型推論に基づく手続き間ポインタ解析アルゴリズムの実装

東京工業大学
理学部
情報科学科

吉羽 和之
学籍番号: 9927000

平成17年度卒業論文

指導教官 佐々 政孝 教授

平成18年2月28日

目 次

第 1 章　はじめに **2**
1.1 背景 2
1.2 ポインタ解析とは 2
1.3 概要 2

第 2 章　準備 **3**
2.1 Disjoint Set 3
2.2 対象とする言語 4
2.3 型 5
2.4 推論規則 5
2.5 COINS の構成 6

第 3 章　ポインタ解析 **8**
3.1 アルゴリズムの概要 8
3.2 Disjoint Set の利用 8
3.3 join と cjoin 9
3.4 計算量 9

第 4 章　LIR 上でのポインタ解析 **12**
4.1 前提条件 12
4.2 LIR の構成 12
4.2.1 L 式の意味 12
4.3 LIR 上での解析方法 13
4.4 C 言語と LIR の対応 14
4.4.1 単純な代入文等 14
4.4.2 ポインタへのポインタがある場合 15
4.4.3 構造体 15
4.4.4 配列 15

第 5 章　関連研究 **18**

第 6 章　おわりに **19**
6.1 まとめ 19
6.2 今後の課題 19

第1章　はじめに

1.1　背景

コンパイラが持つ役割には、プログラムを機械語に変換することのほかに、より実行効率のよいコードに変換するということがある。このような変換をコード最適化と呼ぶ。現在、多くの最適化のアルゴリズムが提案されており、それらの中には、プログラムのある２点間で変数の値が変更されていないことを解析する必要があるものがある。しかし、C言語のようにポインタ変数がある言語ではこのような解析が困難になる。たとえば、ポインタ変数 p が変数 x を指しているとき、x の値は x への代入のほかに $*p$ への代入によっても変更される。したがって、ポインタ変数が存在する場合には、ポインタがどの変数を指しているのかを解析する必要がある。この解析を行うのがポインタ解析である。

1.2　ポインタ解析とは

名前や式が異なるものが同じメモリアドレスに割り当てられているときに、それらは互いに**別名**（alias）であるという。例えば、C言語のようにポインタ変数が許される言語では

```
int *p;
int n;
p = &n;
```

によって、p が n を指すようになり、$*p$ と n が別名となる。このようにポインタ変数が指すものの解析をすることを**ポインタ解析**（points-to analysis）と呼ぶ。

解析の方法には、制御フローやデータフローを考慮して解析する**フロー依存**（flow-sensitive）のものと、すべての文が実行されると仮定する**フロー非依存**（flow-insensitive）のものとに分けられる。一般に、フロー依存の解析のほうが解析の精度が高いが、時間とスペースを多く必要とする。また、手続き呼び出しによって生じる別名関係を解析する**手続き間の解析**（interprocedual analysis）もある。手続き間の解析は**文脈依存**（context-sensitive）のものと**文脈非依存**（context-insensitive）のものとに分けられる。

1.3　概要

本研究では、Bjarne Steensgaard が論文「Points-to Analysis in Almost Linear Time」[1] で提案したポインタ解析アルゴリズムをコンパイラ・インフラストラクチャCOINS[3] 上に実装した。この解析は、文脈非依存の手続き間解析を行うフロー非依存の解析である。

第2章　準備

本章では、ポインタ解析アルゴリズムの説明に必要になる用語や概念について説明する。

2.1　Disjoint Set

ポインタ解析では、ポインタが指す可能性のある変数の集合を求める。この集合をDisjoint Setを用いて表現する。Disjoint Setは互いに素な集合を表すデータ構造である [5]。各集合は、その集合の1要素である**代表元**（representative）によって表される。Disjoint Set上での操作は次のものがある。

- Find(x)：xが属する集合の代表元を返す。
- Union(x,y)：xが属する集合とyが属する集合を合併する。

Disjoint Setの効率の良い実現方法として、図2.1のように代表元を根とする木構造を用いる方法がある。各ノードは親ノードへのリンクを持っていて、根は自身へのリンクを持つ。

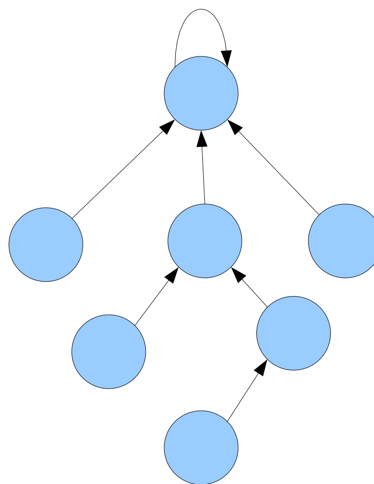

図 2.1: Disjoint Setの木による表現

木による表現を用いると、**経路圧縮**（path compression）と**ランクによる統合**（union by rank）という手法により実行時間を改善できる。

経路圧縮　Find(x)を行うときに、図2.2のようにxからrへのパス上のノードがrを直接指すようにする。

ランクによる統合 各ノードで、そのノードの高さの上界であるランクを管理し、UNION は、図 2.3 のようにランクが小さいほうのルートが大きいほうのルートを指すようにする。

