型安全でないCプログラムのポインタ解析

千代 英一郎†

本論文では，ポインタ型と整数型が相互に変換されるような型安全でない C プログラムのポインタ解析について述べる．キャストや共用体型の使用により生じるこのような変換の結果は処理系依存のため，従来研究では解析の対象外とするか，過度に穏健的な近似を行っている．だが組み込みシステムにみられるような，特定アドレスのメモリ領域が外部デバイスとの入出力に割り当てられている環境では，その領域を参照するためにアドレスを表す整数値からポインタ値への変換が不可欠であり，多くの組み込み用処理系では変換の過程でその値が保存されることを保証している．このような環境において安全で高精度なポインタ解析を行うためには整数値の解析が必要となるが，整数値は多種多様な演算が可能なため，その解析は一般に困難である．本論文では，最初に整数値とポインタ値の間の型変換の意味を形式的に定義し，それに基づき整数値の表しうるメモリ領域の集合を近似的に求める方法について述べる．また，このような型変換の存在のもとで構造体の内部領域を安全に区別する方法についても述べる．解析アルゴリズムは個々の整数値を追跡する代わりに，キャストおよび共用体に着目し，整数値となりうるポインタ値の集合を求めるもので，効率的な実現が可能である．

Pointer Analysis for Type-unsafe C Programs

Eiichiro Chishiro†

We present new pointer analysis for type-unsafe C programs which may contain conversions from an integer type to a pointer type, and vice versa. Since the result of such conversion caused by casts or union types is implementation-defined, it is either not treated or approximated too conservatively in most of previous work. But in the area of embedded systems, it is general that a program interacts with auxiliary devices through the memory area of a certain, system-specific address, which is designated as a pointer-converted integer constant. So many compilers for embedded systems ensure that the representation of the value is preserved through the conversion. In such environment, we need to track integer values in order to perform safe and highly precise pointer analysis, but various integer operations make it difficult to do it. In this paper, we first define formally the semantics of the type conversion between integer values and pointer values, and describe how to approximate the set of memory regions which may be designated by integer values. We also describe how to distinguish the internal region of structure objects under the existence of such conversion. Instead of tracking individual integer values, the algorithm focuses on cast expressions and unions in a program to calculate the set of pointer values which may be treated as integer, and can be implemented simply and efficiently.

1. はじめに

ポインタ解析とは，プログラム中のポインタが実行時に指すメモリ領域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，これまで多くの研究がなされてきている[3),5),7),8),13),15),16),18),19)]．

これらの提案手法の多くは ANSI C[2)] で定められた言語仕様に準拠した C プログラムを前提にしている．ANSI C の言語仕様では，整数型の値をポインタ型の値（アドレス）に型変換したときの結果は処理系定義であり，そのような動作を行うプログラムの可搬性は保証されない．そのため ANSI C に基づくポインタ解析では，対象プログラムがこのような変換を含まないことを前提にしている．

だが，組み込み環境をターゲットとするコンパイラには，整数値とポインタ値の間での型変換の結果を保証する（変換において値の表現を保存する）ものも存在する[20)]．特定アドレスのメモリ領域（定数アドレス領域）が外部デバイスとの入出力に割り当てられている環境では，この保証に基づいて，その領域を参照す

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

```
struct S { int f1, *f2, *f3; } *sp, s;
struct T { int g1, *g2; } *tp, t;
int *p, x;
...
p = (int *)0x5FFFFC0;          /* (S1) */
x = (int)&s + 4;               /* (S2) */
*p = 0x1;                      /* (S3) */
tp = (struct T *)x;            /* (S4) */
t.g1 = 5;                      /* (S5) */
t.g2 = &x;                     /* (S6) */
...
```

図 1 C プログラムの例
Fig. 1 Example C program.

るためのポインタ値を整数値からキャストして構成することができる（**図 1** の文 S1）．

このような環境においては，ポインタの指示先を求めるためにポインタ値だけでなく整数値も考慮する必要がある．だが，整数値は四則演算を含む多種多様な演算が可能であり，その解析は一般に困難である．たとえば整数型の変数間での代入は，制御フローを通して間接的に行うことが可能である☆が，これは通常のデータフロー解析では検出できない．

この問題に対する素朴な解は，整数値は不明なアドレスであり，任意のメモリ領域を表すものとして扱うことである．Andersen はポインタ解析に不明値を意味する要素（*unknown*）を導入して，すべての整数値を表現している[3]．この方法は安全であるが，値が *unknown* となるポインタが間接参照される場合に解析精度が大きく低下するという問題がある．たとえば図 1 のプログラムでは，文 S1 により p の指示先は *unknown* となるため，文 S3 での p の指示先への整数値の代入により，すべてのポインタの指示先が *unknown* となる．

一方でこの問題は構造体のような構造を持つ型の領域内部の取扱いにも関連する．C ではキャストや共用体が存在するため，構造を持つ型の取扱いは単純ではなく，構造体や共用体の各メンバを区別しない手法も多い[1),7),16)]．だがこれらの手法で整数値を扱う場合，同じ構造体のポインタ型メンバと整数型メンバが同一視されるため，整数型メンバへの整数値の代入がポインタ型メンバへの整数値（*unknown*）の代入と見なされ，前述の例と同様の問題が生じる．たとえば図 1 のプログラムでは構造体 t の整数型メンバ g1 とポインタ型メンバ g2 が区別されないため，文 S5 での g1 への整数値の代入により g2 の値が *unknown* となる．

これに対しメンバを区別する手法においては，アドレスのキャストにより領域が異なる型で参照される場合を考慮する必要がある．キャストはたとえば図 1 の文 S2 と S4 のように，整数値を介して行われる可能性があるため，領域内部の対応する位置を正確に求めるにはやはり整数値の解析が必要となる．

提案手法

我々はこの問題に対して，整数値の解析を必要とせず，かつ *unknown* を用いるよりも精度の高い近似を行う解析方法を提案する．*unknown* を用いる方法では，整数値の表す領域に関して何の制限も設けていない．だが，ポインタ解析の結果をコンパイラ最適化に利用するという目的のもとでは，解析対象プログラムはコンパイラの仕様を満たしていることを前提としてよい．本論文ではコンパイラの仕様として，多くのコンパイラにおいて暗黙的に想定されている次の条件を前提とする．

条件 1 プログラムの実行過程のある時点において，整数値 n をメモリ領域 r のアドレスとして利用する動作が定義されているのは，r のアドレスがそれまでに整数型となり，その値が n である場合に限られる．

ここでアドレスが整数型になるとは，アドレスが整数型にキャストされるか，アドレスを格納した領域が整数型として参照可能なことである．詳しくは 2 章で意味論として定義する．

たとえば，プログラム中で変数 x のアドレスが参照されていなくても，適当な整数値 n をアドレスに変換したときにその値がたまたま x のアドレスに一致していれば，n をアドレスとして用いることで間接的に x の値を操作しうる．だが条件 1 を前提とするならば，そのようなプログラムは未定義であり，したがってポインタ解析の結果の安全性を保証する必要はない．

変数の具体的なメモリ配置については言語仕様上の規定はなく，コンパイラの仕様として条件 1 を前提とすることは自然である．このときポインタ解析は，整数値の表す領域を，アドレスが整数型になるすべての領域の集合によって安全に近似することができる．これは *unknown* を用いた場合の近似値（すべての領域の集合）と比べてきわめて限定された範囲になることが期待できる．

たとえば図 1 のようなプログラムでも，p，sp，t，x のアドレスが整数型とならないことが分かれば，文 S3 で整数値を代入された p の指示先は定数アドレス領域または整数型にアドレスがキャストされた領域 s

☆ たとえば `for(i = 0; i < j; i++) ;` により代入 i = j が実現できる．

のみであり，p，sp，t，x は p の指示先にならないと解析できる．

また構造を持つ領域内部についても，異なる型で参照されうる領域（図 1 の s）と唯一の宣言型でしか参照されない領域（図 1 の t）とを区別できれば，前者は単一の領域として扱い，後者はメンバを区別することで前述の問題を回避しつつ解析精度を高めることが可能である．

