平成 24 年度

学士学位論文

C プログラムを対象としたシェープ解析アルゴリズムの実装

An implementation of a shape analysis algorithm for C programs

1130391　　水野 雄介

指導教員　　高田 喜朗

2013 年 3 月 1 日

高知工科大学 情報学群

要 旨

C プログラムを対象としたシェープ解析アルゴリズムの実装

水野 雄介

近年, ソフトウェアが複雑化するにつれて, プログラムの実行効率や信頼性を向上させるためのプログラム解析技術がますます重要になってきている. 重要な静的解析技術のひとつとしてポインタ解析がある. ポインタ解析とは, プログラム内の変数が実行時に指す可能性のあるメモリ領域を求める静的解析である. ポインタの指す先に関する情報はメモリ参照式間の依存関係を正確に解析する事で求められる. しかしポインタ解析では通常, リストのような再帰的なデータ構造を一つの領域として解析することから解析精度が低下してしまう. そこで, ポインタ解析よりも精度の高い解析手法の一つとしてシェープ解析が提案されている.

シェープ解析とは, 述語抽象を行う事によりプログラム内のヒープ割り当て構造を求める静的解析である. シェープ解析では, ポインタ解析では行わない再帰的なデータ構造の解析を行う事ができる.

シェープ解析の効率化や解析対象の拡張に関する研究が近年行われているが, 公開されている入手容易な実装がなく, 利用者が容易に解析を行う事ができない. そこで, 本研究では COINS コンパイラ・インフラストラクチャに対して, Reps が提案しているシェープ解析法を実装する. その結果, ポインタ解析では正確な解析結果を得られない連結リストを含むアルゴリズムを正確に解析することができた.

キーワード　ポインタ解析, シェープ解析, COINS, 再帰的データ構造

Abstract

An implementation of a shape analysis algorithm for C programs

Yuusuke MIZUNO

In recent years, the program-analysis technologies for improving the efficiency and reliability of a program is becoming stil more important as software is complicated. The points-to analysis is one of the important static analysis technologies. The points-to analysis statically computes the memory areas to which a variable in a program may point during the execution of that program. That computation can be achieved by analyzing the dependency between memory references correctly. However, in the points-to analysis, a recursive data strucutre like a list is usually abstracted as one element, and it sometimes causes low analysis accuracy. To solve this problem, the shape analysis is proposed as one of the analysis techniques with higher-precision than points-to analysis.

The shape analysis statically analyses heap-allocated structures in a program. It can analyse recursive data structures that cannot be analysed by the points-to analysis.

Although researches on the extension and improvement of the shape analysis have been performed in recent years, there is no freely and easily accessible implementation of the shape analysis.

Therefore, in this study, we implement on the COINS compiler infrastructure a simple shape analysis algorithm that Reps has proposed. As a result, we could correctly analysed an algorithm using a linked list that could not be analysed by the points-to

analysis.

key words Points-to analysis, Shape analysis, COINS, Recursive data structure

目次

目次

図目次

表目次

第 1 章

はじめに

1.1 背景

近年, ソフトウェアが複雑化するにつれて, プログラムの実行効率や信頼性を向上させるためのプログラムがますます重要になってきている. 重要な静的解析技術の一つにポインタ解析がある. ポインタ解析とは, プログラム内の変数が実行時に指す可能性のあるメモリ領域を求める静的解析である. ポインタの指す情報は, 情報はメモリ参照式間の依存関係を正確に解析することで求めることができる. しかし通常のポインタでは, 動的なデータ構造 (リストや木構造) を解析することができない. そこで, ポインタ解析よりも精度の高い解析手法の一つとしてシェープ解析 [1] が提案されている.

シェープ解析とは, 制御フローを用いて各ノードの経路を解析することによりデータ構造を求める静的解析である. シェープ解析では, ポインタ解析では行わない動的なデータ構造を考慮して解析を行う事ができる.

シェープ解析の効率化や解析対象の拡張 [2] やマルチスレッドへの対応 [3] に関する研究は近年行われているが, 公開されている入手容易な実装がなく利用者が容易に解析を行う事ができない.

本研究では, COINS コンパイラ・インフラストラクチャ(以下 COINS)[4] に対して, Reps[1] が提案している解析手法を実装する.

1.2　対象とする言語

今回の実装では以下に示す言語に対してシェープ解析を行う．この言語は, C のサブセットとリスト操作関数を含んでいる.