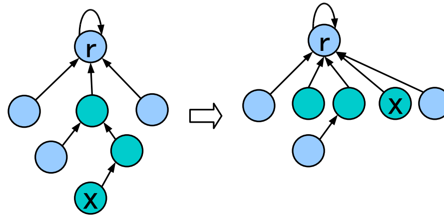

図 2.2: 経路圧縮

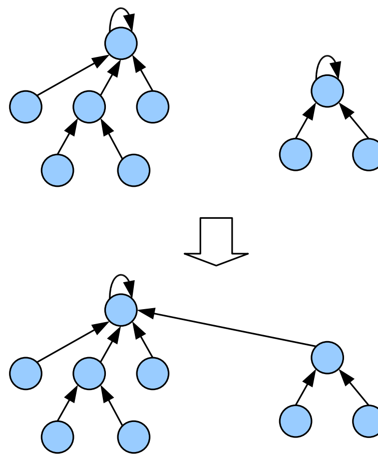

図 2.3: ランクによる統合

2.2 対象とする言語

以下、2.2 節〜2.4 節では Steensgaard[1] に沿って基礎事項を述べる。

以下に示す言語に対してポインタ解析を行う。この言語は C 言語の主な性質を取り出したもので、ポインタ変数、関数へのポインタ、メモリの動的な確保などを含んでいる。op(...) は算術演算のような基本的な演算を表し、allocate(y) はサイズ y のメモリ領域の動

$$
\begin{aligned}
S ::= {} & x = y \\
| \ & x = \&y \\
| \ & x = *y \\
| \ & x = op(y_1 \ldots y_n) \\
| \ & x = allocate(y) \\
| \ & *x = y \\
| \ & x = fun(f_1 \ldots f_n) \rightarrow (r_1 \ldots r_m)\ S^* \\
| \ & x_1 \ldots x_m = p(y_1 \ldots y_n)
\end{aligned}
$$

的な確保を行う。関数の定義は、$\mathrm{x} = \mathrm{fun}(\mathrm{f}_1 \ldots \mathrm{f}_n) \to (\mathrm{r}_1 \ldots \mathrm{r}_m)$ S*によって表される。f_1 から f_n は仮引数であり、r_1 から r_m は戻り値である。関数呼び出しは $\mathrm{x}_1 \ldots \mathrm{x}_m = \mathrm{p}(\mathrm{y}_1 \ldots \mathrm{y}_n)$ のように行われる。この言語ではC言語と異なり、複数の値を返す関数を定義できる。

2.3 型

ポインタ解析を行うために**型**を定義する。

$$\begin{aligned} \alpha &::= \tau \times \lambda \\ \tau &::= \bot \mid \mathrm{ref}(\alpha) \\ \lambda &::= \bot \mid \mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) \end{aligned}$$

τ は変数のメモリ上の位置を表し、λ は関数のシグネチャを表す。たとえば、$\mathrm{ref}(\tau \times \bot)$ は位置 τ を指すポインタを表し、$\mathrm{ref}(\bot \times\ \mathrm{lam}(\alpha_1, \alpha_2)(\alpha))$ は引数を2つとり値を1つ返す関数を表す。そして、単なる変数は $\mathrm{ref}(\bot \times \bot)$ と表される。

2.4 推論規則

推論規則は、代入文や関数呼び出しなどがあった場合に、それぞれの変数の型がどのようであるべきかを定める。たとえば、x=&y に対する規則

$$\frac{A \vdash \mathrm{x} : \mathrm{ref}(\tau \times _) \quad A \vdash \mathrm{y} : \tau}{A \vdash \mathit{welltyped}(\mathrm{x} = \&\mathrm{y})}$$

は、x が指す型と y の型が一致していなければならないと述べている。

x=y に対する規則は、普通に考えると

$$\frac{A \vdash \mathrm{x} : \mathrm{ref}(\alpha) \quad A \vdash \mathrm{y} : \mathrm{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{y})}$$

となりそうである。この規則のもとでは、x=y という文があった場合 x と y は常に同じ変数の集合を指すことになる。しかし、y がポインタでない場合には、これは正確でない。そこで、次のような順序関係 $\trianglelefteq$ を導入する。

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

この順序関係を使って規則を次のように変える。

$$\frac{A \vdash \mathrm{x} : \mathrm{ref}(\alpha_1) \quad A \vdash \mathrm{y} : \mathrm{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{y})}$$

この規則のもとでは、y が $\bot$ を指している限りは、x は集合を指すことはない。

表 2.1 は、各文に対する推論規則である。

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(\alpha_1) \\ A \vdash \mathrm{y} : \mathrm{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(\tau \times _) \\ A \vdash \mathrm{y} : \tau \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x} = \&\mathrm{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(\alpha_1) \\ A \vdash \mathrm{y} : \mathrm{ref}(\mathrm{ref}(\alpha_2) \times _) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x} = *\mathrm{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(\alpha) \\ A \vdash \mathrm{y}_i : \mathrm{ref}(\alpha_i) \\ \forall i \in [1 \ldots n], \alpha_i \trianglelefteq \alpha \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{op}(\mathrm{y}_1 \ldots \mathrm{y}_n))}$$

$$\frac{A \vdash \mathrm{x} : \mathrm{ref}(\mathrm{ref}(_) \times _)}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{allocate}(\mathrm{y}))}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(\mathrm{ref}(\alpha_1) \times _) \\ A \vdash \mathrm{y} : \mathrm{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(*\mathrm{x} = \mathrm{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x} : \mathrm{ref}(_ \times \mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathrm{f}_i : \mathrm{ref}(\alpha_i) \\ A \vdash \mathrm{r}_j : \mathrm{ref}(\alpha_{n+j}) \\ \forall\ \mathrm{s} \in \mathrm{S}^* : A \vdash \mathit{welltyped}(\mathrm{s}) \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x} = \mathrm{fun}(\mathrm{f}_1 \ldots \mathrm{f}_n) \rightarrow (\mathrm{r}_1 \ldots \mathrm{r}_m)\ \mathrm{S}^*)}$$

