A-014

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

Optimization of Instruction Sequences by Rearrangement and Grouping in LMNtal Compiler

櫻井 健* Ken SAKURAI　加藤 紀夫‡ Norio KATO　水野 謙* Ken MIZUNO　上田 和紀† Kazunori UEDA

1 はじめに

1.1 言語モデル LMNtal

LMNtal (elemental と発音) [1] は階層型グラフ構造の書き換えに基づく, 多重集合書き換え言語の一種である. LMNtal は柔軟なデータ表現により様々な計算モデルを統合し, 大規模計算から極小計算まで様々な環境を統合して扱うことのでき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. 現在 Java で実装された処理系が稼動している.§

1.2 研究の目的と意義

LMNtal では複雑なデータ構造 (階層型や循環型) の使用を可能にして様々な計算をモデル化できるように設計されているが, 現在の処理系では実行効率の面で十分な最適化は行われていない. よって本研究では, 実行効率の改善を図るために従来の最適化器に新しい機能を設計, 実装した.

LMNtal は階層型グラフ構造のデータに対してマッチングを行い, マッチした部分の構造をルールによって書き換えることで処理が進行する. 本研究で設計, 実装した最適化機能は, LMNtal コンパイラが生成する中間命令列に対し**並び替え**と**グループ化**という 2 つの処理を施すものであり, どちらもルール適用時のマッチング処理を最適化する. 実際にこれらの最適化を施すことで計算量の改善を実現できた.

2 LMNtal 言語の概要

2.1 LMNtal のプログラム

LMNtal のプログラムの構成要素は引数に 0 個以上のリンクを持つ**アトム**, アトムを繋ぐ**リンク**, **プロセス**を階層化・局所化する**膜** (中括弧で表記), 書き換え規則である**ルール**から成る. リンクの名前は大文字から始まり, アトムの名前はそれ以外で始まる. また, プロセス P の定義は

$$P ::= 0 \mid p(X_1, ..., X_m) \mid P, P \mid \{P\} \mid \text{ルール}$$

となっている.

ルールの構造

ヘッド :- ガード | ボディ

LMNtal におけるプログラムの実行は, ルールによって, ヘッド部にマッチするプロセスのうち, ガード部の制約を満たすプロセスをボディ部のプロセスに書き換えることで処理が進行する. なおルールは, そのルールのある膜内のプロセスにしか適用されないようになっている. すべての膜内で適用可能なルールが無くなった時点でプログラムの実行が終了となる.

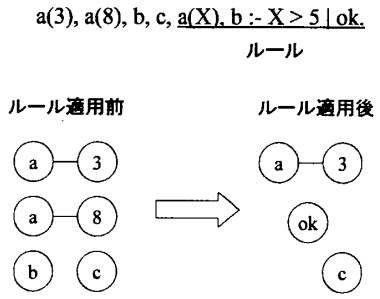

図 1: プログラム例

図 1 のルールは, データ内でリンクを 1 個持つアトム a とリンクを持たないアトム b を探し, a のリンク先のアトムが 5 より大きい整数であるとき, それらの a と b をアトム ok に書き換える. なお LMNtal では, a(X), b(X) を a(b) と略記することを認めている. つまり プログラム中の a(3) は 名前が a でリンクを 1 個持つアトムと, 名前が 3 でリンクを 1 個持つアトムがそれぞれの持つリンクによって繋がっていることを表している.

2.2 現在の処理系

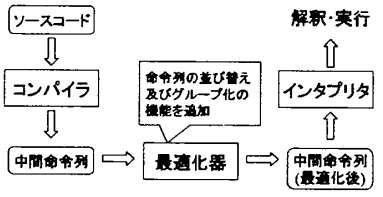

図 2: LMNtal 処理系

現在の LMNtal 処理系を説明する. ユーザの書いたソースコードは LMNtal コンパイラにより中間命令列に変換される. 中間命令列は最適化器 [2] により最適化された後, インタプリタで解釈, 実行される. 図 2 はこれらの処理を示すものである.