本論文では以上の考察に基づき，整数値とポインタ値の間の変換が保証される環境において安全かつ高精度なポインタ解析を行う方法を提案する．解析は領域の持つ型，および整数値とポインタ値の間の型変換に着目するため，プログラム実行過程におけるこれらの意味論を最初に定義する．解析は意味論の自然な近似として導かれる．

本論文の構成は以下のとおりである．2 章では本論文で対象とする言語仕様を定義し，領域の持つ型や型変換を定式化する．3 章ではそれをもとに提案するポインタ解析方法について述べる．4 章では提案手法を実装し，ベンチマークプログラムで評価した結果を示す．5 章では関連する研究についてふれる．6 章は結論である．

2. 言 語 L

本章では C の言語仕様からポインタ解析に関わる要素を取り出したサブセット言語 L を定義し，1 章で述べた，領域が持つ型，整数値の表しうる領域を定式化する．

なお本論文では P が組 $(E_1, \ldots, E_n)$ 型のとき，その各構成要素を $E_i(P)$ と表記する．また _ により，その文脈に配置可能な任意の要素を表す．

2.1 抽 象 構 文

L の抽象構文を**図 2** に示す．説明を簡潔にするため制御構文，関数呼び出し，構造体代入等は省略している．文法はほぼ C に準じるが，C に存在する暗黙の型変換はすべて明示的に記述するものとする

L のプログラムは命令 c であり，これは代入文 $m = (\tau)e$ の列である．代入文の両辺の型は一致するものとし，左辺式 m と右辺式 e の型が異なる場合には，キャスト (τ) により明示的に型変換を記述する．式 e は定数 n，メモリ参照式 m，アドレス参照式 $\&m$，算術演算式 $e\ op\ e$，ポインタ演算式 $e \oplus e$ のいずれかである．なおポインタ演算式は第 1 オペランドがポインタ型，第 2 オペランドが整数型とする．メモリ参照式 m は変数 x，メンバ参照式 $m.f$，配列参照式 $m[e]$ [☆]，間接参照式 $*m$ のいずれかである．

Categories

$$
\begin{array}{lcl}
x & \in & Var \\
f & \in & Field \\
n & \in & N \\
op & \in & ArithOp \\
\oplus & \in & PointerOp
\end{array}
$$

Code

$$
\begin{array}{lcl}
c & ::= & m = (\tau)e \mid c; c \\
e & ::= & n \mid m \mid \&m \mid e\ op\ e \mid e \oplus e \\
m & ::= & x \mid m.f \mid m[e] \mid *m
\end{array}
$$

Type

$$
\tau \quad ::= \quad int \mid \tau* \mid \tau[n] \mid \{f_1, ..., f_n\}
$$

図 2 L の抽象構文
Fig. 2 Abstract syntax of L.

L における型 $Type$ は整数型 int，τ へのポインタ型 $\tau*$，τ 型の N 要素からなる配列型 $\tau[N]$，メンバ $f_1, ..., f_n$ からなる構造体型または共用体型のいずれかである．ここでは簡単のため基本型は int 型およびポインタ型のみであり，両者のサイズは等しいものとする．なお型宣言については C に準じるものとし，ここでは定式化しない．記述を簡潔にするため，誤解が生じない場合には構造体（共用体）の型を型宣言におけるタグ名により表記する．

L のプログラムに対して，変数のメモリ配置および式の型を対応付ける環境を定義する．変数環境 $E(x)$ は，変数 x が実行時に占める具体的なアドレス値を表す．Te，Tm，To は型環境であり，$Te(e)$ はプログラム中の式 e の型，$Tm(f)$，$To(f)$ はそれぞれ構造体（共用体）メンバ f の型およびオフセットを表す．なおすべてのメンバは一意な識別子を持つものとする．

2.2 値

次に L の意味を定義する．L の値 v の定義を**図 3** に示す．

値 v は整数値 (n, int) またはロケーション l のいずれかである．ロケーションはメモリアドレスを抽象化したもので，ベース b，相対位置 r，型 τ の 3 つ組である．

☆ L では配列参照式 $m[e]$ における m は配列型の式に限定し，左辺値として評価する（変数名はそのアドレスへと評価する）．そのため C と異なり，ポインタ p による配列参照式は `p[i]` ではなく `(*p)[i]` と記述する．また配列の添字は 1 次元に限定する．

Val

$$
\begin{array}{lcl}
v & ::= & (n, int) \mid l \\
l & ::= & (b, r, \tau) \\
b & ::= & x \mid constarea \\
r & ::= & n \mid ap \\
ap & ::= & nil \mid ap : [n] \mid ap : f
\end{array}
$$

$State$

$$
\begin{array}{lcl}
S & ::= & (M, Locs, Conc) \\
M & \in & \wp((Loc \times Val)^*) \\
Locs & \in & \wp(Loc) \\
Conc & \in & \wp(Base)
\end{array}
$$

図 3 L の値と状態
Fig. 3 Values and states of L.

ベース b は互いに重なり合わないメモリ領域を区別するもので，変数名 x または定数アドレス領域を表す特別な識別子 $constarea$ のいずれかである．なお $|b|$ により b の領域のサイズを表す．

相対位置 r は構造を持つ領域内部の相対的な位置を表す．r には各領域の持つ静的な型に基づくアクセスパス ap による表現と，領域先頭からのバイトオフセット n による表現の 2 種類が存在する．

アクセスパス ap は空要素 nil から始まる配列参照またはメンバ参照の列である．たとえば式 `s.f[2].g` が参照する領域のアクセスパスは $nil : f : [2] : g$ となる．変数がつねに唯一の宣言型で参照される言語では，アクセスパスによりすべての内部領域を区別できる．だが，C ではキャストにより任意の位置を参照可能なため，バイトオフセットによる表現も必要になる．つねに後者だけを用いて領域内部を表現することも可能だが，ここではキャストされたアドレスとそうでないアドレスを区別するために 2 種類の表現を用いる．なお $\hat{r}$ により r のアクセスパスをバイトオフセットに変換した相対位置を表す．なお，誤解が生じない限りアクセスパスの先頭の nil は適宜省略する．

ロケーションの型 τ はそのロケーションが表す領域の型である．たとえば整数型変数 `x` に対応するロケーションは (x, nil, int) となる．

定数アドレス領域は $constarea$ をベースとし，定数アドレス値をオフセットとするロケーションによって表現する．なお L においては定数アドレス領域と変数領域が重なることはないと仮定する☆．

☆ pragma 等により変数の配置アドレスを指定できるコンパイラも存在する．その場合，その変数は定数アドレス領域の部分領域として扱い，変数領域には加えない．

```
int x, y;
struct S { int *p; struct S *sn } s1, s2;
struct T { int  i; struct T *tn } *tp;

s2.p  = &x;                    /* (S1) */
s1.sn = &s2;                   /* (S2) */
tp = (struct T *)&s1;          /* (S3) */
tp->i = y * tp->tn->i;         /* (S4) */
```

図 4 キャストを用いたプログラムの例
Fig. 4 Sample program using cast.

