参照式 $m.k$ の左辺値型（*AEnv* に相当）は

$$\frac{\tilde{S} \vdash_L\ m : V}{\tilde{S} \vdash_L\ m.k : \{(b, o+k) \mid (b,o) \in V\}}$$

のように，集合およびその要素を用いて定義される（$\tilde{S}$ は *AStore* に相当する右辺値型環境）．これはすでに述べたとおり，効率的な実現を行ううえでの大きな障害となる．

以降では表記を簡潔にするため，左辺値関係を $\underset{L}{\longrightarrow}$，右辺値関係を $\underset{R}{\longrightarrow}$，ポインタ指示関係を $\underset{Pt}{\longrightarrow}$ という中置演算子によって表す（たとえば $aenv(m,l)$ を $m \underset{L}{\longrightarrow} l$ によって表す）．

例 3.1 たとえば，プログラム中で変数 p が x を指す場合，正しいポインタ解析の結果は以下の関係を含む．



□

従来の deductive system によるポインタ解析では，図式の左側の関係 $\underset{Pt}{\longrightarrow}$ を直接導出するのに対し，本手法ではそれを図式の右側の関係 $\underset{R}{\longrightarrow}$ から関係 $\underset{L}{\longrightarrow}$ を媒介として導く点に特徴がある．

最後に，メモリ参照式間の親子関係を定義する．これは 4 章で示す解析の実現で用いる概念である．

定義 6（Parent） プログラム中のメモリ参照式と，それが含む最大の部分式の対応を表す関係を親子関係 *Parent* と呼ぶ．

$$parent(m_1, m_2) \in Parent = AExp \times AExp$$

m_1 の最大の部分式 m_2 を m_1 の親と呼ぶ．親子関係は以下の規則により定義する．

$$\frac{}{parent(*m, m)} \qquad \frac{}{parent(m.k, m)}$$

□

3.4 導出規則

3.3 節で定義した解析領域に基づく解析の導出規則を**図 6** に示す．2 章で説明した Heintze や Whaley の方法では，各規則は実際には rule schema であり，解析は対象プログラム P から各 rule schema を instance 化して得られる規則の集合として構成される．ここでは，本質的に同じだが，より分かりやすい表現を用い

ている．

各規則の意味を以下に述べる．なお規則中の $AExp_P$，$AStmt_P$ はそれぞれ対象プログラム P 中に含まれている（部分）式および文の集合である．規則（*top*），（*field*），（*ind*）はメモリ参照式の左辺値関係 $\underset{L}{\longrightarrow}$ の導出規則である．規則（*top*）は，プログラム P に含まれる（すなわち $AExp_P$ に含まれる）すべての変数 x が左辺値 $(x, 0)$ を持つことを表している．規則（*field*）は，メンバ参照式 $m.k$ の親である m が左辺値 (b, o) を持つとき，そのメンバ参照式は親の左辺値にメンバの offset を加えた左辺値 $(b, o + k)$ を持つことを表している．親となる式 m は間接参照式の場合もあるため，メンバ参照式は複数の左辺値を持ちうる．規則（*ind*）は，間接参照式 $*m$ の左辺値が，その親 m の左辺値 l_1 が持つ右辺値 l_2 であることを表している．これはたとえば，代入文の左辺が`*p`のとき，代入先領域のアドレスは`p`の持つ値になることを考えると分かりやすい．

規則（*assign address*），（*assign simple*）は代入によって生じる右辺値関係 $\underset{R}{\longrightarrow}$ の導出規則である．規則（*assign address*）は，アドレス代入文により，左辺の左辺値 l_1 が右辺の左辺値 l_2 を右辺値として持つことを表している．規則（*assign simple*）は，ポインタ代入文により，左辺の左辺値 l_1 が右辺の左辺値 l_2 の持つ右辺値 l_3 を右辺値として持つことを表している．

関係 $\underset{L}{\longrightarrow}$ と $\underset{R}{\longrightarrow}$ の導入により，type system の場合と同様，解析がプログラムの構文構造に基づく semantics の素直な反映として定式化されていることに注意されたい☆．その結果，2.2 節で指摘した Heintze らの定式化の問題が解消され，自然に field-sensitive な方法が実現されている．また本論文では触れないが，個々の式に対する左辺値が明示的に定義されているため，キャスト式を解析対象言語に追加したときに，その作用を正しく扱う規則の導入も容易である．

4. 実　　現

3 章では deductive system によりポインタ解析を定式化した．本章では関係代数に基づくその実現方法について述べる．

4.1 deductive system の閉包計算

本論文のポインタ解析は flow/context-insensitive なものであり，その解はプログラム実行時に成立しうるすべてのポインタ指示関係 *PointsTo* である．3.3 節で述べたように，提案手法におけるポインタ指示関係 *PointsTo* は左辺値関係 *AEnv* および右辺値関係 *AStore* によって定義されるため，解の計算の実質は，プログラム実行時に成立しうるすべての関係 *AEnv* および *AStore* を求めることである．これは対象プログラム P に対して図 6 の導出規則により導出可能なすべての関係 *AEnv* および *AStore* を求めることに相当する．以降では，deductive system の各規則によって導出可能なすべての fact（関係）を求めることを deductive system の閉包計算と呼ぶ．

deductive system の閉包を計算する方法として，次に述べる top down 法および bottom up 法が存在する．

top down 法　top down 法では最終的に必要とする関係（ここでは *aenv* や *astore*）を起点（goal）とし，その導出につながる規則を順に適用していく．Prolog や saturation-based な自動定理証明系の計算過程がこれに相当する．解が有限であっても，導出木が無限になる可能性があるため，全解探索を行う際には table 化等により，一度探索した規則の instance を記憶しておく必要がある．

bottom up 法　bottom up 法では各規則を繰り返し適用しながら，最終的に必要な結果も含めてすべての導出可能な関係を求める．deductive database 向け言語である datalog の計算過程がこれに相当する．当然だが，導出可能な関係が有限でなければ収束しない．