$$\frac{\begin{array}{c} A \vdash \mathrm{x}_j : \mathrm{ref}(\alpha'_{n+j}) \\ A \vdash \mathrm{p} : \mathrm{ref}(_ \times \mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathrm{y}_i : \mathrm{ref}(\alpha'_i) \\ \forall i \in [1 \ldots n] : \alpha'_i \trianglelefteq \alpha_i \\ \forall j \in [1 \ldots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j} \end{array}}{A \vdash \mathit{welltyped}(\mathrm{x}_1 \ldots \mathrm{x}_m = \mathrm{p}(\mathrm{y}_1 \ldots \mathrm{y}_n))}$$

表 2.1: 各文に対する推論規則

2.5 COINSの構成

一般にコンパイラはフロントエンド (front end) とバックエンド (back end) から構成される。フロントエンドは原始プログラム (source program) を中間コード (intermediate code) と呼ばれる内部形式に変換する。バックエンドは中間コードを計算機の機械コードに変換する。フロントエンドはさらに字句解析器 (lexical analyzer)、構文解析器 (syntax analyzer)、意味解析器 (semantic analyzer) に分けられる。バックエンドは最適化器 (optimizer) とコード生成器 (code generator) に分けられる。これらの各部分はコンパイラのフェーズと呼ばれる。

本研究で用いるコンパイラ・インフラストラクチャCOINS の概念図を図 2.4 に示す。COINS では、複数の入力言語、複数の目的機種に対応する 2 つの中間表現がある。入力言語の論理構造に近いレベルの中間表現を高水準中間表現 (high-level intermediate representation, HIR) と呼び、機械語に近いレベルの中間表現を低水準中間表現 (low-level intermediate represantation, LIR) と呼ぶ。

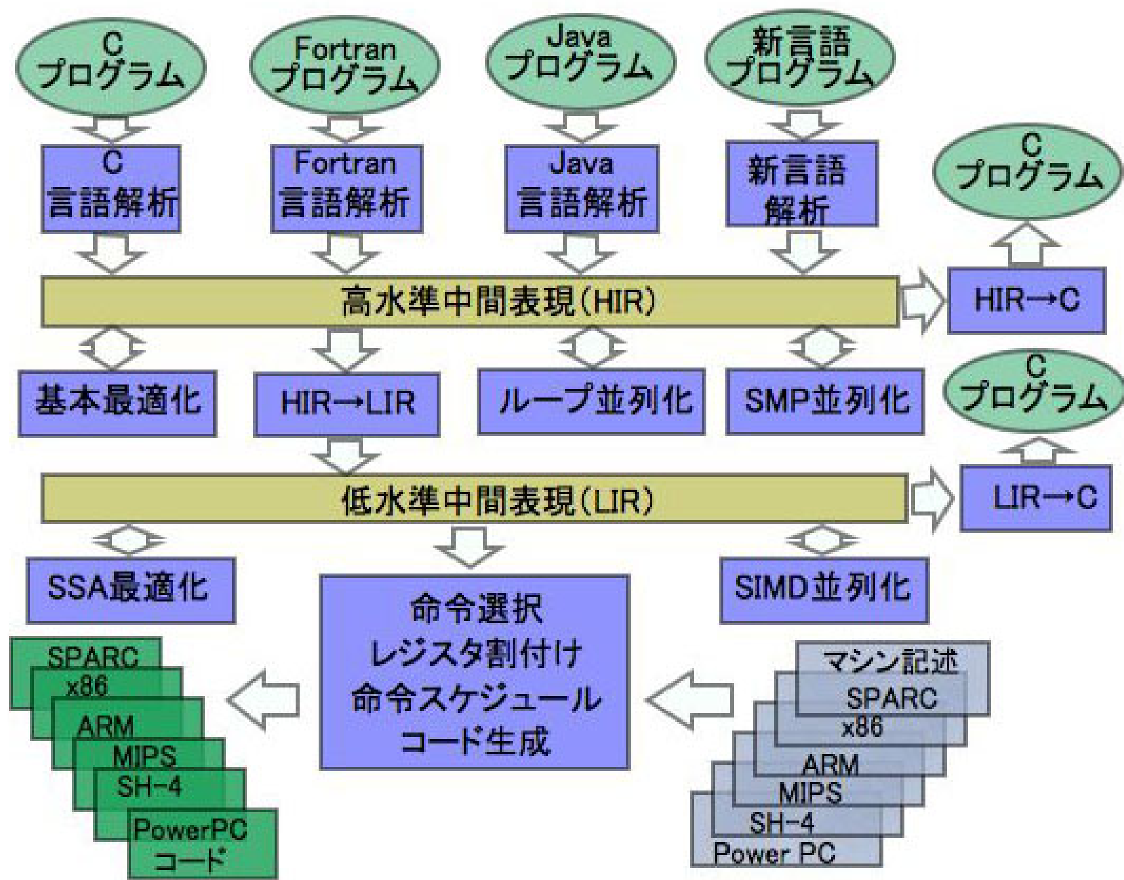

図 2.4: 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図

第3章　ポインタ解析

3.1　アルゴリズムの概要

今回採用したアルゴリズムは次の手順で解析を行う。

1. プログラム中のすべての変数の型を、$\mathrm{ref}(\bot \times \bot)$ で初期化する。
2. 各文を1度ずつ見て、その文が well-typed になるように、表 2.1 各規則にしたがって型の内容を置き換える。

