Pointer Analysis についてのサーベイ

cocoa

September 24, 2001

1 Pointer Analysis とは

pointer analysis とはポインターが何を指しているか求める解析のことで points-to analysis とも (may-)alias analysis とも呼ばれることがある。pointer analysis は dataflow analysis や control flow analysis と本質的には同じである。pointer analysis とは dataflow analysis において record や reference などのコンストラクトが導入された時にそれらを扱う方法であるとも考えることができる。pointer analysis は imperative な言語の解析や最適化と深い関わりがある。例えば次のようなプログラムを考えてみる。

```
*x = 0;
*y = 1;
foo (*x);
```

このプログラムで foo に渡される値が 0 かどうかは x と y が同じものを指しているかどうかに依存する。x と y が指しているものが分からないと constant propagation や common subexpression elimination などが制限されることになる。また整数区間解析などの前段階として pointer analysis が行われることもある。

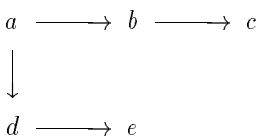
2 基本的な解析アルゴリズム

2.1 Andersen のアルゴリズム [1]

pointer analysis の中で最も基本的なものが Andersen のアルゴリズムである。

```
a = &b;
b = &c;
a = &d;
d = &e;
```

このアルゴリズムでは解析の結果として一つのグラフを生成する。上のようなプログラムの場合は次のようなグラフが生成される。



計算量は $O(n^3)$ である (n は node の数)。$O(n^3)$ は充分に実用的な速度のようにみえる。しかし、現実の数万行のプログラムを解析すると最近のコンピュータを使っても一時間以上かかり、1GB 以上のメモリを消費することもある。

Andersen のアルゴリズムと本質的には同じだが、まず関数ごとに解析を行って、そのあと全体の解析を行うようにした Burke のアルゴリズム [12] もある。このようにすると dead dataflow を計算できるので若干精度が上がることがある。

2.2 Steensgaard のアルゴリズム [19]

Andersen のアルゴリズムを高速化したものの中で最も代表的なのが Steensgaard のアルゴリズムである。このアルゴリズムでは node から出る edge は常に 1 つと決まっている。そのため前述のプログラムを解析した結果は

$$a \longrightarrow \{b, c\} \longrightarrow \{d, e\}$$

となる。この結果、本来あり得ない $c \longrightarrow d$ のような edge が生じてしまう。これを unrealizable path と呼ぶ。計算量は almost linear ($O(n\alpha(n, n))$) である。a の指す先は $\{b, c\}$ と集合になるが、これを a の points-to set と呼ぶ。Steensgaard のアルゴリズムは Tofte, Talpin の region-based system と同等である (ただし monomorphic な variant と)。例えば上記のポインタに関する制約は次のような型付けと解釈することができる。

$$\begin{aligned} a &: \mathrm{ref}_{\rho 1}\ \mathrm{ref}_{\rho 2}\ \alpha \\ b, d &: \mathrm{ref}_{\rho 2}\ \alpha \\ c, e &: \alpha \end{aligned}$$

3 精度を上げるテク

intraprocedural な pointer analysis のアルゴリズムと組み合わせて pointer analysis の精度を上げることができる標準的な 2 種類の方法がある。それが context と flow の概念である。この節ではそれらを含む、精度を上げる手法について説明する。

3.1 Context-sensitive analysis

```
int    *id( int *x ) { return x; }
void   foo() {
    int     a, b, *c, *d;
    c = id (&a);
    d = id (&b);
}
```

というプログラムを考える。関数 id の引数が保持する可能性のあるポインタの集合を x_A とおくと id の呼び出しから $\{\&a, \&b\} \subseteq x_A$ となる。関数 id の帰り値が保持する可能性のあるポインタの集合を x_R とおくと $x_R \supseteq x_A$ だから変数 c と d の取りうる値は $\{\&a, \&b\}$ となる。この不正確さが嫌だと思う人は関数の呼び出しポイントごとに解析を別にしようと考える。このような解析を context-sensitive analysis という。

context-sensitive analysis は関数の段数に関して exponential の計算量になる。また再帰呼び出しがあると止まらない。そのため関数の段数を最大 n 段で抑える制限を加えることがある。これを nCFA とよぶ。

