2004年度 卒業論文

LMNtalコンパイラにおける
並び替えとグループ化を用いた命令列の最適化

提出日 : 2005年2月2日

指導 : 上田 和紀 教授

早稲田大学理工学部情報学科

学籍番号：1G01P047-1

櫻井 健

概 要

LMNtal は並列計算や多重集合書き換えなどの様々な計算モデルを統合して扱うことを目的として開発されている言語モデルである．その目標は, 計算モデルとして簡明であり, 高い汎用性を持つ実用的なプログラミング言語のベースとなることである．

LMNtal の処理は階層型グラフ構造を書き換え規則 (ルール) によって書き換えることで進行する．そのためデータ自体がグラフ構造なので, 手続き型言語では複雑な階層型や循環型のデータ構造も容易に扱えるようになっている．機能面でもモジュールの組み込みや java のインライン展開により拡張が可能であり, また LMNtal で書いたプログラムは直感的に図示しやすいという利点もある．こうした観点から, LMNtal は汎用性の高い言語となりつつある．

しかし, 実行効率の面ではまだ優れているとまでは言えない．本論文では, LMNtal のコンパイラが生成した命令列 (中間コード) を解析し, その原因の 1 つであるルール適用失敗時のバックトラック処理に着目した．その結果, ある命令で失敗した際, 関係の無い部分もしらみつぶしにバックトラックしたり, ルール適用失敗が確定したにも関わらずその時点で処理が終了しない, といった無駄が多く行われていることが分かった．そこで命令列に対し, 並び替えとグループ分けという 2 種類の最適化を施し, 実行効率の改善を図った．並び替えは命令列の内, 早い段階に実行した方がよい命令を可能な限り前に押し上げる．グループ化は命令列を独立した処理の単位であるグループに区切る．

これらの最適化を行い, 前述ような無駄な処理を削減した結果, プログラムによっては最適化をしない場合に比べて 1000 倍以上高速化された．

目 次

図 目 次

表 目 次

第1章　序論

1.1　並列言語モデルLMNtal

近年, コンピュータを取り巻く環境は, 並列計算や広域分散, 大規模なものから組み込みのための小規模なものまで, 様々な環境が想定される. 一般にシステム開発者は, 環境が変わるとそれに適した言語なりモデルなりを選び, それを1から学ばなければならず, これは大変負担になる. LMNtal はこういった様々な環境を統合して扱うことの出来る汎用的なプログラミング言語のベースとなることを目指して開発されている.

環境を統合するには様々な計算をモデル化できなくてはならないが, LMNtal ではプロセスとデータを同等に扱うことができ, 他言語では複雑な階層型などの構造を容易に扱うことができる. また, LMNtal で書かれたコードは図示するのが容易であり, 直感的に分かり易いプログラムを書くことができる. 逆に図でモデルを表現できれば, そこからプログラムを書くこともできるということになる. こういった観点から LMNtal はモデル化に適した言語であると言える. LMNtal を言語モデルと称するのは, このように計算をモデル化するのと同時にプログラムとして実装する機能を持つことを意味する.

1.2　研究の目的と意義

LMNtal はモデル化に適しているだけでなく, プログラミング言語としての機能面でもモジュールの組み込みや java のインライン展開などにより大概のことは出来るようになってきている. しかし, 実行効率の面ではまだ不安を抱えていることは否めない. LMNtal の目標は高い汎用性を持つことなので, あらゆる計算を現実的な時間内で終了させるだけの実行効率を持つことが望ましい. よって少しでも効率を上げることを考えなくてはならない.

LMNtal は, 階層型グラフ構造の特定箇所でマッチングを行い, マッチしたらグラフの書き換えを行うという流れで処理が進行する. 実行効率が悪い原因の1つは, このマッチングに失敗した時の処理にあるということが分かった. 本研究ではこの問題点に焦点を当て, 処理系の解釈する命令列の並び替えおよび新しい命令の追加を行った. それらを用いて最適化を行うことで, マッチング失敗時のバックトラック処理を改善することに成功した.

1.3 本論文の構成

以降本論文の構成は以下のようになっている.

第 2 章 LMNtal の概要について述べる.

第 3 章 LMNtal のコードが処理系内でどう解釈されていくのかを解説する. LMNtal がインタプリタで解釈する各々の命令についてもこの章で解説する. また, その結果から現在の処理系の問題点について考察する.

第 4 章 第 3 章で分かった問題点を解決する機構の設計と実装について述べる.

第 5 章 実装した最適化の性能評価と考察を行う.

第 6 章 本論文のまとめと今後の目標を述べる.

付録 命令列の並び替え (GuardOptimizer クラス) とグループ化 (Grouping クラス) の java コードを載せた.

第2章　LMNtal言語解説

LMNtal は書き換え規則 (ルール) によって, リンクで繋がれたノードで構成された階層的グラフ構造の内, 特定のパターンにマッチした部分の構造を書き換えていくことで処理が進行する. この章では LMNtal の基本的な構成要素と簡単なプログラムを例にルール適用の手順を解説する.

2.1　グラフ構造の構成要素

2.1.1　アトム

アトムはノードを構成する最小単位である. 次のように記述する.

アトムのプログラム上の表現

$a(X_1, X_2, ..., X_n)$
a:アトムの名前　　X:リンク

アトムは 0 個以上のリンクを持ち, 他のアトムと繋がっている. リンクと区別するため, アトムの名前は大文字以外 (小文字や数字) から始まる文字列で表現する. また, アトムの名前と引数 (リンク) の数をまとめたものをファンクタと呼ぶ. アトム名が a, 引数の数が n の場合, そのファンクタは `a_n` と記述する.

2.1.2　リンク

アトムとアトムを 1 対 1 で繋ぐ役割を果たす. ルール上では大文字から始まる文字列で表現する. 例としてアトム a, b をリンクで繋いだ構造をプログラムと図で表すと図 2.1 のようになる.

リンクで繋がれたアトムのプログラム

```
a(X), b(X)
```

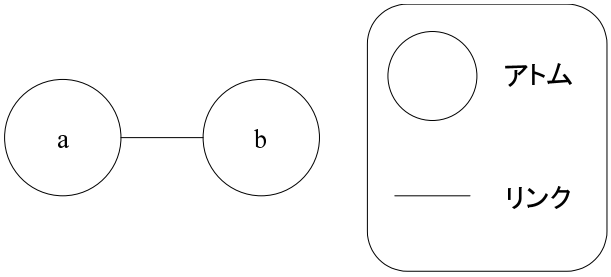

図 2.1: リンクで繋がれたアトム

また, リンクの表現には次のような略記法を使用できる.

- $p(s_1, ..., s_m)$, $q(t_1, ..., t_i, ..., t_n)$ は s_m と t_i が同じリンクなら, $q(t_1, ..., t_{i-1}, p(s_1, ..., s_{m-1}), t_{i+1}, ..., t_n)$ と略記してよい.

これを踏まえた上で先ほどのリンクを表現すると, `a(b)` となり簡潔に表現できる.

2.1.3 膜

膜はノードをまとめる 1 つの単位であり, 膜自身も 1 つのノードとして扱うことが出来る. よって膜内部のアトムから, 他の膜内のアトムにリンクを張ることで階層的グラフ構造を構成する.
ここまでに出てきた基本要素を用いてグラフ構造を表現すると図 2.2 のようになる.

プログラム

```
{{a(A1), a(A2)}, {b(A1)}, c(A2), d, e, f}
```

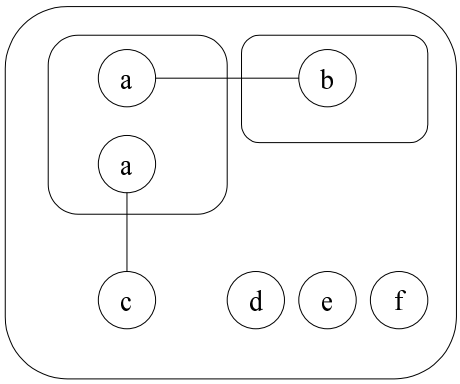

図 2.2: 膜を用いたグラフ構造

2.2 グラフ書き換え規則 (ルール)

ルールはグラフ構造のある部分において, 特定のパターンとマッチした場合にそのグラフ構造を書き換える. ルールもまたグラフ構造を構成するノードの 1 つであり, 同じ膜内でマッチするグラフ構造を検索する.

ここで簡単なプログラムを用いてルール適用の例を説明する.

プログラム

```
a, a, a :- a(a(a)).
```

ルールは, 左辺にマッチした部分を右辺に書き換える機能を持つ. この例では 0 引数のアトム a が 3 個あった場合, それらをリンクで繋ぐ.

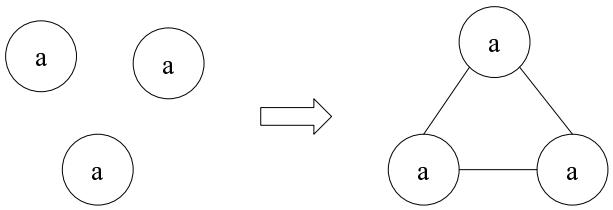

図 2.3: ルール適用の例

なおルールは適用できる限り続けられる.

```
{ a, a, a, a, a, a, a, ( a, a, a :- a(a(a)) ) }
```

このような膜は最終的には

```
{a(a(a)), a(a(a)), a, ( a, a, a :- a(a(a)) )}
```

となる.

2.2.1 ルールの構造

ルールの構造をもう少し詳しく説明する.

ルールの構造

ヘッド　:-　(ガード　|)　ボディ

ヘッド 現在の膜内 (ルールのある膜内) のグラフ構造の中で, マッチングさせたいパターンを記述する.

ガード ヘッドでマッチしたパターンの内から, さらに特定のパターンを取得したい時に, その “特定” を絞り込むような制約を記述する.

ボディ ルール適用時に生成するプロセスを記述. プロセスについては次節を参照.

例として次のプログラムを実行した時を考える.

プログラム

```
a(X) :- int(X) | ok
```

処理系内で各部位が行う処理は次の通り.

ヘッド :ファンクタ `a_1` を持つアトムを探す.

ガード :ヘッドで取得したアトムのリンク先が 1 引数で, 名前が整数のアトムであることを確認する. もしそうでないならヘッドで取得した アトム はこのルールにはマッチしない.

ボディ :ファンクタ `ok_0` を持つアトムを生成する.

以下にガード部に書ける命令を列挙する. またこれ以降, 整数を名前に持つアトムを整数アトムと呼ぶことにする. なおこの他にもいくつかも命令はあるが, 本研究にあまり関係のない命令は割愛する.

ガードに書ける命令の一部

int(X) : X のリンク先が整数アトムであることを確認する.

float(X) : X のリンク先が浮動小数点数アトムであることを確認する.

unary(X) : X のリンク先が 1 引数のアトムであることを確認する. (int(X), float(X) はこの部分集合)

$X > Y(X \geq Y)$: X,Y のリンク先が整数アトムでかつ $X > Y(X \geq Y)$ であることを確認する.

$X < Y(X \leq Y)$: X,Y のリンク先が整数アトムでかつ $X < Y(X \leq Y)$ であることを確認する.

$X > Y$ の X,Y の部分には $A + 10$ 等の演算を書くことも出来る.
(`a(X),b(Y) :- X + Y < 20 | ok.`)
浮動小数点数を扱う時は演算子の後ろに. を付ける.
(`a(X),b(Y) :- X +. Y <. 1.5 | ok.`)

ガード部の命令における失敗時の処理は, 本研究の論点でもある. ガード部分の検査はヘッド部分のマッチングを全て済ませてから行うため, 場合によっては無駄なマッチング検査をしていることになる. その詳細は 3 章で述べる.

2.2.2 プロセス

ここまでにアトム, リンク, 膜, ルールを説明したが, これらの集まりをプロセスと呼ぶ。膜はプロセスの多重集合なので, 膜によってプロセスの階層化や局所化が可能となる.

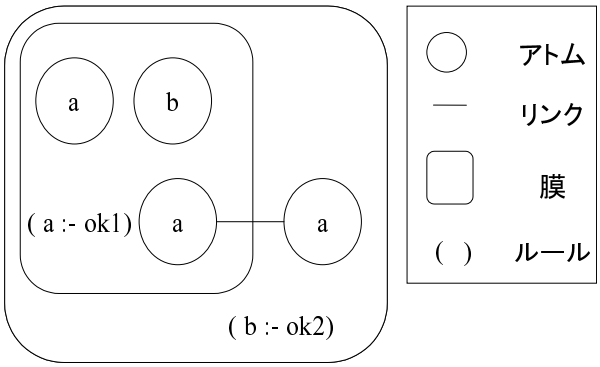

図 2.4: 膜によるプロセスの局所化の例

プロセス 1(内側の膜内) : `a, b, a(X), (a :- ok1)`

プロセス 2(外側の膜内) : `a(X), {a, b, a(X), (a :- ok1)}, (b :- ok2).`

この場合 (a :- ok1) は適用されるが (b :- ok2) は適用されない. これはプロセス 1 はプロセス 2 から局所化されているため, プロセス 2 のルール (b :- ok2) はプロセス 1 の b にマッチしないためである.

2.3 LMNtal の構文規則

LMNtal の構文規則は次のようになっている.

