BDDを利用したCプログラムのfield-sensitiveなポインタ解析

吉羽　和之　　佐々　政孝
東京工業大学大学院情報理工学研究科数理・計算科学専攻

精度の高いポインタ解析は、プログラムの最適化を行う上で重要である。しかし、精度の高い解析や大きいプログラムを対象とした解析はコストが大きくなる。本論文ではfield-sensitiveなポインタ解析を定式化した。field-sensitiveな解析は構造体の各メンバを区別して扱うので精度の良い結果を得ることができる。しかし、それによって解析で扱う集合が大きくなり、メモリ使用量や解析時間が増大することが考えられる。そこで、Whaleyらが Java 言語を対象にして行った研究を参考にし、集合の表現にBDDを用いたfield-sensitiveなポインタ解析を実装し実験を行った。その結果、BDD を使うことでメモリ使用量を減らすことでき、素朴な集合を用いては解析できないプログラムも解析できることを示すことができた。

Field-sensitive Points-to Analysis for C Programs using BDDs

Kazuyuki Yoshiba　　Masataka Sassa
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Tecnology

A precise points-to analysis is important for program optimizations. However, the costs for the precise analysis and the analysis for large programs are high. In this paper, we formalize a field-sensitive points-to analysis. Because field-sensitive analysis treats members of structure independently, a precise result can be obtained. However, it is thought that as the set treated by the analysis grows, the memory usage and analysis time increase. Then, referring to the research of Whaley targeted to the Java language, we implemented and experimented a field-sensitive points-to analysis using BDDs to represent sets. As a result, we show that the memory usage decreases and programs that cannot be analyzed by using simple sets can be analyzed by using BDDs.

1　はじめに

ポインタ解析についての研究は多数あり、各々解析の範囲や精度に差がある。一般に、プログラムの解析では精度を上げようとする場合、解析にかかる時間やスペースのコストも大きくなる。ポインタ解析の場合その傾向が特に顕著であり、効率的な解析のために精度を犠牲にする手法や、データ構造を工夫することで使用するスペースを節約する手法などが提案されている。

本論文では、flow-insensitive、inclusion-based、field-sensitive、context-insensitive なポインタ解析の推論規則を定式化する。そして、Whaley らの研究 [1] を参考にし、定式化された規則を BDD の演算によって表現することで効率的な解析ができることを示す。

2　Binary Decision Diagram

2.1　BDD の概要

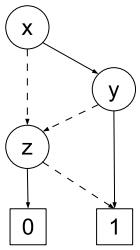
BDD(Binary Decision Diagram) とは、ブール関数を表現するのに使われるデータ構造であり、多くのブール関数をコンパクトに表現できるという特徴がある。

BDD は 1 つの出発点を持つ有向非循環グラフである。2 つの終端ノード（0 終端、1 終端）を持ち、終端ノード以外のノードを決定ノードと呼ぶ。決定ノードには 1 つのブール変数が対応しており、2 つの子ノードを持つ。2 つの子ノードは、ブール変数が真 (1) のときの HIGH ノードと偽 (0) のときの LOW ノードである。

BDD が表現しているブール関数が真 (1) となるような変数への真理値の割り当ては、根のノードから 1 終端ノードまでの経路で表す。あるノードから HIGH ノードへの経路を通る場合、そのノードに対応する変数の値

が 1 であることを示し、LOW ノードの場合は 0 であることを示す（経路上に現われない変数の真理値は関数の値に影響しない）。

下に示すのはブール関数 $f(x,y,z) = x * y + \neg z$ を表す BDD である[1]。HIGH ノードへのエッジは実線で、LOW ノードへのエッジは点線で表す。