またいわゆる polymorphic な解析を行うのも context-sensitive analysis と言える。

context-sensitive analysis では同じ関数を何度も解析することになるが、この時に本当に解析をするとコストが高くつく。そのため関数を解析したときの効果だけを先に計算しておいて、その効果を適用することで高速化を図ることがある。この関数の効果を transfer function [21] と呼ぶ。また解析の (途中) 結果のことを summary と呼ぶ。この意味で関数の効果を summary transfer function と呼ぶこともある。

3.2 Flow-sensitive analysis

```
int     *p;
if (x) {
    *p = 1;
    printf( "%d\n", *p );
}
else
    *p = 2;
```

というプログラムを考える。ここで画面に表示される値はいくつだろうか。それはポインタ p が何を指しているかに依存する。p が指す可能性があるのは $\{1,2\}$ だから画面に表示されるのは 1 か 2 である。

ただの pointer analysis ではこうなるが、この不正確さが嫌だと思う人はプログラムポイントごとに p の値をトレースしようと思うようになる。そうすれば画面に表示されるのは 1 だと分かるが一方でこの処理のために保存しなければならないデータ量がプログラムポイントの数だけ倍加されることになる。(これについてはもちろん control flow の node ごとに差分を記録するような節約法がある。それを sparse representation と呼ぶ。)

3.3 Disjunction の導入 [10]

解析情報の劣化は条件文、再帰関数やループなどで control flow がマージする場所で発生する。この劣化をできるだけ避けるために disjunction を使う研究がある。式 E のポインタに関する情報を $|E|$ と書くことにする。通常の解析では if $p == \mathrm{NULL}$ then A else B という文全体の効果は、情報をマージする演算子 $\sqcap$ を使って $|A| \sqcap |B|$ のように表現される。しかしこの研究では disjunction を使って $(p == \mathrm{NULL} \wedge |A|) \vee (p\ != \mathrm{NULL} \wedge |B|)$ のように表す。これを使うと非常に正確な結果を得ることができる。たとえばリストの要素を順番に free するプログラムを while を使って書いたときに free が、存在しない対象を消したりしていないことを保証できる。欠点はもちろん計算量で最悪 exponential になる (n は変数の出現の数)。

3.4 Strong update

flow sensitive な解析を行う時に、メモリの特定の場所 (例えば変数) に確実に書き込むことが分かる場合、その場所に関する以前の結果を忘れてもよい。これを strong update と呼ぶ。strong update をやるためには参照先がユニークである (配列のようなものはまずい) 必要がある。また解析が monotonic でなくなるので停止性の問題が生じる。

4 解析の精度

pointer analysis の精度は主に二つの方法で測られている。

- points-to set の大きさ、分布を見る。(小さい方が優秀。)
- 実際の optimization に適用してどのくらい速くなるか見る。

points-to set の大きさを見るときに、その逆関数のサイズも見ることがある。なぜなら、例えばすべてのポインタが同じひとつの対象を指しているような状態を考えてみよう。この時に points-to set の大きさは 1 である。しかし、この結果は最も不正確な情報を表している。

4.1 Context sensitivity の影響 [15]

この研究では context sensitivity が精度にどのような影響を与えるかどうか議論している。ベースとなるのは flow sensitive な Andersen である。結果は context sensitive analysis を使うと最大で 11% ほど points-to pair の数が少なくなる。半分以上のケースで向上は 3% 以下である。その理由としてこの論文で推測しているのは次の点である。

- malloc する時に確保されるメモリは、その malloc の occurrence ごとに区別している。より精度の高い他の解析では context によって区別させるようにしているものもある。そのような工夫をしないと context sensitive にした利益が得られないのかも知れない。
- プログラムを解析する前に様々な最適化 (loop invariant code motion, dead code elimination etc.) を行っており context sensitive な解析ならば区別できる曖昧さを持っているコードは少ないかもしれない。