LMNtal 構文規則

```
P :: = 0 (空)
   | p(X1,...,Xm)    (m >= 0) (アトム)
   | P, P   (分子)
   | {P}   (セル)
   | T :- T   (ルール)

T :: = 0 (空)
   | p(X1,...,Xm) (アトム)
   | T,T    (分子)
   | {T}    (セル)
   | T :- T   (ルール)
   | @p    (ルール文脈)
   | $p[X1,...Xm|A] (プロセス文脈)
   | p(*X1,...,*Xm) (m>0)(アトム集団)

A :: = [] (空)
   | *X (リンク束)
```

大きく分けて構文のカテゴリはプロセス P とプロセステンプレート T の 2 つである. プロセスはプログラムの実行で作られるプロセスを実現し, プロセステンプレートはプロセスの書き換え (ルール) の表現に使用される.
なおセルというのは, 膜内のプロセス全体であり, 膜はセルを包む {} のことである.

2.3.1 自由リンクと局所リンク

プロセスに 2 回出現するリンクを局所リンク, 1 回だけ出現するリンクを自由リンクと呼ぶ. つまり局所リンクはプロセス内でリンクが繋がっているて, 自由リンクはプロセスの外にリンクが出ているということになる.

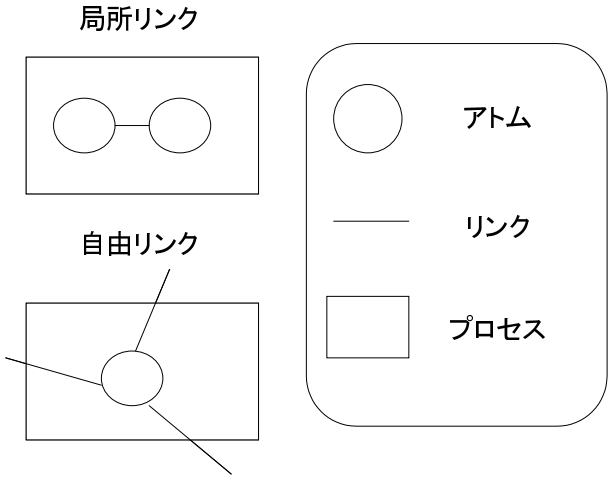

図 2.5: 自由リンクと局所リンク

2.3.2 ルール文脈とプロセス文脈

ルール文脈はルール左辺のセル中の全てのルールにマッチする. プロセス文脈はセルの中にあるルール以外の全てのプロセスとマッチする. なお p というのは名前であり, 先頭が大文字でなければ任意である. またルール左辺のルール文脈とプロセス文脈は, ともにセルの中に出現しなければならない.
例えば, `{a, a, (a :- b)}` を `{$p, @p}` で表現した時 `$p, @p` がどこにマッチしているのかを確かめると,

```
$p = a, a
@p = a :- b
```

ちなみに $p は `$p[|*X]` の略である. また, プロセス文脈 `$p[X1,...,Xm|A]` の意味を説明すると, X1,...,Xm は $p が持つ自由リンクの指す. A はそれ以外のリンクの束を指し, プロセス文脈の剰余引数と呼ぶ.

`$p` を含む膜を左辺に持つルールは空の膜にもマッチするが, ルールを含む膜の場合 `@p` を書かないとマッチしない.

ルール文脈, プロセス文脈のマッチ

- `{}. {$p[]} :- ok.` => マッチする.
- `{(a:-b)}. {$p[]} :- ok.` => マッチしない.
- `{(a:-b)}. {$p[], @p} :- ok.` => マッチする.

2.3.3 ルール構文の各条件

ルールを書くに当たりいくつか制約条件があるのでここに列挙する.

構文条件

1. ルールはルールの左辺には出現しない.
2. アトム集団はルールの右辺には出現しない.
3. ルール文脈とプロセス文脈は, ルール左辺においてはセルの中に出現しなければならない.

回数条件

1. ルールに出現するリンクとリンク束は, そのルールにちょうど 2 回出現しなければならない.
2. ルール左辺の 1 つのプロセス文脈の引数に出現するリンクは互いに異なっていなければならない.
3. ルール左辺のリンク束は互いに異なっていなければならない.
4. ルールに出現するルール文脈名とプロセス文脈名は, そのルールの左辺にちょうど 1 回出現しなければならず, そのルール内の他のルールには出現しない.
5. ルール左辺の各セルの最外部には, ルール文脈とプロセス文脈はそれぞれ 1 回を越えて出現しない.

整合性条件

1. ルール内では, 同じ名前のプロセス文脈の剰余引数に空 ([ ]) とリンク束の両方が存在してはならない.
2. ルール内では, 同じ名前のプロセス文脈の引数の個数 m は全て一致していなければならない.
3. ルールにおいて, 同じリンク束が 2 つのプロセス文脈の剰余引数として出現するなら, それらのプロセス文脈は同じ名前を持たなければならない.
4. ルール内のアトム集団 $p(*X_1, ..., *X_m)(m > 0)$ に対してプロセス文脈名 $q が 1 つ存在して, どの $*X_1$ も, 名前 $q を持つそのルール内のプロセスのどれかに, 剰余引数として出現しなければならない.

なお, 本研究で行う最適化は膜のないプログラムのみを対象としている. 検証に用いるプログラムでも膜を使用しないので, プロセス文脈, ルール文脈, アトム集団, リンク束は使わない. よって本論文を読み進めるにあたっては, これらに関する条件項目を意識する必要は無い.

2.4 略記法

LMNtal のプログラムを書く上で使用できる略記法を以下にまとめる.

略記法

1. ()(0 価のアトムの括弧) 及び `|[]` は省略してよい.
2. `[|*X]` は `(*X)` と略記してよい.
3. ルールに同一の `$p[|*X]` が 2 回出現するなら, 両方を同時に $p と書いてよい.
4. $p(s_1, ..., s_m)$, $q(t_1, ...t_i, ..., t_n)$ は s_m と t_i が同じリンクなら, $q(t_1, ..., t_{i-1}, p(s_1, ..., s_{m-1}), t_{i+1}, ..., t_n)$ と略記してよい.

2.5 機能拡張

LMNtal には現在足りない機能を補うために java のインライン展開とモジュールの組み込みが使用されている.

2.5.1 javaのインライン展開

LMNtalの処理系はjavaで実装されている. インライン展開ではLMNtalのソースコードに直接javaのコードを埋め込むことができる.
インラインコードは `[:/*inline*/ javaのコード :]` の形で記述する. javaコード内でアトムを生成することもできる. 以下にインラインを用いた簡単な例を挙げる.

インライン展開の使用例

```
a.
a :- [:/*inline*/
       System.out.println("Hello World!");
     :],b.
```

インラインもアトムとして生成される. 実行するとアトムaがbに書き変わるのと同時に, コンソールにHello　World! と出力される. また, インラインの中身が実行されるタイミングは他のアトムの生成が終わった後 (上の例ではbの生成) となる.

2.5.2 モジュールの組み込み

モジュールは他言語でいうライブラリの機能を果たす. LMNtalのソースに組み込むことで機能拡張する. 使用方法は直感的には次のようになる.

モジュール名.関数名

モジュールの作成から使用までの例として, 作成したintegerモジュールを用いたプログラムを紹介する. なお, このモジュールは後の検証用プログラムにも使用している.

integer モジュール

```
{
module(integer).
/* 中略 */
H = integer.rnd(N) :- int(N) | H=[:/*inline*/
  int n =
  ((IntegerFunctor)me.nthAtom(0).getFunctor()).intValue();
  Random rand = new Random();
  int rn = (int)(n*Math.random());
  Atom result = mem.newAtom(new IntegerFunctor(rn));
  mem.relink(result, 0, me, 1);
  me.nthAtom(0).remove();
  me.remove();
  :](N).
}.
```

java コードの内インラインならではの部分を順を追って説明すると

- `H = ...`
  これは integer.rnd(N) の戻り値を H のリンク先と接続することを意味する.
- `int n = ...` インラインアトムに渡した 1 個目 (添え字は 0) のアトム (この場合 N) を, 整数を名前に持つアトムとして取得, その名前を整数値として n に渡す.
- `Atom result = ...`
  名前 result で引数が r n のアトム,result(r n) を生成する.
- `mem.relink(...)`
  result の第 1 引数とインラインアトムの第 2 引数 (この場合 H) を繋げる.
- `me.nthAtom(0).remove(); me.remove();`
  インラインアトムの削除.

この場合, インラインでは出力を繋ぐためのリンクを必要としている. つまり使い方は, アトム (`integer.rnd(N)`) であり, 0 から N-1 の整数の値を引数に持ったアトムを生成することになる.

モジュール使用例

```
a(integer.rnd(10)).
b(integer.rnd(100)).
c(integer.rnd(1000)).
```

なお, 実行結果は次のようになった.

実行 1 回目 : a(9), b(72), c(393)

実行 2 回目 : a(1), b(24), c(771)

実行 3 回目 : a(5), b(95), c(815)

第3章　処理系の動作

LMNtal のコンパイラはまずユーザーの書いたコードを, インタプリタで解釈できる命令列にコンパイルする. この章ではコンパイラの生成が各々の命令について解説し, 実際にインタプリタでどう解釈されていくかを追っていく. また現状のインタプリタの動作における問題点を考察する.

3.1　コンパイル

処理系はユーザーの書いたコードを下の構造を持つルールの列として受け取る.

ルールの構造

ヘッド　:-　ガード　|　ボディ

データの生成をするプログラムも, 何もない領域にアトムを作るルールということになる.

```
a. a. a. ===> ( :- (a. a. a.) )
```

この例はファンクタ a_0 を持つアトムを 3 個作るものだが, 'a を 3 個作るルールを作る' というルールとして解釈される. なお, このようにデータやルール等の初期生成を行うルールは最初に 1 度しか実行されない.

LMNtal のコンパイラは, ソースコードを全てルールの列として解釈し, そのルールを各部位ごとにインタプリタで解釈できる内部命令の列へとコンパイルする. できた命令列を各部位ごとにヘッド命令列, ガード命令列, ボディ命令列と呼ぶ.

3.2　処理系内の命令列

実際にプログラムを実行した時に内部に生成される命令列を次に示す.

サンプルプログラム

```
a(5, X) :- X + 1 < 10 | ok.
```

何をするプログラムか述べると,

1. ファンクタ `a_2` を持つアトムを探す.
2. a の第 1 引数が名前が 5 である 1 引数の整数アトムと繋がっていることを確認.
3. a の第 2 引数と繋がっているアトムの名前が整数であり, かつその値 X が $X + 1 < 10$ を満たすことを確認.
4. 1〜3 に全て成功したらファンクタ `ok_0` を持つアトムを生成する.

この例で生成される命令列は, 本論文で扱う範囲の命令をほぼ網羅している. また, 前述の通り実際には ルール `( :- ( a(5, X) :- X + 1 < 10 | ok ) )` の命令列も生成されるのだが, 本論文ではこちらを考察することには意味はないので省略する.

実際に命令列を示す.

生成される命令列

```
--atommatch:
         spec            [2, 6]
--memmatch:
         spec            [1, 3]
         findatom        [1, 0, a_2]
         deref           [2, 1, 0, 0]
         func            [2, 5_1]
         jump            [L162, [0], [1, 2], []]
--guard:L162:
         spec            [3, 7]
         allocatom       [3, 1_1]
         derefatom       [4, 1, 1]
         isint           [4]
         iadd            [5, 4, 3]
         allocatom       [6, 10_1]
         ilt             [5, 6]
         jump            [L163, [0], [1, 2, 3, 4, 5, 6], []]
--body:L163:
         spec            [7, 8]
         commit          [( a(~1,~5) 5(~1) $X[~5] :- 1($~3) '+'
                         ($X,$~3,$~2) 10($~4) '<'($~2,$~4) | ok )]
         dequeueatom     [1]
         removeatom      [1, 0, a_2]
         removeatom      [2, 0, 5_1]
         removeatom      [4, 0]
         newatom         [7, 0, ok_0]
         enqueueatom     [7]
         freeatom        [1]
         freeatom        [2]
         freeatom        [4]
         freeatom        [6]
         freeatom        [5]
         freeatom        [3]
         proceed         []
```

以降, 書く部位ごとに解説していく.

3.2.1 ヘッド命令列

ヘッド命令列の役目はルールのある膜内で, ルール左辺にマッチする箇所を検索することである.

この命令列中, ヘッドに当たるのは次の部分である.

ヘッド命令列

```
--atommatch:
        spec            [2, 6]
--memmatch:
        spec            [1, 3]
        findatom        [1, 0, a_2]
        deref           [2, 1, 0, 0]
        func            [2, 5_1]
        jump            [L162, [0], [1, 2], []]
```

マッチングはアトム主導テストと膜主導テストの 2 種類があり, 命令列もそれぞれ atommatch, memmatch に分かれている.

アトム主導テスト ある膜内で, ルールに関連付けられる (左辺に出現する) アトムをアクティブアトムといい, あるアクティブアトムが実際に適用されるルールが膜内にあるか, という方向性でマッチングを行うこと.

膜主導テスト ある膜内に, 適用できるルールがあるかどうかをチェックしていくこと. 存在する膜全てについて適用できるルールを片っ端から検索するので, 全てのルールについてマッチングを試みることができる. その反面効率は良くない.