それぞれ長所と短所を持つが，ここでは bottom up 法の考え方に基づく実現を行う．top down 法による解析については 6 章で少し触れる．

4.2 関係代数に基づく実現

bottom up 法による閉包計算は，datalog 等の論理型言語を用いるならば，元になる導出規則をほぼそのまま書き換えるだけで実現可能である．これは本論文の目的の 1 つである解析処理系の品質確保という観点からは理想的であるが，解析効率に問題があり，大規模なプログラムを実用的な時間で解析するのは難しい．

本論文では，deductive system で扱う各種関係を関係 database における関係 table として表現し，bottom up 法による閉包計算を関係代数演算に基づいて実現する．導出規則の関係代数に基づく実現は，datalog 等を用いた場合における Horn 節による実現ほど直接的ではないが，各規則と 1 対 1 に対応した宣言的な記述によって行うことが可能であるとともに，関係 database における基本的な手法を用いることで効率的な実現が可能となる．

☆ やや informal に述べるなら，Andersen の解析における型関係：と $\underset{L}{\longrightarrow}$，型の間の制約関係 $\supseteq$ と $\underset{R}{\longrightarrow}$ がそれぞれ対応していると見なすことができる．

$$
\begin{aligned}
AExp &= \{ExpID, ExpKind, Parent, Base, Offset\}\\
Assign &= \{AssignID, AssignKind, LE, RE\}\\
ALoc &= \{LocID, Base, Offset\}\\
AEnv &= \{ExpID, LocID\}\\
AStore &= \{LocID, Content\}
\end{aligned}
$$

$$
\begin{aligned}
Dom(ExpKind) &= \{\texttt{top}, \texttt{indirect}, \texttt{field}\}\\
Dom(AssignKind) &= \{\texttt{simple}, \texttt{address}\}\\
Dom(Base) &= String\\
Dom(otherwise) &= Nat
\end{aligned}
$$

図 7 L_A の関係 schema と定義域
Fig. 7 relational schema and domains of L_A.

RuleTop = **insert ignore into** AEnv
(**select** ExpID, ALocID
from AExp, ALoc
where AExp.Base = ALoc.Base **and** ExpKind = "top" **and** ALoc.Offset = 0);

図 8 規則（*top*）の SQL
Fig. 8 SQL for rule (*top*).

4.2.1 解析領域の実現

最初に 3.3 節で定義した解析領域を関係 table として実現する．L_A の解析に用いる関係 schema および各属性の domain の定義を**図 7** に示す．

関係 schema *AExp* はプログラム中のすべての式 *AExp* を保持するためのものである．属性 *ExpID* は各式を区別するための一意な識別子（整数値）である．属性 *ExpKind* は式の種類を表すもので，`top` は単純な変数式，`indirect` は間接参照式，`field` はメンバ参照式をそれぞれ意味している．

本手法における解析対象言語 L_A の式 *AExp* は再帰的な木構造を持つ．これを関係 table 上に表現するために，関係 schema *AExp* は 3.3 節で定義した式の親子関係（*Parent*）を属性として含んでいる．属性 *Parent* は，親が存在する式の場合（間接参照式またはメンバ参照式の場合），その親の式の *ExpID* を表す．属性 *Base* は式の種類が `top` の場合（変数式の場合），その変数名を表す文字列を表す．属性 *Offset* は式の種類が `field` の場合，そのメンバの offset 値を表す．

関係 schema *Assign* についても同様である．属性 *AssignID* は一意な識別子（整数値）を表す．属性 *AssignKind* は代入文の種類を表しており，`simple` は $e_1 = e_2$ の形の単純代入文，`address` は $e_1 = \&e_2$ の形のアドレス代入文をそれぞれ意味している．属性 *LE* および *RE* はそれぞれ代入文の左辺および右辺の式の *ExpID* を表す．

その他の関係 schema *ALoc*，*AEnv*，*AStore* の意味は明らかである．

4.2.2 導出規則の実現

前述したように，deductive system の bottom up な閉包計算では，各導出規則ごとに，その前提となるすべての fact が存在するならばその結論となる fact を system に追加する．この導出の関係代数に基づく実現は，前提に使用される関係に対応する関係 table 間の結合演算，およびその結果の挿入により行うことが可能である．

たとえば，図 6 に示した規則（*top*）による導出は，**図 8** に示す SQL 文として実現できる☆．図 8 の SQL 文は，table *AExp* と table *ALoc* を属性 *Base* に関して結合した table から，*ExpKind* = `top` かつ *ALoc.Offset* = 0 である record を選択し，属性 *ExpID* および *LocID* に関して射影した結果を table *AEnv* に挿入する．図 6 の規則（*top*）と比較すれば，その意味は明らかである．図 6 のその他の規則についても同様の実現が可能である（付録 A.1 参照）．