4.2 Flow sensitivity の影響 [13]

この論文は Andersen のアルゴリズムに対する flow sensitivity の影響を調べている。結果は points-to 集合の大きさで比較している。結論は flow sensitivity は精度の向上にほとんど影響がない、というものだった。この論文ではその理由を次のように推測している。ポインタ変数の更新を考えても、あまり効果はない。なぜならポインタ変数というのはそう更新されるものではない。あるとすればループをなめる時であるが、この場合は配列をなめているので情報はつぶれている。したがって精度はあまり向上しない。

4.3 Flow and context sensitivity の影響 [20]

この論文では modification side effects という問題を利用して flow and context sensitive analysis と flow and context insensitive analysis の精度を比較している。modification side effects とは、代入文で代入する可能性のある reference の集合を求める解析である。reference はそれが生成された program point で区別される。alias analysis には Steensgaard と等価なアルゴリズムを使用している (この論文は後で述べる relation-based なアルゴリズムを使用しており Steensgaard のアルゴリズムと見かけはかなり異なっているが少なくとも僕の見た限り等価なアルゴリズムのように見える)。静的に確保されるメモリはいいとして、動的に確保されるメモリは program point を使って区別する。45 の C のプログラムを使っている。sensitive analysis では、更新される可能性のある平均の reference の数は 1.3 一方 insensitive analysis では 5.1 という差になっている。

4.4 Polymorphism の影響 [9]

この研究では polymorphism がどのくらい役に立つのかを実験で調べている。比較されるのは次の 4 通りの解析である。

- monomorphic Steensgaard
- polymorphic Steensgaard
- monomorphic Andersen
- polymorphic Andersen

すべて flow-insensitive である。また structure field は区別せずに解析する。解析対象のプログラムは最大 60000 行。2GB のメモリでは解析しきれなかったものがちらほらある。解析時間は 60000 行で 515sec (monomorphic Steensgaard) 5574sec (polymorphic Andersen). 結果は points-to set の大きさで判断する。

結論は次のようになる。

$$\text{monomorphic Steensgaard} < \text{polymorphic Steensgaard} < \text{polymorphic Andersen}$$

$$\text{monomorphic Andersen} = \text{polymorphic Andersen}$$

ただし $<$ は有意な差があることを示す。$=$ はほとんど差がないことを示す。

monomorphic Andersen = polymorphic Andersen という結果が暗示しているのは C 言語では input に応じて動作を変える関数がほとんどで input に応じて return value が変化する (points-to 解析の意味で) 関数はまれ、ということである。

4.5 最適化への影響 [3]

この研究は complete C features に対応していると書いてある。pointer analysis は Andersen に基づいており flow insensitive だが context sensitive な方法を採用している。structure field は区別するようにしている。ただし cast に対応するために field name を使わず常に offset でアクセスする。20 万行の 126.gcc を Pentium II 450 MHz で解析し、経過時間は 600 秒、使用メモリは 240 MB であったという。

pointer analysis の結果を利用して SPEC を最適化し、どのくらい速くなるかを調べた。その結果、関数間解析を行わなかった場合の speedup は 7% であり、行った場合の speedup は 43% であったという。また pointer analysis の情報は register promotion と instruction scheduling のどちらにも効くという結果が得られている。

4.6 性能比較 [14]

この論文では次の 4 つの解析の比較を行っている。

- Steensgaard
- Andersen
- Burke
- Choi – 元論文は読んでいないのだが、この論文の解説によると flow sensitive にした Andersen のようである。

各解析の性能は次の基準によって計っている。

mod/ref precision ポインタへの代入、参照において対象となるポインタの集合を求める。
live variable live variable の数を求める。
reaching definition reaching definition の数を求める。
constant propagation constant と決定できた node の数を求める。
analysis time
memory usage