なお “ ある膜” という言い回しをしているが, ランダムに膜を選んでいるわけではない. 膜は実行膜スタックという膜に積まれており, 親膜の方が子膜より下となっている. このスタックから順次膜を取得していくので, ここでいう “ある膜” とは実行膜スタックの先頭に積まれた膜のことである.
また, 膜はそれぞれその膜内のアトムを積む実行アトムスタックを持っており, マッチングで検査されるアトムはこの先頭から取り出していく.

一般にアトム主導テストの方が効率がいいが, デフォルトではアトム主導テストは必ず失敗する. よって atommatch の命令列がほぼ空の状態になっている. アトム主導テストで失敗しても, 膜主導テストで全てのマッチングが行われるようになっている. なお, 本論文の最適化手法は, 膜主導テストの命令列の書き換えのみを扱い, アトム主導テストに関しての最適化は行わないこととする.

次に膜主導テストの命令列を順次追っていく.
処理系内でアトムや膜は変数番号で管理されている. 引数の-dstatom や srcatom などがそれに当たる. また先頭に-が付いているのは出力する引数であり, -dstatom は dstatom という変数番号を持つアトムを新たに定義するという意味である.

spec `[formal, local]`
: インタプリタに対し, 処理に必要な仮引数の個数 formal と局所変数の個数 local を宣言する. 各命令列の先頭には必ず spec 命令がある.

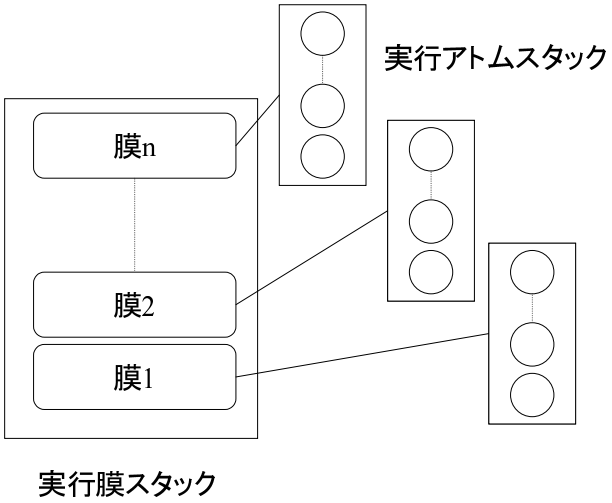

図 3.1: 実行膜スタックと実行アトムスタック

findatom `[-dstatom, srcmem, funcref]`
膜srcmemにあってファンクタfuncrefを持つアトムへの参照を次々にdstatomに代入する．srcmemが0のものはルールのある膜内ということであり, 膜を用いない場合ではsrcmemの値は全て0である．なお“次々に”というのをどうやって実現しているかというと, findatom命令の所でインタプリタの再帰呼び出しを行うことで実現している．従ってガード命令列も含め, これより下の命令で失敗した場合このfindatom命令に戻ってくる．そして先程とは違う参照を-dstatomに代入する．これらの処理は3-3節で詳しく述べる．

deref `[-dstatom, srcatom, srcpos, dstpos]`
アトムsrcatomの第srcpos引数のリンク先が, 第dstpos引数に接続していることを確認したら, リンク先のアトムへの参照をdstatomに代入する．

func `[srcatom, funcref]`
アトムsrcatomがファンクタfuncrefを持つことを確認する．

jump `[instructionlist, [memargs...], [atomargs...], [varargs...]]`
指定の命令列でラベル付き命令列instructionlistを呼び出す．指定した命令列で失敗するとこの命令は失敗し, 成功すると終了する．
この場合, ヘッド命令列で取得した膜リストmemargsとアトムリストatomargsの情報を得た状態で, L162とラベルされた命令列, すなわちガード命令列の実行に移ることになる．

例に挙げた命令列の処理の流れをまとめると,

1. 膜 0 の中からファンクタ a_2 を持つアトム a を見つけ, 参照を変数番号 1 に代入. (findatom [1, 0, a_2])

2. a の第 1 引数 (添え字は 0) のリンク先が, あるアトムの第 1 引数であることを確認し, その参照を変数番号 2 に代入. (deref [2, 1, 0, 0])

3. 変数番号 2 のアトムのファンクタが 5_1 であることを確認. (func [2, 5_1])
ここまでで a(5, X) を取得したことになる. X はヘッド命令列中ではリンク先は明示されていない. このあとのガード命令列で調べることになる.

4. ガード命令列へ移行. (jump)

3.2.2 ガード命令列

ガード命令列の役目は, ヘッド命令列を実行してマッチングに成功したパターンに対し, さらにその中から特定のパターンを取得するために様々は制約を付けることである.
先程の例だと, ヘッド命令列ではアトム a(5, X) を取得しているが, これは “ ファンクタが a_2 で, その第 1 が整数アトム 5 へ繋がっているアトム” を取得していることを意味する. この段階では第 2 引数は言わば “何でもいい” という状態だが, ガード命令列でそれに対し制約を付けて, より限定されたパターンの取得をできる.
ガード命令列の最後の jump 命令まで行ければ, ルール適用成功ということになる. そしてボディ命令列で, ルール右辺に書かれた処理の実行に移る.
ガード命令列は次の部分になる.

ガード命令列

```
--guard:L162:
        spec            [3, 7]
        allocatom       [3, 1_1]
        derefatom       [4, 1, 1]
        isint           [4]
        iadd            [5, 4, 3]
        allocatom       [6, 10_1]
        ilt             [5, 6]
        jump            [L163, [0], [1, 2, 3, 4, 5, 6], []]
```

各命令について解説する.

allocatom [-dstatom, funref]
ファンクタ funcref を持つ所属膜を持たない新しいアトムを作成し, 参照を

dstatom に代入する. ガード検査で使われる定数アトムを生成するために使われる.

derefatom `[-dstatom, srcatom, srcpos]`
アトム srcatom の第 srcpos 引数のリンク先のアトムへの参照を dstatom に代入する。

isint `[atom]`
アトム atom が整数アトムであることを確認する。整数でなければ失敗となる.
似たような命令に isfloat, isstring, isunary がある. それぞれアトムが浮動小数, 文字列, 1 引数のアトムであることを確認する.

iadd `[-dstintatom, intatom1, intatom2]`
整数アトムの加算結果 (intatom1+intatom2) を表す所属膜を持たない整数アトムを生成し, dstintatom に代入する. 扱うアトムが整数なので,
`a(X) :- X > 10 | ok.` と書いた場合, `a(X) :- int(X), X>10 | ok.` と書いたことと同じ扱いになる.
また似たような命令に, 整数型の四則演算の結果および剰余を表すアトムを生成する isub, imul, idiv, imod, 浮動小数点数の四則演算の結果を表す fadd, fsub, fmul, fdiv がある. これらのうち 0 で割る可能性のある idiv, imod, fdiv のみ失敗することがある. それ以外は必ず成功する.

ilt `[intatom1, intatom2]`
整数アトムの値の大小比較が成り立つことを確認する. 成り立たない場合は失敗ということになる. ilt(¡) 以外の場合も処理は全く同じ. 次のような命令もある.
`ile(=<), igt(>), ige(>=)`
浮動小数点数アトムの値の比較の場合,
`flt(<.), fle(=<.), fgt(>.), fge(>=.)`

例のプログラムのガード部分の処理をまとめると,

1. 整数アトム 1 をつくり, その参照を変数番号 3 に代入する. (`allocatom [3, 1_1]`)
2. 変数番号 1 のアトム (ヘッドで取得した a(5, X)) の第 1 引数の接続先のアトムへの参照 (X) を, 変数番号 4 に代入. (`derefatom [4, 1, 1]`)
3. 変数番号 4 が整数アトムであることを確認. X が整数であるという制約を付ける. (`isint [4]`)
4. X+1 の計算結果のアトムを生成し, その参照を変数番号 5 に代入. (`iadd [5, 4, 3]`)

5. 整数アトム10を作り, その参照を変数番号6に代入. (`allocatom [6, 10_1]`)

6. 変数番号5の参照する整数アトム < 変数番号6の参照する整数アトムとなることを確認する. (`ilt [5, 6]`)
   ここまででヘッド側で自由リンクだったXが, $X+1<10$ という制約を守っていることを確認する.

7. ボディ命令列へ. (jump)

3.2.3 ボディ命令列

ボディ命令列は, 実際にルールが適用された場合にリンクの繋ぎ換え, アトムやルールの生成などを行う. ボディ命令列は以下の部分に相当する.

ボディ命令列

```
spec            [7, 8]
commit          [( a(~1,~5) 5(~1) $X[~5] :- 1($~3) '+'
                ($X,$~3,$~2) 10($~4) '<'($~2,$~4) | ok )]
dequeueatom     [1]
removeatom      [1, 0, a_2]
removeatom      [2, 0, 5_1]
removeatom      [4, 0]
newatom         [7, 0, ok_0]
enqueueatom     [7]
freeatom        [1]
freeatom        [2]
freeatom        [4]
freeatom        [6]
freeatom        [5]
freeatom        [3]
proceed         []
```

各々の命令の意味は次のようになっている.

commit `[ruleref]`
: 現在の実引数ベクタでルール ruleref に対するマッチングが成功したことを表す. 処理系は, この命令に到達するまでに行った全ての分岐履歴を忘却してよい.

dequeueatom `[srcatom]` アトム srcatom がこの計算ノードにある実行アトムスタックに入っていれば, スタックから取り出す.

rmoveatom `[srcatom, srcmem, funcref]`

(膜 srcmem にあってファンクタ funcref を持つ) アトム srcatom を現在の膜から取り出す.

newatom `[-dstatom, srcmem, funcref]` 膜 srcmem にファンクタ funcref を持つ新しいアトム作成し, 参照を dstatom に代入する. アトムはまだ実行アトムスタックには積まれない.

enqueueatom `[srcatom]`
アトム srcatom を所属膜の実行アトムスタックに積む.

freeatom `[srcatom]`
srcatom がどの膜にも属さず, かつこの計算ノード内の実行アトムスタックに積まれていないことを表す.

proceed
この proceed 命令が所属する命令列の実行が成功したことを表す. この命令が実行されれば, ルール適用の処理全て完了したことになる.

本論文ではボディ命令列に関しては最適化を行わないので, あまり詳しくは触れない. 簡潔にまとめると, ルール左辺のアトム (`a_2, 5_1`) を削除して, ルール右辺に書かれたファンクタ `ok_0` を持つアトムを新たに生成していることになる.

3.3 インタプリタの動作とその問題点

処理系内で解釈, 実行される命令列を紹介したが, 実際にその命令列を解釈するインタプリタの動作を説明する.
また, 現在のインタプリタの問題点を挙げる.

3.3.1 インタプリタの動作

処理系内部で動くインタプリタのソースの一部を示す.

インタプリタ

```
boolean interpret(List insts, int pc) {
   Iterator it;
   Atom atom;
   AbstractMembrane mem;
   Link link;
   Functor func;
   while (pc < insts.size()) {
      Instruction inst = (Instruction) insts.get(pc++);
      switch (inst.getKind()) {
         case Instruction.FINDATOM:// [-dstatom, srcmem, funcref]
         func = (Functor) inst.getArg3();
         it = mems[inst.getIntArg2()].atoms.iteratorOfFunctor(func);
         while (it.hasNext()) {
            Atom a = (Atom) it.next();
            atoms[inst.getIntArg1()] = a;
            if (interpret(insts, pc))
               return true;
            }
            return false;
         case Instruction.DEREF://[-dstatom, srcatom, srcpos, dstpos]
            link = atoms[inst.getIntArg2()].args[inst.getIntArg3()];
            if (link.getPos() != inst.getIntArg4()) return false;
                atoms[inst.getIntArg1()] = link.getAtom();
               break;
         case Instruction.FUNC://[srcatom, funcref]
            if (!((Functor)inst.getArg2()).equals(
               atoms[inst.getIntArg1()].getFunctor()))
               return false;
            break;
         case Instruction.ISINT://[atom]
            if (!(atoms[inst.getIntArg1()].getFunctor()
                instanceof IntegerFunctor)) return false;
               break;
        case Instruction.PROCEED:
           return true;
         ....
         }
      }
      return false;
}
```

ここでは, 前節でその動作を紹介した命令 (DEREF, FUNC 等) を実現している部分のコードを挙げた.

インタプリタ全体の処理の流れは, 命令列を受け取り, プログラムカウンタ pc

に従って順次命令を取得, その命令の種類に応じて, 変数ベクタで管理しているアトムや膜のリスト (atoms, mems) を操作していく. 変数ベクタは, フィールドに持つ情報であり, jump 命令実行時やルール適用時などにまとめて更新される.

3.3.2 インタプリタの再帰呼び出しを行う制御命令

インタプリタが boolean 型を戻り値としているのは, findatom のようにインタプリタを再帰呼び出しする命令があるためである. findatom の他には例えば, 膜を順次取得していく anymem 命令がある.

findatom ではそれ以下の命令列を引数にインタプリタを再帰呼び出しする. その戻り値が true なら findatom 命令は true を返し, false なら別のアトムを取得し直して再び findatom 以下の命令列を実行する.

なお, このようなループする命令以外では true を返すのは proceed 命令のみである. よって findatom 命令内に

```
if(interpret(insts, pc)) return true;
```

という箇所があるが, ここで実際に true が返ってくるのはボディ命令を完了時, つまりルール適用完了時のみである. それ以外で, 途中 isint 命令などで失敗したりすると, ここに false が返される.