- 代入文
- 条件文
- ループ文 (while)
- データ読み込み
- goto 記述
- 適切な述語 (atom, null, 比較)
- リスト操作関数 (cons, car, cdr)
- 言語の型 (int,double)

数値や文字列などリストではない値については atom と置き換えて表現する．Reps の解析法では, プログラム言語 Lisp を解析対象とし, リスト操作関数 cons, car, cdr を使って生成され操作されるリストの解析を行う．本研究では, これらのリスト操作関数が予め定義されている C 言語を仮定し, Reps の解析法と同様にこれらの関数によって生成され操作されるリストを解析対象とする.

以下にリスト操作関数についての説明を行う.

- cons 関数

  cons は, 第一引数を先頭要素, 第二引数を第二要素以降のリストとするリストを返す関数
- car 関数

  car は, 引数であるリストの先頭要素を戻り値として返す関数
- cdr 関数

  cdr は, 引数であるリストの第二要素以降の値を戻り値として返す関数

1.3　本論文の構成

以降, 本論文では 2 章でシェープ解析, 3 章で実装を行う COINS の説明, 4 章で COINS 上でのシェープ解析を説明し, 5 章で本研究で作成したツールによる解析実験と解析結果を記述する. また, 6 章でシェープ解析の効率化や解析対象の拡張に関する関連研究を記述する.

第 2 章

シェープ解析

本章では, Reps によって提案されたシェープ解析手法について説明を行う.

シェープ解析とは, ソフトウェア解析手法の一つでプログラムを実行せずソースコード上で解析を行うものであり, プログラム内の動的データ構造 (リストや木構造など) に着目して解析を行う.

解析結果として各ポインタ変数の指す先が取り得る値を示すことができる.

主な応用として, ソーティングアルゴリズムなどのリストを用いたアルゴリズムの振舞い検証や, メモリーリークの検出, プログラムの実行中に不正なメモリ領域を指さないかの検証などがある.

2.1 Reps のシェープ解析アルゴリズム

Reps の解析法は,Lisp 言語で作成されたプログラムコードに対して解析を行い, まず制御フローグラフの作成を行う. そして作成した制御フローグラフに対して到達可能性を調べる事で解析結果を得る. 図で示すと図 2.1 のようになる.

詳しい解析の手法について 2.1.1 節から 2.1.2 節に分けて説明を行う.

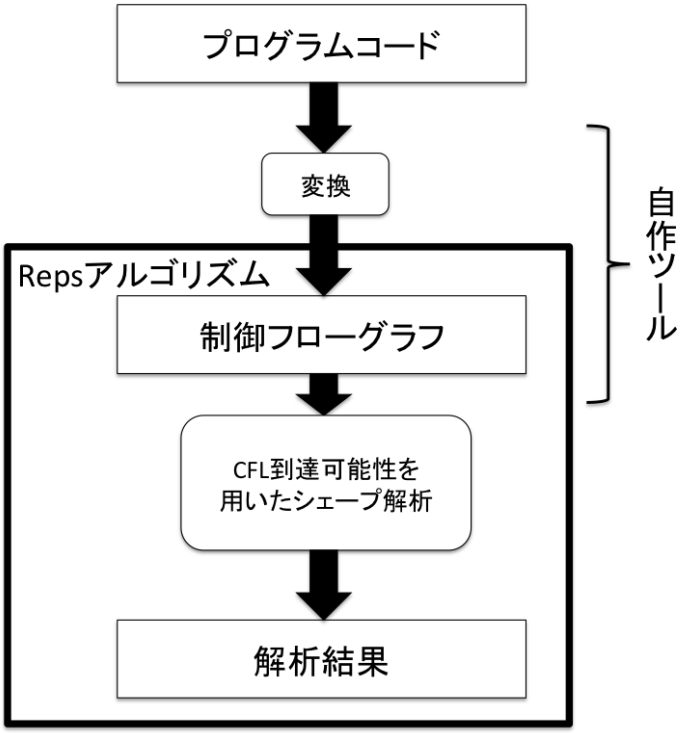

図 2.1　解析手順

```
x := nil;
read(z);
while z ≠ nil do
  x := cons(z, x);
  read(z);
od;
y := nil;
while x ≠ nil do
  temp := car(x);
  y := cons(temp, y);
  x := cdr(x);
od;
```

図 2.2　Reps の解析対象プログラム [1]

2.1.1　制御フローグラフ

制御フローグラフについて説明を行う.

制御フローグラフは, プログラム行と変数の組み合わせを頂点とし, 変数が取り得る値同士の関係をラベルで表したものである. 変数が取り得る値同士の関係は表 2.1 のようになる.