5. 評　　価

本章ではベンチマークプログラムを用いて提案手法

☆ SQL の文法は処理系により細かな差異がある．ここでは MySQL 4.1 [20] 系の文法に基づいて記述する．なお **ignore** は重複する record の挿入を無視する keyword である．

表 1 ベンチマークプログラムの特徴
Table 1 Benchmark characteristics.

name	lines	Funcs	AExp	Assign	CallSite	ALoc
dhry2	755	170	769	331	75	844
a2time	11996	327	3507	1587	518	2732
130.li	7597	515	7328	4792	1245	6153
124.m88ksim	19916	416	9011	4421	1514	7089
126.gcc	205601	2165	95540	60145	19873	68743

表 2 解析結果
Table 2 Analysis result.

name	AEnv	AStore	iterations	time-base(s)	time-tuned(s)	memory(MB)
dhry2	709	265	9	0.06	0.05	0.2
a2time	2811	3053	9	0.59	0.31	0.8
130.li	8587	18926	19	5.47	2.19	2.3
124.m88ksim	9992	14501	11	4.08	1.41	2.4
126.gcc	93777	89869	15	379.85	23.33	25.9

の有効性を解析効率の観点から評価する．素朴な実装に対して関係 database における基本的な最適化を施すだけで，実用的な解析時間が得られることを示す．なお，解析精度に関しては筆者が文献 33) で示した解析方式と同等であり，その評価についてはそこに示されているため，ここでは省略する．

使用したベンチマークプログラムの特徴を**表 1** に示す．dhry2 は Dhrystone 2.1，a2time は EEMBC/A2TIME01 [10]，124.m88ksim，126.gcc および 130.li は SPECint95 [24] の同名のベンチマークプログラムを表す．表中の数は，標準ライブラリの効果を模倣する 158 個の stub 関数を含んでいる．測定環境は CPU: Pentium 4 3 GHz，memory: 2 GB，OS: Windows XP である．

3 章で示した解析対象言語 L_A に含まれていない関数呼び出し，復帰，構造体代入等の C の構文要素についても，L_A と同様に導出規則を定義できる．またキャストや共用体については文献 33) の UT/VT 領域解析を用い，複数の異なる型で参照される領域については field-independent な方法で取り扱うことで，解析の安全性を確保している．これは文献 33) の型規則をほぼそのまま導出規則に書き換えることで可能であり，ここでは詳細は省略する．

5.1 基本実装（nested loop）

4.2.2 項で示した関係代数演算により実現した導出規則の適用処理において，その中心となるのは複数の関係 table を結合する部分である．これを実現する最も素朴な方法は，結合する関係 table ごとに，そこに含まれるすべての record をたどる loop を構成し，結合条件を満たす record の組を探す，というものである．これは nested loop 方式と呼ばれるもので，関係代数演算で記述された導出規則を最も仕様に近い形で実現できる．ただし，その計算量は結合する各 table i に含まれる record 数 N_i の積 $O(\prod_i N_i)$ であり，table の record 数が増加するにつれ効率が大きく低下する．

表 2 の time-base は nested loop 方式で結合を実現したときのポインタ解析時間（秒）である．表中の iterations は，すべての規則を 1 度ずつ適用することを基本単位（1 iteration）とし，閉包計算が終了するまでに要した iteration 数を表している☆．

5.2 最適化（hash-clustering）

table の結合は関係 database における基本的な演算であり，様々な最適化が存在する．ここでは，その代表的な最適化である hash-clustering 方式[17] を利用することで，効率が改善されることを示す．

hash-clustering 方式では，結合する各 table を hash 関数により，結合する属性に関して同じ hash 値を持つ record 群（bucket と呼ぶ）に分類する．結合は同じ hash 値を持つ bucket ごとに行うため，その計算量は，結合する各 table i の bucket k に含まれる record 数を m_{ik} としたとき，bucket ごとの結合計算量の和 $O(\sum_k \prod_i m_{ik})$ となる．すなわち bucket に含まれる record 数 m を十分小さくできれば，table の record 数が増加しても効率の低下を抑えることができる．実現は nested-loop 方式より若干複雑になるが，変化するのは join におけるループ制御部分のみであり，プ

☆ 当然，規則の適用順序を工夫したり，すべての規則を等しく適用したりするのではなく特定の規則の適用回数の割合を増やす等の最適化が考えられるが，本論文ではこれらに関する評価は実施していない．

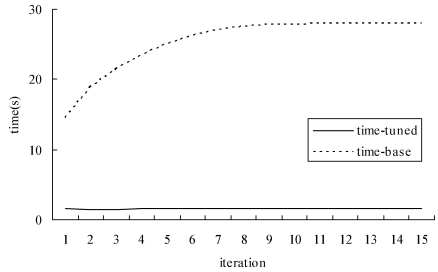


図 9 iteration ごとの解析時間（126.gcc）
Fig. 9 Analysis time per iteration for 126.gcc.

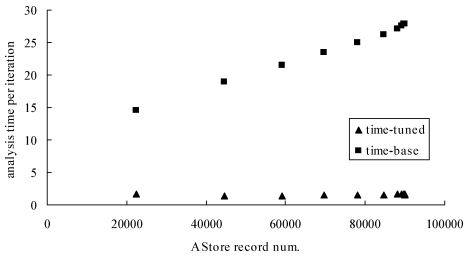


図 11 *AStore* の record 数と解析時間の相関
Fig. 11 Correlation between AStore record num. and analysis time for 126.gcc.