その場合, 別のアトムを取得するのだが, その仕組みについて触れておく. アトムの取得にはイテレータを使う. findatom の役割はデータの中から第 3 引数で与えられたファンクタを持つアトムを取得することであり, この一連の処理がイテレータで行われる. 感覚的には, この findatom で取得されるアトムの候補がいくつもあり, イテレータでそれを取得する. その後それ以下の命令列実行時に false が返ってきたら, イテレータによって別のアトムを取得できる.

例として, 次のデータの中からファンクタ `a_1` を持つアトムを取得する場合の動作を確認する.

データ

a(1), a(5), b(1), a, a(3).

1. a(1) を取得.
2. 以降の命令列を実行していった結果 false が返ってきた.
3. a(5) を取得.
4. 以降の命令列を実行していった結果 false が返ってきた.
5. a(3) を取得.
6. 以降の命令列を実行していった結果 false が返ってきた.

7. 取得できるアトムがもうないので, この findatom より上にある findatom に false を返す. 上に findaotm がなければルール適用失敗.

ここから分かるように, findatom 命令は失敗時は 1 つ前に出てきた findatom 命令に戻るようになっている. これによって全てのアトムの取得パターンを総当りできる. しかし, 逆に言えば無駄な探索もそれだけ多いということになる.

3.3.3 実行時に無駄の生じるプログラムの例 1

前述の通り無駄が生じる例を, プログラムを交えて説明する.

プログラム 1

データ:
```
a(2), a(5), b(2), b(5).
```
ルール:
```
a(X), b(Y) :- X > 3, Y > 3 | ok.
```

生成されるヘッド命令列 (膜主導テスト部) とガード命令列を以下に示す.
データを生成する部分は省略する.

```
--memmatch:
              spec            [1, 3]
              findatom        [1, 0, a_1]
              findatom        [2, 0, b_1]
              jump            [L100, [0], [1, 2], []]
--guard:L100:
              spec            [3, 7]
              allocatom       [3, 3_1]
              derefatom       [4, 1, 0]
              isint           [4]
              igt             [4, 3]
              allocatom       [5, 3_1]
              derefatom       [6, 2, 0]
              isint           [6]
              igt             [6, 5]
jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

実行時にデータがどう処理されるかを追っていく.

1. a(2) を取得. (findatom [1, 0, a_1])
2. b(2) を取得. (findatom [2, 0, b_1])
3. 整数アトム 3 を生成. (allocatom [3, 3_1])
4. X の参照を 2 とする. (derefatom [4, 1, 0])

5. 2 が整数であるか確認. (`isint [4]`)

6. $2 < 3$ なので失敗. (`igt [4, 3]`)
   2 つ目の findatom へバックトラック.

7. `b(5)` を取得. (`findatom [2, 0, b_1]`)

8. 3〜6 を繰り返す.

9. `b_1` を持つアトムは全て試したので 2 つ目の findatomu 命令は失敗となる.
   よって 1 つ目の findaotm へバックトラック.

10. `a(5)` を取得. (`findatom [1, 0, a_1]`)

11. 2〜3 を繰り返す.

12. X の参照を 5 とする. (`derefatom 4, 1, 0]`)

13. 5 が整数であるか確認. (`isint [4]`)

14. $5 > 3$ であることを確認. (`igt [4, 3]`)

15. 整数アトム 3 を生成. (`allocatom [5, 3_1]`)

16. Y の参照を 2 とする. (`derefatom [6, 2, 0]`)

17. 2 が整数であるか確認. (`isint [6]`)

18. $2 < 3$ なので失敗. (`igt [6, 5]`)
    2 つ目の findatom へバックトラック.

19. `b(5)` を取得. (`findatom [2, 0, b_1]`)

20. 整数アトム 3 を生成. (`allocatom [3, 3_1]`)

21. 12〜15 を繰り返す.

22. Y の参照を 5 とする. (`derefatom [6, 2, 0]`)

23. 5 が整数であるか確認. (`isint [6]`)

24. $5 > 3$ であることを確認. (`igt [6, 5]`)

25. a(5), b(5) でルール適用成功.

この内無駄な箇所を挙げていく.

7〜9 この処理は 6 に起因するもの. 6 で失敗しているのは a(2) を取得したせいなので, `b_1` をいくら取り直しても解決しない.

15, 16 など 定数アトムの生成処理だが, X, Y それぞれと比べるための整数アトム 3 が 1 つずつあればいい. これを何度作り直したところで値が変わるわけではないので, 2 回目以降の生成は無駄である.

これらの無駄がなぜ生じたかを考えると, ガード命令列の出現の遅さに原因があると考えられる. ガード命令列による検査は, ヘッド命令列が全て終わった後で実行しなければいけないわけではない. ルールは 3 部位に分かれているとはいえ, インタプリタで解釈されるのは 1 つの長い命令列である. よってガード命令列中の命令をヘッド命令列の中へ移動することは可能である. 命令列の並び替えにより上のような無駄を解決するための最適化手法を第 4 章で述べる.

最後に処理の流れをバックトラックの様子を図でまとめる.

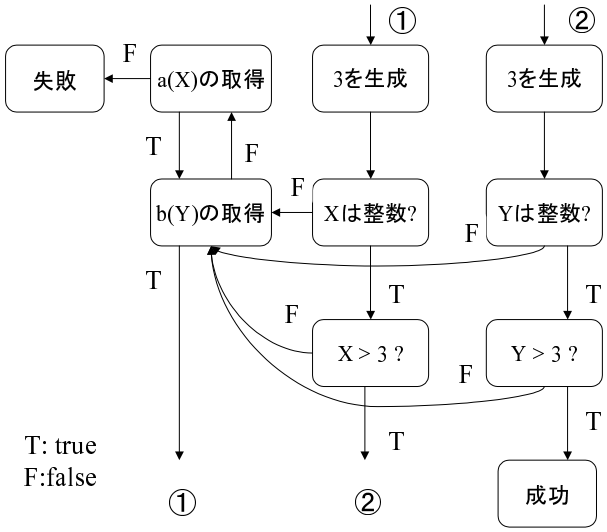

図 3.2: バックトラックの様子

なお, この図において b(Y) の取得に失敗した時 a(X) の取得に戻っているが, 実際に a(X) を取得した後, 再び b(Y) の取得を試みる際, ファンクタ `b_1` を持つアトムを取得する findatom 命令を最初からやり直すことになる. つまり, b(Y) 取得用のイテレータが初期化されるので, b(Y) はまた最初から選び直される.

3.3.4 実行時に無駄の生じるプログラムの例 2

もうひとつ無駄の生じるプログラムを挙げる.

プログラム 2

データ:

```
a, a, a, a, a.
```

ルール:

```
a, b :- ok.
```

命令列は次のようになる.
このプログラムにはガード部の記述がないのでガード命令列は省略する.

```
--memmatch:
   spec           [1, 3]
   findatom       [1, 0, a_0]
   findatom       [2, 0, b_0]
   jump           [L153, [0], [1, 2], []]
```

ガード命令命令列が無いので, 先ほどの例に比べるととてもシンプルである. このルールは, データの中から a と b を探しアトム ok を生成するというもの. しかし, データは a が 5 個あるのみで b がない. よってこのルールは失敗するのだが, その失敗が分かるまでのプロセスを以下に示す.

1. a を取得.
2. b を取得しようとするが失敗.
3. 2 個目の a を取得.
4. b を取得しようとするが失敗.
5. 3 個目の a を取得.
6. b を取得しようとするが失敗.
7. 4 個目の a を取得.
8. b を取得しようとするが失敗.
9. 5 個目の a を取得.
10. b を取得しようとするが失敗.
11. a を取得しようとするが失敗.

このプロセスの内無駄な部分はどこかを考えると, 実は 3〜11 が全て無駄になっている. b が無いからといって a を取得し直したところで b が生成されるわけではないからである. どう動くのが望ましいかというと, b の取得に失敗した時点でルール適用失敗となって欲しいわけである. この問題を解決するための手法も第 4 章で述べる.

第4章　並び替えとグループ化による命令列の最適化

3章3節で説明した問題点を解決するため次の2つを実装した. また, 本論文の最適化は膜の無いプログラムを対象とする.

1. 命令列の並び替え
   無駄の生じる例1の解決を目指したもの. ガード命令列の命令をなるべく早い段階で済ませるため, 命令列を並び替える.
   なお, 今回移動の対象となる命令はガード部に書いた算術演算子, 比較演算子とそれらに関係する命令とする. findatom など元々ヘッド部にあった命令の並び換えは行わない.

2. 命令列のグループ分け
   無駄の生じる例2の解決を目指したもの. ルール適用失敗が確定してもルール適用を試みるという問題を解決する. そのために命令列をお互いに関連付けられるグループ単位に分割することで, 命令列を独立した処理がいくつか集まったものとして扱うようにする. そのグループの取得に失敗したら即座にルール適用失敗となるようにした.

これらを順に解説していく. なお, これらの最適化はOオプションで実行される通常の最適化器にはまだ統合されていない. 本論文の最適化はZオプションで行うものとする.

Z1 以上 命令列の並び替え

Z4 以上 命令列のグループ分け+Z1

これは暫定的な最適化オプションであり, いずれ通常の最適化器に統合する予定である.

4.1　命令列の並び替え

繰り返しになるが, もう一度無駄の生じる例1のプログラムを見てみる.

無駄の生じる例 1

データ:

```
a(2), a(5), b(2), b(5).
```

ルール:

```
a(X), b(Y) :- X > 3, Y > 3 | ok.
```

命令列:

```
--memmatch:
   spec           [1, 3]
   findatom       [1, 0, a_1]
   findatom       [2, 0, b_1]
   jump           [L100, [0], [1, 2], []]
--guard:L100:
   spec           [3, 7]
   allocatom      [3, 3_1]
   derefatom      [4, 1, 0]
   isint          [4]
   igt            [4, 3]
   allocatom      [5, 3_1]
   derefatom      [6, 2, 0]
   isint          [6]
   igt            [6, 5]
   jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

このプログラムを動かした際の無駄を解決するための設計, 実装を次に述べる.

4.1.1 設計

手始めにヘッド命令列とガード命令列を 1 つにまとめる必要がある. ヘッド命令列最後の jump 命令はガード命令列を続けて実行するというものなので, 単純に jump の位置にガード命令列を展開すればよい. この処理は最適化レベル 1(O1 オプション) で行われるようになっている. ちなみに Z オプションによる最適化では, この処理は前提条件となっている. よって Z1 以上で処理系を動かすと, 通常の最適化レベルは強制的に 1 になるようになっている.

ヘッドとガードをまとめた命令列は次のようになる.

命令列 (jump 命令展開後)

```
--memmatch:
   spec           [1, 7]
   findatom       [1, 0, a_1]
   findatom       [2, 0, b_1]
   allocatom      [3, 3_1]
   derefatom      [4, 1, 0]
   isint          [4]
   igt            [4, 3]
   allocatom      [5, 3_1]
   derefatom      [6, 2, 0]
   isint          [6]
   igt            [6, 5]
   jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

この命令列にしても処理の流れは全く変わらないので, 無駄な処理が解決しているわけではない. まずはこの命令列がどういう並びになると, 命令失敗時のバックトラック処理が最適化されるかを考える.

この命令列中, 失敗する可能性があるのは次の命令.

findatom, isint, igt

この内 findatom は, a(X) と b(Y) の取得順を変えても意味がない上, 現在の処理系だと, ある findatom より下にある findatom は上の findatom の再帰呼び出しの中に入っていることになっているので, 単純にこの順番を入れ替えるわけには行かない. 4 章の冒頭で述べた通り, 移動による最適化の対象はガード命令列にあった命令に限る. よってこの場は isint, igt での失敗時について考える.

isint, igt はともに失敗時にはそれより上の最寄の findatom にバックトラックする. そして別のアトムを取得して命令列を実行してくるわけだが, 失敗に関係したアトムを変更するのではなく単に最後に取得したアトムを変更するということに問題がある. 図で無駄なバックトラックを表現すると次のようになる.

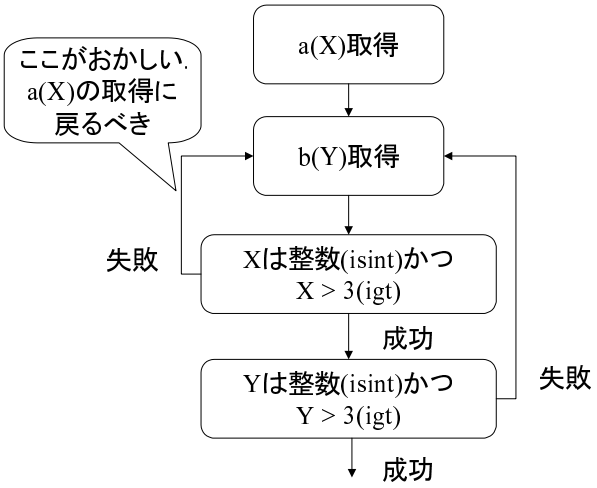

図 4.1: isint, igt 命令失敗時のバックトラック

この性質に合わせると, 失敗する可能性のある命令が, その命令に関係のあるアトムを取得した直後にあれば解決すると思われる. 直感的には, 次のようにプログラムを書いたのと同じ命令列に並び替えたい. (実際にはユーザーはこういうプログラムは書けない).