- ラベル id は前後の変数の値が等しい事を表す.
- ラベル hd は辺の先の変数の値であるリストの先頭要素が, 辺の元の変数の値に等しいことを表す.
- ラベル tl は辺の先の変数の値であるリストの第二要素が, 辺の元の変数の値に等しいことを表す.
- ラベル hd_inv は辺の先の変数の値が, 辺の元の変数の先頭要素と等しいことを表す.
- ラベル tl_inv は辺の先の変数の値であるリストの値が, 辺の元の変数の第二要素以降に等しいことを表す.

ソースコードの記述	制御フローグラフの辺	ラベル
x := a (a は数値や文字列)	atom → (q, x)	id
scanf(x)	atom → (q, x)	id
x := nil	empty → (q, x)	id
x := y	(p, x) → (q, y)	id
x := cons(y, z)	(p, y) → (q, x)	hd
	(p, z) → (q, x)	tl
x := car(y)	(p, y) → (q, x)	hd_inv
x := cdr(y)	(p, y) → (q, x)	tl_inv

表 2.1 ラベル付け

図 2.3 は, 図 2.2 の解析対象プログラムの二つ目の while 文を抜き出したものである.

ラベルを省略した辺は, ラベル id すなわち前後の変数の値が等しい事を示している. ラベル hd の辺は, 辺の先の変数の値であるリストの先頭要素が, 辺の元の変数の値に等しいことを表している. プログラムコードからこのような制御フローグラフを作成し, グラフ上の経路を調べる事でシェープ解析が行える.

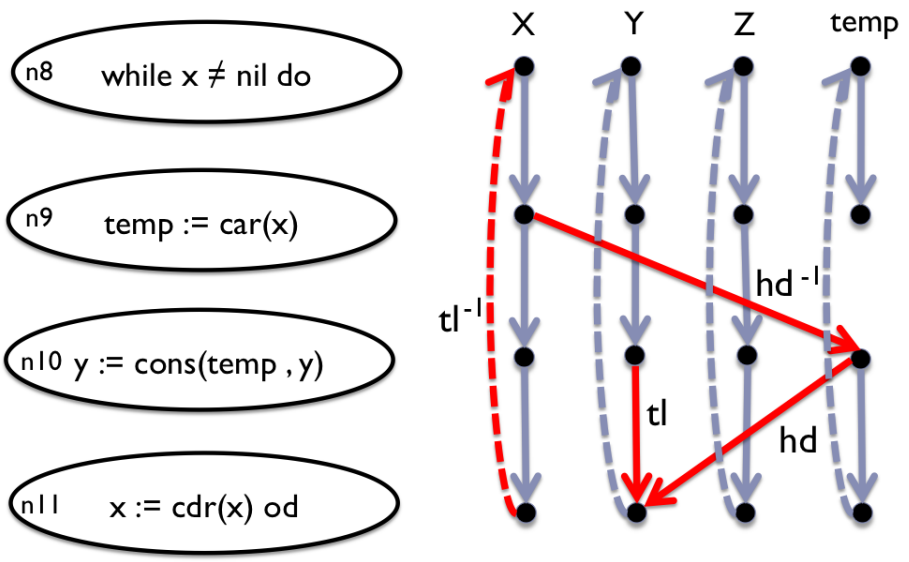

図 2.3　制御フローグラフ

2.1.2　Context-Free-Language 到達可能性

制御フローに対して Context-Free-Language 到達可能性の解析を行うことでシェープ解析を行う. 解析では, 利用者が指定した二頂点の間に, ラベル系列が表 2.2 の文脈自由文法で表されるような経路が存在するかどうかを調べる. 以下では頂点 (p, x) と (q, y) の間の経路について考える.

- L 1 のパターンにマッチする経路が存在するとき, 行 p における変数 x の値と行 q における変数 y の値は等しい.
- L 2 のパターンにマッチする経路が存在するとき, 行 p における変数 x の値は, 行 q における変数 y の値であるリストの先頭が取り得る値になる.
- L 3 のパターンにマッチする経路が存在するとき, 行 p における変数 x の値は, 行 q における変数 y の値であるリストの第二要素が取り得る値になる.
- L 4 のパターンにマッチする経路が存在するとき, 行 p における変数 x の値は, 行 q における変数 y の値であるリストのリストの任意の要素が取り得る値になる.