値に型を対応付ける関数として φ_r，φ_l を以下のように定義しておく．

$$
\begin{array}{l}
\varphi_l((b, r, \tau)) = \tau \\
\varphi_r((b, r, \tau)) = \tau* \\
\varphi_r((n, int)) = int
\end{array}
$$

φ_l はロケーションを左辺値としてみたときの型，φ_r は右辺値としてみたときの型を表す．

2.3 状　　態

次に L のプログラムの状態を定義する．一般にプログラムの状態は領域が持つ値を表すが，ここでは 1 章で述べたとおり，領域が持つ型，および整数値の表す領域の集合を含めて状態を定義する．

2.3.1 有効ロケーション集合

例として**図 4** のプログラムを考える．文 S3 の実行の結果，`s1` の領域は `S` 型だけでなく `T` 型としても参照可能となる．すなわち文 S4 のように，`s1` のポインタ型メンバ `s1.p` の領域を `T` 型構造体の整数型メンバ `tp->i` としても参照できるようになる．

また複数の型を持つ領域においては，その指示先もさらに複数の型を持つようになる（間接的なキャスト）．例では文 S3 でのキャストの結果，`s1.sn` の領域は `T*` 型を持つようになり，その指示先である `s2` は `T` 型の構造体として参照可能となる．すなわちポインタ型メンバ `s2.p` の領域も整数型メンバ `tp->tn->i` によって参照できるようになる．

本論文では，このような領域の持つ型の動的な変化を表すために**有効ロケーション集合（$Locs$）**をプログラムの状態を構成する要素として導入する．$Locs(S)$ は，状態 S において領域の参照に利用可能なロケーションの集合である．

プログラムの初期状態 S_0 における有効ロケーション集合 $Locs(S_0)$ は，すべての領域をその宣言型に基づき最小のメモリ参照単位に分解してできる部分領域のロケーションの集合である．たとえば図 4 のプロ

$$update(S,l,v) = S'' \ \ where$$
$$S' = S[M \mapsto (l, conv(S, \varphi_r(l), v)) : M]$$
$$Locs(S'') = Locs(S') \cup newlocs$$
$$Conc(S'') = Conc(S') \cup newconc$$
$$newlocs = \begin{cases} \bigcup_{n\in\omega}\{locs(S')(l)(n)\}\backslash Locs(S') & if\ \varphi_l(l) \neq \varphi_r(v) \vee VT(S,l) \\ \phi & otherwise \end{cases}$$
$$newconc = \begin{cases} \{b|(b,\_,\_) = read(S', l'') \wedge l' \in newlocs \wedge l'' \simeq l' \\ \quad \wedge \varphi_l(l') = int\} \cup \{b|v = (b,\_,\_) \wedge \varphi_l(l) = int\} & if\ \varphi_l(l) \neq \varphi_r(v) \vee VT(S,l) \\ \phi & otherwise \end{cases}$$

$$locs(S)(l)(0) = \bigcup\{layout(read(S,l'))|l' \in Locs(S) \wedge l \simeq l' \wedge \varphi_l(l') = \_*\}$$
$$locs(S)(l)(n+1) = \bigcup\{layout(read(S,l'))|l' \in locs(S)(l)(n) \wedge \varphi_l(l') = \_*\}$$

$$read(S,(b,r,\tau)) = \begin{cases} v_k & if\ l_k = l \wedge \forall i \leq k.\ \neg(l_i \simeq l) \\ conv(S, \varphi_l(l), v_k) & if\ l_k \simeq l \wedge \forall i \leq k.\ \neg(l_i \simeq l) \end{cases}$$
$$where M(S) = [(l_1, v_1), ..., (l_m, v_m)]$$

$$conv(S, \tau*, (b,r,\tau')) = \begin{cases} (b, \hat{r}, \tau) & if\ \tau \neq \tau' \\ (b,r,\tau) & otherwise \end{cases}$$
$$conv(S, \tau*, (n, int)) = (b, n - E(b), \tau) \quad if\ b \in Conc(S) \wedge E(b) \leq n \leq E(b) + |b|$$
$$conv(S, int, (b,r,t)) = E(b) + \hat{r}$$
$$conv(S, int, (n, int)) = (n, int)$$

図 5 L のメモリモデル
Fig. 5 Memory model of L.

グラムでは $\{(s1,p,int*),\ (s1,sn,S*),\ (s2,p,int*),\ (s2,sn,S*),\ (tp,nil,T*),\ (x,nil,int),\ (y,nil,int)\}$ となる．有効ロケーション集合はプログラムの実行過程において，キャストもしくは複数の型（ロケーション）を持つ領域への代入により増加する．文 S3 の実行後の状態を S_3 とすると，$Locs(S_3)$ は $Locs(S_0) \cup \{(s1,\hat{p},int),\ (s1,\hat{sn},T*),\ (s2,\hat{p},int),\ (s2,\hat{sn},T*)\}$ となる．なおキャストにより生成されるロケーションはバイトオフセット表現で表すため ˆ を用いている．

本論文では複数のロケーション（型）を持つ領域を**VT 領域**，これに対し唯一の宣言型のみを持つ領域を **WT 領域**と呼び区別する．状態 S においてロケーション l が VT 領域に属するかどうかは次の述語

$$VT(S,l) \stackrel{\text{def}}{=} \exists l' \in Locs(S).l \simeq l' \wedge l \neq l'$$

により定義できる．ただし $l \simeq l'$ は l と l' が同じ内部領域を表すロケーションであることを表す．

VT 領域には，共用体のように宣言型により初期状態から複数の型を持つ領域と，実行過程において動的に複数の型を持つようになった領域の 2 種類が存在する．本論文では前者を**静的 VT 領域**，後者を**動的 VT 領域**と呼び区別する．

L では有効ロケーション集合は代入文によってのみ変化する．代入による $Locs$ の変化を定式化したものが**図 5** の $newlocs$ と $locs$ である．$newlocs$ は代入により新たに増加するロケーションの集合である．代入先のロケーションを l とすると，l がポインタ型ロケーションで，l と異なる型の値がキャストを用いて代入されるとき（$\varphi_l(l) \neq \varphi_r(v)$），もしくは l が VT 領域に属するため l と同じ領域を表す異なるポインタ型のロケーション l' が存在するとき（$VT(S,l)$），l，l' の指示先領域は新たな型（l，l' の型）で参照可能となる．$locs(S)(l)(0)$ は，それらの指示先領域を $layout$ により新しく型付けして得られるロケーションの集合である．$layout(l)$ は型の構造に基づき l の領域を分解して参照可能となるロケーションの集合を表す．なお $read(S,l)$ は状態 S における l の領域の値を表す．$layout(l)$ については，そこに含まれるロケーションはすべて l と同じ領域（ベース）に属する，という自明な性質のみを定め，ここでは具体的な定義を省略

表 1 VT/UT 領域の分類
Table 1 Classification of VT/UT region.

	静的	動的	
		明示的	非明示的
VT	共用体	アドレスが別のポインタ型にキャストされる	アドレスを格納した領域が別のポインタ型を持つ
UT	定数アドレス領域	アドレスが整数型へキャストされる	アドレスを格納した領域が整数型を持つ

する．

新たに増加したロケーションがポインタ型の場合，さらにその指示先も新たな型（ロケーション）で参照可能となる．$locs(S)(l)(n{+}1)$ は，新たに増加したロケーションの集合をもとに前述と同様の型付けを行うことで生じるロケーションの集合である．$newlocs$ はこのようにして再帰的に定義される集合族 $locs(S)(l)(n)$ の最小上界となる．

2.3.2 UT 領域集合

次に整数値が表す領域の集合を定義する．これは 1 章の条件 1 で述べた，アドレスが整数型となる領域の集合である．

本論文ではロケーション（アドレス）が整数型の値として参照可能なとき，そのロケーションの属する領域を **UT 領域**と呼ぶ．UT 領域には定数アドレス領域のように初期状態から存在する領域と，実行過程において，キャストまたは $Locs$ の増加によりそのロケーションが整数型として参照可能になる領域の 2 種類が存在する．前者を**静的 UT 領域**，後者を**動的 UT 領域**と呼び区別する．

各状態における整数値の表す領域集合として，**UT 領域集合**（$Conc$）をプログラムの状態の構成要素に導入する．プログラムの初期状態 S_0 における UT 領域集合 $Conc(S_0)$ は定数アドレス領域 $constarea$ のみである．プログラムの実行過程において，キャストによりロケーション l が整数型に変換されるか，ロケーション l を保持する領域が $Locs$ の増加により新たに整数型のロケーションを持つようになったとき，l の属する領域が UT 領域集合に含まれるようになる．たとえば図 4 では，文 S3 の実行後，新たに整数型ロケーション $(s1, \hat{p}, int)$，$(s2, \hat{p}, int)$ が参照可能となるため，$(s2, \hat{p}, int)$ と同じ領域のポインタ型ロケーション $(s2, p, int*)$ の指示先領域のロケーションは整数型としても参照されるようになる．すなわち $x \in Conc(S_3)$ となる．