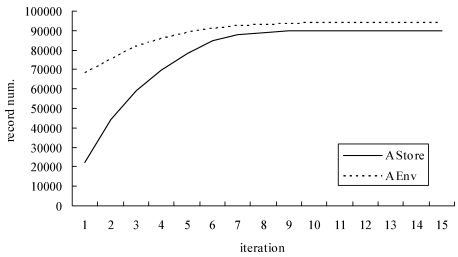


図 10 iteration ごとの record 数（126.gcc）
Fig. 10 The number of records for 126.gcc.

ログラムの構造は変わらないため，品質確保も容易である．

ここでは，特に規模が大きくなる table *AEnv* の hash 関数として *ExpID* mod *HASHSIZE*，table *AStore* の hash 関数として *LocID* mod *HASHSIZE* を用い，*HASHSIZE* の値を十分大きく（10 万程度）とることで，bucket にはほぼ結合対象 record のみが含まれるようにした．

表 2 の time-tuned は hash-clustering 方式で結合を実現したときのポインタ解析時間（秒），memory は解析で使用した総メモリ量（MB）である．nested loop 方式と比較すると，プログラムの規模が大きくなるにつれて，解析時間が大きく改善されていることが分かる．

図 9，**図 10** に 126.gcc の解析時間および record 数の変化を示す．図 9 は，各 iteration ごとに要した解析時間，図 10 は各 iteration 終了時の table *AEnv* および table *AStore* の record 数を表している．解析が進み record 数が増加していくにつれて，nested loop 方式（time-base）では規則の適用，すなわち結合演算に要する時間が増加していくが，hash-clustering 方式（time-tuned）では，その増加は緩やかなものにとどまっている．たとえば，table *AStore* の record 数 s と各 iteration ごとの解析時間 t（秒）について回帰分析を行うと，nested loop 方式の場合の回帰直線が

$$t = 1.98 \times 10^{-4} s + 9.95 \quad (1)$$

であるのに対して，hash-clustering 方式の回帰直線は

$$t = 1.52 \times 10^{-6} s + 1.44 \quad (2)$$

となり，その傾きに 130 倍以上の差が生じていることが分かる．**図 11** に table *AStore* の record 数と各 iteration ごとの解析時間の散布図を示す．

なお，*HASHSIZE* を database 中の総 record 数より 1 桁少ない 1 万程度の値にした場合の 126.gcc の解析時間は約 25.5 秒，1 千程度では約 36 秒であり，いずれも nested-loop 方式の解析時間に対して 1 桁以上小さな値となる．すなわち本節の評価に用いた「*HASHSIZE* と総 record 数がほぼ等しい」という条件は，hash-clustering 方式による性能向上要因としては二義的なものである．

5.3 グラフに基づく実現との比較

著者が文献 33) で示したポインタ解析では，2 章で述べた Andersen の解析と同様に，type system による定式化を行っている．その実現はグラフと集合に基づく一般的なもので，ポインタ指示関係をグラフによって表現し，解析対象プログラムに type system を適用して得られる制約関係を 1 つずつ処理しながら，すべての制約が満たされるまでグラフにエッジを追加していくというものである．図 1 から分かるように，そこで使用される制約関係は主にメモリ参照式の持つ値（抽象 location の集合）の間の包含関係であり，間接参照式やメンバ参照式の場合は，式の構造に即して再帰的に計算を行う．グラフと集合に基づく実現は，ポインタ解析に限らずプログラム解析の実現として一般的なものであり，柔軟で効率的な方法として広く認知されている．

本節では，文献 33) で行ったグラフと集合に基づく実現方法と，本論文で提案する table に基づく実現方法の解析効率を比較する．なお両者の解析精度は，その表現方法に差はあるがほぼ同等のものである．

表 3 に，同じベンチマークプログラムに対する両方

表 3 グラフに基づく実現との比較
Table 3 Efficiency comparison with graph-based implementation.

name	graph-based(s)	table-based(s)
dhry2	0.02	0.05
a2time	0.22	0.31
130.li	1.10	2.19
124.m88ksim	0.61	1.41
126.gcc	611.10	23.33

法の解析時間を示す．表中の table-based の値は表 2 の time-tuned の値と同じものである．これを見ると，126.gcc を除く小規模なベンチマークにおいてはグラフによる実現の方が効率的であるが，126.gcc では両者の効率が大きく逆転していることが分かる．大規模なプログラムにおける急激な効率低下の直接原因は，解析過程において頻繁に行われる式の値（抽象 location の集合）の計算に必要となる，集合の union 操作に要する時間の増大である．大規模なプログラムの解析において集合の union 操作が高 cost であることはよく知られている事実であり，文献 14)，30) でも，その回数を減らすために様々な最適化が行われている．そのなかで最も効果的なのが一度行った計算結果の cache 化であるが，これは必要に応じて再計算を行ってその値の妥当性を維持するために十分な考慮を必要とする．表 3 の実現でも cache 化を行っているが，その実現の正しさについて完全な確信を得ることが難しかったため，やや穏健な再計算 policy を用いたものになっている．これを改善したり，また集合の表現方法等を工夫したりすることで，大規模なプログラムにおいても高い解析効率を維持できる可能性はあるが，本論文での table に基づく方法に比べて実現は複雑になり，その品質確保はより困難になると思われる☆．