L1	<id_path> → hd <id_path> hd_inv <id_path>
	<id_path> → tl <id_path> tl_inv <id_path>
	<id_path> → id <id_path>
	<id_path> → ε
L2	<hd_path> → <id_path> hd <id_path>
L3	<tl_path> → <id_path> tl <id_path>
L4	<unmatched_path> → <id_path> hd <unmatched_path>
	<unmatched_path> → <id_path> tl <unmatched_path>
	<unmatched_path> → <id_path>

表 2.2 CFL 到達可能性

第 3 章

COINS

3.1 構成

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 3.1 に示す．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) によって構成される．フロントエンドでは原始プログラム (source program) を中間コード (intermediate code) と呼ばれる形式に変換する．バックエンドはフロントエンドで生成した中間コードを機械コードに変換する．フロントエンドを細かく分類すると字句解析器 (lexical analyzer), 構文解析器 (syntax analyzer) と意味解析器 (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．

3.2 中間言語

COINS では, 複数の入力言語, 複数の目的機種に対応する二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高水準中間表現 (high-level intermediate representation, HIR) と, 機械語に近いレベルの低水準中間表現 (low-level resentation, LIR) である．

COINS へのシェープ解析の実装では, 機械語に近いレベルの LIR を解析対象として採用した. 採用の理由としては, シェープ解析で用いる制御フローを解析しやすいからである。

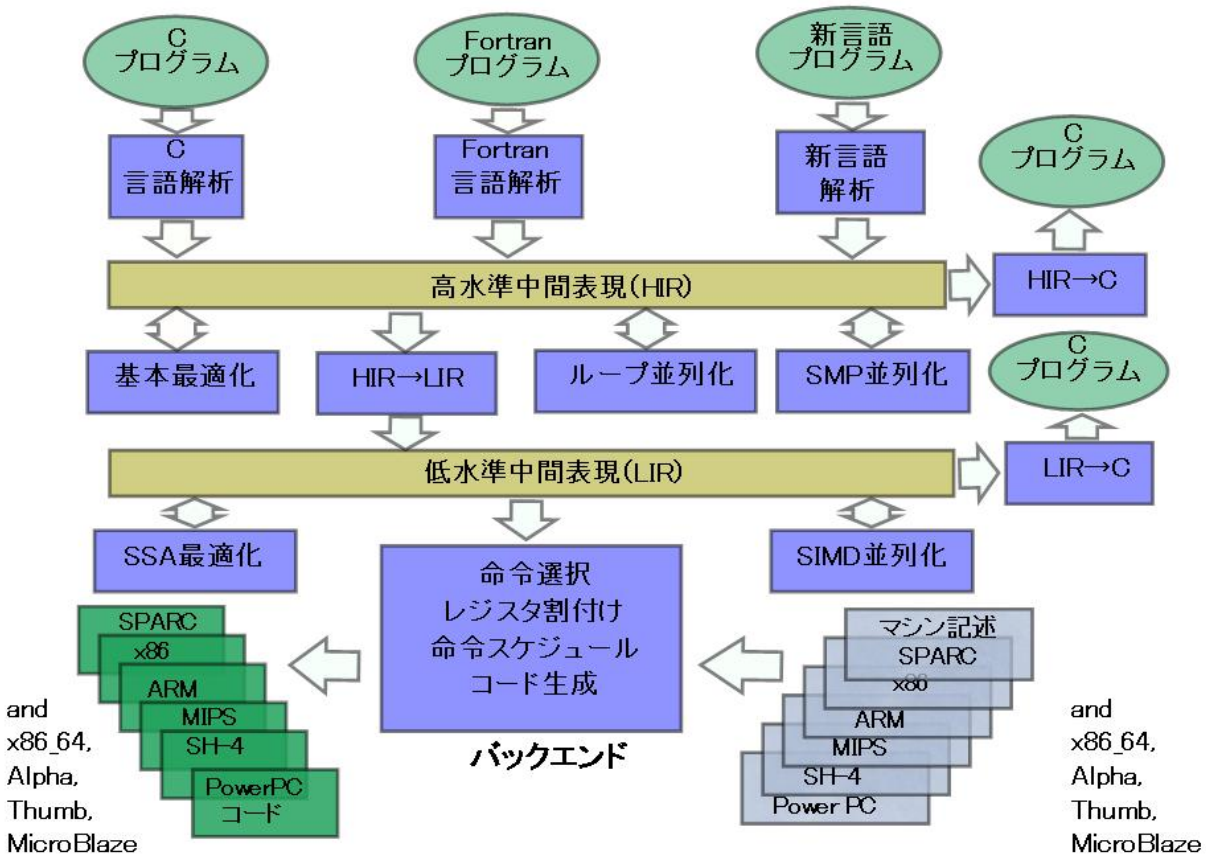

図 3.1　並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図 [4]

第 4 章

LIR 上でのシェープ解析

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.
- HIR から LIR に変換された直後の LIR コードを使用.