* 早稲田大学大学院 理工学研究科 情報・ネットワーク専攻
† 早稲田大学理工学部 コンピュータ・ネットワーク工学科
‡ 現所属は産業技術総合研究所システム検証研究センター
§ http://www.ueda.info.waseda.ac.jp/lmntal/

このうち，本研究では最適化器に命令列の並び替えとグループ化という 2 つの機能を設計，実装した．これらの最適化については 3 章で述べる．

2.3 ルールの構造と中間命令列

ルール適用は処理系内部では次のように行われている．

ヘッド部 データ内でヘッドに記述されたプロセスにマッチする部分を探す．マッチしたらガード部へ．

ガード部 ガードの制約を満たすかを確認する．制約を満たす場合ボディ部へ．満たさない場合はヘッド部にバックトラック．

ボディ部 ヘッドに記述されたプロセスの除去およびボディに記述されたプロセスの生成．

ヘッド，ガード，ボディはそれぞれ別のコンパイラに通すため，コンパイル後の中間命令列も 3 つの部分に分かれている．なお従来の最適化器 [2] では，主にボディ部の最適化を行っているが，本研究の最適化対象はヘッド部とガード部とする．

例として，`a(X), b :- X>5 | ok.` をコンパイルした中間命令列のうち，ヘッド命令列の直後にガード命令列を展開した部分を挙げる．

中間命令列

```
findatom  [1, 0, a_1]
findatom  [2, 0, b_0]
allocatom [3, 5_1]
derefatom [4, 1, 0]
isint     [4]
igt       [4, 3]
```

図中の各命令の意味は次のようになっている．

findatom [-dstatom, srcmem, funcref]
: 膜 srcmem にあって**ファンクタ** (名前と引数の個数)funcref を持つアトムへの参照を次々に dstatom に代入する．

allocatom [-dstatom, funref]
: ファンクタ funcref を持つアトムを作成し，参照を dstatom に代入する．

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 に代入する。

isint [atom]
: atom が整数アトムであることを確認する。

igt [intatom1, intatom2]
: intatom1 > intatom2 を確認する．

allocatom 命令以下はガード命令列である．命令列中では，アトムや膜は変数番号で管理される．1 行目の findatom 命令を例に説明すると，リンクを 1 個持つアトム a を番号 0 の膜から取得し，その参照として変数番号 1 を定義せよという意味である．findatom 命令では第 1 引数の変数番号を定義した後，その下の命令列を実行する．もしどこかで失敗したら，再びその findatom に戻り，別のアトムを取得して再度以下の命令列の実行を試みる．取得できるアトムが無くなったら findatom は失敗となる．つまり命令列上で複数の findatom がある場合，失敗時には 1 つ前の findatom に順次バックトラックすることで総当りのパターンマッチングを実現している．しかし，総当りであるということは無駄なマッチングも含まれるので，この無駄をいかに減らすかに着目した．

3 並び替えとグループ化

3.1 命令列の並び替え

2.3 節で挙げた命令列において，X > 5 を検査する igt 命令で失敗すると 1 つ前の findatom にバックトラックするが，この findatom は b を取得するものであり，X の値を変更するものではない．X の値を変更するにはリンクを 1 個持つ a(以後 a(X) と呼ぶ) を取得する必要があるが，a(X) を取得する findatom にバックトラックするには b を取得する findatom で失敗しなければならない．

この問題はガード命令列がすべてヘッド命令列の後で実行されていることに起因する．実際には X > 5 の制約のチェックは a(X) を取得した直後で行う方が自然である．そこでガード命令列のうち，早い段階で実行できる命令を移動させて命令列を並び替える機能を最適化器に設計，実装した．並び替えの手法として，変数番号がすべて単一代入であることに着目した．単一代入であるということは，各命令はその命令で使用している変数番号が定義された後ならば，どこで実行しても結果は同じであると言える．よって並び替える命令 (ガード命令列の命令) を，その命令が使用している変数番号を定義している命令の直後まで移動させた．