6. 関 連 研 究

ポインタ解析の定式化に関する従来研究については 2 章ですでに述べた．本章では提案手法と関連するその他の従来研究について述べる．

6.1 deductive system によるプログラム解析

deductive system によるプログラム解析に関しては，特に論理型言語や deductive database との関連で研究が行われてきている．

Ullman は文献 29) で到達定義解析の dataflow 方程式を logic program として記述した例を示している．これはごく素朴な例であるが，deductive system によるプログラム解析の可能性を示唆し，その後の研究の発展のきっかけになっている．

Dawson らは文献 9) で，SLG 導出に基づく論理型言語 XSB [27] を用いて簡単な言語に対するプログラム解析を記述している．そこでは XSB が備えている table 化機能を用い，すでに導出した規則の instance を記憶しておくことで，4 章で触れた top down 法による閉包計算を実現している．本論文で提案したポインタ解析を XSB で記述することは可能であるが，すでに述べたとおり，現在のところ実用的な解析効率を得るのは難しい．

Horwitz らは文献 16) で，Prolog 風の言語で dataflow 方程式を記述し，それに magic set 変形を適用することで bottom up 法による on demand なプログラム解析を実現している．ただし，上に述べた Dawson らの table 化による方法よりも解析効率は低い．また magic set 変形の結果 debug が複雑になり，品質確保の点でも問題がある．

6.2 BDD の利用

従来 model checking による hardware 検証の分野で使用されてきた BDD（Binary Decision Diagram）[5] を用いて，解析が扱う大量の関係や集合を効率的に処理する研究が最近さかんに行われている [4),18),31),32)]．たとえば文献 31) では BDD を用いて 10^{14} オーダの大量の関係を扱う解析を約 20 分程度で行うのに成功している．

BDD で表現できるのは，2 値の論理関数 $Bool^N \rightarrow Bool$ なので，deductive system によるプログラム解析に BDD を利用する際には，解析が扱う関係を論理関数として encode し，各導出規則を BDD 上の論理関数の合成演算として実現する必要がある．文献 4)，31)，32) では，構造体内部を区別しない C プログラムのポインタ解析や Java プログラムのポインタ解析，escape 解析等を BDD 上で実現しているが，本論文で提案したような，複雑な構文要素を取り扱う field-sensitive な解析はまだ発表されていない．これは 2.2 節で述べた，従来手法での deductive system によるポインタ解析が単純な言語のみを対象にしているのと同じ事情による．提案手法を BDD を用いて実現する場合，構造体内部の offset 計算に必要な集合に対する加算演算，および構造体代入に必要な集合に対する比較演算の 2 つが特に問題となる．これらは今後の課題である．

6.3 解 析 速 度

ポインタ解析の速度は，対象言語の複雑さやその解析方式，測定環境に大きく左右されるため，定量的な

☆ 126.gcc を対象として行った解析の tuning や debug の苦労は本研究の大きなきっかけの 1 つである．

比較は難しい．ここでは，あくまでおおざっぱな目安として，gcc の解析に成功している☆1代表的ないくつかのポインタ解析の結果をあげるにとどめる．

現在までに発表されている Andersen 風（flow/context-insensitive, inclusion-based）のポインタ解析のうち，最も高速なのは Heintze らの解析[14] である．文献 14) によると，126.gcc の解析を field-based に行った場合の解析時間は約 0.17 秒である（Pentium 800 MHz☆2× 2 with 2 GB memory）．ただし，2 章で述べたように文献 14) の方法は，処理系依存動作を含む C プログラムに対する解析の正しさの面で問題がある．また field-based/independent な方法は field-sensitive な方法に比べて解析精度が大きく低下することが知られている[30),☆3] ．

Cheng らの解析[6),7)] は，field-sensitive な解析を実現している数少ない例である．文献 6) によると，126.gcc の解析に 520.85 秒，130.li の解析に 149.59 秒要している．また 126.gcc の解析におけるメモリ使用量は約 238 MB となっている．Cheng らの解析は若干の context-sensitivity を持っており，その点を割り引いて考える必要があるが，式の持つ値を求める部分が集合の union 操作をともなう再帰関数として定式化されており，5.3 節で述べたように，本手法のように単純な手法で効率向上を達成することは難しいと思われる．

Rountev らは文献 22) で，ポインタ解析を実施する前にプログラム変換を行い，同じ指示先集合を持つ変数を同一の変数に置換しておくことで解析速度を向上させる手法を提案している．文献 22) では，その手法を用いて Andersen 風のポインタ解析（ただし field-independent）を実装した結果を示しており，その結果は gcc の場合☆4で，変換時間が 113.3 秒，解析時間が 214.7 秒（計 328.0 秒），最大メモリ使用量は 121.4 MB である（SGI Origin 195 MHz with 1.5 GB memory）．変換を行わなかった場合の結果は 942.3 秒，347.4 MB であり，3 倍近く高速化されている．

2.1 節で触れた Foster らの定式化に基づく解析については，その後文献 11) で online cycle elimination 最適化が導入された後，文献 28) でさらに projection merging 最適化が追加され，gcc 2.8.1☆5の解析結果が示されている．結果は cycle elimination のみ実施した場合が 11445.80 秒，projection merging を加えた場合が 503.41 秒となっている（SPARC Enterprise-5000 with 2 GB memory）．