これを定式化したものが図 5 の $newconc$ である．第 1 項は $newlocs$ によって増加したロケーションのうち，整数型のロケーションと重なるポインタ型のロケーションについて，その指示先領域を $Conc$ に追加している．第 2 項はキャストにより整数型に型変換されたロケーションの表す領域を $Conc$ に追加している．

以上で定義した VT/UT 領域の，生成要因による分類を**表 1** に示す．前述したロケーション集合 $locs$ の再帰的な定義は，ポインタが別の型を持つことによりその指示先も別の型を持つようになり，その結果がさらにその指示先へと波及していくという非明示的（間接的）な領域の型の変化を定式化したものになっている．

2.3.3 状態の定義

以上をもとに L のプログラムの状態を定義する（図 3）．状態 S はメモリ M，有効ロケーション集合 $Locs$，UT 領域集合 $Conc$ からなる 3 つ組である．本論文ではメモリ M はメモリに対してなされた書き込みのトレース列として表す．各書き込みは書き込み先のロケーションと書き込む値の組である．同じロケーションに対する書き込み結果は値が更新されたメモリではなく，これまでの M の先頭に新しい書き込みの組が追加されたものとなる．$Locs$ および $Conc$ はすでに述べたとおりである．

状態の更新・参照は図 5 の $update$，$read$ により定義する．$update$ は状態 S においてロケーション l に値 v を書き込んだ結果の状態 S'' を返す．S'' は，状態 S のメモリ $M(S)$ に新しい書き込み結果を追加した状態 S' に対して，増加するロケーション集合 $newlocs$ および UT 領域集合 $newconc$ を求め，それぞれ $Locs(S')$ および $Conc(S')$ に追加した結果の状態である．また $read$ は状態 S におけるロケーション l に対応する領域の値を返す．

$update$，$read$ は必要に応じて $conv$ を用いて値を型変換する．ロケーションどうしの型変換はその指示先の型のみを変更する．整数値からロケーションへの変換は，その整数値の表す領域が UT 領域集合に含まれている場合にのみ定義される．これは 1 章で述べた条件 1 を意味論として定式化したものである．ロケーションから整数値への変換は，変数環境 E によって定まる領域の具体的なアドレス値に相対位置のバイトオフセットを加えたものになる．

2.4 操作意味論

L の操作意味論を**図 6** に示す．評価規則は式の左辺値に関するもの（$\vdash_l$），式の右辺値に関するもの（$\vdash_r$），

$$\frac{}{\vdash_l < x, S > \rightarrow (x, nil, Te(x))}\ (var) \qquad \frac{\vdash_l < m, S > \rightarrow l \quad \vdash_r < e, S > \rightarrow v}{\vdash_l < m[e], S > \rightarrow array(l, v)}\ (array)$$

$$\frac{\vdash_l < m, S > \rightarrow l}{\vdash_l < m.f, S > \rightarrow field(l, f)}\ (field) \qquad \frac{\vdash_r < m, S > \rightarrow l}{\vdash_l < *m, S > \rightarrow l}\ (ind) \qquad \frac{}{\vdash_r < n, S > \rightarrow (n, int)}\ (const)$$

$$\frac{\vdash_l < m, S > \rightarrow l}{\vdash_r < \&m, S > \rightarrow l}\ (address) \qquad \frac{\vdash_l < m, S > \rightarrow l \quad l \in Locs(S)}{\vdash_r < m, S > \rightarrow read(S, l)}\ (mem)$$

$$\frac{\vdash_r < e_1, S > \rightarrow v_1 \quad \vdash_r < e_2, S > \rightarrow v_2}{\vdash_r < e_1\ op\ e_2, S > \rightarrow eval(op, v_1, v_2)}\ (op) \qquad \frac{\vdash_r < e_1, S > \rightarrow l_1 \quad \vdash_r < e_2, S > \rightarrow v_2}{\vdash_r < e_1 \oplus e_2, S > \rightarrow pointer(l_1, v_2)}\ (pointer)$$

$$\frac{\vdash_l < m, S > \rightarrow l \quad \vdash_r < e, S > \rightarrow v}{\vdash_c < m = (\tau)e, S > \rightarrow update(S, l, v)}\ (assign) \qquad \frac{\vdash_c < c_1, S > \rightarrow S_1 \quad \vdash_r < c_2, S_1 > \rightarrow S_2}{\vdash_c < c_1; c_2, S > \rightarrow S_2}\ (seq)$$

$$field(l, f) = \begin{cases} (b, n + To(f), Tm(f)) & if\ l = (b, n, \tau) \\ (b, ap : f, Tm(f)) & if\ l = (b, ap, \tau) \end{cases}$$

$$array(l, v) = \begin{cases} (b, n + |\tau| * v, \tau) & if\ l = (b, n, \tau[\_]) \\ (b, ap : [v], \tau) & if\ l = (b, ap, \tau[\_]) \end{cases}$$

$$pointer(l, v) = \begin{cases} (b, ap : [i + v], \tau) & if\ l = (b, ap : [i], \tau) \\ (b, n + v * |\tau * |, \tau) & if\ l = (b, n, \tau) \end{cases}$$

図 6 L の操作意味論
Fig. 6 Operational semantics of L.

および命令に関するもの（$\vdash_c$）からなる．これは Andersen による定式化[3] とほぼ同様であるが，前節で述べた状態の定義および 2 種類のロケーションを区別する点で異なっている．

規則 (var) は変数の左辺値を定めている．規則 ($field$) は構造体（共用体）メンバの左辺値を関数 $field$ によって定めている．$field$ はメンバ f の親の左辺値 l がオフセット表現の場合には f のオフセットを加算し，アクセスパス表現の場合にはその末尾に f を加えたロケーションを返す．この扱いは配列要素の左辺値を定める規則 ($array$)，ポインタ算術演算の結果を定める規則 ($pointer$) についても同様である．なお明示的にキャストされない限り，ポインタ算術演算の対象となるロケーションは配列の範囲内の配列要素型であり，そうでない動作は ANSI C と同様に未定義とする．

規則 (op) は算術演算の評価を $eval$ により定めている．$eval$ の内容については，整数型の結果を返すこと以外の詳細は定めない．規則 (mem) はメモリ参照式の右辺値を定めている．読み出すロケーションは現在の状態 S の $Locs$ に含まれている必要がある．規則 ($assign$) は代入文の評価を定めている．これらの規則で用いられている $read$，$update$ については前節で述べたとおりである．

3. 解析方式

前章では，領域の持つ型をもとに，アドレスが整数型となりうる領域，すなわち整数値をアドレスとみたときにそれが表しうる領域の集合（UT 領域集合 $Conc$）を定義した．整数値がポインタとして利用される場合，その指示先が UT 領域集合を含むように近似することで，安全なポインタ解析の結果を得ることができる．だが，前章の図 5 における定義に基づいて UT 領域集合を求めるためには整数値の解析が必要であり，これは一般には困難である．

本章では，整数値を解析することなく UT 領域集合の安全な近似集合を求め，これを整数値の表す領域の集合とするポインタ解析方式について述べる．UT 領域集合は，表 1 での分類における非明示的な生成要因のため，複数の型を持つ領域の集合（VT 領域集合）に依存しており，解析の際には VT 領域集合をあわせて求める必要がある．領域が複数の型を持つかどうか

は，構造体や配列のような，構造を持つ型の領域内部の区別にも関連しており，VT 領域集合を用いてその解析精度をあげることが可能である．

提案するポインタ解析方式のうち，VT/UT 領域の扱いに関する部分以外については，おおむね Andersen の方式[3] に基づいている．これはプログラムの制御の流れを考慮しないフロー非依存な解析であり，プログラム中の各代入文に対して，左辺の表す領域が右辺の表すロケーションを指示先に含むという制約関係を生成し，その最小解を解析結果として求めるものである．

3.1 表　　現

最初にポインタおよびその指示先の表現について考える．個々の変数領域および定数アドレス領域はその識別子（ベース）により区別可能だが，構造を持つ型の領域内部については，粒度の異なるいくつかの方法が存在する．