BDD を利用する利点は次のようなものがある。

- 多くのブール関数をコンパクトに表現できる。
- 関数同士の演算がグラフのサイズにほぼ比例する時間で実行できる。

BDD はブール関数の表現であるが、ビット列の集合を表現するものとみなすこともできる。たとえば、例で示した BDD は、長さ 3 のビット列の集合

$$\{v_1v_2v_3 | f(v_1,v_2,v_3) = 1\} = \{000, 010, 100, 110, 111\}$$

を表しているとみなすことができる。このように、n 変数のブール関数を表す BDD は、長さ n のビット列の表現と見ることができる。

2.2 ドメインと BDD 同士の演算

ここでは、本論文で提案するポインタ解析手法の説明に必要となる BDD の用語について説明する。

ドメイン BDD を操作する際、ビット列の各桁を 1 つずつ扱うより、いくつかのビットをまとめて扱えたほうが便利な場合がある。そのようなビットのまとまりをドメインと呼ぶ。例えば集合 S = { 0000, 0001, 0100, 0101, 1000, 1001, 1010 } の最初の 2 ビットをドメイン $D1$、次の 2 ビットをドメイン $D2$ とすれば、S は $D1$、$D2$ に属する長さ 2 のビット列の組の集合 { (00,00), (00,01), (01,00), (01,01), (10,00), (10,01), (10,10) } と見ることができる。このようにビット列をドメインに分割することで n 個のビット列からなる組の集合を BDD で表現することができる。

あるビット列 b がドメイン D に属することを $b \in D$ と書き、集合 S がドメイン $D1$、$D2$ からなることを $S : D1 \times D2$ と書くこととする。

[1] ここで $*$ は AND、$+$ は OR、$\neg$ は NOT を表す。

relational product relational product は、2 つの BDD からある種の積集合を求める演算である。2 つの BDD $X : D1 \times D2$、$Y : D1 \times D3$ があるとき X、Y の relational product $R = relprod(X, Y, D1)$ は次のように定義され、$R : D2 \times D3$ となる。

$$\begin{aligned} R &= relprod(X, Y, D1) \\ &= \{(a,b) | \exists d \in D1[(d,a) \in X \wedge (d,b) \in Y]\} \end{aligned}$$

また、$X : D2 \times D1$、$Y : D1 \times D3$ のような場合でも同じように定義できる。さらに、BDD が $Z : D1 \times D2 \times D3$、$W : D1 \times D2 \times D4$ のように 3 つのドメインからなる場合、

$$\begin{aligned} &relprod(X, Y, D1 \times D2) \\ &= \{(a,b) \mid \exists (d1,d2) \in D1 \times D2 \\ &\quad [(d1,d2,a) \in X \wedge (d1,d2,b) \in Y]\} \end{aligned}$$

のようにドメインの組を 3 番目の引数にすることもできる。

例として $X : D1 \times D2$、$Y : D1 \times D3$、$X = \{(00,01),(01,00),(01,10)\}$、$Y$ ={ (00,00), (01,01), (10,10) } から $R = relprod(X, Y, D1)$ を求めることを考える。X の第 1 要素 (00, 01) と Y の第 1 要素 (00, 00) はドメイン $D1$ の値が 00 で等しいので、それぞれの要素の中で 00 と組になっている 01 と 00 の組 (01, 00) が R に含まれる。同じことを繰り返すことで $R = \{(00,01),(01,00),(10,01)\}$ が求まる。

この例では定義にもとづいて relational product を求めたが、2 つの BDD からグラフを直接操作して relational product を効率よく求める関数がライブラリに用意されているので、実装時にはそれを利用した。

replace replace は、ある BDD のドメイン名を別のドメイン名に置き換えた BDD を求める演算である。$X : D1 \times D2$ であるとき、$R = replace(X, D1, D3)$ によって得られる BDD R のドメインは $D3$ と $D2$ になる。この演算は、単にドメイン名を変えるだけでグラフの形は変えない。

3 BDD を利用したポインタ解析

3.1 推論規則

ポインタ解析で利用する推論規則を後の表 3 に示す。規則 1〜5 は構造体がない場合の一般的な flow-insensitive、inclusion-based なポインタ解析の規則である。