```
a(X), b(Y) :- X > 3, Y > 3 | ok.
```

⇩

```
a(X), X > 3, b(Y), Y > 3 :- ok.
```

図 4.2: 直感的なプログラム

また, バックトラックして以前実行した命令を再実行する際に, 定数アトムを生成する allocatom 命令は何度も実行しても意味がないと思われる. allocatom 命令は何度実行しても結果は同じなので, ルールの先頭で行ってもよいものである. よって allocatom 命令は spec 命令の直後に移動させることにする.

以上を踏まえ,

- 移動できる命令は, 関係するアトムが生成された直後に移動させる.
- allocatom 命令はルールの先頭, すなわち spec 命令の直後に移動させる.

という方針で実装した.

目標達成後の命令列は次のようになっているはずである.

目標状態

```
--memmatch:
   spec           [1, 7]
   allocatom      [3, 3_1]
   allocatom      [5, 3_1]
   findatom       [1, 0, a_1]
   derefatom      [4, 1, 0]
   isint          [4]
   igt            [4, 3]
   findatom       [2, 0, b_1]
   derefatom      [6, 2, 0]
   isint          [6]
   igt            [6, 5]
   jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

4.1.2 実装

この命令列に対し何をすべきかをまとめると, 命令列を上から見ていきながら下の処理を行う.

1. allocatom 命令を spec 命令の直後に移動.
2. derefatom 命令を第 2 引数の変数番号が定義された findatom 命令の下に移動.
3. isint 命令を引数の変数番号が定義された derefatom 命令の下に移動.
4. igt 命令を第 2, 第 3 引数のアトムが両方とも isint 命令で整数であることが確認された直後に移動.

ここで 2, 3 で述べている “定義された” というフレーズについて説明する.
3 章の繰り返しになるが derefatom, isint の両命令の意味を再確認する.

derefatom `[-dstatom, srcatom, srcpos]`
: アトム srcatom の第 srcpos 引数のリンク先のアトムへの参照を dstatom へ代入する.

isint `[atom]`
: アトム atom が整数アトムであることを確認する.

これを見て分かるのは, derefatom は srcatom(第 2 引数) を参照して, 結果定義するアトムを dstatom(第 1 引数) に代入している. よって derefatom は srcatom を変数番号に持つアトムが定義された後になければならない. このプログラム上で srcatom がどこで定義されているかを見ると, findatom 命令の第 1 引数である. 定義されるアトムは出力引数として命令列に出現している.
(`findatom [-dstatom, srcmem, funcref]`)
isint に関しても同様に, 引数のアトムを参照するので atom が定義された後でなければならない.

以上から “ 定義された” というのは, “その命令で使用されるアトムが他の命令の出力引数として出現している” ということ意味するものとする.

具体的な実装を以下に述べる. 基本的な流れとしては, ヘッド命令列 (ガード命令列を末尾に展開後) を先頭から見ていき, 移動対象の命令があったら可能な限り前の方へ押し上げるというもの.

まず ALLOCATOM の移動のソースは以下のようになる.

ALLOCATOM の移動

```
public static void guardMove(List head){
   for(int hid=1; hid<head.size()-1; hid++){
      Instruction insth = (Instruction)head.get(hid);
      //ALLOCATOM 命令は先頭の SPEC の直後に移動
      if(insth.getKind() == Instruction.ALLOCATOM){
         head.remove(hid);
         head.add(1, insth);
      }
      ...
}
```

命令のリストとして記述された命令列 head を先頭から先頭から見て, ALLOCATOM があったら SPEC 命令の直後, リストのインデックスで言うと 1 の位置に移動する. このやり方で実行と次の図のように元の ALLOCATOM の並びと逆になるが, ALLOCATOM 命令は定数アトムの生成であり, 生成にあたり他のアトムより前にあってはならないという制約はないので問題ない.

他の命令をどうするか.
まず最初に実装したのは, 移動させたい命令を見つけたらその命令で使用しているアトムを定義している命令の直後に移動させるというものだった. ただしこの実装の問題点は, 移動先の候補となる命令の種類が移動させる命令によって異なるということで, それを考慮してプログラムを書くと非常に長くなることだった. 命令別の移動先の候補は次のようになっている.

derefatom findatom か deref の直後.

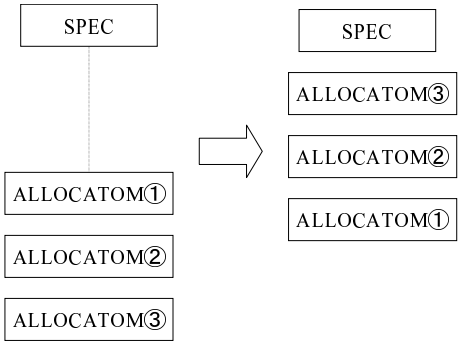

図 4.3: allocatom 命令の移動

isint derefatom の直後

iadd isint, allocatom, iadd, isub, imul, idiv, imod の直後.

ilt isint, allcoatom, iadd, isub, imul, idiv, imod の直後.

プログラムの 1 部

```
case Instruction.DEREFATOM:
  for(int hid2=hid-1; hid2>0; hid2--){
     Instruction insth2 = (Instruction)head.get(hid2);
        switch(insth2.getKind()){
           case Instruction.FINDATOM:
           case Instruction.DEREF:
              if(insth.getIntArg2() == insth2.getIntArg1()){
                 head.add(hid2+1, insth);
                 head.remove(hid+1);
              }
              break;
           .....
```

このプログラムでやっているのは DEREFATOM の移動で, その流れは次のようになっている.

1. 命令列中に DEREFATOM を発見.
2. その位置から命令列をさかのぼって FINDATOM か DEREF を探す.
3. FINDATOM, DEREF の第1引数が出力引数となっているので, これと DEREFATOM の第 2 引数を比較. 同じだったらこの FINDATOM か DEREF の直後に DEREFATOM を移動させる.

これと似たようなコードが各命令ごとにあるわけだが, この実装の問題点は他にもある. まず 1 つは新しい命令を追加するとまたコードに追加しなければならないこと. もう 1 つは, 例では DEREFATOM の移動先の候補は findatom または deref の直後ということになっているが, 将来的には変わるかもしれない, というより現段階でも他にあるかもしれない. つまり, 移動させたい命令とその移動先を漏れなく記述しなければならないというのが問題ということになる.

よって実装を変更した.
命令列で着目すべき点として, アトムや膜の変数番号は全て単一代入であるということである. つまり, ある変数番号が定義されるタイミングは命令列全体で 1 回のみである. よって定義済みの変数番号を使う命令は, その変数番号が初めて定義された命令より後ろである限り, 命令列の前の方へ押し上げることが出来るということになる. igt や iadd のように 2 つ以上の変数番号を使う命令の場合は, 2 つとも定義済みである限り押し上げることができる.
ここで便宜上, 命令の引数の内, アトムや膜の変数番号を表したものを次の 2 種類に分類する.

def 変数番号を初めて定義している引数. 出力引数がこれにあたる. また, 出力引数を持つ命令の場合, 出力は第 1 引数となっている. 1 つの命令で複数の変数番号を定義することはない.

use 変数番号を使用している引数.

これら 2 つを用いて移動を制御することを考える. 図にすると次のようになる.

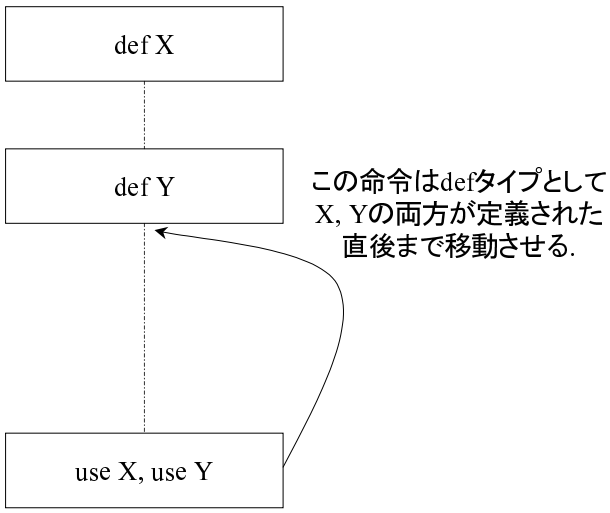

図 4.4: 命令移動の様子

この制御を利用し, 次のように実装し直した.

まず, 本論文で移動させる命令はガード命令列に書かれた命令のみということから, ヘッド命令列とガード命令列を統合した命令列の内, ガード命令列の命令がどこから始まるかをチェックしなければならない. ここではガード命令列の先頭にありそうな命令が見つかるまで命令列を検索し, その位置の値を guardinstsstart に代入することで実現した.

冒頭からその部分までのプログラムは次のようになる.

命令列の並び替えのプログラム 1

```
public static void guardMove(List head){
   int guardinstsstart = 0;
   for(int hid=1; hid<head.size()-1; hid++){
      Instruction insth = (Instruction) head.get(hid);
      switch(insth.getKind()){
      //ガード命令列の先頭にありそうな命令
         case Instruction.ALLOCATOM:
         case Instruction.DEREFATOM:
         case Instruction.GETLINK:
         case Instruction.NATOMS:
         case Instruction.NMEMS:
         case Instruction.NORULES:
            guardinstsstart = hid;
            hid = head.size();
            break;
            default: break;
         }
      }
      ...
```

“ガード命令列の先頭にありそうな” というのはかなり曖昧に見えるが, 本節の検証で使うのは膜の無いプログラムであり, かつ移動させたい命令は算術演算子や比較演算子に関係する命令のみ. この場合ガード命令列の先頭にある命令は DEREFATOM か ALLOCATOM である. よって本論文の検証内では必ず最適化されるので, 今回はこのまま議論を進める.

次に guradinstsstart 以下にある命令の移動を行う.

ALLOCATOM は前述の通り, 単純に SPEC 命令の直後に移動させる.

その他の命令は, 命令の引数の内 use のもの全てが, 他の命令で def として出現した後に移動させたい. ここで命令の持つ引数の内, use の変数番号のリストを返すメソッド getVarArgs() を用いる. 上の命令が, このリストに含まれる変数番号を定義するものでない限り, 命令を押し上げられる.

しかしこの方針で実装すると, 同じ変数番号を使用する命令があった場合, 前の方にある命令が先に動くということから, その前後関係が逆転してしまう. (igt

の方が isint より前に来る, など)

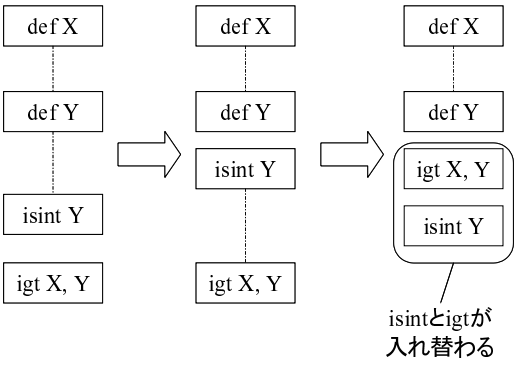

図 4.5: 移動の不具合の例

この問題を解決するため, isint 命令や isfloat 命令のような, 同じ変数番号を使用する他の命令より優先度の高い命令は追い越せないようにした.

この部分の実装をしているプログラムを次に挙げる. これは先程のプログラムの続きである.

命令列の並び替えのプログラム 2

```
...
if(guardinstsstart != 0) //ガード命令列が空の場合は実行しない.
   for(int hid=guardinstsstart; hid<head.size()-1; hid++){
      Instruction insth = (Instruction)head.get(hid);
      ArrayList list = insth.getVarArgs();
      boolean moveOk = true;
      //ALLOCATOM 命令は先頭の SPEC の直後に移動
      if(insth.getKind() == Instruction.ALLOCATOM){
         head.remove(hid);
         head.add(1, insth);
      }
      else for(int hid2=hid-1; hid2>0; hid2--){
            Instruction insth2 = (Instruction)head.get(hid2);
            ArrayList list2 = insth2.getVarArgs();
            for(int i=0; i<list.size(); i++){
               if((insth2.getOutputType() != -1
                && list.get(i).equals(insth2.getArg1()))
                || orderConstraintCheck(insth, insth2, list.get(i))){
                  moveOk = false;
                  break;
               }
            }
            if(moveOk){
               //移動対象の命令 insth の位置は hid2+1
               head.remove(hid2+1);
               //hid2 に移動
               head.add(hid2, insth);
            }
            else break;
         }
   }
}
```

このプログラムの内, 移動できる限界点を調べているのが

```
if((insth2.getOutputType() != -1
  && list.get(i).equals(insth2.getArg1()))
  || orderConstraintCheck(insth, insth2, list.get(i))){ ...
```

の部分である.

getOutputType というメソッドは命令が出力引数を持つものでない場合-1 を返す. また, orderConstraintCheck は第 1 引数 (移動対象の命令) と第 2 引数 (1 つ上の命令) が共に第 3 引数の変数番号を使っている場合に, この 2 つの命令の順序を入れ替えられるかをチェックする. よってここで行っている終了判定は次の 2 つ.