最も粗い方法はすべての領域は単一の領域であると見なし，その内部の区別を行わないものである[7),10),16)]．この方法では同じ構造体に属する整数型メンバとポインタ型メンバが同一領域にあるものと見なすため，1 章で述べたとおり，整数型メンバに整数値が代入されたときに *unknown* の伝播による解析精度の低下が生じる．また整数値を無視しうる場合でも，構造体メンバを区別する手法に比べて解析精度が大きく低下するという報告がある[11),19)]．

次に粗い方法は，構造体の各メンバは区別するが，配列は単一領域と見なしその内部を区別しないものである[15),19)]．この方法では，配列内部にある構造体の各メンバを区別できないことによる精度低下が生じる．

Wilson らは宣言型に関する情報を用いず，領域内部をオフセット o とストライド s の組により区別する方法を提案している[17),18)]．(o, s) の表す領域は領域先頭から $o + ns$（n は整数）の相対位置にある部分領域の集合である．ストライド s は配列要素参照される領域を表すための値で，あるオフセット o に対して配列要素参照がなされた場合，o に配列要素サイズ s の整数倍（負の値を含む）を加えた位置がすべて参照されうると見なす．これは安全ではあるが，配列範囲を超えた参照は未定義とする言語仕様のもとでは過度に穏健的である．またこの結果，整数型の配列を含む領域はその全体が整数型で参照可能であると見なされ，*unknown* の伝播による解析精度の低下が生じる．

Cheng らはメンバはオフセットおよびサイズで表し，配列添字は無視することで領域内部を区別している[5)]．これは本質的にはすべての配列要素をその先頭要素で代表しているとみることができる．この方法は

```
union U {
    struct S {
        int *f1, *f2, *f3, *f4;
    } s;
    struct T {
        int *g1, *g2;
    } a[2];
} u;
```

図 7 構造体・共用体・配列の組み合わせ
Fig. 7 Combination of struct, union and array.

配列要素を区別せず，メンバを区別するという点で望ましい性質を持っているが，キャストや共用体が存在している場合に解析結果が安全でないという問題がある．

たとえば**図 7** のプログラムにおいて，`u.s.f3` と `u.a[1].g1` は同じ領域を参照する．だが，配列添字を無視した表現では，前者は $u.8\_4$，後者は $u.0\_4$（o_s はオフセット o，サイズ s の領域を表す）という異なる表現になってしまう．

Cheng らの方法が問題となるのは，キャストや共用体により異なる型で参照されうる領域を表現する場合である．本論文で導入した WT/VT 領域の区別を利用すると，WT 領域については Cheng らと同様の方法で内部領域を詳細に区別し，VT 領域については単一領域と見なし内部領域を区別しないようにすることで，解析精度と安全性を両立させることが可能となる．

次節では，この議論をもとに提案するポインタ解析方式を定式化する．

3.2 解析の定式化

ポインタ解析では，具体的なメモリおよびロケーション（アドレス）を抽象化して扱うのが一般的である．本論文では，両者を抽象化したものをそれぞれ抽象メモリ AM および抽象ロケーション $ALoc$ と呼ぶ．ポインタ解析の結果は抽象メモリ AM であり，これはポインタ（の領域を表す抽象ロケーション）からその指示先（の領域を表す抽象ロケーション）集合への写像である．

具体的なロケーションを抽象ロケーションに対応付ける関数を抽象化関数 α と呼ぶ．3.1 節で述べた領域内部の区別に関する各手法の違いは，抽象化関数の定義の違いとして把握することができる．たとえば構造体メンバを区別しない手法では，その抽象化関数はすべてのメンバのロケーションを同じ抽象ロケーションに対応付けている．

提案方式では整数値の表す領域を精度良く近似するため，また 3.1 節で述べた方針に基づいて領域内部

$ALoc$

$$al \quad ::= \quad (b, o, d)$$

$AState$

$$\begin{array}{lll} AS & ::= & (AM, D, U) \\ AM & \in & ALoc \to \wp(ALoc) \\ D & \in & \wp(Base) \\ U & \in & \wp(Base) \end{array}$$

図 8 解析空間
Fig. 8 Analysis domain.

を区別するため，抽象メモリに加えて VT 領域集合および UT 領域集合の近似集合を解析する．VT 領域集合のうち，静的 VT 領域集合（共用体）については，抽象化関数の定義に組み込んで扱うことができる．一方動的 VT 領域集合については，ポインタ解析の結果と相互に依存しているため，並行して解析する必要がある．本論文では，これを抽象化したものを**抽象動的 VT 領域集合**（D）と呼ぶ．また UT 領域集合を抽象化したものを**抽象 UT 領域集合**（U）と呼ぶ．提案方式におけるポインタ解析ではこれらの 3 つ組 (AM, D, U) からなる抽象状態（**図 8**）を求める．

以下，解析を構成する各要素について詳しく述べる．

3.2.1 抽象ロケーション

提案方式における抽象ロケーション al はベース b，抽象オフセット o，それに VT 領域に属しているかどうかを表す属性フラグ d の 3 つ組である．抽象オフセット o は領域内部を区別するためのもので，領域先頭からの相対位置を表す．なお属性フラグ d の意味については後述する．

ロケーションを抽象ロケーションに対応付ける抽象化関数 α の定義を**図 9** に示す．α は 3.1 節の議論に基づき，ロケーションのベース b が動的 VT 領域に属する場合には内部領域を区別せず，すべて同じ抽象ロケーション $(b, 0, T)$ に対応付ける．また静的 VT 領域に属する場合，すなわち共用体内部にある場合には，その内部位置を区別せず，すべて共用体の先頭を表す抽象ロケーションに対応付ける．配列はすべての要素をその先頭要素で代表する．$index$ はアクセスパスに対応する抽象オフセットを求める関数で，配列添字は無視し，アクセスパスの途中に共用体型のメンバがあればその先はたどらずにそこまでの値を抽象オフセットとしている．なお in_union はロケーションが共用体内部に属することを表す述語である．

図 10 に抽象ロケーションの例を示す．なお $s \notin D$ とする．`s.u` の内部は共用体（静的 VT 領域）のため，

$$\begin{array}{ll} \alpha((b, \_, \_)) = \{(b, 0, T)\} & if\ b \in D \\ \alpha((b, ap, \_)) = & \\ \quad \{(b, index(b, ap), in\_union(b, ap))\} & otherwise \\ \alpha((n, int)) = \{(b, 0, T) | b \in U\} & \end{array}$$

$$\begin{array}{l} index(b, nil) = 0 \\ index(b, ap : [i]) = index(b, ap) \\ index(b, ap : f) = \end{array}$$

$$\begin{cases} index(b, ap) & if\ in\_union(b, ap) \\ index(b, ap) + To(f) & otherwise \end{cases}$$

図 9 抽象化関数
Fig. 9 Abstraction function.

```
struct S {
    struct T {              /* ALoc     */
        int *f1;            /* (s,0,F)  */
        int *f2;            /* (s,4,F)  */
    } a[2];
    union U {
        int g1, *g2;        /* (s,8,T)  */
    } u;
    int *h;                 /* (s,12,F) */
} s;
```

図 10 抽象ロケーションの例
Fig. 10 Example of abstract location.