ここで、プログラム

```
if(...)
   p = &x;
else
   p = &y;
```

を例に、アルゴリズムの動きを説明する。まず、すべての変数の型の初期状態は図 3.1(A) のように $\mathrm{ref}(\bot \times \bot)$ である。次に、最初の文が p=&x なので規則により、p の型の左の $\bot$ は x の型 τ_2 に置き換えられて (B) になる。2番目の文は p=&y なので、p の型の中の τ_2 は y の型 τ_3 に置き換えられて (C) になり、これが結果になる。つまり、p の型が指す型は τ_3 であり、型が τ_3 である変数は x と y なので、p が指す可能性のある変数は x と y である。

$$
\begin{array}{ccccc}
\mathrm{p}:\tau_1 = \mathrm{ref}(\bot\times\bot) & & \mathrm{p}:\tau_1 = \mathrm{ref}(\underline{\tau_2}\times\bot) & & \mathrm{p}:\tau_1 = \mathrm{ref}(\underline{\tau_3}\times\bot) \\
\mathrm{x}:\tau_2 = \mathrm{ref}(\bot\times\bot) & \Rightarrow & \mathrm{x}:\tau_2 = \mathrm{ref}(\bot\times\bot) & \Rightarrow & \mathrm{x}:\underline{\tau_3} = \mathrm{ref}(\bot\times\bot) \\
\mathrm{y}:\tau_3 = \mathrm{ref}(\bot\times\bot) & & \mathrm{y}:\tau_3 = \mathrm{ref}(\bot\times\bot) & & \mathrm{y}:\tau_3 = \mathrm{ref}(\bot\times\bot) \\
\text{(A)} & & \text{(B)} & & \text{(C)}
\end{array}
$$

図 3.1: ポインタ解析の例

3.2　Disjoint Set の利用

図 3.1 で示した例のように、型を置き換える方法では、一度に複数の型を置き換える必要がでてくる。そこで、τ や $\bot$ などを Disjoint Set の要素であると考えて、$\bot$ を τ で置き換える代わりにそれらの Union をとることにする。この方法で 3.1 節と同じ例を解析すると、図 3.2 のようになる。

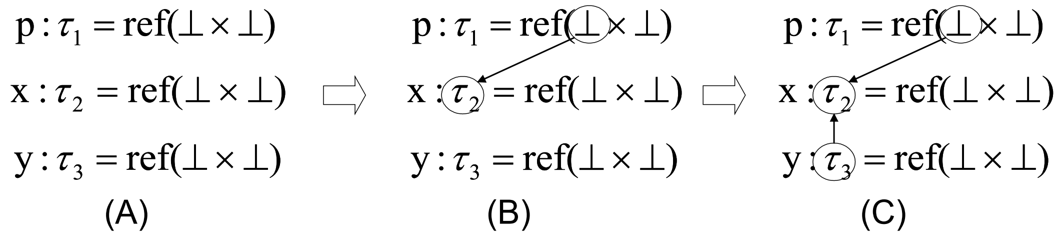

図 3.2: Disjoint Set の利用．矢印は子ノードからその親ノードへのポインタを表す．

3.3 join と cjoin

x = y のように規則の中に $\trianglelefteq$ がある場合には、その文を見ただけでは 2 つの型を Union すべきかどうか分からない。もし $\trianglelefteq$ の左辺が $\bot$ 以外の場合は、規則にあわせるために 2 つの型を Union しなければならない。一方 $\trianglelefteq$ の左辺が $\bot$ の場合には Union する必要はないが、ほかの式を処理したときに $\bot$ 以外のものに変わる可能性があり、そのときには右辺の型と Union しなければならない。

このような状況を扱うために、型 $\bot$ には、それが $\bot$ 以外の型に変わったときに Union しなければならない型の集合を覚えさせておく。これらを行うのが表 3.1 にある手続き cjoin と join である。cjoin は左辺の型が $\bot$ のときには、左辺の型が覚えておく集合（pending）に右辺の型を加え、そうでないときには、join を呼んで 2 つの型を Union する。

join は、２つの型を Union する。そして、$\bot$ が $\bot$ 以外の型と Union されたときには $\bot$ が覚えていた集合をの要素も join する。

各規則の中で、２つの型の関係が $\trianglelefteq$ であると定められているときには、それらの型を cjoin し、そうでないときには join を行う。規則が定める条件に一致するように変数に型を与える手続きを表 3.2 に示す。これらは、表 2.1 にある各推論規則を逆方向に見て、各文が well-typed になるように変数の型を定める手続きである。

3.4 計算量

図 3.2 で示したポインタ解析にかかる時間は、プログラム中の各文を訪れるのにかかる時間と、型に対して Union や Find を行うのにかかる時間によって決まる。各文を訪れるのにかかる時間は入力プログラムのサイズに比例する。また、n 個の要素に関する Disjoint Set 上での操作を m 回行った場合の実行時間は $O(m\alpha(m,n))$ であることが証明されている [5]。$\alpha(m,n)$ はアッカーマン関数の逆関数で、非常にゆっくり増加する関数である。