構造体に関する規則　規則 6〜10 が本論文で新たに提案する field-sensitive なポインタ解析を行うための規則である。解析では、構造体の各メンバを別々に扱う。たとえば、s.f1 や ∗(u.f2)、(∗p).f などを 1 つの変数として扱う。s.f1 や ∗(u.f2) への代入による pt(s.f1)[2] と pt(∗(u.f2)) の変化は規則 1〜5 で求めることができる。規則 6〜10 はそれ以外の場合（たとえば (∗p).f への代入によって p が指す構造体のメンバ f が指す集合が変化する）についての規則である。

規則 7、8 は規則 3、4 と似た意味である。規則 3、4 は ∗p への代入は p が指すすべての変数への代入とみなすと言っているのに対し、規則 7、8 は (∗p).f への代入は p が指しうるすべての変数 s のメンバ f への代入とみなすと言っている。

規則 9 は (∗p).f が指す変数についての規則である。(∗p).f が指す変数は (∗p).f への代入以外でも変更される。p が q を指しているとき、q.f が x を指すと (∗p).f も x を指すようになる。

最後に、規則 10 は構造体同士の代入文に対する規則である。構造体同士の代入ではメンバの値がコピーされるので、t.f がポインタ型であるなら t.f が指す変数は s.f が指す変数に含まれることになる。

関数間の解析　規則 11〜13 は関数間の解析のための規則である。$actuals \ni$ (f, n, q) は n 番目の実引数が q である f の呼び出しがあることを示し、$formals \ni$ (f, n, p) は f の n 番目の仮引数が p であることを示す。そして、$returns \ni$ (f, p) は f の中に return p という文があることを示す。関数呼び出し時に実引数 q が仮引数 p に渡されたとき、その効果は代入 p = q と同じと考える。そして、関数 f に return q という文があるとき、文 p = f(...) の効果は p = q と同じと考える。つまり、pt(q) ∋ x ならば pt(p) ∋ x となる（規則 11、12）。

また、C 言語には関数ポインタが存在する。規則 13 では、x = (∗fp)(q) のように関数ポインタ fp から関数を呼び出している場合、$actuals \ni$ (fp, 1, q) となる。そして pt(fp) に含まれるすべての関数が呼び出されていると考える。たとえば、pt(fp) ∋ f であり $formals \ni$ (f, 1, p) ならば、(∗fp)(q) の効果は p = q と同じと考え、pt(q) ∋ x ならば pt(p) ∋ x となる。

3.2 BDD による実現

3.2.1 集合の BDD による表現

解析のためにはプログラム中の代入文やポインタ指示関係を BDD で、つまりビット列の集合で表す必要がある。そのためにまず、プログラム中の各変数をビット列で表す。これは以下のようにたんに異なる変数にビット列を順に割り当てていくだけである。

```
int **p, *q;        p  ⇔ 000
int a = 0;          *p ⇔ 001
int b = 1;          q  ⇔ 010
p = &q;             a  ⇔ 011
q = &a;             b  ⇔ 100
```

次に代入文やポインタ指示関係を変数の組で表す。たとえば、p = q という代入文を (p, q) で、pt(p) ∋ x を (p, x) で表す。各変数はビット列で表されているので、代入文の集合やポインタ指示関係の集合はビット列の組の集合で表され、BDD で表現できる。

代入文の集合やポインタ指示関係を表す BDD とそのドメインを次のように定義する。

- $Assign$: $dest \times source$
  - 代入文の集合。p = q があるならば (p, q) ∈ $Assign$
  - $dest$ は代入先の変数を表すドメイン、$source$ は代入元の変数を表すドメイン
- $PointsTo$: $var1 \times var2$
  - ポインタ指示関係を表す集合。pt(p) ∋ x ならば (p, x) ∈ $PointsTo$
  - $var1$ はポインタ変数を表すドメイン、$var2$ はポインタが指す変数を表すドメイン