`s.u.g1`, `s.u.g2` の抽象ロケーションはともに $(s, 8, T)$ となる．また構造体配列 `s.a[]` の内部は，配列添字を捨象し先頭要素におけるメンバのオフセットで区別する．すなわち `s.a[0].f1`，`s.a[1].f1` は $(s, 0, F)$，`s.a[0].f2`，`s.a[1].f2` は $(s, 4, F)$ となる．

3.2.2 抽象動的 VT 領域集合 D

α はロケーションが VT 領域に属するかどうかという情報をもとに抽象ロケーションを定めている．前述したように，動的 VT 領域を正確に求めるためにはポインタ解析の結果が必要だが，ポインタ解析を正確に行うためには動的 VT 領域の解析が必要となる．提案方式では動的 VT 領域は存在しないという仮定（$D = \phi$）のもとでポインタ解析と動的 VT 領域の解析を並行して行い，両者の最小不動点を解として求める．

2 章での議論および図 5 の $newlocs$ の定義から，動的 VT 領域となりうるのは，ロケーションがキャストされた領域，および VT 領域からその指示先をたどり到達可能な領域であることが分かる．提案方式では

これらの領域の集合を D とし，動的 VT 領域集合を近似する．正確には，動的 VT 領域となるのは異なる型のロケーションが存在する領域であり，アップキャストのような互換性のある型変換によって動的 VT 領域が増加することはないが，ここではすべての型変換を互換性のない型変換と見なした安全な近似を行っている．

3.2.3 抽象 UT 領域集合 U

2 章での議論から，整数値の表す領域は UT 領域集合に包含されることが分かる．抽象 UT 領域集合 U は UT 領域集合の近似集合である．L では静的 UT 領域となるのは定数アドレス領域のみであり，U は $constarea$ を含む必要がある．動的 UT 領域については，$newconc$ の定義から，ポインタ型から整数型へのキャストが存在する場合（明示的な生成要因），および代入によって増加した整数型ロケーションの領域がポインタ値（ロケーション）を保持している場合（非明示的な生成要因）に増加することが分かる．

動的 UT 領域が増加する要因のうち，前者についてはロケーションが整数型へキャストされた領域の集合，後者については VT 領域の指示先集合によって安全に近似できる．提案方式ではこれらの和集合を U とする．

3.2.4 抽象メモリ AM

AM は抽象ロケーションの表す領域の持つ値を表す．$AM(al)$ の値は，al が VT 領域に属さない（属性フラグが F である）場合には al ごとに独立の値となるが，al が VT 領域に属する可能性がある（属性フラグが T である）場合には，すべて共通の値（UT 領域を表すすべての抽象ロケーションの集合）となる．この近似の理由は以下のとおりである．

VT 領域の値を読み出す場合，型によっては $conv$ によって整数型からポインタ型に変換されたロケーションを読み出す可能性がある．図 5 の $read$，$conv$ の定義から，そのロケーションとなりうるのは UT 領域に属するものに限られる．すなわち

VT 領域の指示先 $\supseteq$ UT 領域集合

である．一方 VT 領域は複数の型を持つため，その指示先も宣言型と異なる型で参照されうる．そのため，ポインタ型の領域が整数型として参照されうることになる．$newconc$ の定義から，VT 領域のロケーションから到達可能なすべての領域は UT 領域となりうることが分かる．すなわち

VT 領域の指示先 $\subseteq$ UT 領域集合

である．以上から

VT 領域の指示先 $=$ UT 領域集合

という近似が導かれる．

この結果は，メモリ領域間の指示関係が，各領域の種別により限定されることを意味している．WT 領域のポインタはすべての領域を指しうるが，VT 領域のポインタは UT 領域のみを指す．また UT 領域に属さない VT 領域は WT 領域のみから指され，他の VT 領域から指されることはない．

3.3 型システムによる実現

図 11 はこれまでの議論をもとにポインタ解析を標準的でない型システムとして定式化したものである．型付け規則はそれぞれ図 6 の意味論の各規則に対応している．これは Andersen による定式化[3] と同様である．

式の型の直感的な意味はその式の評価値を抽象化したものである．規則 $(pvar)$ は変数 x の抽象ロケーションを定めている．属性フラグ d は x が VT 領域に属しているかどうかを表す．規則 $(parray)$ は配列要素を区別しないことを表している．規則 $(pfield)$ は，構造体 m の抽象ロケーションが VT 領域に属しているかを属性フラグにより区別し，VT 領域に属していなければ，そのメンバ参照式 $m.f$ の抽象ロケーションは m のオフセットに f のオフセットを加えたものになることを表している．メンバが共用体であれば，その内部の抽象ロケーションは VT 領域に属することになる．また m の抽象ロケーションが VT 領域に属している場合には，その内部領域は区別せず，オフセットは変化しない．規則 $(pind)$，$(paddress)$ は対応する意味論の規則 (ind)，$(address)$ と同じである．規則 $(pconst)$，(pop) をみると明らかなように，整数値はすべて $U(AS)$ によって近似される．規則 $(pmem)$ は，メモリ参照式 m の値は m の表すすべての抽象ロケーションの持つ値の和集合として近似することを示している．規則 $(ppointer)$ はポインタ算術演算においても配列要素を区別しないことを表している．指示先が VT 領域に属する場合，指示先は配列要素に限定されないが，VT 領域の内部は区別しないので同じ結果になる．規則 $(passign)$ は，代入の効果を付帯条件により表している．最初の条件は，左辺の領域の値が右辺の領域の値を包含することを定めている．また 2 番目の条件は整数型にキャストされた抽象ロケーションのベースが UT 領域に含まれること，3 番目の条件は異なるポインタ型にキャストされた抽象ロケーションのベースが VT 領域に含まれることを定めている．付帯条件から，キャストのない整数型の代入文に関しては何の制約もなく，したがって解析が不要であることが分かる．

$$\frac{}{AS \vdash_{pl}\ x : \{(x, 0, x \in D(AS) \vee is\_union(x))\}}\ (pvar) \qquad \frac{AS \vdash_{pl}\ m : A}{AS \vdash_{pl}\ m[e] : A}\ (parray)$$

$$\frac{AS \vdash_{pl}\ m : A}{AS \vdash_{pl}\ m.f : \{(b, n + To(f), is\_union(f)) | (b, n, F) \in A\} \cup \{(b, n, T) | (b, n, T) \in A\}}\ (pfield)$$

$$\frac{AS \vdash_{pr}\ m : A}{AS \vdash_{pl}\ *m : A}\ (pind) \qquad \frac{}{AS \vdash_{pr}\ n : U(AS)}\ (pconst) \qquad \frac{AS \vdash_{pl}\ m : A}{AS \vdash_{pr}\ \&m : A}\ (paddress)$$

$$\frac{AS \vdash_{pl}\ m : A}{AS \vdash_{pr}\ m : \bigcup\{AM(AS)(al) | al \in A\}}\ (pmem) \qquad \frac{}{AS \vdash_{pr}\ e_1\ op\ e_2 : U(AS)}\ (pop)$$

$$\frac{AS \vdash_{pr}\ e_1 : A_1}{AS \vdash_{pr}\ e_1\ \oplus\ e_2 : A_1}\ (ppointer)$$

$$\frac{AS \vdash_{pl}\ m : A_1 \quad AS \vdash_{pr}\ e : A_2}{AS \vdash_{pc}\ m = (\tau)e}\ (passign)$$

$$\begin{aligned} where \quad & Te(m) = \_* \Rightarrow \forall al \in A_1.\ A_2 \subseteq AM(AS)(al) \\ & Te(m) \neq Te(e) \wedge Te(m) = int \wedge Te(e) = \_* \Rightarrow \{b | (b, \_, \_) \in A_2\} \subseteq U(AS) \\ & Te(m) \neq Te(e) \wedge Te(m) = t_1* \wedge Te(e) = t_2* \wedge t_1 \neq t_2 \Rightarrow \{b | (b, \_, \_) \in A_2\} \subseteq D(AS) \end{aligned}$$

$$\frac{AS \vdash_{pc}\ c_1 \quad AS \vdash_{pc}\ c_2}{AS \vdash_{pc}\ c_1; c_2}\ (pseq)$$

$$\begin{aligned} & constarea \in U(AS) && (constarea) \\ & AM(AS)((\_, \_, T)) = \{(b, 0, T) | b \in U(AS)\} && (U) \\ & U(AS) \subseteq D(AS) && (D) \end{aligned}$$

図 11 ポインタ解析のための型システム
Fig. 11 Type system for pointer analysis.