型の個数と、Union、Find の回数は入力プログラムのサイズ N に比例するので、解析にかかる時間は $O(N\alpha(N,N))$ である。

```
cjoin(e1, e2)
  if type(e2) = ⊥ then
    pending(e2) ← {e1} ∪ pending(e2)
  else
    join(e1, e2)
```

```
join(e1, e2)
  let t1 = type(e1)
      t2 = type(e2)
      e = ecr-union(e1, e2) in
    if t1 = ⊥ then
      type(e) ← t2
      if t2 = ⊥ then
        pending(e) ← pending(e1) ∪ pending(e2)
      else
        for x ∈ pending(e1) do join(e, x)
    else
      type(e) ← t1
      if t2 = ⊥ then
        for x ∈ pending(e2) do join(e, x)
      else
        unify(t1, t2)
```

表 3.1: 手続き cjoin と join. **pending**(e) は e が覚えておく集合. ecr-union(e_1, e_2) は e_1 と e_2 を UNION した後に新しくできた集合の代表元を返す。

$x = y$
let $\mathrm{ref}(\tau_1 \times \lambda_1) = \mathrm{type}(\mathrm{ecr}(x))$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(y))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{cjoin}(\lambda_1, \lambda_2)$

$x = \&y$
let $\mathrm{ref}(\tau_1 \times _) = \mathrm{type}(\mathrm{ecr}(x))$
$\tau_2 = \mathrm{ecr}(y)$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{join}(\tau_1, \tau_2)$

$x = *y$
let $\mathrm{ref}(\tau_1 \times \lambda_1) = \mathrm{type}(\mathrm{ecr}(x))$
$\mathrm{ref}(\tau_2 \times _) = \mathrm{type}(\mathrm{ecr}(y))$ **in**
if $\mathrm{type}(\tau_2) = \bot$ **then**
$\mathrm{settype}(\tau_2,\mathrm{ref}(\tau_1 \times \lambda_1))$
else
let $\mathrm{ref}(\tau_3 \times \lambda_3) = \mathrm{type}(\tau_2)$ **in**
if $\tau_1 \neq \tau_3$ **then** $\mathrm{cjoin}(\tau_1, \tau_3)$
if $\lambda_1 \neq \lambda_3$ **then** $\mathrm{cjoin}(\lambda_1, \lambda_3)$

$x = \mathrm{op}(y_1 \ldots y_n)$
for $i \in [1 \ldots n]$ **do**
let $\mathrm{ref}(\tau_1 \times \lambda_1) = \mathrm{type}(\mathrm{ecr}(x))$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(y_i))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{cjoin}(\lambda_1, \lambda_2)$

$x = \mathrm{allocate}(y)$
let $\mathrm{ref}(\tau \times _) = \mathrm{type}(\mathrm{ecr}(x))$ **in**
if $\mathrm{type}(\tau) = \bot$ **then**
$\mathrm{settype}(\tau,\mathrm{ref}(\bot \times \bot))$

$*x = y$
let $\mathrm{ref}(\tau_1 \times _) = \mathrm{type}(\mathrm{ecr}(x))$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(y))$ **in**
if $\mathrm{type}(\tau_1) = \bot$ **then**
$\mathrm{settype}(\tau_2,\mathrm{ref}(\tau_2 \times \lambda_2))$
else
let $\mathrm{ref}(\tau_3 \times \lambda_3) = \mathrm{type}(\tau_1)$ **in**
if $\tau_1 \neq \tau_3$ **then** $\mathrm{cjoin}(\tau_3, \tau_2)$
if $\lambda_1 \neq \lambda_3$ **then** $\mathrm{cjoin}(\lambda_3, \lambda_2)$

$x = \mathrm{fun}(f_1 \ldots f_n) \rightarrow (r_1 \ldots r_m)$ S*
let $\mathrm{ref}(_ \times \lambda) = \mathrm{type}(\mathrm{ecr}(x))$ **in**
if $\mathrm{type}(\lambda) = \bot$ **then**
$\mathrm{settype}(\lambda,\mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\mathrm{ref}(\alpha_i) = \mathrm{type}(\mathrm{ecr}(f_i))$, **for** $i \leq n$
$\mathrm{ref}(\alpha_i) = \mathrm{type}(\mathrm{ecr}(r_{i-n}))$, **for** $i > n$
else
let $\mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathrm{type}(\lambda)$ **in**
for $i \in [1 \ldots n]$ **do**
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(f_i))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{join}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{join}(\lambda_2, \lambda_1)$
for $i \in [1 \ldots m]$ **do**
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(r_i))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{join}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{join}(\lambda_1, \lambda_2)$

$x_1 \ldots x_m = p(y_1 \ldots y_n)$
let $\mathrm{ref}(_ \times \lambda) = \mathrm{type}(\mathrm{ecr}(p))$ **in**
if $\mathrm{type}(\lambda) = \bot$ **then**
$\mathrm{settype}(\lambda,\mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\alpha_i = \bot \times \bot$
let $\mathrm{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathrm{type}(\lambda)$ **in**
for $i \in [1 \ldots n]$ **do**
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(y_i))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{cjoin}(\lambda_1, \lambda_2)$
for $i \in [1 \ldots m]$ **do**
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathrm{ref}(\tau_2 \times \lambda_2) = \mathrm{type}(\mathrm{ecr}(x_i))$ **in**
if $\tau_1 \neq \tau_2$ **then** $\mathrm{cjoin}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ **then** $\mathrm{cjoin}(\lambda_2, \lambda_1)$

表 3.2: 各変数に型を与える手続き. ecr(x) は変数 x の型の属する集合の代表元. settype(τ,t) は型 τ の中身を t に変える.