COINS に付属のツールを使って制御フローを解析し, 各ノードに関する情報を記述した LIR コードを入手することもできるが, 制御フローグラフに変換するのに必要ではない記述が多く出力され解析に手間がかかるので, ここではこのツールは使わずに, HIR から LIR に変換された直後の LIR コードに対して解析を行うよう実装した.

4.2 LIR の構造

C 言語プログラムは LIR コード (L-module) に変換される. L-module にはモジュール名, (グローバル) シンボルテーブル, L 関数の定義, L データからなる. L 関数は（ローカル）シンボルテーブル, L シーケンスからなる. L シーケンスは, L 式のリストであり PROLOGUE 式で始まり, EPILOGUE で終わる. 図 4.1 で LIR の内部構造を示す.

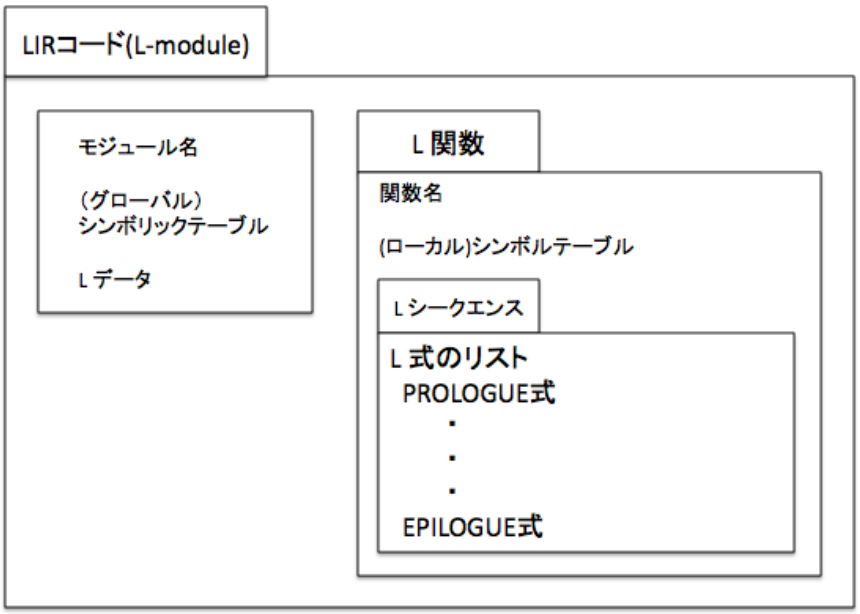

図 4.1　LIR の内部構造

4.2.1　L 式のセマンティックス

L シーケンスで用いられる L 式について, 本研究で必要な部分のみ説明を行う.

Const 式

Const 式は, 整数値と浮動小数値を表す. 整数値は (INTCONST I32 50) と表され, 32 ビットの 50 を表わしている. 浮動小数値は (FLOATCONST I32 50.0) と表され, 32 ビット浮動小数値の 50.0 を表わしている.

Addr 式

Addr 式は, グローバル変数とローカル変数のアドレスを表す. グローバル変数 x のアドレスは (STATIC I32 ”x”) と表される. ローカル変数 x のアドレスは (FREAM I32 ”x”) と表される.

Pure 式

Pure 式は, 加減乗除や比較などの算術演算式である. 数多くの種類があるため一部のみ説明する. 加算 1+1 は (ADD I32 (INTCONST I32 1) (INTCONST I32 1)) と

表される. 比較 (x ≠ 0) は (TSTNE I32 (FRAME I32 ”x”) (INTCONST I32 0)) と表される.

Mem 式

Mem 式は, データメモリのアドレスに格納している値を表す. (MEM I32 (FRAME I32 ”x”)) は, データメモリのアドレス x に格納されている 32 ビットの値を表している.

Set 式

Set 式では, データメモリのアドレスや, レジストリに値を代入する. (SET I32 (MEM I32 (FRAME I32 ”x”))(MEM I32 (INTOCONST I32 ”0”)) は代入式, x = 0 を表している.

Jump 式

Jump 式では, ラベルで示されたロケーションへジャンプする. (JUMP (LABEL I64 ”lab1”)) は, lab1 のロケーションにジャンプすることを表している.