型規則のほかにポインタ解析の解 AS に関する制約として，規則 ($constarea$) により定数アドレス領域が $U(AS)$ に含まれること，規則 (U) により VT 領域の指示先は $U(AS)$ に含まれるすべての領域 b の抽象ロケーション $(b, 0, T)$ の集合となること，規則 (D) により UT 領域は VT 領域になることを定めている．これらは前節で述べた議論をそのまま表したものである．

たとえば図 4 のプログラムに対する最小解は

$$\begin{aligned} & D(AS) = \{constarea, s1, s2, x\} \\ & U(AS) = \{constarea, s2, x\} \\ & AM(AS)((tp, 0, F)) = \{(s1, 0, T)\} \\ & AM(AS)((s1, 0, T)) = AM(AS)((s2, 0, T)) \\ & \quad = \{(constarea, 0, T), (s2, 0, T), (x, 0, T)\} \end{aligned}$$

となる．図 4 のプログラムでは，`y` の値が 1 のとき，文 S4 の代入により `s1.p` は `x` を指す．本解析では `y * tp->tn->i` のような整数型の式の値を個別に追跡するのではなく，文 S3 と規則 ($passign$) により `s1` が $D(AS)$ に含まれることと，文 S1，S2 と規則 ($passign$)，(U)，(D) により `s2`，`x` が $U(AS)$ に含まれることから，`s1.p` が `x` を指示することを導き出している．

4. 評　　価

本章では 3 章で述べたポインタ解析を SHC コンパイラ[20] に実装し，評価を行った結果を示す．評価に用いた実装は，解析対象プログラムの各ソースファイルごとに，そこで定義されている関数に含まれるポインタに関する代入文，コールサイト，キャスト式をサマリファイルへ出力した後，プログラム全体のサマリファイルをもとにポインタ解析を行うという構成をとっている．

関数呼び出しにおける実引数と仮引数の対応付けは Cheng らの方法[5] と同様に，インタフェース用の一時

表 2 ベンチマークプログラムの特徴
Table 2 Characteristics of benchmark programs.

program	#line	#stmt	#callsite (indirect)	#assign pointer	#assign other	#cast pointer	#cast other
dhrystone2	621	186	72(0)	29	204	57	2
099.go	28547	14043	2083(0)	9	5931	33	0
124.m88ksim	17940	11145	1511(3)	710	6373	980	82
129.compress	1424	620	63(0)	88	346	70	15
130.li	6916	3779	1241(4)	1561	489	645	99
bench1	137231	5399	37(0)	178	3566	118	1
bench2	378816	40230	4133(15)	5831	24085	4614	2757
EEMBC/auto_indy							
harness	9843	3019	510(20)	333	1419	373	200
a2time01	2153	423	19(0)	16	315	13	2
aifftr01	9194	330	26(0)	29	210	13	2
aifirf01	3347	387	28(0)	143	182	49	2
aiifft01	9119	297	23(0)	29	192	13	2
bitmnp01	2204	590	29(0)	47	321	28	20
basefp01	4964	179	16(0)	19	110	14	3
cacheb01	1302	275	28(1)	122	110	39	2
canrdr01	4369	349	25(0)	50	222	22	2
idctrn01	17478	478	91(0)	17	341	13	2
iirflt01	3506	543	34(0)	167	254	43	2
matrix01	1527	704	49(0)	162	365	196	67
pntrch01	1020	205	28(0)	20	115	22	2
puwmod01	6026	305	25(0)	17	172	13	2
rspeed01	1780	159	16(0)	17	101	13	2
tblook01	905	164	17(0)	22	85	16	2
ttsprk01	26507	467	77(0)	48	295	37	2

変数を導入し両者の間の代入文を生成することで行っている．また標準ライブラリ関数については同じ処理を行うスタブ関数を用意し，ベンチマークプログラムとあわせて解析している．メモリ確保関数が返すヒープ領域については，*heap* という単一の領域を用意し，つねにそれを返すようにしている．これについてはメモリ確保関数の呼び出し元ごとに別の領域を割り当てるようにすることでさらに解析精度をあげることが可能である．

解析アルゴリズムは，3 章で記述した型規則による制約条件を，抽象ロケーションをノード，指示関係をエッジとするグラフで表現し，反復解法により解くものである．ポインタどうしの代入は Heintze らの方法[11] と同様に依存エッジで表現し，指示先集合が必要になったときに依存エッジから推移的閉包を求めている．

これらはすべて C で記述し，SHC コンパイラ Ver.8 のデータフロー解析部の 1 機能として実装している．解析の実行は Hitachi FLORA 330 DK4（PentiumIII 1.13 GHz，メモリ 512 MB）上で行っている．

表 2 は評価に用いたプログラムの特徴である．#line はプログラムの行数（コメント，空行を含む），#stmt はコンパイラ中間語における文の数，#callsite は静的な関数呼び出しの数（かっこ内は間接呼び出しの数），#assign はコンパイラ中間語におけるポインタ代入文（pointer）およびそれ以外の代入文（other）の数，#cast はポインタ型へのキャスト（pointer）および非ポインタ型へのキャスト（other）の数を表す．

ベンチマークに利用したプログラムは dhrystone ver. 2.1，SPEC[14] Cint95 の一部，組み込み用実アプリケーション（bench1，bench2）および組み込み分野の標準ベンチマーク EEMBC[9] Automotive/Industrial（auto_indy）である．SPECint95 プログラムは，SHC が未提供の標準ライブラリの使用や制限値を超える条件式ネスト等の理由により，そのままではコンパイルできない．そのため測定に使用したプログラムは動作に影響を与えない範囲でソースを修正している．実アプリケーション bench1，bench2 は，SHC コンパイラ開発グループが所有している自動車・家電等のアプリケーション群から無作為に選択したものである．auto_indy は harness と呼ばれる共通のテスト用関数群と個々のベンチマークプログラムからなる．表 2 では両者を分けて記述しているが，解析は両者をあわせて行っている．

#assign における pointer の数と other の数を比べると，ポインタ代入文以外を無視できるようにポイン

表 3 解析結果
Table 3 Analysis result.

program	#dereference-set 1	2	3	4≤	U	#U	#address	#U / #address	analysis time(s)
dhrystone2	37	0	0	0	11	1	7	0.14	0.02
099.go	4	5	2	11	0	1	748	0.00	0.34
124.m88ksim	270	113	24	234	564	69	961	0.07	0.61
129.compress	15	7	2	0	0	3	20	0.15	0.02
130.li	23	0	3	40	1536	89	638	0.14	1.10
bench1	88	18	0	2	2	1	252	0.00	0.01
bench2	206	68	4	2	800	194	648	0.30	6.07
EEMBC/auto_indy									
a2time01	111	20	1	12	111	12	90	0.13	0.22
aifftr01	123	20	3	12	111	12	96	0.13	0.22
aifirf01	111	20	1	12	122	14	90	0.16	0.24
aiifft01	123	20	3	12	111	12	96	0.13	0.23
bitmnp01	114	20	1	12	114	16	94	0.17	0.20
basefp01	111	20	1	12	111	12	90	0.13	0.23
cacheb01	111	20	1	12	160	12	90	0.13	0.24
canrdr01	111	23	1	12	112	18	129	0.14	0.22
idctrn01	111	20	1	12	111	12	90	0.13	0.21
iirflt01	111	20	1	12	124	14	90	0.16	0.23
matrix01	111	20	1	12	121	12	90	0.13	0.25
pntrch01	111	20	1	12	117	12	90	0.13	0.21
puwmod01	111	20	1	12	111	12	90	0.13	0.20
rspeed01	111	20	1	12	111	12	90	0.13	0.20
tblook01	114	20	1	12	113	12	93	0.13	0.19
ttsprk01	112	21	2	14	127	12	105	0.11	0.22

タ解析を設計することの効果が分かる☆．130.li を除くと代入文全体に対するポインタ代入文の割合は 0〜30%程度であり，解析の計算量が線形オーダの場合，数倍から数十倍の効率向上が期待できる．

表 3 はポインタ解析の結果である．ポインタ解析の精度は間接参照されるポインタの指示先の数で評価している．#dereference-set は，間接参照式をその対応する領域の数で分類したものである．領域の数が 1 の間接参照式は，そこに含まれるポインタの指示先が完全に正確に解析できていることを表す．U は対応する領域が UT 領域となる間接参照式の数である．130.li では U の数が 1536 と突出しており，#dereference-set の平均はほぼ UT の数（89）に近くなる．これらのほとんどはヒープ領域のメモリセルを参照する式であり，整数値とポインタ値の相互変換を考慮しない従来のポインタ解析においても，130.li の#dereference-set の平均は数百のオーダになることが報告されている[10]．ただし，中間語や指示先の表現方法等が異なるため，数値の厳密な比較は困難である．