対象は SPEC を含む 23 の C のプログラムである。

結果を次に載せる。ただし Andersen と Burke はほとんど同じ結果になったので並べて載せる。

	mod	ref	live	def	constant	time (s)	memory (MB)
Steensgaard	1.04	1.75	20.13	22.04	10.6	0.90	1.15
Andersen/Burke	0.871	1.540	18.36	20.21	10.7	29.60/32.92	8.52/3.15
Choi	0.867	1.536	18.30	20.16	10.7	79.49	12.19

5 解析時間を短くするテク

5.1 構造体の要素を区別しない

配列の要素を区別しない解析は当然のように思うかもしれないが、実は構造体も一点に潰してしまう解析も結構ある。

5.2 k-limiting

主にメモリ使用量を減らすためにポインタが作る構造の深さを制限したいと思うことがある。これを k-limiting と呼ぶ。k は最大の深さを表す。

しかしそれでも $O(n^k)$ で node が増える可能性がある。これを避けるために実際の解析では k を小さくしがちだが、そうしてしまうと今度は深い構造を解析した時の精度を極端に悪くする。そこで、この研究 [2] では node をマージするときは、同じ program point で作られた node 同士に限るという制限を与えた。これにより構造をできるだけ保存したまま解析を行うことができる。

5.3 k-outedge

Shapiro によるこの手法 [17, 18] は node の outedge を高々 k 個に制限してしまう方法である。そのためもし node から $k+1$ 個以上の node を指す必要がある時はどれかの node が unify される。$k=1$ の場合が Steensgaard のアルゴリズムと考えることができる。計算量は $O(k^2n\alpha(k^2n,k^2n))$ となり almost-linear である。

結果は points-to set のサイズで比較している。このアルゴリズムの精度は安定的ではない。Andersen のアルゴリズムに肉薄するケースもままある。しかし、極端に悪くなるケースもある。

ちなみに比較のために実装された Andersen のアルゴリズムは 24000 行の C プログラムを解析するのに SS20 上で 500 秒かかっている。

5.4 k-visiting

この研究 [5] ではデータフロー解析の途中に同じ node を k 回以上アクセスするとその node をその次の node とマージする、という最適化について調べている。この最適化は他の最適化のある種の一般化となっており、Andersen や Steensgaard を表すことができるという。解析結果は points-to set が singleton となる割合と解析結果を使って Cecil でコンパイルした時の実行速度の違いである。結果を見ると、singleton 集合の割合と実行速度の向上には強い相関がある。また結果は次の 2 種類に分割することができる。

- $k=10$ あたりで性能はサチってしまい、それ以上詳しい解析をしても無駄である。
- $k=10$ ~ 50 あたりで性能はサチるのだが、$k=50$ と ∞ の間には大きな差がある。

この結果が意味しているのは、ある種のアプリでは同じ node をひんぱんにアクセスすることがあるが、その node を別の node とマージすると、極端な解析精度の低下を招く、ということである。

5.5 型情報を利用

この研究 [8] では 3 つの解析の有効性を比較している。

TypeDecl 同じ型および subtype は alias される可能性があると考える。

FieldTypeDecl 同じ型でも field が違っており、その field のアドレスを取られていないならば alias されていないと考える。

SMFielTypeDecl flow-insensitive dataflow analysis をして、同じ集合に入ったら alias されると考える。(Steensgaard のアルゴリズムのように見える。)

なお対象は Modula-3 である。Modula-3 には pointer arithmetic はないが、field を参照渡しで渡すことができ、field への alias が作れるらしい。

結果は points-to set の大きさで計っており、割と予想通りで TypeDecl が極端に悪く、他の二つはあまり差がない。

次に実際の最適化に適用してみてどのくらい効くのかを調べた。ここで対象とした最適化は次のものである。

Redundant Load Elimination (RLE):

> loop の中の load を loop の外に動かす変換。これを行うためには load の対象が loop の中で書き換えられないということを知る必要がある。これには alias 解析が必要。

結果は RLE の最適化も points-to set の正確さにほぼ比例している。TypeDecl が悪く。FieldType と SMFielTypeDecl はほとんど変わらない。理想的な状況と比較して RLE がどのくらい残っているか調べると、解析により無駄な load の 30%-80% を除去できたことがわかった。残りの原因を調べてみても、静的解析では調べるのが難しい依存性によっている (たとえば配列に同じポインターが入っていた) ことが分かった。