これらの BDD を使いポインタ解析を行う方法を規則 2 を例にして説明する。規則 2 は、

> 代入文 p = q があり、かつ、pt(q) ∋ x、ならば pt(p) ∋ x

という意味である。これは、規則 2 によって新たに得られるポインタ指示関係の集合を $NewPointsTo$ とし、$PointsTo$ と $Assign$ を用いることで、

$$NewPointsTo = \{\ (p, x) \mid (p, q) \in Assign \wedge (q, x) \in PointsTo\ \}$$

と言い換えることができる。これは前節で説明した relational product の定義とほぼ同じである。つまり、規則 2 によって得られるポインタ指示関係は $Assign$ と $PointsTo$ の relational product をとることで求めることができる。実際はドメイン名を指定する必要がある。具体的には次のようになる。

[2] 変数 p が指しうる変数の集合を pt(p) と書く。

Architecture	Superscalar SPARC Version 9
CPU	750MHz UltraSPARCIII × 2
Cache	64KB データ 32KB インストラクション
Memory	1GByte
OS	SunOS 5.8

表 1: Sun Blade 1000 の主な仕様

	BDD		HashSet	
	Time(sec)	Mem(MB)	Time(sec)	Mem(MB)
164.gzip	0.79	2.22	34.0	35.1
175.vpr	7.47	5.01	59.5	189
184.mcf	0.42	2.19	3.73	24.3
197.parser	2.83	3.98	31.9	126
254.gap	137	19.9	-	-
255.vortex	269	33.7	-	-
256.bzip2	0.42	1.66	2.62	28.5
300.twolf	11.1	8.79	316	254
177.mesa	307	36.9	-	-
179.art	0.35	1.44	1.84	21.6
183.equake	0.31	1.67	2.29	25.7
188.ammp	9.42	8.55	224	207

表 2: 実験結果

$$
\begin{aligned}
tmp &= replace(Assign, source, var1) \\
NewPointsTo &= relprod(tmp, PointsTo, var1) \\
NewPointsTo &= relpace(NewPointsTo, dest, var1) \\
PointsTo &= PointsTo \cup NewPointsTo
\end{aligned}
$$

1 行目では $Assign$ と $PointsTo$ の relational product をとるために、replace を行って (p, q) $\in Assign$ の q のドメイン名と (q, x) $\in PointsTo$ の q のドメイン名を一致させている。そして、relational product の結果を $PointsTo$ に加える前にもう一度 replace を行い $NewPointsTo$ のドメイン名を $PointsTo$ に合わせている。

他の規則に関しても、replace でドメイン名を一致させて relational product をとることを繰り返すことで計算することができる。

4 実験と評価

実験は提案手法をコンパイラ・インフラストラクチャ COINS[5] の高水準中間表現である HIR 上に実装し行った。また、BDD を扱うために JavaBDD[7] を利用した。これは Java プログラムで BDD を扱うためのライブラリで、2.2 で説明した機能を持っている。

4.1 実行環境とテストプログラム

実験は、Sun Microsystems の、Sun Blade 1000 で行った。この主な仕様は、表 1 の通りである。

テストプログラムは、SPEC CPU2000 のベンチマークプログラムより以下の 12 個を用いた。

164.gzip, 175.vpr, 181.mcf, 197.parser, 254.gap, 255.vortex, 256.bzip2, 300.twolf, 177.mesa, 179.art, 183.equake, 188.ammp

4.2 実験内容とその目的

実験では各プログラムに対して次の計測をおこなった。

1. BDD を用いたポインタ解析の解析時間とメモリ使用量
2. BDD の代わりに java.util.HashSet を用いた場合の解析時間とメモリ使用量

メモリ使用量の測定には Java に標準で付属する hprof を用いた。またメモリを最大限に使うためオプションに-Xmx1024M を指定した。各プログラムに対して 3 回ずつ計測を行いその中央値をとった。

HashSet を用いるにあたり、HashSet 間の relational product が必要になるのでそれは手で実装した。

1 と 2 の結果を比較し、BDD を用いた方のメモリ使用量や解析時間が少なければ本手法が有効であるということになる。

4.3 実験結果

実験結果は表 2 である。表の中で「-」となっているのはプログラムの実行中に OutOfMemoryException が発生し計測できなかったことを示す。図 1 は HashSet を 100 としたときの解析時間とメモリ使用量である。