1. 1 つ上にある命令が出力引数を持つ命令であり, かつそこで定義される変数

番号が移動させようとしている命令で使われているものである場合.

2. 1 つ上にある命令と移動させようとしている命令で使われている変数番号が同じであり, 上の命令の方が優先度が高い場合.

以上を実装して, 例のプログラムを Z1 オプションで実行したと次のような命令列が生成された.

無駄が生じる例 1 最適化後

```
a(X), b(Y) :- X > 3, Y > 3 | ok.
--memmatch:
   spec           [1, 7]
   allocatom      [5, 3_1]
   allocatom      [3, 3_1]
   findatom       [1, 0, a_1]
   derefatom      [4, 1, 0]
   isint          [4]
   igt            [4, 3]
   findatom       [2, 0, b_1]
   derefatom      [6, 2, 0]
   isint          [6]
   igt            [6, 5]
   jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

allocatom の順序が入れ替わっているが, 目標状態の通りに並び替えることができた.

4.1.3 検証

詳しい検証はグループ分け (Z4) の設計, 実装の説明の後 5 章でも行うが, 無駄が生じる例 1 について簡単に性能評価をしておく.

検証用データの生成プログラム

```
num(0). num(X) :- X<300 |
   a(integer.rnd(10)), b(integer.rnd(10)), num(X+1).
```

0 から 9 までの整数をランダムに引数とし, ファンクタ `a_1` , `b_1` を持つアトムを 300 個ずつ生成する.

2 章で紹介した integer モジュールを使用して, ランダムな整数を得ている. このプログラムを用いて 10 通りのデータを生成し, それらのデータに対し, ルー

ル `a(X), b(Y) :- X > 3, Y > 3 | ok.` を最適化レベル Z0, Z1 それぞれで実行する.

ルール適用開始から終了までの時間は次のようになる.

表 4.1: Z1 オプションの性能評価

データ	`Z0[ms]`	`Z1[ms]`
1	27900	200
2	34120	190
3	30554	207
4	28351	210
5	29892	211
6	31335	193
7	23364	210
8	23364	210
9	30964	208
10	19658	210
平均	29410.3	208
性能比	1.00	141

この例では, 平均で比較すると Z1 オプションで最適化した場合, 最適化なしに比べると約 141 倍高速である.

これは $X > 3$ で失敗した時に, Z0 では b(Y) の取り直しという無駄な作業があるのに対し, Z1 では即座に a(X) の取り直しを行うことによる差である. この無駄な作業は 1 回の $X > 3$ での失敗につき, ファンクタ `b_1` を持つアトムの個数分だけ行われることになるためこれ程の差がついていると思われる.

4.2 命令列のグループ分け

無駄が生じる例 2 をもう一度見てみる.

無駄が生じる例 2

データ:

```
a, a, a, a, a.
```

ルール:

```
a, b :- ok.
```

命令列:

```
--memmatch:
   spec            [1, 3]
   findatom        [1, 0, a_0]
   findatom        [2, 0, b_0]
   jump            [L153, [0], [1, 2], []]
```

ここでやりたいことは命令列をグループ単位に分けて, あるグループの取得に失敗したら即座にルール適用失敗とすることである. 直感的には上の命令列を次のようにグループ分けしたい.

目標状態

```
--memmatch:
   spec            [1, 3]
   group     [
      findatom        [1, 0, a_0]  ]
   group     [
      findatom        [2, 0, b_0]  ]
   jump            [L153, [0], [1, 2], []]
```

group 命令の定義は次の通り.

group `[subinsts]`

命令列 subinsts を実行する.

失敗したらルール適用は失敗となる.

例で挙げてあるデータに対し, この命令列でルール適用を試みた場合, 2 つ目のグループの取得は 1 回目で失敗する. よってファンクタ `a_0` を持つアトムの取得を無駄に何度も行うことはない.

4.2.1 設計

設計に先立ってアトムグループという概念を定義する.

アトムグループ

ルール左辺においてリンクでつながれているアトムの集合.

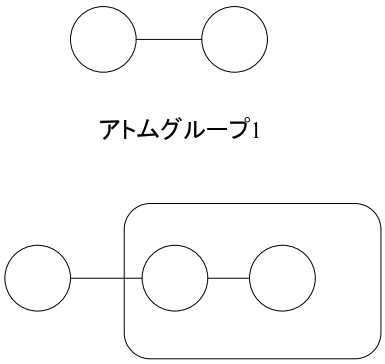

図 4.6: アトムグループ

図 4.6 にアトムグループの例を示す. この定義に従うと, 命令列上ではどこからどこまでが 1 つのアトムグループになっているかと言うと, 単純に findatom から findatom の間ということになる. よって命令列の先頭から検索して findatom を見つけたら, そこから次の findatom の直前までの命令を命令列として, それを引数に持つ group 命令を作ればアトムグループ単位でのグループ分けは実現できる.

しかし, 実際にこの方針で実装したところ次のような問題点があった.

(1) 同じファンクタを持つアトムを取得するグループが 2 つ以上ある場合.

プログラム例

```
データ:
a(3), a(10).
ルール:
a(X), a(Y) :- X > 1, Y < 5 | ok.
命令列 (Z1 オプションによる最適化後とする):
--memmatch:
   spec          [1, 7]
   allocatom     [5, 5_1]
   allocatom     [3, 1_1]
   group   [
      findatom   [1, 0, a_1]
      derefatom  [4, 1, 0]
      isint      [4]
      igt        [4, 3]   ]
   group   [
      findatom   [2, 0, a_1]
      derefatom  [6, 2, 0]
      isint      [6]
      igt        [6, 5]
      neqatom    [1, 2]
      jump       [L158, [0], [1, 2, 3, 4, 5, 6], []]
```

本論分ではここまでに説明していなかった neqatom 命令の意味は次の通り.

neqatom `[atom1, atom2]`

atom1 と atom2 が異なるアトムを参照していることを確認する。

プログラムを実行すると,

1. 1 つ目のグループに a(3) がマッチ.
2. 2 つ目のグループに a(10) はマッチしない.
3. 2 つ目のグループのマッチングに失敗したのでルール適用失敗.

となってしまう.

実際には, 1 つ目のグループで a(10) を取得し直せば, 2 つ目のグループで a(3) がマッチするのでルール適用成功となるはずである. このように, 同じファンクタを持つアトムを取得するグループが 2 つ以上あるとうまく行かない.

(2) アトムグループを跨る命令があった場合

プログラム例

```
ルール:
a(X), b(Y) :- X > Y | ok.
命令列:
group   [
   a(X)   ]
group   [
   b(Y)   ]
X > Y
```

2つのアトムグループを跨る命令の行き場が問題となる. 上ではグループ外に置いているが, 命令列上ではグループ単位で処理が独立しているため, この場合 $X > Y$ で失敗すると, バックトラックせずルール適用失敗となる. 仮に 2 つ目のグループの中に $X > Y$ を入れたとしても, b(Y) の取得をし直すだけで a(X) のアトムは最初に 1 つ目のグループで取得したものから変更できない.

これら 2 点を踏まえると, 命令列を区切るグループの単位はアトムグループではないことが分かる. 従って次に定義するグループ単位で分けることにする.

グループの定義

ファンクタが同じアトムを取得したり, ある命令でお互いに関連付けられるアトムグループを 1 つの集団としてまとめたもの. 命令列はグループ単位で処理が独立するものとする.

この定義に従い実装を行った.

4.2.2 実装

無駄が生じる例 2 のプログラムでは少々単純すぎるので, 次のプログラムを例にグループ化を考える.

プログラム例

```
a(X), b(Y), c(Z) :- X > Y, int(Z) | ok.

グループ (1) [a(X), b(Y), X > Y]
グループ (2) [c(Z), int(Z)]
```

上のようにグループを (1) と (2) に分けることを目標とする.

また命令列の並び替えは, グループ分けを行った後で各々のグループ内の命令列に対して行うものとする. よって次の命令列をうまくグループ化することを考える.

命令列

```
--memmatch:
   spec            [1, 7]
   findatom        [1, 0, a_1]
   findatom        [2, 0, b_1]
   findatom        [3, 0, c_1]
   derefatom       [4, 1, 0]
   derefatom       [5, 2, 0]
   isint           [4]
   isint           [5]
   igt             [4, 5]
   derefatom       [6, 3, 0]
   isint           [6]
   jump            [L142, [0], [1, 2, 3, 4, 5, 6], []]
```

実装の方法として, 次の 2 つのハッシュマップを用いた.

var2DefInst 変数番号→変数番号を定義した命令 (変数番号が def として出現) へのマップ.

Inst2GroupId 命令→命令が所属するグループ番号へのマップ.

最終的に Inst2GroupId マップで同じグループ番号を値として持つキー (命令) を同一のグループとみなすことが出来る. 初期状態ではグループ番号は全て異なるとする.(行番号を値とする) ただし spec 命令と jump 命令はグループ化の対象外なので, マッピングもしない.

実際に初期化した時の var2DefInst と Inst2GroupId は次のようにマップされている.

var2DefInst

```
1 -> findatom   [1, 0, a_1]
2 -> findatom   [2, 0, b_1]
3 -> findatom   [3, 0, c_1]
4 -> derefatom  [4, 1, 0]
5 -> derefatom  [5, 2, 0]
6 -> derefatom  [6, 3, 0]
```

Inst2GroupId

```
findatom   [1, 0, a_1]   -> 1
findatom   [2, 0, b_1]   -> 2
findatom   [3, 0, c_1]   -> 3
derefatom  [4, 1, 0]     -> 4
derefatom  [5, 2, 0]     -> 5
isint      [4]           -> 6
isint      [5]           -> 7
igt        [4, 5]        -> 8
derefatom  [6, 3, 0]     -> 9
isint      [6]           -> 10
```

実際に初期化後, マッピングしていく部分のプログラムを次に挙げる.

プログラム

```
private void createGroup(List head){
   for(int hid=1; hid<head.size()-1; hid++){
      Instruction insth = (Instruction)head.get(hid);
      Object group = null;
      Object changegroup = null;
      ArrayList list = insth.getVarArgs();
      for (int i = 0; i < list.size(); i++) {
         if (list.get(i).equals(new Integer(0))) continue;
         group = Inst2GroupId.get(var2DefInst.get(list.get(i)));
         changegroup = Inst2GroupId.get(insth);
         changeMap(changegroup, group);
      }
   }
   ...
}
```

順を追って説明していくと,

1. 命令列の先頭から順次命令を取得する.(spec, jump は除く)
2. 取得した命令の引数の内, use であるものを list に入れる. getVarArgs() と利用.
3. list から順次変数番号を取得し, その変数番号が定義された命令の所属するグループ (group) を取得する.
4. 1 で取得した命令のグループを changegroup として取得.
5. Inst2GroupId マップにおいて, 値が changegroup となっている全てのキーに対し, その値を group に書き換える. この処理を行うメソッドが changeMap(changegroup, group) である. この処理によって, 互いに関連づけられる命令のグループ番号が統一される.

最終的に Inst2GroupId マップは次のようになる.

Inst2GroupId

```
findatom   [1, 0, a_1]   -> 2
findatom   [2, 0, b_1]   -> 2
findatom   [3, 0, c_1]   -> 3
derefatom  [4, 1, 0]     -> 2
derefatom  [5, 2, 0]     -> 2
isint      [4]           -> 2
isint      [5]           -> 2
igt        [4, 5]        -> 2
derefatom  [6, 3, 0]     -> 3
isint      [6]           -> 3
```

ここまで出来てしまえば, 後は先頭から同じグループに所属する命令をかき集めて命令列を作り, それを group 命令の引数 subinsts にすればいい. この時, subinsts の先頭に spec, 末尾に proceed 命令を入れる.

spec 命令は現在の処理系の仕様上, 命令列の先頭には spec 命令があることになっているため必要. spec 命令はインタプリタ呼び出しに先立って参照され, インタプリタの仮引数と局所変数の数を指定する. インタプリタ実行中に spec 命令があった場合, その局所変数の数を, 現在の値から spec 命令の第 2 引数の値に拡張する. 元の値より spec 命令の第 2 引数の値の方が小さい場合, spec 命令はスルーされる. subinsts においては局所変数の数を拡張させる必要は無いので spec 命令はスルーすればいい. 第 2 引数が 0 であれば必ずスルーされるので, spec[0, 0] を subinsts を入れることにする.

proceed 命令は, グループ命令の引数の命令列 subinsts が成功した時に true を返してくれるようにするために必要. インタプリタ上で group 命令は

```
case Instruction.GROUP:
   subinsts = ((InstructionList)inst.getArg1()).insts;
   if(!interpret(subinsts, 0)) //true だったら次の命令へ
    return false;              //false だったらルール適用失敗
   break;
```

という処理を行っている. よって必ず true か false を返してくれないと困る. group 命令内部の命令列の最後までいったら true を返してくれるようにすればいいので, 単純に true を返すだけの proceed 命令を subinsts の末尾に入れておく.

そして最後に subinsts を Z1 オプション時に使うのと同じ, 命令列の並びかえを行う最適化器に通せば完成となる.

グループ分け後の命令列