Call 式

Call 式は, (CALL x_1 (x_2 ⋯ x_n) (y_1 ⋯ y_m)) と表され, x_1 の値で示されるプログラムメモリに L 関数があると仮定して, 引数 (x_2 ⋯ x_n) を与え, 結果の値を (y_1 ⋯ y_m) に代入することを表す.

Interface 式

Interface 式は, PROLOGUE と EPILOGUE の 2 つからなり, それぞれ L 関数の開始処理, 終了処理を表す. PROLOGUE 式では, 仮引数のリストを指定する. EPILOGUE 式では, 返り値のリストを指定する.

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 L シークエンスに対して,

1. L 関数から main メソッドの Prologue 式, Epilogue 式を調べ, (ローカル) シンボルテー

ブルから変数を取り出す.

2. L シークエンスの中の L 式を調べ, L 式に対応する C プログラムに基づいて制御フローグラフを作成する.
3. 制御フローグラフに対して, 利用者が指定した頂点への到達可能性を解析する.

以上のように, 解析するためには C 言語プログラムがどのように L 式に変換されたかを理解する必要がある.

4.4　C 言語と LIR の対応

この章では, C 言語プログラムの各文が LIR 式にどのように変換されているかを説明する. これらの対応関係に基づいて, LIR コードから制御フローグラフを抽出する.

4.4.1　代入文や型

代入文や言語の型は次のような L 式に変換される.

- 代入文

  x = y

  ⇒ (SET I32 (MEM I32 (FREAM I64 ”x”))(MEM I32 (FRAME I64 ”y”))

  x = *y

  ⇒ (SET I32 (MEM I32 (FREAM I64 ”x”))(MEM I32 (MEM I64 (FRAME I64 ”y”)

- データの型

  int x ⇒ (FRAME I32 ”x”)

  double x ⇒ (INCONST I32 1)

4.4.2　条件文やループ文

構造的な制御構造は LIR 式には組み込まれていない. そのため, 制御構造は比較文と指定のラベルへの JUMP 式で記述される.

- if 文

  if(x == 0)
  ⇒ (JUMPC (TSTNE I32 (MEM I32 (FRAME I64 ”x”)) (INTCONST I32 0))
  (LABEL I64 ”_lab3”) (LABEL I64 ”_lab5”))

- while 文

  while(x != 0)
  ⇒ (JUMPC (TSTNE I32 (MEM I32 (FRAME I64 ”x”)) (INTCONST I32 0))
  (LABEL I64 ”_lab5”) (LABEL I64 ”_lab3”))

比較演算＝と≠はどちらも TSTNE に変換され, 条件付きジャンプ命令 JUMPC のジャンプ先引数によって違いが表されている. if 文と while 文の違いは, 指定されたラベルの先に, while 文の先頭への Jump 文が記述されているかどうかで表される.

4.4.3　関数呼び出し

リスト操作関数 car, cons, cdr やデータ読み込み関数, 書き込み関数の呼び出しは次のような L 式に変換される.

- データ読み込み関数の呼び出し

  scanf(”%d”, x)
  ⇒ (CALL (STATIC I64 ”scanf”) ((STATIC I64 ”string”) (FRAME I64 ”z”))
  ((MEM I32 (FRAME I64 ”functionvalue”)))

- car 関数の呼び出し

  car(x, sizeof(x)/sizeof(int), y, sizeof(y)/sizeof(int))

⇒ (CALL (STATIC I64 ”car”)
((FRAME I64 ”x”) (DIVS I32 (INTCONST I32 200)(INTCONST I32 4))
(FRAME I64 ”y”) (DIVS I32 (INTCONST I32 200) (INTCONST I32 4))) ())

4.5　到達可能性の解析

抽出された制御フローグラフに対し, 以下のように到達可能性を解析する.

1. 利用者に頂点すなわちプログラム行と変数の組み合わせを指定してもらう.
2. 指定された頂点を起点として, 辺を逆向きにたどりながら探索を行う. このとき, 辺のラベルの組み合わせによって, 到達した頂点を id_path, hd_path, tl_path, unmatched_path という 4 つの集合に分ける.
3. 2 で到達した頂点を起点として 2 を繰り返す. empty か atom に到達すると探索を終了する.

以下に 2 の詳しい説明を行う.