[16] でも型を使って計算量を下げている。この研究ではベンチマークとして C のプログラムを対象にしているが、この研究の手法は Pascal のような pointer の semantics に限定されている。この手法は次のような observation に基づいている。

```
struct foo {
    int *p;
    int *q;
};
```

という構造体があったとしよう。このプログラムは `struct foo` 型の値と `int` 型の値を扱っている。`struct foo*` 型の変数を操作すると `struct foo` 型の値を扱うことができる。一方 `int*` 型の変数を操作すると `struct foo` 型の値と `int` 型の値の両方を変更する可能性がある。

そこでまずプログラム中の `struct foo*` 型の変数だけに着目してデータフロー解析を行い、その結果を覚えておく。次に `int*` 型の変数と最初の解析の結果を利用して、より詳しい解析を行う。このようにすることでピーク時に必要なメモリ量を減らすことができる。

SPEC と思われるベンチマークを対象に実験を行ったところプログラムを十数個 (4 ~ 35) の部分に分割することができ、ピーク時のメモリ使用量を 1.3 ~ 7.2 倍縮小させることができた。

5.6 解析の分割によるコストの削減

[3] ではプログラムから生成されたデータフローグラフを強連結成分分解し、下流から順にやっていくことでメモリの最大使用量を減らしている。

[23] では points-to relation を連結成分分解し、成分ごとに解析している。

5.7 One level flow algorithm [4]

abstract によると 140 万行のプログラム (MS Word97) を 2 分で解析でき、結果も Andersen のアルゴリズムと同程度だと言う。flow-insensitive, context-insensitive である。基本的な方針は「ポインターを (他のポインターによって指されていない) 変数によって受け渡しする (関数の引数も含む) 場合を正確に解析し、メモリに書かれたらそれ以上の解析は諦める」ことにある。これは C のプログラマの直観に合っている。この方針は一段階だけ Andersen の方法によって解析し、それ以降の指されているものを扱うときは Steengaard の方法によって unify する、というやり方で実装できる。具体的に見てみよう。

```
bar( struct s *q ) { *q.b = 4; }

foo( struct s *p ) { *p.a = 3; bar (p); }

foo (&s1);
foo (&s2);
bar (&s3);
```

というプログラムがあった時に Andersen のアルゴリズムでは

$$p \to \{s1, s2\}, q \to \{s1, s2, s3\}$$

という結果になり、Steensgaard のアルゴリズムでは

$$p = q \to \{s1, s2, s3\}$$

とやや荒い結果になる。

この場合は構造体にポインタが埋め込まれていないので、一段階だけ Andersen のアルゴリズムでやるとまず

$$p \to \{s1, s2\}, q \to \{s3\}$$

という状態になる。次に `bar(p)` という呼び出しを処理するのだが、この解析では代入 (と引数の受渡し) だけ flow sensitive な扱いをする。これが名前の one level flow の由来であるが、この場合は `p` から `q` へのデータフローがあると解釈する。したがって最終的に

$$p \to \{s1, s2\}, q \to \{s1, s2, s3\}$$

となり、Andersen と同じ結果が得られる。計算量は $O(n^2)$ である。解析精度は points-to set の大きさの平均と分布で計っているが Andersen のアルゴリズムとほとんど同じである。

6 解析結果の表現法 [6]

pointer の作る構造を表現するには代表的な 2 つの方法がある。

store-based model store を使う。
storeless model right-regular equivalence relation を使う。

store-based model では次のような (Env,Store) でポインタの参照関係を表現する。

$$\mathrm{Env} = \mathrm{Id} \to \mathrm{Loc}$$
$$\mathrm{Store} = \mathrm{Loc} \to \mathrm{Val}$$

$$\mathrm{Val} = \mathrm{Const} \cup (\mathrm{Id} \to \mathrm{Loc})$$

ちなみに Val の定義で使われる $(\mathrm{Id} \to \mathrm{Loc})$ は構造体などを表すのに使われる。

一方 storeless model では

$$\mathrm{Tree} = \mathrm{Id}^* \to \mathrm{Const} \cup \{\Omega\}$$
$$\mathrm{Alias} = \mathcal{P}(\mathrm{Id}^* \times \mathrm{Id}^*)$$