```
--memmatch:
   spec           [1, 7]
   group          [
      spec           [0, 0],
      findatom       [1, 0, a_1],
      derefatom      [4, 1, 0],
      isint          [4],
      findatom       [2, 0, b_1],
      derefatom      [5, 2, 0],
      isint          [5],
      igt            [4, 5],
      proceed        [] ]
   group          [
      spec           [0, 0],
      findatom       [3, 0, c_1],
      derefatom      [6, 3, 0],
      isint          [6],
      proceed        [] ]
    jump           [L101, [0], [1, 2, 3, 4, 5, 6], []]
```

4.2.3 検証

無駄が生じる例 2 のプログラムで Z0 との性能を比較する.

データ生成のプログラム

```
num1(0). num1(X) :- X<100000 | a, num1(X+1).
num2(0). num2(Y) :- Y<100    | b. num2(Y+1).
```

a を 100000 個, b を 100 個生成する.

この内, a の数は 1000, 10000, 30000, 100000 の 4 通りで計測する. これらのデータに対し, ルール `a, b :- ok.` を最適化レベル Z0, Z4 それぞれで実行する.

表 4.2: Z4 オプションの性能評価

a の個数	Z0[ms]	Z4[ms]
1000	110	97
10000	161	110
30000	304	161
100000	2904	360

Z0 では a が尽きるまで無駄なルール適用が繰り返されるが, Z4 では b が尽きた時点でルール適用失敗が確定する. よって b の数を固定すれば, a の数が 1 億だろうが 1 兆だろうがルール適用にかかる時間はそれほど増加しないはずである. 実際には Z4 も a の個数の増加に合わせて時間が増加しているが, これはデータが増えるにつれメモリを食うようになる等が原因であると思われる. ルール適用を試みる回数自体は a の個数には依存しない.

第5章　検証と考察

この章では 4 章で実装した最適化レベル間での効率の差を検証し, その結果を考察する.

なお, 本論文で性能評価を行った PC の環境は次の通りである.

CPU: Intel Pentium Ⅲ 845MHz
メモリ: 256MB
OS: Microsoft Windows XP Home Edition version 2002 Service Pack 2

検証には次のプログラムを使用する.

検証用プログラム

```
ルール:
a(A), b(B), c(C), d(D) :- A > B, C > D | ok(A, B, C, D).
データ生成部
num1(0). num1(AB) :- AB > 100 |
 a(integer.rnd(100)), b(integer.rnd(100)), num(AB+1).
num2(0), num2(CD) :- CD > 100 |
 c(integer.rnd(100)), d(integer.rnd(100)), num(CD+1).
```

データはファンクタ a_1, b_1, c_1, d_1 を持つアトムの数を変動させて計測する必要がある.

なおこのルールは, Z4 オプションでグループ化すると次のような命令列が生成される.

グループ化後の命令列

```
--memmatch:
   spec     [1, 9]
   group    [
             spec           [0, 0],
             findatom       [1, 0, a_1],
             derefatom      [5, 1, 0],
             isint          [5],
             findatom       [2, 0, b_1],
             derefatom      [6, 2, 0],
             isint          [6],
             igt            [5, 6],
             proceed        [] ]
   group    [
             spec           [0, 0],
             findatom       [3, 0, c_1],
             derefatom      [7, 3, 0],
             isint          [7],
             findatom       [4, 0, d_1],
             derefatom      [8, 4, 0],
             isint          [8],
             igt            [7, 8],
             proceed        [] ]
   jump           [L101, [0], [1, 2, 3, 4, 5, 6, 7, 8], []]
```

よって a(A), b(B) 及び c(C), d(D) がそれぞれ同じグループに属しているので, それらの数はセットで変動させる.

データの例

a_1 × 200, b_1 × 200
c_1 × 400, d_1 × 400

Z0, Z1, Z4 別の計測結果を以下に挙げる.
計測時間はルールの適用が始まってから終わるまでであり, 単位はミリ秒. 異なるデータで 10 回ずつ計測した.
また, 表中の N_{ab}, N_{cd} はそれぞれ ファンクタ a_1, b_1 及び c_1, d_1 を持つアトムの数を表す.

見て分かるように Z0 だけ桁違いに遅い. Z1 と Z4 はまだ差が小さいのでもっとデータを大きくしたいところだが, そうすると Z0 での計測が困難になる. よって Z1 と Z4 だけの性能比較を大きいデータで後で行うものとし, ここでは Z0 の遅い原因について考察する.

最適化なしの Z0 オプションによるバックトラックの様子を図 5-1 に示す.

Z0 オプションの場合, どの命令で失敗しても, 命令列中で直前に出てきた findatom の所へ順次バックトラックする. 例えば $A > B$ で失敗した場合の処理の流

表 5.1: $N_{ab} = 100$, $N_{cd} = 100$

データ	Z0[ms]	Z1[ms]	Z4[ms]
1	158628	691	240
2	524785	391	370
3	326850	251	230
4	409048	561	241
5	329524	911	230
6	284048	310	290
7	1006738	370	360
8	149465	301	260
9	360579	401	331
10	462295	260	251
平均	401196	444.7	280.3
性能比	1.00	902	1430

れを示す.

1. d(D) の選び直しをする.
2. $A > B$ で失敗.
   d(D) の候補があるなら 1 へ. 無いなら 3 へ.
3. c(C) の選び直しをする.
4. $A > B$ で失敗.
   c(C) の候補があるなら 1 へ. 無いなら 5 へ.
5. $A > B$ で失敗.
6. b(B) の選び直しをする.
7. $A > B$ で成功なら次 ($C > D$) へ.
   失敗の場合, b(B) の候補がまだあるなら 1 へ. 無いなら 7 へ.
8. a(A) の選び直しをする.
9. $A > B$ で成功なら次 ($C > D$) へ.
   失敗の場合, a(A) の候補がまだあるなら 1 へ. 無いならルール適用失敗.

この処理の内 1 から 5 は無駄であるが, Z0 オプションだと $A > B$ で失敗する度にこの無駄な処理が入るため, 最適化ありの場合と比べて桁違いに遅くなってい

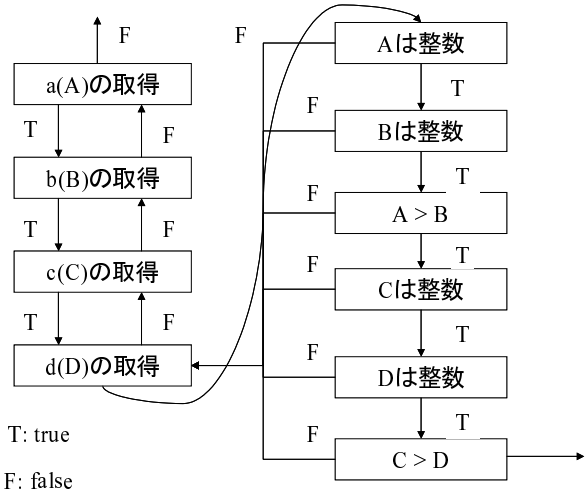

図 5.1: Z0 のバックトラック

ると考えられる. 最適化ありの場合のバックトラックを示す図は, それぞれ図 5-2, 5-3 を参照.

前述の通り, Z0 オプションではこれ以上大きいデータで計測するのは困難なので, 以降は Z1 と Z4 の性能比較を行う.

まず, Z1 と Z4 におけるバックトラックの様子をぞれぞれ図 5.2, 図 5.3 に示す. また, これ以降便宜上 a(A), b(B) を取得するグループをグループ 1, c(C), d(D) を取得するグループをグループ 2 と名付ける.

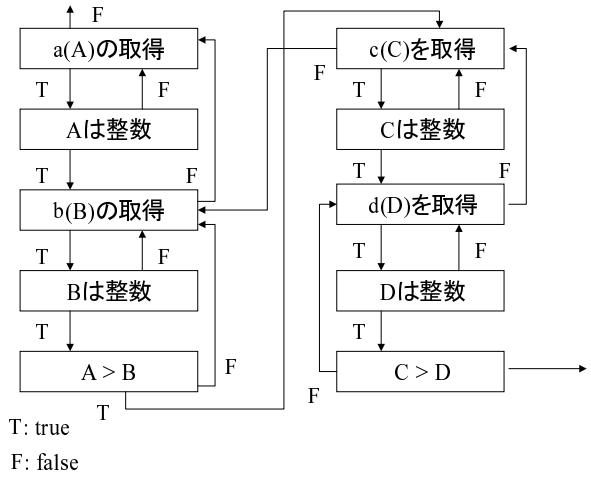

図 5.2: Z1 のバックトラック

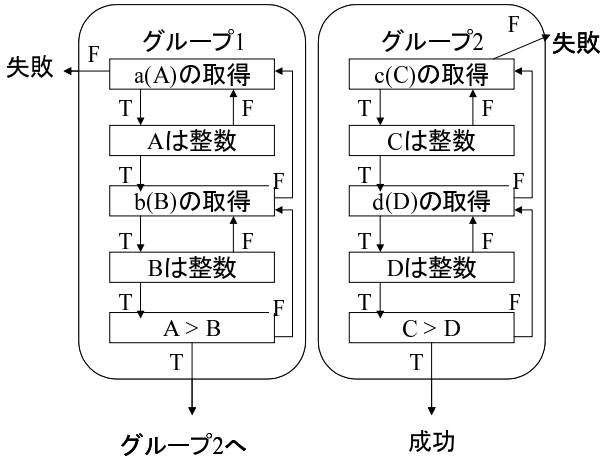

図 5.3: Z4 のバックトラック

Z1 と Z4 の違いは, Z4 ではグループごとに処理が独立しているため, どれか 1 つでもグループの取得に失敗したらその時点でルール適用を終了できる点である. Z1 における終了条件は a(A) の取得に失敗すること, すなわち Z4 で言うとグループ 1 の取得に失敗することである. グループ 2 の取得に失敗することが確定していても, グループ 1 が取得できる限り処理が続く. この無駄な処理が性能の差となっている.

章末の図 5.2 から図 5.5 までにその計測結果の表を挙げる.

以下, 各表ごとに考察する. なおグループが無くなるというのは, グループの取得に成功できるアトムが無くなるという意味である.

表 5.2 と表 5.5 グループ 1 とグループ 2 がほぼ同時に無くなる場合.

ルールが a(A), b(B), c(C), d(D) を 1 つずつ消費するものであり, どちらのグループの igt 命令 (>) も成功する確率は同じなので, これら 4 つのアトムの数が等しい場合グループ 1 とグループ 2 は同じようなタイミングで取得に失敗する.

表の結果を見ると, Z1 の方が微妙に速い場合と, Z4 の方がかなり速い場合とに分類できる. 前者はグループ 1 が先に無くなった場合であり, 後者はグループ 2 が先に無くなった場合である. 前者で Z1 が微妙に速いのは, Z4 ではグループ化をしている関係で spec と proceed という命令がグループの数だけ追加されているため, 同じバックトラックの手順で実行した場合, Z1 に比べて実行する命令の数自体は多くなるからである. しかし, 表を見れば分かる通りこの差は微々たるもので, 逆にグループ 2 が先に無くなると Z1 は Z4 に比べてかなり遅くなる. 全体的に見れば Z4 の方がムラがなく高性能であると言える.

表 5.3 グループ 1 が先に無くなる可能性が高い場合.

グループ 1 で取得できるアトムの数が, グループ 2 の半分しかないのでほとんどの場合グループ 1 が先に無くなるであろう例. 全体的に見て Z1 の方が Z4 より微妙に速い. グループ数によらず, 命令列上で先頭にあるグループが最初に尽きるような場合, Z1 の方が Z4 より速くなることが示された.

しかし, その差もそれ程大きいものではなく, どのグループが最初に尽きるかを考えないで済む分, Z4 の方が汎用性が高いと言える.

また Z1 に着目すると, 表 5.3 よりもデータ量の少ない表 2 より何故か速くなっている. これはグループ 2 が尽きた時にグループ 1 が残っていた場合, グループ 1 でアトムの取得パターンを全て試してから (グループ 1 に失敗してから) でないと終了できないためである. 表 3 ではグループ 2 が尽きた時にグループ 1 が残っていることはあまりないが, 表 2 ではほぼ 5 分の確率でグループ 1 が残っているので, 終了までに無駄な処理が必要な確率が高い.

表 5.4 グループ 2 が先に無くなる可能性が高い場合.

Z4 が効果を存分に発揮できることを想定して試した例. 予想通り, Z1 に比べて圧倒的に Z4 が速くなった.

Z1 では, グループ 2 が取得できなくなっても, グループ 1 でアトムを全パターン取得してからでないと終了できない. この無駄な処理が差となっている.

なおこの例は Z4 の成果を試すのにちょうどいいので, Z0 を計測し比較を試みたが, データ 1 を Z0 オプションで実行したところ 1 時間経っても終わらなかったので断念した.

表 5.2: $N_{ab} = 200$, $N_{cd} = 200$

データ	Z1[ms]	Z4[ms]
1	942	982
2	1392	981
3	2183	771
4	771	791
5	3305	611
6	661	681
7	1322	1022
8	2143	661
9	801	821
10	1332	1372
平均	1485.2	869.3
性能比	1.00	1.71

表 5.3: $N_{ab} = 200$, $N_{cd} = 400$

データ	Z1[ms]	Z4[ms]
1	962	952
2	811	911
3	561	681
4	1271	1272
5	661	671
6	751	781
7	911	942
8	1092	1071
9	882	892
10	661	671
平均	856.3	884.4
性能比	1.00	0.968

表 5.4: $N_{ab} = 400$, $N_{cd} = 200$

データ	Z1[ms]	Z4[ms]
1	123328	1312
2	220186	1432
3	129937	802
4	309214	992
5	218504	882
6	134153	921
7	154011	1122
8	221769	1011
9	160000	1192
10	97601	992
平均	176870	1065.8
性能比	1.00	166

表 5.5: $N_{ab} = 400$, $N_{cd} = 400$

データ	Z1[ms]	Z4[ms]
1	5758	5738
2	6148	7000
3	21331	5878
4	5057	4757
5	6039	5649
6	21321	4456
7	19659	5117
8	31194	5668
9	6909	5898
10	6910	7090
平均	13032.6	5725.1
性能比	1.00	2.28

第6章　まとめと今後の課題

6.1　まとめ

本論文で行った最適化をまとめると.

Z1 オプション 命令列を並び替える. 具体的にはガード命令列の命令を, ヘッド命令列上のなるべく早い位置に移動させる.

Z4 オプション 命令列を処理の独立した単位であるグループごとに区切る.

の 2 つである. どちらも最適化無し (Z0) の場合に比べかなりの成果を挙げることができた.

特に目立ったのは Z0 の場合の効率の悪さである. 5 章の検証では他の 2 つのオプションと比べて 1000 倍程度の時間がかかっていた. あえて Z0 の苦手そうな例を選んだせいもあるが, ここまでの時間短縮ができたことは大きな成果だと思われる.

ルール適用失敗が確定した後にも処理が無駄に続くという問題を Z4 で解決できたので, Z1 と Z4 の間でもプログラムによっては 100 倍以上もの差を出すことに成功した.

しかし, Z4 オプションでもグループ内部ではまだ無駄なバックトラックが行われる場合もある. 今後の課題はこれらを解決していくことである.

6.2　今後の課題

次の 2 つのプログラムの性能を比較してみる.