4.4 結果の考察

まず図 1 を見てすぐわかるように、BDD を用いた方がはるかにメモリ使用量が少ないことが分かる。

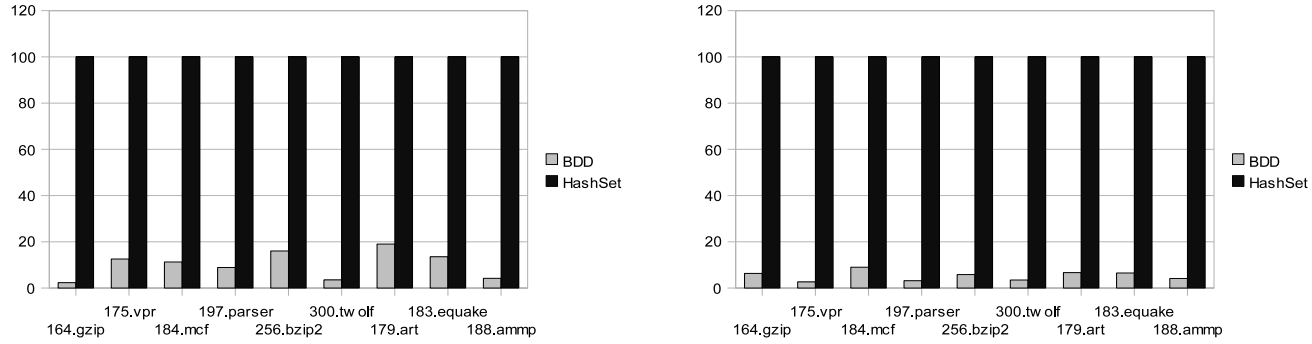


図 1: HashSet との比率　左が解析時間、右がメモリ使用量

さらに、254.gap、255.vortex、177.mesa の 3 つのプログラムは BDD を用いないと解析できなかった。このことから本研究が提案する手法が有効であることが示された。

解析時間に関しても BDD のほうがまさっているが、これは BDD ライブラリが提供する relational product 演算と、手で実装した HashSet 間の relational product 演算の性能の差によるところがあると考えられるので一概に比較するのは難しい。254.gap、255.vortex、177.mesa の 3 つのプログラムの解析時間が他のプログラムに比べて極端に長くなっている。BDD の演算が必ずしもサイズに比例した時間で実行できるわけではないようである。原因は分らないが変数の順序付けが影響している可能性がある。変数の順序付けが解析にどう影響するかについての考察は今後の課題である。

5　まとめ

本研究では、一般的に精度が良いとされる field-sensitive なポインタ解析を行うための推論規則を定めた。この規則に従い構造体の各メンバをそれぞれ異なる変数として扱うことで解析を行うが、それにより扱う集合が大きくなり、集合を素朴な方法で表現すると大きなプログラムを解析できなくなる。そこで本研究では、BDD（binary decision diagram）を集合の表現として用いることでコンパクトに集合を表現した。そして、BDD を用いる場合と用いない場合を COINS の HIR 上に実装し、それぞれのメモリ使用量と解析時間を比べることで、BDD を用いた方が少ないメモリかつ少ない解析時間で解析できることを示した。

6　今後の課題

今後の課題は次のとおりである。

最適化への応用　今回の研究は、ポインタ解析アルゴリズムの実装のみにとどまったが、このポインタ解析の結果を使ったデータフロー解析や最適化を行い、どれほど効果があるかを調べる必要がある。

context-sensitive な解析の実装　今回実装したポインタ解析は、context-insensitive なものであるが、context-sensitive なものに拡張してより精度のよい結果を得ることを目指す。context-sensitive な解析ではさらに大きな集合を扱うことになるが、BDD を用いることで十分に解析可能であると考えられる。

変数の順序付けの検討　BDD のサイズは変数の順序付けによって変化する。今回は実装に利用したライブラリのデフォルトの順序付けを使ったが、よりよい順序がある可能性があり、それについて検討する必要がある。Marc らは変数の順序とメモリ使用量についての詳細な実験をしている [8]。

参考文献

[1] Whaley, J., Lam, M. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 131-144, 2004

[2] L. O. Andersen. Program analysis and specialization for C programming language, PhD Thesis, Dept. of Computer Science, University of Copenhagen, Denmark 1994.