第4章　LIR上でのポインタ解析

4.1　前提条件

本研究では、次のような前提条件のもとで解析を行う。

- 入力言語はC言語。
- LIR は HIR から変換された直後の状態。

4.2　LIR の構成

原始プログラムはL-module と呼ばれる LIR コードに変換される。L-module はグローバルシンボルテーブルといくつかの L-function を含む。L-function はローカルシンボルテーブルと L-sequence を含む。L-sequence は L 式のリストであり `PROLOGUE` 式で始まり `EPILOGUE` 式で終わる。図 4.1 に LIR の内部構造を示す。

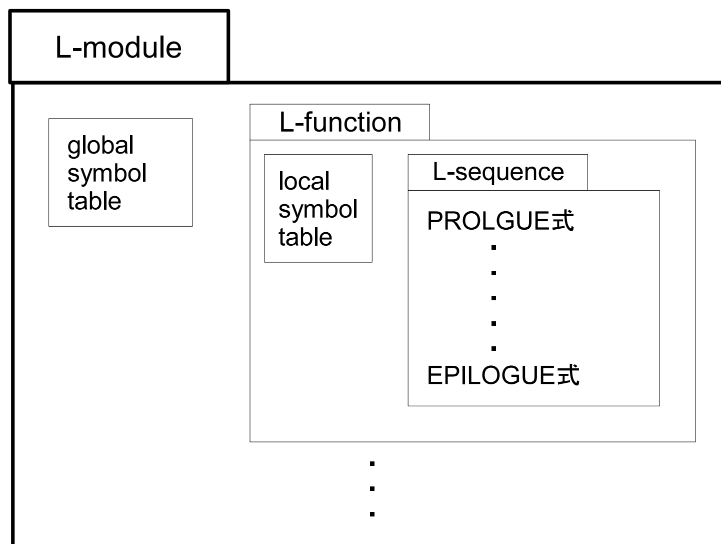

図 4.1: LIR の内部構造

4.2.1　L 式の意味

ここで、一部の L 式についてその意味を説明する。

Const 式

Const 式には、Intconst 式と Floatconst 式がある。(`INTCONST I32 1`) は 32 ビット整数の 1 を表し、(`FLOATCONST F32 2.0`) は 32 ビット浮動小数点数の 2.0 を表す。

Addr 式

Addr 式は、変数のアドレスを表すためのものであり、グローバル変数のアドレスを表す Static 式と、ローカル変数のアドレスを表す Frame 式がある。たとえば、(`FRAME I32` "a") はローカル変数 a のアドレスを意味する。

Mem 式

Mem 式は、メモリ上の値を表す。たとえば、(`MEM I32` (`STATIC I32` "a")) はグローバル変数 a の値を表す。また、ポインタ変数 p の先にしまわれている値は、(`MEM I32` (`MEM I32` (`FRAME I32` "p"))) のように参照される。

Set 式

Set 式は、第一引数の値に第二引数の値を代入する。

(`SET I32` (`MEM I32` (`FRAME I32` "x")) (`MEM I32` (`FRAME I32` "y")))

は x = y を表す。

Call 式

(`CALL` x_1 ($x_2 \ldots x_n$) ($y_1 \ldots y_m$)) は、x_1 が表すメモリにある関数に引数 $x_2 \ldots x_n$ を与え、結果の値を $y_1 \ldots y_m$ に代入する。

Interface 式

Interface 式には、Prologue 式と Epilogue 式がある。Prologue 式は仮引数のリストを持ち、Epilogue 式は返り値のリストを持つ.

これらの他に、算術演算を行う Add 式や Mul 式がある。

4.3 LIR 上での解析方法

LIR 上での解析は次のように行う。
L-module 内の各 L-function に対して、

1. Prologue 式、Epilogue 式から仮引数と返り値を調べ、表 3.2 中の関数定義に対する手続きを呼ぶ。
2. L-sequence 中の L 式を調べ、それが Set 式か Call 式であったらその L 式に対応する C コードに基づいて、表 3.2 中の手続きを呼ぶ。

以上のように、解析のためには入力プログラムの各文がどのような L 式に変換されるかを知る必要がある。

4.4 C 言語と LIR の対応

ここでは入力プログラムの各文と LIR コードがどのように対応するかを説明する。まず、2.2 節で示した言語の文がどのような L 式に変換されるかを見る。次に、ポインタへのポインタや構造体、配列といった 2.2 節の言語で扱っていないものをどう解析するかを述べる。