- 起点および id_path に属する頂点からラベル id の辺のみたどって到達できる頂点を id_path に所属させる.
- id_path に属する頂点とラベル hd の辺で隣接している頂点を hd_path に所属させる. また, その頂点を起点としてラベル id の辺のみたどって到達できる頂点も hd_path に所属させる. hd_path に所属する頂点から id 以外の辺をたどって到達できる頂点を unmathced_path に所属させる.
- id_path に属する頂点とラベル tl の辺で隣接している頂点を tl_path に所属させる. また, その頂点を起点としてラベル id の辺のみたどって到達できる頂点も tl_path に所属させる. tl_path に所属する頂点から id 以外の辺をたどって到達できる頂点を unmathced_path に所属させる.
- id_path に属する頂点とラベル hd_inv の辺で隣接している頂点に対し, その頂点を起点としたときに id_path に属する頂点を再帰的に探し, その頂点からラベル hd の辺で隣

接している頂点を id_path に所属させる.

- id_path に属する頂点とラベル tl_inv の辺で隣接している頂点に対し, その頂点を起点としたときに id_path に属する頂点を再帰的に探し, その頂点からラベル tl の辺で隣接している頂点を id_path に所属させる.

第 5 章

解析実験

実装したシェープ解析ツールを評価するために, Reps の論文中にて用いられているリストの中身を反転するアルゴリズム (図 2.2) を, 図 5.1 のように C 言語に書き換えて解析を行った.

```
int main (void){
  int z, temp;
  List x = {0};
  scanf("%d", &z);
  while (z != 0){
    cons(x, size(x), z, x, size(x));
    scanf("%d", &z);
  }
  List y = {0};
  while (! isEmpty(x)){
    car(&temp, size(temp), x, size(x));
    cons(y, size(y), temp, y, size(y));
    cdr(x, size(x), x, size(x));
  }
  return 0;
}
```

図 5.1　解析対象プログラム

COINS を用いて C 言語の解析対象を LIR コードに変換する. LIR コードでは while 等のループ文や if 等の条件文は JUMP 文に変換されているため, 解析ではコード内で定義されているラベルを解析することにより制御フローグラフの作成を行う. 解析対象を制御フロー情報に変換すると図 5.2 になる.

得られた制御フローグラフに対して到達可能性を解析することにより, 図 5.3 のような解析結果を得られる. 図 5.3 は, プログラムの終端 (n12) における変数 y に関する解析結果を示している.

```
%- n1: start -> n2: x := null
edge(v(n1, x), v(n2, x), id ).
edge(v(n1, y), v(n2, y), id ).
edge(v(n1, z), v(n2, z), id ).
edge(v(n1, temp), v(n2, temp), id ).

%- n2: x := null -> n3: scanf (z)
edge(empty , v(n3, x), id ).
edge(v(n2, y), v(n3, y), id ).
edge(v(n2, z), v(n3, z), id ).
edge(v(n2, temp), v(n3, temp), id ).

%- n3: read (z) -> n4: while z != null
edge(v(n3, x), v(n4, x), id ).
edge(v(n3, y), v(n4, y), id ).
edge(atom , v(n4, z), id ).
edge(v(n3, temp), v(n4, temp), id ).

%- n4: while z != null -> n5: x := cons(z, x)
edge(v(n4, x), v(n5, x), id ).
edge(v(n4, y), v(n5, y), id ).
edge(v(n4, z), v(n5, z), id ).
edge(v(n4, temp), v(n5, temp), id ).

%- n5: x := cons(z,x) -> n6: scanf (z)
edge(v(n5, x),v(n6, x), tl ).
edge(v(n5, z),v(n6, x), hd ).
edge(v(n5, y), v(n6, y), id ).
edge(v(n5, z), v(n6, z), id ).
edge(v(n5, temp), v(n6, temp), id ).

%- n6: read (z) -> n4: while z != null
edge(v(n6, x), v(n4, x), id ).
edge(v(n6, y), v(n4, y), id ).
edge(atom , v(n4, z), id ).
edge(v(n6, temp), v(n4, temp), id ).

%- n4: while z != null -> n7: y := null
edge(v(n4, x), v(n7, x), id ).
edge(v(n4, y), v(n7, y), id ).
edge(v(n4, z), v(n7, z), id ).
edge(v(n4, temp), v(n7, temp), id ).

%- n7: y := null -> n8: while x != null
edge(v(n7, x), v(n8, x), id ).
edge(empty , v(n8, y), id ).
edge(v(n7, z), v(n8, z), id ).
edge(v(n7, temp), v(n8, temp), id ).

%- n8: while x != null -> n9: temp := car(x)
edge(v(n8, x), v(n9, x), id ).
edge(v(n8, y), v(n9, y), id ).
edge(v(n8, z), v(n9, z), id ).
edge(v(n8, temp), v(n9, temp), id ).

%- n9: temp := car(x) -> n10: y := cons(temp, y)
edge(v(n9, x), v(n10, x), id ).
edge(v(n9, y), v(n10, y), id ).
edge(v(n9, z), v(n10, z), id ).
edge(v(n9, x),v(n10, temp), hd_inv ).

%- n10: y := cons(temp,y) -> n11: x := cdr(x)
edge(v(n10, x), v(n11, x), id ).
edge(v(n10, y),v(n11, y), tl ).
edge(v(n10, temp),v(n11, y), hd ).
edge(v(n10, z), v(n11, z), id ).
edge(v(n10, temp), v(n11, temp), id ).

%- n11: x := cdr(x) -> n8: while x != null
edge(v(n11, x),v(n8, x), tl_inv ).
edge(v(n11, y), v(n8, y), id ).
edge(v(n11, z), v(n8, z), id ).
edge(v(n11, temp), v(n8, temp), id ).

%- n8: while x != null -> n12: Exit
edge(v(n8, x), v(n12, x), id ).
edge(v(n8, y), v(n12, y), id ).
edge(v(n8, z), v(n12, z), id ).
edge(v(n8, temp), v(n12, temp), id ).
```