ただし alias は right-regular equivalence relation でなければならない。この定義で分かる人はあまりいないと思うので [23] から具体例を見てみよう。

$o ::=$	(object name)
$\quad v$	(object value)
$\quad \mid \&v$	(address-of)
$v ::=$	(object value)
$\quad x$	(variable)
$\quad \mid v.f$	(field selection)
$\quad \mid *v$	(pointer dereference)

これらの定義はポインタ演算を含む式を表現している。C で使われる `x->f` は $(*x).f$ の省略形と解釈することができる。形式的な簡潔さのために pointer dereference は特殊な field selection であると解釈することもある。そうなると v は Id^* と同じである。v が primitive value を指す時に、その値を対応させ、そうでない時に特殊な定数 Ω と対応させるような関数の領域が Tree である。再帰的な list や tree を考えてみると v は無限に長い可能性もあることが分かる。そのような場合は普通は regular tree を使うが、[6] によれば nonregular なものを扱う研究もある。

さて storeless model では pointer alias を relation で表す。例えばプログラムで $*p = x$ という代入があると $(*p)Rx$ であるような right-regular equivalence relation R でこの情報を表す。では right-regular であるとはどういうことだろうか。

R が right-regular であるための条件は

$$\text{if } aRb \in \text{ then } (a.l)R(b.l)$$

である。つまり $*p$ と x が同じものを指していたら $(*p).f$ と $x.f$ も同じものを指しているという意味である。

storeless model には、ごみがでない、location の renaming に対して不変という特徴がある。言い替えると (Tree,Alias) は (Env,Store) に isomorphic up to garbage-collection and location-renaming である。

[7] ではより拡張された storeless model を提案している。ここでは parameterize された tree を使い $x \to (\mathrm{tl} \to)^i \mathrm{hd}$ where $i \geq 5$ のような構造を表すことができる。[7] の手法を使うととても正確な解析を行うことができる。例えば

```
struct tree { void *hd; struct tree *tl; } *x, *y;
x = y->tl;
```

というプログラムの持つ制約は

$$x \to (\mathrm{tl} \to)^i \mathrm{hd} = y \to (\mathrm{tl} \to)^j \mathrm{hd} \quad \text{where } i = j + 1$$

のように表すことができる。この解析の問題はコストである。彼らの prototype implementation では 50 行以下のプログラムは 30 秒以内に解析できると、書いてある。

ポインタは次のような集合で表すこともある [21] 。

$$\{ax + b | x \in Z\}$$

ただし b は整数か、シンボルと整数の和かどちらかである。この集合は $a = 1$ で最悪の情報を意味する。この表現法は構造体の配列をアクセスする時のパターンを捉えている。

7 C 言語の解析

C 言語の full set を対象とした解析アルゴリズムは少ない。ほとんどは C 言語の適当な subset か、Modula 3 を対象としている。

C 言語の仕様の決め方には 3 種類のカテゴリーがある。

Implementation-defined 実装によって定められる仕様を意味する。
Unspecified 実装が定めてもいいし、定めなくても良い。
Undefined 違法なものを意味する。

問題なのは unspecified の場合である。どんな実装に対しても有効な解析をしようと思うと曖昧な結果になってしまう。また C の仕様に忠実であろうとするのは非常に難しい (例えば以下のプログラムの動作は undefined である。)

```
struct s { char string[30]; } s1, s2;
strcpy( s1.string, "Hello,World\n" );
s2 = s1;    // uninitialized value による initialization
```