最後に，1 章で触れた Pearce らの解析は，提案手法と同じ flow/context-insensitive, inclusion-based, field-sensitive なものである[21)]．gcc を用いた評価結果は示されていないが，近い規模のプログラム gs の解析時間は 2510.4 秒となっている（Athlon 900 MHz with 1 GB memory）．彼らの定式化および実現はグラフと集合に基づくもので，特殊な indexing によりすべての抽象 location を整数値として表現する等，様々な最適化がなされているが，その解析効率については否定的である．

7. おわりに

本論文では，flow/context-insensitive, inclusion-based, field-sensitive な解析の定式化および実現方法について述べた．提案手法では式の左辺値と右辺値に関する関係を導入し，そこから間接的にポインタ指示関係を定義することで，field-sensitive な解析を deductive system として定式化することに成功している．定式化された deductive system は，導出規則を関係代数演算に書き換えることで，関係 database の基本的な最適化手法を利用し効率的に実現できる．各導出規則は実現においても互いに独立しており，これは実用的なポインタ解析処理系を構成する際に必要な品質確保を非常に容易にする．また 5 章および 6 章で示したように，Andersen 風（flow/context-insensitive, inclusion-based）の従来の他の解析と比べ，その解析効率は field-based な文献 14) の解析についで高く，解析精度は最も高い．

今後の研究課題としては，flow/context-sensitive な解析への拡張，導出規則から実現の自動生成等がある．また解析効率をさらに向上させる方法として，閉包計算の incremental 化が考えられる．incremental な閉包計算では，各 iteration で新たに生成された fact を区別し，前 iteration との差分に対して導出規則を適用する．導出規則が複数種類の fact を前提に含む場合，fact の種類ごとにその差分に対して導出規則を適用する必要があり，実現はより複雑になる．だが，たとえば 126.gcc の解析では，閉包計算の半分近くがほとんど変化しない fact 集合に対して行われており（図 10），提案手法をより大規模なプログラムに対して適用する場合，閉包計算の incremental 化は効果的であると思われる．

☆1 これは 20 万行程度のプログラムの解析が可能な実用的な方法であることを意味する．
☆2 詳細なプロセッサ名は示されていない．
☆3 文献 14) には解析精度に関する評価結果は示されていない．
☆4 バージョンは示されていない．
☆5 126.gcc は version 2.5.3 である．

本論文の直接の対象はポインタ解析であるが，提案手法の適用範囲はポインタ解析に限定されるものではない．汎用的なプログラム解析の定式化および実現の枠組みとして，その可能性と限界を調査していきたい．

謝辞 本研究に関してご討論いただいたシステム開発研究所の佐川暢俊氏，久島伊知郎氏，佐藤真琴氏，千葉雄司氏および 203 ユニットの方々に深く感謝いたします．

参 考 文 献

1) Aiken, A., Fahndrich, M. and Foster, J.S.: Partial Online Cycle Elimination in Inclusion Constraint Graphs, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.85–96 (1996).
2) American National Standard Institute, Inc.: *ANSI/ISO 9899-1990: American National Standard for Programming Languages — C* (1990).
3) Andersen, L.O.: Program Analysis and Specialization for the C Programming Language, Ph.D. thesis, DIKU, University of Copenhagen (1994).
4) Berndl, M., Lhotak, O., Qian, F., Hendren, L. and Umanee, N.: Points-to Analysis Using BDDs, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.103–114 (2003).
5) Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation, *IEEE Trans. Comput.*, Vol.C-35, No.8, pp.677–691 (1986).
6) Cheng, B.: Compile-time Memory Disambiguation for C Programs, Ph.D. thesis, University of Illinois (2000).
7) Cheng, B. and Hwu, W.W.: Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation and Evaluation, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.57–69 (2000).
8) Das, M.: Unification-based Pointer Analysis with Directional Assignments, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.35–46 (2000).
9) Dawson, S., Ramakrishnam, C.R. and Warren, D.S.: Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.117–126 (1996).
10) Embedded Microprocessor Benchmark Consortium (2000). http://www.eembc.org
11) Fahndrich, M., Foster, J., Su, Z. and Aiken, A.: A Partial Online Cycle Elimination in Inclusion Constraint Graphs, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.85–96 (1998).
12) Foster, J.S., Fahndrich, M. and Aiken, A.: Flow-insensitive Points-to Analysis with Term and Set Constraints, Technical Report UCB/CSD-97-964, Computer Science Division (EECS), University of California Berkeley (1997).
13) Foster, J.S., Fahndrich, M. and Aiken, A.: Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C, Technical Report UCB/CSD-00-1097, Computer Science Division (EECS), University of California Berkeley (2000).
14) Heintze, N. and Tardieu, O.: Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.254–263 (2001).
15) Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet?, *SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.54–61 (2001).
16) Horwitz, S., Reps, T. and Sagiv, M.: Demand Interprocedural Dataflow Analysis, *SIGSOFT Symposium on the Foundations of Software Engineering*, pp.104–115 (1995).
17) Kitsuregawa, M., Tanaka, H. and Motooka, T.: Application of Hash to Data Base Machine and its Architecture, *New Generation Computing*, Vol.1, No.1, pp.66–74 (1983).
18) Lhotak, O. and Hendren, L.: Jedd: a BDD-based Relational Extention of Java, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.158–169 (2004).
19) McAllester, D.: On the Complexity Analysis of Static Analysis, *International Static Analysis Symposium*, pp.312–329 (1999).
20) MySQL Inc. (1995). http://www.mysql.com/
21) Pearce, D.J., Kelly, P.H.J. and Hankin, C.: Efficient Field-sensitive Pointer Analysis for C, *SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.37–42 (2004).
22) Rountev, A. and Chandra, S.: Off-line Variable Substitution for Scaling Points-to Analysis, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.1–10 (2000).
23) Shapiro, M. and Horwitz, S.: Fast and Accurate Flow-Insensitive Points-To Analysis, *Symposium on Principles of Programming Lan-*