例えば derefatom [4, 1, 0] は変数番号 1 と 0 を用いて新たに変数番号 4 を定義する命令なので，変数番号 1 と 0 が定義された直後まで移動可能である．このうち変数番号 0 はルールのある膜の番号なので，処理系の仕様により始めから定義されている．よって変数番号 1 が定義される findatom [1, 0, a_1] の直後まで移動させることが可能である．ただし，同じ変数番号を使用する命令 (isint と igt など) は並び替え前の前後関係が入れ替わらないようにした．

実際に並び替えを行った命令列は次のようになる．

並び替え後の中間命令列

```
allocatom [3, 5_1]
findatom  [1, 0, a_1]
derefatom [4, 1, 0]
isint     [4]
igt       [4, 3]
findatom  [2, 0, b_0]
```

この命令列なら X > 5 で失敗したとき，a(X) の取得へバックトラックできる．

3.2 命令列のグループ化

並び替え前および並び替え後の命令列では，どちらも b の取得 (2 つ目の findatom) に失敗したとき，a(X) の取得 (1 つ目の findatom) にバックトラックする．しかし，データ領域に b

が存在しない以上, a(X) を取得し直しても b の取得に成功することはない. そして a(X) の取得に失敗して初めてルール適用失敗が確定する. つまり, a(X) の数だけ取得できるはずのない b の取得を試みてしまう. この問題を解決するためにグループ化という概念を導入した.

ここでは a(X) と b を取得する処理はそれぞれ独立したものであり, 片方で取得したアトムを選び直すことでもう片方に影響を与えることはない. 逆に a(X), b(Y) :- X>Y | ok. のようなルールの場合, a(X) の取得により, マッチする b(Y) が変わってくる. このような場合は 2 つのアトムの取得は独立した処理ではないと言える. また, 同じファンクタを持つアトムを取得する命令同士も独立した処理ではない. なぜなら, a(X), a(Y) :- X>5, Y>10 | ok のようなルールがあったとき, a(X) を取り直すことで a(Y) の取得に成功する場合もあるからである.

このような独立した処理ごとに命令列をグループ分けし, どれか 1 つのグループ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. つまりグループ化とは, 全体で 1 つの長いマッチング命令列を, いくつかのお互いに独立した短いマッチング命令列に区切ることである.

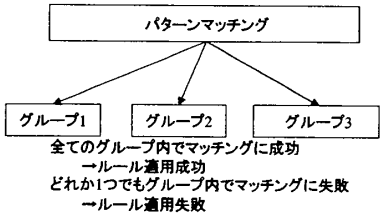

図 3: グループ化

実際に命令列をグループに分割する手法を述べる. 独立した処理ということは, 異なるグループ間で同じ変数番号を使用することはない.* よって変数番号で関わりを持つ命令同士を同じグループにすることを考える. まず, すべての命令に異なる ID(この場合行番号) を与える. そして, ある変数番号を使う命令があった場合, その変数番号を定義した命令および同じ変数番号を使用している命令に同じ ID を与える. この操作を命令列全体に行い, 最終的に同じ ID になった命令を 1 つのグループにまとめることで命令列がグループ化される. これにより, 例えばガード部に X > Y があるとき, 変数番号 X, Y に関わる命令はすべて同じグループとなる.

グループ化を行うにあたり, 中間命令 group を追加した.

group [insts]

命令列 insts を実行する. insts 内で成功したなら次の命令を実行し, 失敗ならルール適用失敗を確定する.

ユーザが次のようなルールを書くと,

a(X), b :- X>5 | ok.

処理系内部で次のようなルールとして解釈されるようになる.

group[a(X), X>5], group[b] :- ok.*

これにより, b の取得に失敗したとき, つまり group[b] で失敗したとき a(X) が残っていてもそれ以上のマッチングを行わず, ルール適用失敗を確定する.

4 性能評価

4.1 測定

測定を行った環境は, CPU が PentiumIII 845MHz, メモリが 256MB, OS は WindowsXP Home Edition である.

また, 最適化オプションは Z0 が最適化なし, Z1 が命令列の並び替え, Z3 は Z1+命令列のグループ化である.