また C に対応とは言ってもここで紹介しているものはすべて setjmp/longjmp, signal handler には対応していない。

また alias 解析の場合は、ポインタ同士の関係を求めることが重要であるのでポインタが NULL であるかどうかを問題にしない場合が多い。

7.1 整数 $\leftrightarrow$ ポインタのキャストの扱い

整数は解析の時にポインタだと思って追跡する。[3] では C 言語レベルの型情報をすべて忘れてしまい、構造体、共用体は単なるワード列であると見なして解析を行っている。プログラム上で整数へのキャストを通してポインタ演算されていても問題ではない。なぜなら二つの heap-allocated object の間の距離は undefined となっているのでポインタ演算をしていたとしてもそれはヒープ上に確保された連続領域の中のアクセスを常に意味するからである。つまりポインタに整数を加えている場合、それは別の malloc で確保された領域を指すことはできない。また別の malloc で確保されたポインタ同士を引いた場合、その結果は undefined である。同じ理由からポインタの下位ビットにマスクを掛けていても正しく扱うことができる。やっかいなのは別のポインタの値を、直接代入するのではなく char を経由して 4 回でコピーしている場合である。(そもそも ISO ではどうなっているのか？)

```
void    copy_pointer( int *p, int *q ) {
    char    *s = (char*)p;
    char    *t = (char*)q;
    s[0] = t[0];
    s[1] = t[1];
```

```
    s[2] = t[2];
    s[3] = t[3];
}
```

また以上のような理由から bcopy, memcpy は polymorphic な型を持つ複数の代入として解釈する必要がある。

7.2 C の仕様に忠実にやるには

論文 [22] の contribution は微妙な言い回しである。casting があって portable な形で解析を行う最初の論文であるという。構造体の要素のオフセットに関して ISO では次のように決まっている。

1. 最初の要素のオフセットは 0 である。
2. 異なる構造体で、全く同じ型の要素が続いている時にその次の要素のオフセットは同じである。

したがって次の構造体の場合

```
struct s {                    struct t {
    int     s1;                   int     t1;
    char    *s2;                  int     t2;
    int     s3;                   int     t3;
}                             }
```

s2 と t2 は同じオフセットから始まるが、s3 と t3 に関しては不明である。

それでこういった問題があるにも関わらずどうやって portable に解析をするかだが、この論文の解決法は驚くほど簡単である。構造体の field が与えられたときにその要素のオフセットを返す関数 lookup をパラメタライズして解析アルゴリズムを書き、実際の解析をする時にコンパイラ固有 lookup 関数を与えてやればちゃんと解析ができる、というものである。

さすがにこれだけではまずいと思ったのか以下の 4 つの解析アルゴリズムの性能比較を行っている。

collapse always 構造体を一点につぶすもの
collapse on cast キャストするとオフセットがわからなくなるもの
common initial sequence common initiali sequence が同じなら別々の field として扱うがそれ以外の型が合わないフィールドはすべて collapse させるもの
offset オフセットを計算するもの

結果は points-to set の大きさで計っている。結果はこの順番でよくなる。collapse always はだんとつで悪く (offset の 2 ~ 10 倍くらい悪い)、他の 3 つは比較的近い。collapse on cast と common initial sequence の違いはあまりない。また驚くべきことに解析時間は collapse always がひどく悪い。おそらく考えなければならない alias の組の数が増大するからだと思われる。

また本題とは関係ないのだが興味深い話がいろいろと載っている。キャストを扱うときにこの研究では次のような正規形に変換している。

```
type_s s;
type_p *p;

s = (type_s)&t.f
s = (type_s)&((*p).f)
s = (type_s)t.f
```

```
s = (type_s)*q
*p = (type_p)t
```

ただし f は possibly empty sequence of field selection である。このパターンは SUIF や他の解析などでも使われているという。

また扱いの難しい例がいくつか挙げられている。

```
struct S { int *s1 } *p;
int  *q, x;
q = &x;
*p = (struct S)q;
```

構造体はその最初の要素としても見ることができるという例である。

```
struct V {
    int *v1;
    char *v2;
    int *v3;
} v;

struct W {
    int *w1;
    struct R {
        int *r1;
        char *r2;
    } r;
    int *w3;
} w;

v = (struct V)w.r;
```