guages, pp.1–14 (1997).
24) Standard Performance Evaluation Corporation (1995). http://www.spec.org
25) Steensgaard, B.: Points-to Analysis by Type Inference of Programs with Structures and Unions, *Computational Complexity*, pp.136–150 (1996).
26) Steensgaard, B.: Points-to Analysis in Almost Linear Time, *Symposium on Principles of Programming Languages*, pp.32–41 (1996).
27) Stony Brook University (1990). http://xsb.sourceforge.net/
28) Su, Z., Fahndrich, M. and Aiken, A.: Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs, *Symposium on Principles of Programming Languages*, pp.81–95 (2000).
29) Ullman, J.D.: *Principles of Database and Knowledge-base Systems: volume II*, Computer Science Press (1989).
30) Whaley, J. and Lam, M.S.: An Efficient Inclusion-based Points-to Analysis for Strictly-typed Languages, *International Static Analysis Symposium*, pp.180–195 (2002).
31) Whaley, J. and Lam, M.S.: Cloning-based Context-sensitive Pointer Analysis Using Binary Decision Diagrams, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.131–144 (2004).
32) Zhu, J. and Calman, S.: Symbolic Pointer Analysis Revisited, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.145–157 (2004).
33) 千代英一郎：型安全でない C プログラムのポインタ解析，情報処理学会論文誌：プログラミング，Vol.45, No.12, pp.52–66 (2004).

付　　録

A.1 L_A の導出規則に対応する SQL

4 章では，図 6 の導出規則（*top*）に対応する SQL のみを示した（図 8）．その他の導出規則に対応する SQL を**図 12** に示す．

A.2 例

簡単なプログラムに対する提案手法の適用例を示す．

例 A.2.1 次の C プログラム

```
struct S { int *f1, *f2; } s, *p;
int x, y, *q, *r;
p = &s;
q = &p->f2;
*q = &x;
s.f1 = s.f2;
r = &y;
```

を解析用言語 L_A に変換したプログラム

$$P = \{\ p = \&s,\ q = \&((*p).4),\ *q = \&x,\ s.0 = s.4,\ r = \&y\ \}$$

を考える．

□

表 4 Table *AExp*
Table 4 Table *AExp*.

ExpID	ExpKind	Base	Parent	Offset	注
1	top	p	-	-	p
2	indirect	-	1	-	*p
3	field	-	2	4	(*p).4
4	top	q	-	-	q
5	indirect	-	4	-	*q
6	top	s	-	-	s
7	field	-	6	0	s.0
8	field	-	6	4	s.4
9	top	x	-	-	x
10	top	r	-	-	r
11	top	y	-	-	y

表 5 Table *Assign*
Table 5 Table *Assign*.

AssignID	AssignKind	LE	RE	注
1	address	1	6	p = &s
2	address	4	3	q = &(*p).4
3	address	5	9	*q = &x
4	simple	7	8	s.0 = s.4
5	address	10	11	r = &y

これに図 6 の導出規則を適用することで**図 13** の導出木を得ることができる．

提案手法では，図 13 と同じ結果を関係代数演算によって求める．例 A.2.1 のプログラムは，図 7 の関係 schema に基づき，**表 4**，**表 5**，**表 6** のような関係 table に変換することができる．これらの表に対して図 8 の SQL 文 *RuleTop* を適用した結果は**表 7**，さらに図 12 の SQL 文 *RuleAssignAddress* を適用した結果は**表 8** のようになる．表 8 において，複数のポインタ代入文（$p = \&s$，$r = \&y$）の作用が一度の演算で得られている点に注意されたい．これは関係代数演算による実現の大きな特徴である．

図 8，図 12 の SQL を table が変化しなくなるまで適用することで，求める結果である**表 9**，**表 10** の関係 table を得ることができる．

(平成 17 年 5 月 2 日受付)
(平成 17 年 8 月 3 日採録)

```
RuleInd = insert ignore into AEnv
    ( select    AExp.ExpID, AStore.Content
      from      AExp, AEnv, AStore
      where     ExpKind = "indirect" and Parent = AEnv.ExpID and AEnv.LocID = AStore.LocID );
RuleField = insert ignore into AEnv
    ( select    t.e, LocID
      from      ALoc,
                ( select  AExp.ExpID as e, ALoc.Base as b, ALoc.Offset + AExp.Offset as o
                  from    AExp, AEnv, ALoc
                  where ExpKind = "field" and Parent = AEnv.ExpID
                        and AEnv.LocID = ALoc.LocID ) as t
      where     t.b = Base and t.o = Offset );
RuleAssignAddress = insert ignore into AStore
    ( select    t1.LocID, t2.LocID
      from      Assign, AEnv as t1, AEnv as t2
      where     AssignKind = "address" and LE = t1.ExpID and RE = t2.ExpID );
RuleAssignSimple = insert ignore into AStore
    ( select    t1.LocID, AStore.Content
      from      Assign, AEnv as t1, AEnv as t2, AStore
      where     AssignKind = "simple" and LE = t1.ExpID and RE = t2.ExpID
                and t2.LocID = AStore.LocID );
```

図 12 各導出規則の SQL
Fig. 12 SQL for rules.