図 5.2　制御フロー解析

```
?id_path(A,n12,y).
        A=v(n12,y).
        A=v(n8,y).
        A=v(n11,y).
        A=empty,.
?tl_path(A,(n12,y)).
        A=v(n10,y).
        A=v(n11,y).
        A=v(n9,y).
        A=v(n8,y).
        A=empty,.

?hd_path(A,(n12,y)).
        A=v(n10,temp).
        A=v(n5,z).
        A=v(n4,z).
        A=atom,.
?unmatched_path(A,(n12,y)).
        A=v(n12,y).     A=v(n5,z).
        A=v(n8,y).      A=v(n4,z).
        A=v(n11,y).     A=atom,.
        A=empty,.       A=v(n10,y).
        A=v(n10,temp).  A=v(n9,y).
```

図 5.3　解析結果

第 6 章

関連研究

シェープ解析に関する既存研究として以下のようなものが挙げられる.

分離論理を用いたシェープ解析

H.Yang らによる論文 [3] では, 本研究のように特定のリスト操作関数を仮定せず, メモリ確保/解放のような低水準の関数のみ仮定してシェープ解析を行う方法を提案している. この解析法では, 分離論理と呼ばれる形式論理に基づいて, ヒープ領域やヒープ操作関数をモデル化する. リスト操作関数を仮定せずデータ構造を解析するため, ダングリングポインタやメモリーリーク, 間接参照の解析などを行える.

同時シェープ解析による効率化

R.Manevich らによる論文 [2] では, 解析するデータ構造を分割ことにより並列でシェープ解析を実行する手法を提案している. 並列でシェープ解析を行う事により, シェープ解析の問題点として挙げられる実行時間の大幅な短縮することができる.

第 7 章

まとめ

本論文では, Reps の提案したシェープ解析を COINS 上で実装した. 結果としてリストを用いたアルゴリズムのデータ構造の解析を容易に行うことができるようになった. しかし, 今回決まったリスト操作関数を使うプログラムのみを解析対象としているので, この制限を緩和することが必要と考える. また今回はシェープ解析の基本アルゴリズムを実装しただけであり, 実用上はより高度なシェープ解析法を実装することが望まれる.

謝辞

本研究を進めるにあたり, ご指導, 御鞭撻を賜りました主指導教員である本学情報学群高田 喜朗 准教授に心より深く感謝致します. また, 副査を引き受けていただいた酒井敬一講師ならびに松崎公紀准教授に心より感謝致します. 様々な御助言, 御指導を賜った情報学群教員, 秘書の皆様方に心より感謝致します.

参考文献

[1] Reps, T., “Shape Analysis as a Generalized Path Problem,” 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pp.1–11, 1995

[2] Manevich, R., Lev-Ami, T., Ramalingam, G., “Heap Decomposition for Concurrent Shape Analysis,” 2008 SAS Static Analysis, 15th International Symposium, pp.363–377, 2008.

[3] Yang, H., Oukseh, L., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O’Hearn, P., “Scalable Shape Analysis For Systems Code,” Proceedings of the 20th International Conference on Computer Aided Verification, pp.385–398, 2008.

[4] “COINS コンパイラ・インフラストラクチャ”, http://coins-compiler.sourceforge.jp/.