軽量な LMNtal 実行時処理系 SLIM の設計と実装

石川　力 *　　堀　泰祐 *　　村山　敬 †　　岡部　亮 †　　上田　和紀 ‡

*早稲田大学理工学部コンピュータネットワーク工学科

†早稲田大学理工学研究科 情報・ネットワーク専攻　　‡早稲田大学理工学術院

LMNtal[1] は階層型グラフ書換えに基づく言語であり, 並行計算と多重集合書き換えを含む多様な計算を統一的に扱う実用言語を目指している. 現在 Java 言語で実装されたコンパイラ及び実行時処理系が稼働しているが, 拡張性を重視した設計であるため, メモリ使用量や実行速度に関しては改善の余地があった. そこで, 拡張機能を制限したシンプルで軽量な処理系を目指し, C 言語による実行時処理系 SLIM (Slim Lmntal IMplementation) の設計と実装を行った. その結果従来の実装と比較して, 簡単な例においてはメモリ使用量は 6 分の 1, 処理時間は 50 分の 1 という結果が得られた. 本論文では, SLIM の設計と実装及び, その評価について述べる.

1　LMNtal

LMNtal が扱う階層グラフは, アトムを膜とリンクの二つの手段で構造化したものである. 各アトムは名前と引数の数で決まる種類 (ファンクタ) と, 引数として順序のある 0 本以上のリンクをもち, 各リンクはアトムどうしを一対一で（無方向的に）つないでグラフ構造を構成する. 膜は階層構造の表現手段であり, 0 個以上のアトムや膜を囲んでそれらの多重集合を構成する. ルールはアトム, リンク, 膜で表現した階層グラフの書換え規則で, 接続構造と階層構造の動的再構成を表現する. ルールは膜の中に記述することもでき, それによってルールの適用範囲を限定することができる.

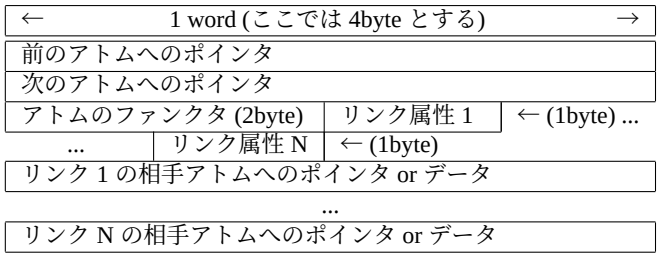
実行の際には, まず LMNtal コンパイラにより, グラフマッチングと書き換えのための小粒度の操作の列 (中間命令列) に変換される. その後実行時処理系によってこれを解釈実行するか, Java 言語のソースコードへ変換後, コンパイルしてから実行する.

従来の Java 言語による実行時処理系の実装は, 非同期実行, 実行中プログラムのグラフによる可視化, Java コードの埋め込みといった豊富な拡張機能を持っている. それに対し, LMNtal を, 組み込みや検証用途などのメモリ量や実行速度が重要になる問題にも利用するため, 言語として必要なものに機能を絞った, 省メモリ, 高速動作を意識した実行時処理系 SLIM を設計し, 実装を C 言語にて行った.

2　SLIM の設計

2.1　データ構造

SLIM において, アトムのデータ構造は次のように設計した.



前後のアトムとは, 同一ファンクタのアトムをリスト状に管理するもので, このリストをアトムリストと呼ぶ. これはグラフマッチングにおいて条件にあう始点を探す際に利用される. リンク属性は各リンクの情報で, 基本的には相手の何番目に接続されているかという情報であるが, この値が負である場合, リンクのポインタ部にデータが埋め込まれていることを示している. ポインタ部にデータを埋め込むことで, プログラム実行時に出現する整数等のデータ 1 つにつきアトム 1 つ分のメモリを節約できる.

膜をまたぐマッチングの際には, プロキシアトムと呼ばれる, 膜の境界を示す特殊なアトムを含むグラフへのマッチングによって階層構造の把握を行うため, 各アトムは自分がどの膜に所属しているかを示す情報を省略できる.

以上の設計より, 1 ワードが Wbyte の環境において, N 本のリンクを持つアトムのサイズは $2+\lceil(2+N)/W\rceil+N$ ワードとなる.

膜は, 親, 前の兄弟, 次の兄弟, 子の代表へのポインタを持つことで膜の階層構造を, その他ファンクタからアトムリストへのハッシュマップなどにより, ルールやアトムの階層構造を構成している.