測定用プログラム

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

A, B, C, D はそれぞれ 0〜99 のランダムな整数. a(A), b(B) の数を N_{ab}, c(C), d(D) の数を N_{cd} として変動させた.

このプログラムは非常に単純なものであまり実用的であるとは言えないが, バックトラックの手順の分かり易くオーダー計算が容易なこと, 並び替えおよびグループ化による最適化効果がはっきり表れることが容易に想定できたことから測定用プログラムとして採用した. もっと複雑かつ実用的な例による性能評価は今後の課題とする.

計測結果は表の通り. プログラムの実行時間を, 1 つのデータに対し各最適化レベルで計測した. なお, それぞれの時間は異なる 10 個のデータで計測した平均である. 4 時間を越えても終わらなかった項目は測定不能とした. また N_{ab} = 200, N_{cd} = 200 における Z0 オプションの計測時間はデータ 1 個でしか測定していないので参考記録とし, 性能比は Z1 と Z3 のみで比較する.

4.2 各最適化レベルの計算量

最適化後の命令列は処理系内部では次のように解釈される.

```
Z1  a(A),b(B),A>B, c(C),d(D),C>D :- ok(A,B,C,D).
Z3  group[a(A),b(B),A>B],
    group[c(C),d(D),C>D] :- ok(A,B,C,D).
```

このうち Z3 オプション時のルール適用の流れを図 4 に表す.

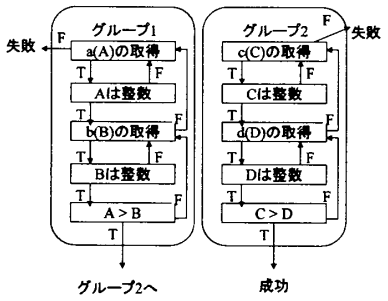

図 4: グループ化時の処理の流れ

Z1 の場合は, 図中の c(C) の取得に失敗したとき, b(B) の取得にバックトラックする. なお, これ以降便宜上 a(A), b(B), A>B の部分のマッチングをグループ 1, c(C), d(D), C>D の部分のマッチングをグループ 2 と呼ぶこととする.

* 変数番号 0 はルールのある膜の番号なので例外.
* このルールは最適化後の中間命令列を表現したものであり, 実際にユーザがこのようなルールを記述できるわけではない.

最適化レベル	N_{ab} = 100, N_{cd} = 100		N_{ab} = 200, N_{cd} = 200		N_{ab} = 200, N_{cd} = 400		N_{ab} = 400, N_{cd} = 200	
	時間 [ms]	性能比	時間 [ms]	性能比	時間 [ms]	性能比	時間 [ms]	性能比
Z0	401196	1.00	3405838	-	計測不能	計測不能	計測不能	計測不能
Z1	444.7	902	1485.2	1.00	856.3	1.00	176870	1.00
Z3	280.3	1430	869.3	1.71	884.4	0.968	1065.8	166

各最適化レベルに共通して言えるのは，途中で失敗すると1つ前のアトムから取得し直すことである．ただし，Z3 では group 内で先頭のアトムの取得に失敗するとその時点でルール適用失敗となるのに対し，他の最適化レベルでは，先頭のアトムである a(A) の取得に失敗したときにしかルール適用失敗を確定できない．これを踏まえると，あるタイミングで a(A), b(B) が N_{ab}' 個，c(C), d(D) が N_{cd}' 個残っているとき，ルール適用を1回成功させるのに必要なマッチングの回数 (findatom の実行回数) は最悪の場合次のようになる．