4.4.1 単純な代入文等

まず、2.2 節で示した言語の文は LIR 上では次のように変換されている。

```
x = y
  ⟹ (SET I32 (MEM I32 (FRAME I32 "x")) (MEM I32 (FRAME I32 "y")))

x = &y
  ⟹ (SET I32 (MEM I32 (FRAME I32 "x")) (FRAME I32 "y"))

x = *y
  ⟹ (SET I32 (MEM I32 (FRAME I32 "x"))
              (MEM I32 (MEM I32 (FRAME I32 "y"))))

*x = y
  ⟹ (SET I32 (MEM I32 (MEM I32 (FRAME I32 "x")))
              (MEM I32 (FRAME I32 "y")))

x = allocate(y)
  ⟹ (CALL (STATIC I32 "malloc")
            (MEM I32 (FRAME I32 "y"))
            (MEM I32 (FRAME I32 "functionvalue")))
     (SET I32 (MEM I32 (FRAME I32 "x")) (MEM I32 (FRAME I32 "functionvalue")))

x = p(y1...yn)
  ⟹ (CALL (STATIC I32 "p")
            ((MEM I32 (FRAME I32 "y1"))...(MEM I32 (FRAME I32 "yn")))
            ((MEM I32 (FRAME I32 "x"))))
```

算術演算式は、x - (y - z) と (x - y) - z のように括弧のつき方によって、異なるコードに変換される。また、$x \times y + z$ のように異なる種類の演算が含まれる場合もある。しかし、今回用いたアルゴリズムでは、演算の順番や種類は考える必要はなく、 Set 式の第二引数が Add 式や Mul 式の場合には、それらの引数を再帰的にたどりオペランドとなる変数を得ればよい。たとえば、x - (y - z) と $x \times y + z$ は同じ op(x, y, z) として扱う。

また、グローバル変数は、別のファイルの中で使われている可能性などがあるので、保守的な方法で解析をする。

4.4.2　ポインタへのポインタがある場合

ポインタ変数 p が別のポインタを指しているときには、$**p$ のような使われ方が考えられる。例えば、$**p=y$ という文は

```
(SET I32 (MEM I32 (MEM I32 (MEM I32 (FRAME I32 "p"))))
          (MEM I32 (FRAME I32 "y")))
```

と変換される。このような L 式を見つけたときには、もとの文が、

```
t = *p;
*t = y;
```

であるものとして解析する。

4.4.3　構造体

構造体のメンバーへの代入は、構造体全体を表す単なる変数への代入として扱う。たとえば、次のように構造体が宣言されたとする。

```
struct point{
  int x;
  int y;
}
struct point pt;
```

このとき、変数 pt はメモリ上で次の図のように表現される。

<table><tr><td rowspan="2">pt:</td><td>pt.x</td></tr><tr><td>pt.y</td></tr></table>

そして、pt.y = a という文は

```
(SET I32 (MEM I32 (ADD I32 (FRAME I32 "pt") (INTCONST I32 4)))
          (MEM I32 (FRAME I32 "a")))
```

という L 式に変換される。このような L 式を見つけたときには、もとの文が pt=a であったとして解析する。

4.4.4　配列

配列も単なる変数として扱う。

1 次元配列

変数 a が 1 次元の配列であるとすると、1 要素のサイズを 4 バイトとして a[1] の値は、

```
(MEM I32 (ADD I32 (FRAME I32 "a")
                  (MUL I32 (INTCONST I32 1) (INTCONST I32 4))))
```

のように参照される。このような L 式は、

```
(MEM I32 (FRAME I32 "a"))
```

であるとみなす。つまり、a[i] = x という文は、a = x という文とみなして解析する。

多次元配列

C 言語では、多次元配列を、配列の配列として表現する方法 (A) と、ポインタの配列として表現する方法 (B) がある。

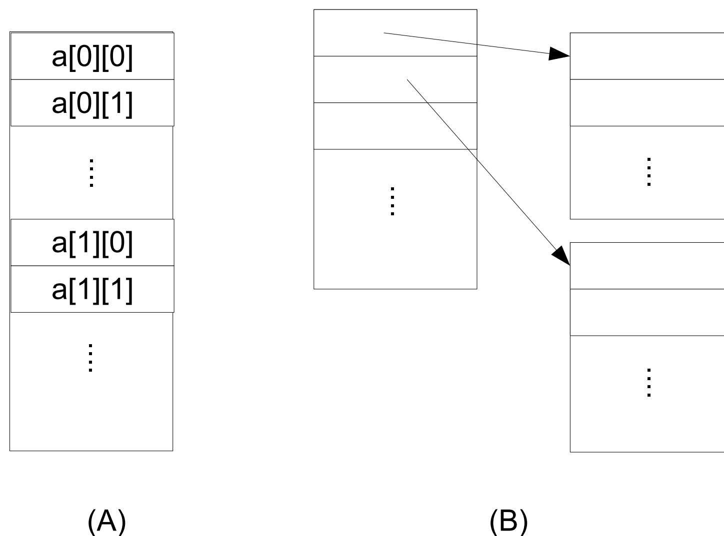

図 4.2: 多次元配列の表現

配列 a が方法 (A) で表された 10×10 の配列だとすると、a[1][1] の値は、

```
(MEM I32 (ADD I32 (ADD I32 (FRAME I32 "a")
                           (MUL I32 (INTCONST I32 1) (INTCONST I32 40)))
                  (MUL I32 (INTCONST I32 1) (INTCONST I32 4))))
```

のように参照される。このような L 式は、

```
(MEM I32 (FRAME I32 "a"))
```

であるとみなす。つまり、a[i][j] = x という文は、a = x という文とみなして解析する。

また、配列 a が方法 (B) で表された 10×10 の配列だとすると、a[1][1] の値は、

```
(MEM I32 (ADD I32 (MEM I32 (ADD I32 (FRAME I32 "a")
                                    (MUL I32 (INTCONST I32 1)
                                             (INTCONST I32 4)))
                  (MUL I32 (INTCONST I32 1) (INTCONST I32 4)))))
```