表 6 Table *ALoc*
Table 6 Table *ALoc*.

LocID	Base	Offset	注
1	p	0	(p,0)
2	q	0	(q,0)
3	s	0	(s,0)
4	s	4	(s,4)
5	x	0	(x,0)
6	r	0	(r,0)
7	y	0	(y,0)

表 7 Table *AEnv* (*(top)* 適用後)
Table 7 Table *AEnv* (*(top)* applied).

ExpID	LocID	注
1	1	p $\xrightarrow[L]{}$ (p,0)
4	2	q $\xrightarrow[L]{}$ (q,0)
6	3	s $\xrightarrow[L]{}$ (s,0)
9	5	x $\xrightarrow[L]{}$ (x,0)
10	6	r $\xrightarrow[L]{}$ (r,0)
11	7	y $\xrightarrow[L]{}$ (y,0)

表 8 Table *AStore* (*(assign address)* 適用後)
Table 8 Table *AStore* (*(assign address)* applied).

LocID	Content	注
1	3	(p,0) $\xrightarrow[R]{}$ (s,0)
6	7	(r,0) $\xrightarrow[R]{}$ (y,0)

表 9 Table *AStore* (最終結果)
Table 9 Table *AStore* (final).

LocID	Content	注
1	3	(p,0) $\xrightarrow[R]{}$ (s,0)
2	4	(q,0) $\xrightarrow[R]{}$ (s,4)
3	5	(s,0) $\xrightarrow[R]{}$ (x,0)
4	5	(s,4) $\xrightarrow[R]{}$ (x,0)
6	7	(r,0) $\xrightarrow[R]{}$ (y,0)

表 10 Table *AEnv* (最終結果)
Table 10 Table *AEnv* (final).

ExpID	LocID	注
1	1	p $\xrightarrow[L]{}$ (p,0)
2	3	*p $\xrightarrow[L]{}$ (s,0)
3	4	(*p).4 $\xrightarrow[L]{}$ (s,4)
4	2	q $\xrightarrow[L]{}$ (q,0)
5	4	*q $\xrightarrow[L]{}$ (s,4)
6	3	s $\xrightarrow[L]{}$ (s,0)
7	3	s.0 $\xrightarrow[L]{}$ (s,0)
8	4	s.4 $\xrightarrow[L]{}$ (s,4)
9	5	x $\xrightarrow[L]{}$ (x,0)
10	6	r $\xrightarrow[L]{}$ (r,0)
11	7	y $\xrightarrow[L]{}$ (y,0)

$$\frac{p = \&s \quad \dfrac{}{p \underset{L}{\longrightarrow} (p,0)}\ (top) \quad \dfrac{}{s \underset{L}{\longrightarrow} (s,0)}\ (top)}{(p,0) \underset{R}{\longrightarrow} (s,0)}\ (assign\ address) \tag{1}$$

$$\frac{q = \&((*p).4) \quad \dfrac{}{q \underset{L}{\longrightarrow} (q,0)}\ (top) \quad \dfrac{\dfrac{\dfrac{}{p \underset{L}{\longrightarrow} (p,0)}\ (top) \quad \dfrac{(1)}{(p,0) \underset{R}{\longrightarrow} (s,0)}}{*p \underset{L}{\longrightarrow} (s,0)}\ (ind)}{(*p).4 \underset{L}{\longrightarrow} (s,4)}\ (field)}{(q,0) \underset{R}{\longrightarrow} (s,4)}\ (assign\ address) \tag{2}$$

$$\frac{*q = \&x \quad \dfrac{\dfrac{}{q \underset{L}{\longrightarrow} (q,0)}\ (top) \quad \dfrac{(2)}{(q,0) \underset{R}{\longrightarrow} (s,4)}}{*q \underset{L}{\longrightarrow} (s,4)}\ (ind) \quad \dfrac{}{x \underset{L}{\longrightarrow} (x,0)}\ (top)}{(s,4) \underset{R}{\longrightarrow} (x,0)}\ (assign\ address) \tag{3}$$

$$\frac{s.0 = s.4 \quad \dfrac{\dfrac{}{s \underset{L}{\longrightarrow} (s,0)}\ (top)}{s.0 \underset{L}{\longrightarrow} (s,0)}\ (field) \quad \dfrac{\dfrac{}{s \underset{L}{\longrightarrow} (s,0)}\ (top)}{s.4 \underset{L}{\longrightarrow} (s,4)}\ (field) \quad \dfrac{(3)}{(s,4) \underset{R}{\longrightarrow} (x,0)}}{(s,0) \underset{R}{\longrightarrow} (x,0)}\ (assign\ simple) \tag{4}$$

$$\frac{r = \&y \quad \dfrac{}{r \underset{L}{\longrightarrow} (r,0)}\ (top) \quad \dfrac{}{y \underset{L}{\longrightarrow} (y,0)}\ (top)}{(r,0) \underset{R}{\longrightarrow} (y,0)}\ (assign\ address) \tag{5}$$

図 13 例題プログラム A.2.1 の導出木
Fig. 13 Derivation tree of example program A.2.1.

千代英一郎（正会員）

1999 年東京大学大学院工学系研究科情報工学専攻修士課程修了．同年日立製作所（株）入社．システム開発研究所にてコンパイラの研究開発に従事．