2.2　高速化

今回の高速化は, 中間命令列を変更せずに可能な範囲で行った. SLIM の実装では, アトムの N 番リンクのポ

Design and implementation of SLIM : a lightweight runtime of LMNtal

*School of Science and Engineering, Waseda University

†Graduate School of Science and Engineering, Waseda University

‡Faculty of Science and Engineering, Waseda University

インタ部分を取得する, といった基本的な操作にマクロを利用し, C 言語コンパイラの最適化がかかりやすくなることを期待する方針をとっている.

中間命令列にコンパイルする性質上, この命令列をほぼ機械的に C 言語のコードに変換することも可能であり, その場合はルール全体にも C 言語コンパイラによる最適化が期待できる. 予備実験において機械的な変換を行ったところ解釈実行の 2 倍以上の高速化が実現した. また, 中間命令列においてアトムなどを表す変数に型をつけることで更なる高速化が望める.

2.3 マッチングの計算量

Java 版の処理系では, 膜がもつ全てのルールの実行を逐一試みる膜主導テストと, 書き換えられる可能性のあるアトムを出発点として実行を試みるアトム主導テストを併用していた. アトム主導テストを用いることで計算量が改善されることが多いが, その計算量は予測不可能なことが多い. そこで, SLIM では膜主導テストのみを採用し, 膜主導テストの計算量を改善することと, 計算量の予測を容易にすることを試みている.

具体的にはアトムについての繰返し命令に対し, アトムリストに履歴管理用アトムを挿入し, 探索作業の履歴を実装した. 複数のルールが 1 つのアトムリストを探索することがあるが, 別の履歴管理用アトムが挿入されるので, 並行する探索にも対応している. 現状の実装では計算量の改善は限定的だが, 将来的にはその他の計算量が悪化する要因を排除することを検討している.

3 性能評価

3.1 アトムの生成削除

```
del(gen(10000)).
del(N), n(M,N) :- del(M).
gen(N,R) :- N=:=0 | R=[].
gen(N,R) :- N=\=0,M=N-1 | n(gen(M),R).
```

これは 10000 個のアトムを鎖状に生成した後, それらを削除する LMNtal プログラムの例である.

この例題において生成するアトムの個数を変化させて性能計測をしたところ, 次のグラフの通り, SLIM のピークメモリ使用量は Java 版実行時処理系と比べて 6 分の 1 程度となった.

処理時間についてはグラフは省略するが, Java 版実行時処理系ではアトム 400000 個で 33 秒かかるのに対し SLIM では 0.6 秒であった. これはおよそ 50 分の 1 程度であり, メモリ操作を非常に高速に行えていることがわかる.

3.2 ラムダ計算

応用例として, LMNtal にエンコードされたラムダ計算 [1] において, チャーチ数の演算 3^N を行った.

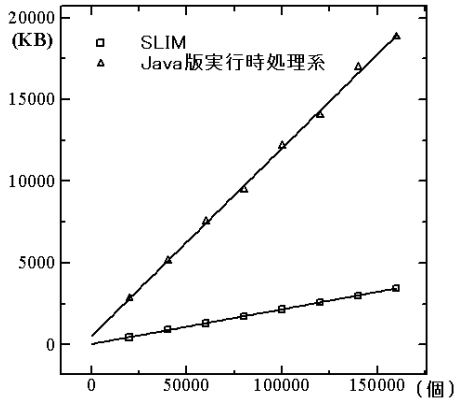


図 1: アトムの生成削除でのメモリ使用量

N を変化させて計測した処理時間のグラフは次のようになった. 指数オーダーの演算となっているが, 実験を行った範囲では SLIM が 100 倍程度高速である.

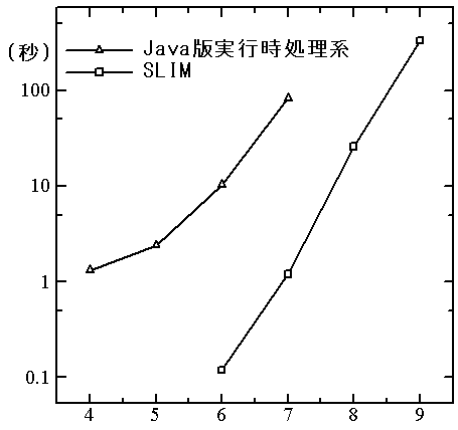


図 2: ラムダ計算での計算時間

4 まとめ

SLIM はピークメモリ使用量, 実行速度ともに大きく性能向上しており, 軽量な LMNtal 実行時処理系としての有用性を確認できた.

一方, 一部の問題に対しては Java 版実行時処理系と比較して計算量が悪化する場合があり, 探索のアルゴリズムの修正が課題である.

参考文献

[1] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀 : “階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal” コンピュータソフトウェア, Vol. 25, No. 1, pp.124–150, 2008.