解析に UT 領域を導入したことによる効果は，UT 領域の数#U とアドレスをとられた領域の数（#address）を比較することで評価している．後者は，不明値（*unknown*）を持つポインタの指示先としてよく用いられる近似集合である．前者の後者に対する割合は 0〜30%の範囲にあり，平均して約 12%である．これはキャストや共用体の影響により指示先不明となるようなポインタ（#dereference-set の U 欄に分類されるもの）の指示先に関して，解析精度が大幅に改善されることを示している．表中で bench2 の比率が 0.30 と突出しているのは，変数のアドレスを `unsigned long` 型にキャストし，整数の算術演算によりオフセットを調節した後にポインタ型に戻すというコーディングが随所でなされているためである．このような実アプリケーションの存在は，整数値とポインタ値の間の変換を保証する環境における安全なポインタ解析の必要性を示している．

表 3 の analysis time はポインタ解析に要する時間（秒）である．解析時間はほぼ 1 秒以下で，最大のプログラム bench2（約 30 万行）でも 6.07 秒となっている．表 2 と比較すると，プログラム中のポインタ代入文の数に対する解析時間はほぼ線形オーダであり，本解析の実行効率が実用上問題ない範囲であることが分かる．

☆ EEMBC/harness 以下の各ベンチマークについては harness との和がプログラム全体の値となる．

5. 関 連 研 究

ポインタ解析については多くの研究がなされてきている．本論文のポインタ解析の枠組みはフロー非依存であり，かつ WT 領域については代入によって包含関係の制約が生成される点で Andersen の方法[3] に分類される．これは代入による制約を指示先の等値関係とする Steensgaard の方法[16] に比べて解析時間はかかるが解析精度は高くなることが知られている[7),10),13)]．

構造を持った領域内部の表現に関してはすでに 3 章で述べた．本論文の方式は WT 領域については変数もメンバもともに区別し，VT 領域については単一の領域として扱うという 2 種類の粒度を併用しており．WT 領域については最も正確な区別が可能となっている．

構造を持った領域内部の区別とキャストの相互作用については，Steensgaard[15] および Yong ら[19] の研究でも取り上げられている．これらは互換性のない複数の型で参照される領域を単一の領域として取り扱う方法を述べているが，いずれも定数アドレス領域を用いるプログラムを安全に解析することはできない．またどちらも配列を単一領域として扱っており，本論文の手法にくらべて内部領域の表現の粒度が粗い．Yong らの方法は 130.li の解析に 7 時間かかることが報告されており，スケーラビリティの面で問題があると思われる．

Chandra らは C プログラムでのキャストの使用傾向を調べ，オブジェクト指向言語の継承機能を C で実現するために使用されるキャストについて，キャストする前の型とキャスト後の型の物理的なメモリレイアウトが正しくサブタイプ関係（physical subtype 関係）になっていることを検証する方法を提案している[4]．本論文の解析方式においても physical subtype 関係を解析し，そのようなキャストでは領域の持つ型が変化しないものとして扱うことでより解析精度を高めることが可能である．

Necula らは型安全でない C プログラムを安全に実行するため，CCured と呼ばれる C の型システムの拡張を提案している[6),12)]．CCured では C プログラムのポインタ型を，宣言型通りに用いるもの（SAFE），配列を指すもの（SEQ），キャストにより宣言型以外で用いられるもの（WILD）に分類する．コンパイラはポインタの間接参照式に対して，それぞれのポインタ型に応じた実行時チェックを挿入することで型安全性を保証することができる．CCured はポインタの指示先を求めるものではない（ポインタ解析ではない）が，唯一の宣言型で参照される（型安全な）領域と，そうでない領域を区別する点で本論文と同じ方向性を持っている．大きな相違点は，本論文では領域を直接区別しているのに対し，CCured では領域を指すポインタの種類を区別している点である．CCured で WILD と解析されるポインタが指す領域は，本解析における VT 領域と UT 領域に対応し，両者を区別することができないため，提案手法のように整数値の表す領域を精度良く求めることはできない．

最後に，本研究とほぼ同時に発表されたため参考にはできなかったが，関口は抽象解釈に基づき領域内部の区別および整数値の処理を行うポインタ解析方法を提案している[21]．関口の方法では領域内部を整数区間の一次式によって表現する．これは 3.1 節で述べた Wilson らの方法に，範囲を制限する要素を追加したものとみることができる．また，アドレスが整数値になった後もその値を可能な限り追跡する．そのため本論文の方法に比べて解析精度が向上する場合もあるが，1 章で述べたように一度整数値になったアドレスを完全に追跡するのは困難なため，解析結果が安全でなくなる場合が生じるか，もしくは *unknown* を利用した場合と同様の過度に穏健な近似が必要になると思われる（後者については，本論文の提案手法を併用することで精度の改善が可能である）．また論文では触れられていないが，Wilson の方法と同様，複数の型を持つ領域内部の扱いに関して，部分的な重なり合いの処理等十分な考慮が必要であり，解析の安全性は自明ではない．

6. 結　　論

本論文では，整数値とポインタ値の間の型変換の結果を保証する言語仕様のもとで，安全かつ高精度にポインタ解析を行う方法について提案した．プログラムの状態を構成する要素として領域が持つ型および整数値が表す領域の集合を考えることで，ポインタ解析を意味論の素直な抽象化として定義することができる．解析はプログラム中のキャスト式に注目するもので，個々の整数値の解析は不要であり，効率的な実装が可能である．ベンチマークプログラムによる評価の結果は，従来方法ではキャスト等の影響により指示先が不明となるようなポインタの指示先に関して，解析精度が大きく改善されることを示している．

謝辞 本研究の遂行を支援していただいたシステム開発研究所の久島伊知郎氏，野々村洋氏，また理論・実験の両面に関して貴重な助言を下さった金藤栄孝氏，千葉雄司氏に深く感謝いたします．

参 考 文 献

1) Aiken, A., Faehndrich, 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) Chandra, S. and Reps, T.W.: Physical Type Checking for C, *Workshop on Program Analysis For Software Tools and Engineering*, pp.66–75 (1999).
5) 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).
6) Condit, J., Harren, M., Necula, G.C., McPeak, S. and Weimer, W.: CCured in the Read World, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.232–244 (2003).
7) Das, M.: Unification-based Pointer Analysis with Directional Assignments, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.35–46 (2000).
8) Emami, M., Ghiya, R. and Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.242–256 (1994).
9) Embedded Microprocessor Benchmark Consortium: http://www.eembc.org (2000).
10) Foster, J.S., Faehndrich, 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).
11) 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).
12) Necula, G.C., McPeak, S. and Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code, *Symposium on Principles of Programming Languages*, pp.128–139 (2002).
13) Shapiro, M. and Horwitz, S.: Fast and Accurate Flow-Insensitive Points-To Analysis, *Symposium on Principles of Programming Languages*, pp.1–14 (1997).
14) Standard Performance Evaluation Corporation: http://www.spec.org (1995).
15) Steensgaard, B.: Points-to Analysis by Type Inference of Programs with Structures and Unions, *Computational Complexity*, pp.136–150 (1996).
16) Steensgaard, B.: Points-to Analysis in Almost Linear Time, *Symposium on Principles of Programming Languages*, pp.32–41 (1996).
17) Wilson, R.P.: Efficient Context-Sensitive Pointer Analysis for C Programs, Ph.D. thesis, Stanford University (1998).
18) Wilson, R.P. and Lam, M.S.: Efficient Context-Sensitive Pointer Analysis for C Programs, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.1–12 (1995).
19) Yong, S.H., Horwitz, S. and Reps, T.: Pointer Analysis for Programs with Structures and Casting, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.91–103 (1999).
20) ルネサステクノロジ：SuperH RISC engine C/C++コンパイラ，アセンブラ，最適化リンケージエディタユーザーズマニュアル (2003).
21) 関口龍郎：C 言語のための現実的なポインタ解析，第 6 回プログラミングおよびプログラミング言語ワークショップ，pp.52–64 (2004).

(平成 16 年 2 月 24 日受付)
(平成 16 年 5 月 10 日採録)

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．