```
(1) a(X), b(Y), c(Z) :- X > Y, X < Z | ok(X, Y, Z).
(2) b(Y), a(X), c(Z) :- X > Y, X < Z | ok(X, Y, Z).
```

データはファンクタ `a_1, b_1, c_1` を持つアトムをそれぞれ 100 個, 引数は 0 から 999 までの整数である.

次のような結果が得られた.

表 6.1: (1), (2) の実行時間

データ	(1)[ms]	(2)[ms]
1	952	351
2	1933	731
3	802	351
4	1803	671
5	3611	1152
6	1622	811
7	2163	390
8	1392	591
9	4466	1382
10	3415	741
平均	2215.9	717.1
性能比	1.00	3.09

a(X) と b(Y) の順番を入れ替えただけで 3 倍程度効率が変わっている. このルールは a(X), b(Y), c(C) がお互いに igt 命令 (>) と ilt 命令 (<) で関連付けられるので全て 1 つのグループにまとめられる. その内部はおおよそ図 6.1 のようになっている.

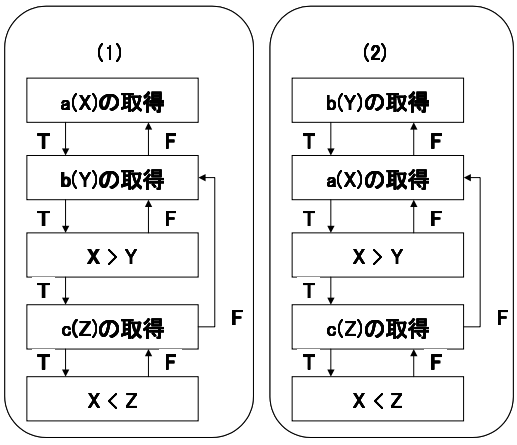

図 6.1: (1), (2) のバックトラック

(1) がなぜ効率が悪いかというと, $X < Z$ で失敗し c(Z) を変更し尽くした際, 関係の無い b(Y) の取得にバックトラックしているためである. これは失敗した命令に関係無く, 1 つ上の findatom 命令へバックトラックしていくため起こる. (2) のようにこの性質を利用してうまく並び替えることで効率を上げることもできるが, 一般的には並び替えだけでは解決しそうもない.

例:`a(A), b(B), c(C), d(D) :- A > B, A > C, A > D | ok.`

この場合は a(A) と b(B) を入れ替えることである程度効率は上がるが, $A > D$ で失敗すると c(C) の取得が無駄に行われる.

これらを踏まえると, 失敗した命令に応じて最適なバックトラックをできるようにしたいところだが, 現在グループ内では findatom 命令で再帰呼び出しが行われている. このため命令に失敗しても 1 番内側の再帰呼び出し元に戻るだけであり, 特定の findatom 命令までバックトラックするようなことはできない.

そこで当面の目標として, 再帰呼び出しを行わずに命令列を解釈できるインタプリタの設計を予定している. 概要は次のようになっている.

- インタプリタはさまざまな情報を積んだスタックを持ち, 命令列実行中に失敗したら, このスタックから取り出した情報を元にバックトラックをする.
- グループの切れ目には終了を示す flag を積んでおく. スタックから pop してバックトラックを試みている際にこれを pop したらその場でルール適用失敗とする.
- findatom 命令でアトムを取得したら, その findatom 命令の命令列上の位置 (プログラムカウンタ) やイテレータ等の情報を積む. この情報を pop したら, 該当する findatom 命令実行し, 先程取得したアトムと別アトムを取得し直す.
- スタックに積む情報を工夫することで, やり直したい findaotm 命令が出てくるまで pop といった処理を可能にしたい. そうすることで, 失敗元の命令に応じて思想的なバックトラックができそうである.

今後, 膜のあるプログラムなども考慮しつつこの案を煮詰めていきたい.

謝辞

本研究を進めるにあたり, ご指導して頂いた上田和紀教授に深く感謝致します.
　また, 様々な助言をして頂いた上田研究室言語班の方々に深く感謝致します.
　特に加藤紀夫氏と水野謙氏には設計実装において大変お世話になりました. ここに感謝の意を表します.

参考文献

[1] 上田和紀, 加藤紀夫. 言語モデルLMNtal. コンピュータソフトウェア, Vol. 21, No. 2, pp. 44–60, 2004.

[2] David Jeffery Christian Holzbaur, María García de la Banda and Peter J. Stuckey. Optimizing Compilation of Constraint Handling Rules. *ICLP 2001, LNCS 2237*, pp. 74–89, 2001.

[3] 水野謙. LMNtal 処理系における最適化器の設計と実装. 早稲田大学卒業論文, 2003.

[4] 原耕司. LMNtal 処理系のモジュール機能と他言語接続機能の設計と実装. 早稲田大学卒業論文, 2003.

付 録A　GuardOptimizerクラス

```
package compile;

import java.util.*;
import runtime.Instruction;
//import runtime.InstructionList;


/**
 * @author sakurai
 *
 * ガード関係の最適化を行うメソッドを持つクラス.
 * 現在はガード命令列の命令を可能な限り
 * 前に移動させるだけ.
 */
public class GuardOptimizer {
public static void guardMove(List head){
int guardinstsstart = 0;
for(int hid=1; hid<head.size()-1; hid++){
Instruction insth = (Instruction) head.get(hid);
switch(insth.getKind()){
//ガード命令列の先頭にありそうな命令
case Instruction.ALLOCATOM:
case Instruction.DEREFATOM:
case Instruction.GETLINK:
case Instruction.NATOMS:
case Instruction.NMEMS:
case Instruction.NORULES:
guardinstsstart = hid;
hid = head.size();
break;
default: break;
}
}
if(guardinstsstart != 0) //ガード命令列が空の場合は実行しない.
for(int hid=guardinstsstart; hid<head.size()-1; hid++){
Instruction insth = (Instruction)head.get(hid);
ArrayList list = insth.getVarArgs();
boolean moveOk = true;
```

```
//ALLOCATOM 命令は先頭の SPEC の直後に移動
if(insth.getKind() == Instruction.ALLOCATOM){
head.remove(hid);
head.add(1, insth);
}
else
for(int hid2=hid-1; hid2>0; hid2--){
Instruction insth2 = (Instruction)head.get(hid2);
ArrayList list2 = insth2.getVarArgs();
for(int i=0; i<list.size(); i++){
if((insth2.getOutputType() != -1
   && list.get(i).equals(insth2.getArg1()))
   || !(orderConstraintCheck(insth, insth2, list.get(i)))){
    moveOk = false;
    break;
   }
}
if(moveOk){
head.remove(hid2+1); //移動対象の命令 insth の位置は hid2+1
head.add(hid2, insth); //hid2 に移動
}
else break;
}
}
}

//inst1 の 1 つ上の命令 inst2 が，inst1 と同じ変数番号 var を使用している場合，
//inst1 が inst2 より前に出られるかをチェックする．
//出られる場合 true，出られない場合 false を返す．
//ISINT，ISFLOAT は同じ変数番号を使う命令の中では優先順位が高い．
//(IGT などは ISINT より後になる)
private static boolean orderConstraintCheck(Instruction inst1,
                                    Instruction inst2, Object var){
ArrayList list = inst2.getVarArgs();
if(list.contains(var)){
switch(inst2.getKind()){
case Instruction.ISINT:
case Instruction.ISFLOAT:
return false;
default: return true;
}
}
else return true;
}

}
```

付 録B　Groupingクラス

```
package compile;

import java.util.*;
import runtime.Instruction;
import runtime.InstructionList;
/**
 * @author sakurai
 *
 * ヘッド命令列をグループごとに分ける
 * group[ findatom
 *        deref
 *        func
 *        insint ]
 * group []
 * のような形になる
 */
public class Grouping {
HashMap var2DefInst;          //変数番号→変数番号を定義した命令
HashMap Inst2GroupId;         //命令→グループ識別番号

public Grouping(List head){
var2DefInst = new HashMap();
Inst2GroupId = new HashMap();

if(((Instruction)head.get(0)).getKind() != Instruction.SPEC) return;
if(((Instruction)head.get(head.size()-1)).getKind() != Instruction.JUMP) return;

//グループ番号を割り振る
//spec, jump 以外の全ての命令に行番号をグループ番号として割り振る
for(int hid=1; hid<head.size()-1; hid++){
Inst2GroupId.put(head.get(hid), new Integer(hid));
}

//変数番号→命令番号にマップを張る
for(int hid=1; hid<head.size()-1; hid++){
Instruction insth = (Instruction)head.get(hid);
if (insth.getOutputType() != -1) {
var2DefInst.put(insth.getArg1(), insth);
```

```
}
}
createGroup(head);
}

//グループ分け
//命令番号→グループ番号とし，同じグループに入る命令は
//同じグループ番号へマップが張られる
private void createGroup(List head){
for(int hid=1; hid<head.size()-1; hid++){
Instruction insth = (Instruction)head.get(hid);
Object group = null;
Object changegroup = null;
ArrayList list = insth.getVarArgs();
for (int i = 0; i < list.size(); i++) {
if (list.get(i).equals(new Integer(0))) continue;
group = Inst2GroupId.get(var2DefInst.get(list.get(i)));
changegroup = Inst2GroupId.get(insth);
changeMap(changegroup, group);
}
}
//マップ生成終了

//GROUP 生成
for(int hid=1; hid<head.size()-1; hid++){
Instruction insth = (Instruction)head.get(hid);
Object group = Inst2GroupId.get(insth);
InstructionList subinsts = new InstructionList();
subinsts.add(new Instruction(Instruction.SPEC,0,0));
for(int hid2=hid; hid2<head.size()-1; hid2++){
Instruction insth2 = (Instruction)head.get(hid2);
if(group.equals(Inst2GroupId.get(insth2))){
subinsts.add(insth2);
head.remove(hid2);
hid2 -= 1;
}
}
subinsts.add(new Instruction(Instruction.PROCEED));
head.add(hid, new Instruction(Instruction.GROUP, subinsts));
}

guardMoveOptimize(head);
//mapView();
}
```

//Inst2GroupId の内，値 group1 をもつ全てのキーに対し，値 group2 へマップを張り替える．

```
public void changeMap(Object group1, Object group2){
Iterator it = Inst2GroupId.keySet().iterator();
while (it.hasNext()) {
Object key = it.next();
if (group1.equals(Inst2GroupId.get(key))) {
Inst2GroupId.put(key, group2);
}
}
}

//GROUP 内の並び替えによる最適化
private void guardMoveOptimize(List list){
//ガード命令の移動
for(int i=0; i<list.size(); i++){
Instruction inst = (Instruction)list.get(i);
if(inst.getKind() == Instruction.GROUP){
InstructionList subinsts = (InstructionList)inst.getArg1();
GuardOptimizer.guardMove((List)subinsts.insts);
}
}
}

//生成されたマップの確認 デバッグ用
public void mapView(){
Set set1 = var2DefInst.entrySet();
Set set2 = Inst2GroupId.entrySet();

Iterator it1 = set1.iterator();
Iterator it2 = set2.iterator();

System.out.println("var2DefInst :- ");
while(it1.hasNext()){
Map.Entry mapentry = (Map.Entry)it1.next();
System.out.println(mapentry.getKey() + "/" + mapentry.getValue());
}

System.out.println("Inst2GroupId :- ");
while(it2.hasNext()){
Map.Entry mapentry = (Map.Entry)it2.next();
System.out.println(mapentry.getKey() + "/" + mapentry.getValue());
}
}
}
```