Z0　$O(N_{ab}'^2 \times N_{cd}'^2 + N_{cd}'^2)$
Z1, Z3　$O(N_{ab}'^2 + N_{cd}'^2)$

このオーダーに関して説明すると，まず Z1, Z3 の場合 A > B で成功するには最悪の場合，a(A), b(B) の全組合わせを試す必要がある．つまりマッチングは $N_{ab}'^2$ 回．同じく C > D で成功するには，$N_{cd}'^2$ 回なので合わせると上記のオーダーとなる．一方，Z0 の場合 b(B) の取得をやり直すには c(C) の取得に失敗しなければならない．c(C) の取得に失敗するのは c(C), d(D) の全組合わせを試して失敗した後なので，A > B で成功するのに必要なマッチングは，最悪の場合 $N_{ab}'^2 * N_{cd}'^2$ 回となる．よって，C > D に成功するのに必要なマッチング回数 ($N_{cd}'^2$ 回) と合わせて上記のオーダーとなる．

このように，Z0 は Z1, Z3 とオーダーが異なるため極端に遅くなっている．N_{ab} = 100, N_{cd} = 100 の場合で Z1 の 902 倍，Z3 の 1430 倍の時間がかかった．

なお，Z1, Z3 ではルール適用成功時のオーダーは等しいが，失敗時のバックトラックに違いがある．グループ2でマッチしなくなってルール適用失敗する場合，失敗を確定するのに必要なマッチングの回数は次の通り．

Z0, Z1　$O(N_{ab}'^2 \times N_{cd}'^2)$
Z3　$O(N_{cd}'^2)$

Z3 の場合，c(C), d(D) の全組合わせを試して終了なのでこのオーダーとなる．Z0, Z1 のオーダーは C > D で成功する c(C), d(D) が無いのに，残っている a(A), b(B) のすべてが A > B を満たしているという最悪の場合である．この場合 a(A), b(B) の組合わせをすべて試しつつ，その度に C > D を満たす c(C), d(D) が無いことを確認するため非常に無駄が多い．

グループ1より先にグループ2でマッチしなくなった場合，Z1 だとグループ1でマッチするアトムが残っている限り，ルール適用は終了しない．Z3 の場合グループ2でマッチしなくなった時点でルール適用失敗となる．よってこのような状況に陥り易い N_{ab} = 400, N_{cd} = 200 のときに Z1, Z3 の差が顕著に表れており，Z3 は Z1 の 166 倍高速となった．

逆にグループ1が先にマッチしなくなる場合は，Z1, Z3 の間に差は出ない．なぜならどちらもルール適用失敗は a(A) の取得の失敗によって決まるからである．N_{ab} = 200, N_{cd} = 400 の測定結果に着目すると，むしろ中間命令が増えている分 Z3 の方が遅くなっている．

4.3 考察

測定結果より，グループ化による最適化 (Z3 オプション) を行う場合，先頭となるグループが最初にマッチングに失敗するときには，最適化効果は得られないということが分かった．しかし，先頭以外のグループのマッチングに失敗する場合には大きな効果を得ることができた．また，並び替えによる最適化 (Z1) は，本研究では並び替えで動く命令をガード命令列の命令に限っているので，ガード部の無い命令では最適化効果は得られない．

結論としては，今回の測定用プログラムのようにガード部があり，かつグループを2つ以上に分けられるような場合には，非常に大きな最適化効果を得ることができるといえる．

5 まとめと今後の課題

命令列に対して並び替えとグループ化という2種類の最適化を用いることで，測定用プログラムにおいては計算量を大幅に削減することができた．今後の課題としては，より多くのプログラムに対して最適化効果を得られるようにすことがあげられる．例えば次のルールがあったとする．

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

これをグループ化まで最適化すると，処理系内部では次のように解釈される．

```
group[a(A), b(B), A>B, c(C), A>C] :- ok.
```

このルールの場合，グループは1つである．現在の処理系ではグループ内のどこかで失敗すると，1つ前のアトムから取得し直す．例えば A>C で失敗すると c(C) → b(B) → a(A) の順でバックトラックする．しかし，理想的には c(C) → a(A) の順でバックトラックさせたい．今後はこのようにさらに intelligent なバックトラックを行えるように，処理系を改良したい．また，多重集合のマッチングに関する更なる最適化 [3] も考えたい．

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 (2004), pp. 44-60.

[2] 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.

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

[4] 櫻井健, 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. 早稲田大学卒業論文, 2004.