[3] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 32-41, 1996.

[4] Bjarne Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. *Computational Complexity*, pp. 136-150, 1996.

[5] COINS Project. Coins project home page. http://www.coins-project.org/.

[6] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet?. *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 54-61, 2001.

[7] JavaBDD. A Java library for manipulating BDDs. http://javabdd.sourceforge.net/index.html.

[8] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 103-114, 2003.

規則 1: $\dfrac{\mathrm{p} = \&\mathrm{x}}{pt(\mathrm{p}) \ni \mathrm{x}}$ 規則 2: $\dfrac{\mathrm{p} = \mathrm{q} \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{p}) \ni \mathrm{x}}$

規則 3: $\dfrac{*\mathrm{p} = \&\mathrm{x} \quad pt(\mathrm{p}) \ni \mathrm{r}}{pt(\mathrm{r}) \ni \mathrm{x}}$ 規則 4: $\dfrac{*\mathrm{p} = \mathrm{q} \quad pt(\mathrm{p}) \ni \mathrm{r} \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{r}) \ni \mathrm{x}}$

規則 5: $\dfrac{pt(\mathrm{p}) \ni \mathrm{r} \quad pt(\mathrm{r}) \ni \mathrm{x}}{pt(*\mathrm{p}) \ni \mathrm{x}}$

規則 6: $\dfrac{\mathrm{p} = \&((*\mathrm{q}).\mathrm{f}) \quad pt(\mathrm{q}) \ni \mathrm{s}}{pt(\mathrm{p}) \ni \mathrm{s.f}}$ 規則 7: $\dfrac{(*\mathrm{p}).\mathrm{f} = \&\mathrm{x} \quad pt(\mathrm{p}) \ni \mathrm{s}}{pt(\mathrm{s.f}) \ni \mathrm{x}}$

規則 8: $\dfrac{(*\mathrm{p}).\mathrm{f} = \mathrm{q} \quad pt(\mathrm{p}) \ni \mathrm{s} \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{s.f}) \ni \mathrm{x}}$

規則 9: $\dfrac{pt(\mathrm{p}) \ni \mathrm{q} \quad pt(\mathrm{q.f}) \ni \mathrm{x}}{pt((*\mathrm{p}).\mathrm{f}) \ni \mathrm{x}}$ 規則 10: $\dfrac{\mathrm{s} = \mathrm{t} \quad pt(\mathrm{t.f}) \ni \mathrm{x}}{pt(\mathrm{s.f}) \ni \mathrm{x}}$

規則 11: $\dfrac{actuals \ni (\mathrm{f}, \mathrm{n}, \mathrm{q}) \quad formals \ni (\mathrm{f}, \mathrm{n}, \mathrm{p}) \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{p}) \ni \mathrm{x}}$

規則 12: $\dfrac{\mathrm{p} = \mathrm{f}(\ldots) \quad returns \ni (\mathrm{f}, \mathrm{q}) \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{p}) \ni \mathrm{x}}$

規則 13: $\dfrac{actuals \ni (\mathrm{fp}, \mathrm{n}, \mathrm{q}) \quad pt(\mathrm{fp}) \ni \mathrm{f} \quad formals \ni (\mathrm{f}, \mathrm{n}, \mathrm{p}) \quad pt(\mathrm{q}) \ni \mathrm{x}}{pt(\mathrm{p}) \ni \mathrm{x}}$

表 3: ポインタ解析のための推論規則