このような例があるために結局 C レベルでの型情報はあまり役に立たず、word 列として認識しなければならない (共用体もそうだが)。

```
struct R { int *r1; int *r2; } r;
double d;
d = (double)r;
```

これは ISO ではどうなっているのか？

8 Shape Analysis

pointer analysis によってポインタが何を指しているかが分かると、その結果を利用してポインタがどんなデータ構造になっているか (例えば tree であるとか DAG であるとか) 分かることがある。このような解析を shape analysis と呼ぶ。shape analysis は dependence analysis や parallelization などへの応用を目的としている。shape analysis を扱った研究はいくつかあるが、我々の興味とは離れているのでここでは代表的な論文を一つだけ紹介する。

8.1 Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C [11]

この論文ではポインタによるデータ構造が次のいづれかであるかを判断する。

tree 任意の 2 node の間のパスは 1 つしかない。
DAG パスは 1 つ以上だが半順序を付けられる。
cycle そうではない。

まず C 言語を次のような中間言語に落として、その上で解析を行っている。

1. `p = malloc();`
2. `p = q;`
3. `p = q->f;`
4. `p = &(q->f);`
5. `p = q` op `k;`
6. `p = NULL;`
7. `p->f = q;`
8. `p->f = NULL;`

この言語を (D,I,A) のデータ構造を利用しながら解析を行っていく。

direction matrix D p から q へ path があるとき D[p,q] = 1 となる。そうでなければ D[p,q] = 0.
interference matrix I p と q から同じ object に reachable であるとき I[p,q] = 1 となる。そうでなければ I[p,q] = 0.
attribute matrix A A[p] = tree, DAG or cycle

中間言語を仮想的なヒープを使いながら実行し、(D,I,A) が fixed point に達するまで繰り返すことで結果を求める。また仮想実行の前に pointer analysis を行うが、これはできるだけ正確な結果が欲しいので context-sensitive, flow-sensitive な pointer analysis を行う。

参考文献

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

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990.

[3] Ben-Chung Cheng and Wen mei W. Hwu. Modular Interprocedural Pointer Analysis using Access Paths: Design, Implementation, and Evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[4] Manuvir Das. Unification-Based Pointer Analysis with Directional Assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, June 2000.

[5] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[6] Alain Deutsch. Semantic Models and Abstract Interpretation Techniques for Inductive Data Structures and Pointers. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'95*, pages 226–228, 1995. June.

[7] Alain Deutsh. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[8] Amer Diwan, Kathryn S. McKinley, and J. Elliot B. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[9] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Monomorphic versus Polymorphic Flow-insensitive Points-to Analysis for C. In *Proceedings of the 7th International Static Analysis Symposium*, pages 175–198, 2000.

[10] Pascal Fradet, Ronan Gaugne, and Daniel Le Métayer. Static Detection of Pointer Errors: an Axiomatisation and a Checking Algorithm. In *Proceedings of European Symposium on Programming, ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 125–140, 1996.

[11] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Conference Record of POPL'96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[12] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[13] Michael Hind and Anthony Pioli. Assessing the Effects of the Flow-Sensitivity on Pointer Alias Analyses. Technical report, IBM Research Report, June 1998.

[14] Michael Hind and Anthony Pioli. Which Pointer Analysis Should I Use? In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, August 2000.

[15] Erik Ruf. Context-Insensitive Alias Analysis Reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[16] Erik Ruf. Partitioning Dataflow Analyses using Types. In *Conference Record of POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, January 1997.

[17] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-to Analysis. In *Conference Record of POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[18] Marc Shapiro and Susan Horwitz. The Effects of the Precision of Pointer Analysis. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 16–34, 1997.

[19] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Conference Record of POPL'96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[20] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing Flow and Context Sensitivity on the Modification-Side-Effects Problem. Technical report, Department of Computer Science, Rutgers University, August 1997. DCS-TR-335.

[21] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[22] Suan Yong, Susan Horwitz, and Thomas Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[23] Sean Zhang, Barbara G. Ryder, and William Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *the Fourth Symposium on the Foundations of Software Engineering (FSE4)*, October 1996.