のように参照される。このようなL式は、

```
(MEM I32 (MEM I32 (FRAME I32 "a")))
```

であるとみなす。つまり、a[i][j] = x という文は、$*a = x$ という文とみなして解析する。

第5章　関連研究

本章では、ポインタ解析に関して行われている研究について述べる。

フロー依存の解析 本論文で実装したポインタ解析の方法はフロー非依存の解析であった。それに対し、フロー依存の解析には、Emami らによる方法 [6] や Landi らによる方法 [7] などがある。

一般に、フロー依存の解析によって得られる結果のほうがフロー非依存の解析による結果よりも精度が良い。しかし、フロー依存の解析による精度の改善はそれほど大きくないとする研究がある [8]。

文脈依存の解析 文脈依存の解析にかかる時間は、最悪時には指数関数的になるので、効率的に解析を行う方法が提案されている [6, 9, 10]。しかし、フロー依存の解析の場合と同様に文脈依存の解析による精度の改善はそれほど大きくないようである [11, 12, 13]。

また、部分的に文脈依存の解析手法 [14, 15] も提案されており、これらは有効な手法であるようである。

SSA 形式上でのポインタ解析 SSA 形式（static single assignment form）とは、プログラムの表現形式の一つで、そのプログラム上のすべての変数の使用に対して、その使用に対応する定義は一カ所しかないように表現したものである。

ポインタ解析の方法にはフロー依存のものとフロー非依存のものがあると述べたが、SSA 形式に変換されたプログラムではフロー情報が反映された形式になっている。したがって、SSA 形式のプログラムでフロー非依存の解析をしても、もとのプログラムでフロー依存の解析をしたのと同じような結果が得られることが期待される。

Hasti らの論文 [16] では SSA 形式上でのポインタ解析の方法が述べられている。これは、各 $*p$ に対し p が指す可能性のある変数の集合を求め、$*p$ をそれらに置き換えるというものである。

第6章　おわりに

6.1　まとめ

本論文では、フロー非依存のポインタ解析である Steensgaard の方法を COINS の LIR 上で実装した。今回採用したポインタ解析アルゴリズムの特徴についてまとめる。

型と推論規則による解析 変数のメモリ上での位置を記述するための型を導入し、推論規則によって変数に型を与えることでポインタ解析を行う。また、推論規則に $\unlhd$ という順序関係を入れることで解析の精度が落ちないようにした。

手続き間の解析 手続き呼び出しがある場合でも、他の代入文と同じように扱うことができる。通常、手続き間の解析を行う場合には、手続きの呼び出し関係をグラフ化した呼び出しグラフを作成するが、このアルゴリズムではその必要がない。

計算量 今回採用したアルゴリズムは、入力プログラムのサイズを N とすると $O(N\alpha(N,N))$ であり、入力プログラムのサイズにほぼ比例する時間でポインタ解析を行える。これは他のポインタ解析アルゴリズムよりも高速である。

6.2　今後の課題

今後の課題は次のとおりである。

前提条件を緩める 今回は、LIR 上で最適化が行われていないことを前提としたが、コンパイル時には複数の最適化を行うのが普通である。したがって、最適化によって書き換えられたプログラム上でもポインタ解析が行えるようにするべきである。そのためには、書き換えられたプログラムが表 3.2 の手続きで解析できるのか、できない場合にどうするのかを考える必要がある。

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。

構造体や配列の解析 今回は、構造体を単なる変数として扱ったが、構造体のメンバーについても詳しく解析するようにしたい。これについては、今回のアルゴリズムを拡張したものが Steensgaard によって提案されている [2]。また千代の提案もある [17, 18]。配列についても要素ごとの解析を行うアルゴリズムがいくつか提案されているので、それらについても検討していきたい。

謝辞

本研究に進めるにあたり多大なる御指導御鞭撻を頂いた、東京工業大学数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。

また、お忙しい中多くのご指導、助言をして下さった佐々研究室の皆様に心から御礼申し上げます。

参考文献

[1] 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.

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

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

[4] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest 共著.
浅野哲夫, 岩野和生, 梅尾博司, 山下雅史, 和田幸一 共訳.
『アルゴリズム イントロダクション 第２巻』. 近代科学社, 1995.

[6] M. Emami, R. Ghiya and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 242-256, 1994.

[7] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 235-248, 1992.

[8] 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.

[9] R. P. Wilson. Efficient Context-Sensitive Pointer Analysis for C Programs. PhD thesis, Stanford University, Dec. 1997.

[10] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp. 1-12, 1995

[11] J. S. Foster, M. Fähndrich and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. *Seventh International Static Analysis Symposium*, 2000.

[12] M. Das, B. Liblit, M. Fähndrich and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. *Seventh International Static Analysis Symposium*, 2001.

[13] E. Ruf. Context-insensitive alias analysis reconsidered. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp. 13-22, 1995

[14] M. Fähndrich, J. Rehof and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp. 253-263, 2000.

[15] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic aubtyping to CFL-reachability. *28th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, 2001.

[16] Rebecca Hasti and Susan Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pp. 97-105, 1998.

[17] 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, vol. 45, no. SIG 12 (PRO 23), pp. 52-66, Nov 2004.

[18] 千代英一郎, Deductive System によるCプログラムのポインタ解析, 情報処理学会論文誌, vol. 47, no. SIG 2 (PRO 28), pp. 1-17, Feb 2006.