素性構造処理言語 LiLFeS の最適化技術

吉田 稔　　牧野 貴樹　　鳥澤 健太郎　　辻井 潤一

東京大学理学部情報科学科

{mino,mak,torisawa,tsujii}@is.s.u-tokyo.ac.jp

1. はじめに

本論文では、言語 LiLFeS のコンパイラの最適化技術について述べる。LiLFeS は、実用的な自然言語処理のための型付き素性構造 (Typed Feature Structures[7]、以下 TFS) 処理言語であり、その処理系は、現在我々が開発している HPSG システム[3][4][5]の核となっている。TFS は、自然言語処理システムの構築に有用なデータ構造であり、HPSG パーザをはじめとする様々な応用が考えられる。

LiLFeS の処理系の最初のバージョンは WAM[6] のエミュレータをベースとし、素性構造を従来の処理系よりも高速に取り扱うことができた。しかし、確定節の処理などが非効率であったため、情報抽出や対話システムなどの現実的な応用のためにより効率的な実装が必要とされていた。

そこで現在これを高速化する計画を進めている。従来の WAM ベースの処理系を、Aquarius Prolog[1]の抽象機械である "BAM(Berkeley Abstract Machine)" をベースとしたものに変更し、さらにエミュレータで処理されていたものをネイティブコードにコンパイルするというものである (図2)。Aquarius Prolog は処理速度の面を主眼に置いた Prolog の処理系であり、その抽象機械である BAM は、インストラクションをより精度の細かいものにすることにより、コンパイル結果の処理の細かい部分を変更することを可能にしている。

本論文では、この計画の一部である最適化の中で、おもに TFS 処理言語に特有の最適化について論じる。

```
my_list <- [bot].
e_list <- [my_list].
ne_list <- [my_list] + [FIRST:bot, REST:my_list].

append( e_list, X, X ).
append(  (FIRST:A & REST:X),
         Y,
         (FIRST:A & REST:Z)   )
  :- append(X,Y,Z).
```

図 1　LiLFeS のプログラム例：リストの append

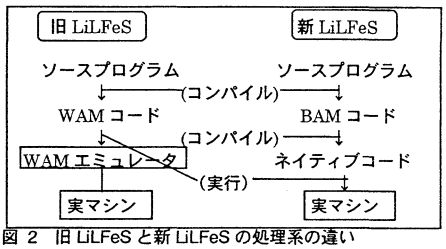

図 2　旧 LiLFeS と新 LiLFeS の処理系の違い

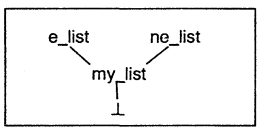

図 3　型階層の例

2. LiLFeS コンパイラの概要

LiLFeS 言語のプログラム形式は Prolog に似ており、確定節の集合と型定義の集合から成っている。図1が LiLFeS のプログラム例である。最初の3行で型の定義をしている。型 my_list を定義し、それに包含される型として[1]素性 FIRST と REST を持つ型 ne_list と、空リストを表す型 e_list を定義する。残りの行が確定節の定義である。述語 append の最初の節が空リストの処理、2番目の節が空でないリストの処理で、第1引数のリストの最初の要素を第3引数のリストの最初に挿入する操作をして再帰的に append を呼び出している。

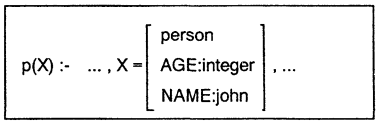

図 4　変数と TFS の単一化

[1] なお、このような包含関係を表す型の半順序集合を**型階層**と呼び、図3のように表す。型は、下に行く程一般的になる。図3では、⊥が最も一般的な型であり、my_list がその次に一般的な型である。

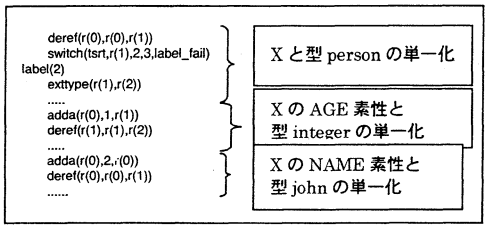

図 5　生成される抽象機械コード

本研究で扱っているのは、LiLFeS コンパイラのうち、おもに単一化に関する部分についてである。最適化技術の説明に入る前に、まずソースプログラムの単一化部分がどのように抽象機械コードにコンパイルされるかの概要を示す。

例として、図 4 のような変数と TFS の単一化を記述したプログラムのコンパイル過程を説明する。抽象機械は、変数の中身を「ヒープ」と呼ばれる配列中に格納し、プログラム中に現れる TFS を命令列として表す。図 4 のような単一化では、変数 X を右辺の TFS と単一化するため、「X の型を表すヒープの要素と型 person を単一化する命令列」「X の AGE 素性を表すヒープの要素と型 integer と単一化する命令列」「X の NAME 素性を表すヒープの要素と型 john と単一化する命令列」が生成される（図 5）。

なお、LiLFeS では、素性構造の型によって素性の種類が制限されている[2]ので、どの素性がヒープのどの位置にあるか（TFS の先頭からの相対位置）は型から一意に定まる[3]。

3. 最適化技術

我々は次の 2 つの方針で最適化を実装した。ひとつは「ソースプログラムの変形」、もうひとつは「生成コードの特化」である。前者はソースプログラムの不要な処理を省くものであり、後者は生成コードの不要な処理を省くものである。以下、それぞれの最適化を順に説明していく。なお、以下に述べる手法のうち、「ファクタリング」「データフロー解析」と「決定性変換」は、Aquarius Prolog における最適化技術を参考にしている。

[2] より正確に言えば、LiLFeS の扱う素性構造はすべて totally well-typed である。totally well-typed の詳しい定義については、論文[7]を参照されたい。

[3] ただし、Prolog と異なり、型には包含関係があるため、単一化の結果、型が変化して素性の種類や相対位置が変わることがある。

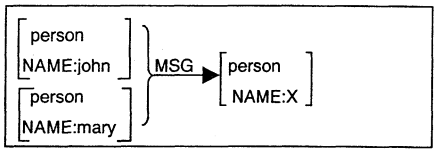

図 6　MSG の計算

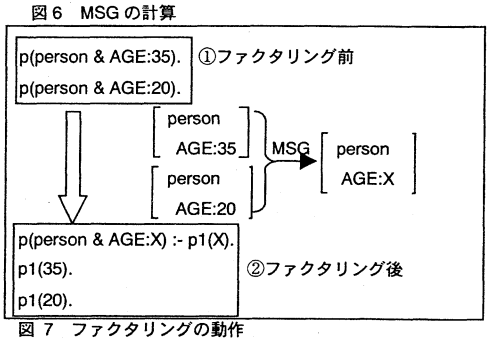

図 7　ファクタリングの動作

3.1. ソースプログラムの変形

ソースプログラムは、人間に分かりやすく書かれている反面、それをそのまま抽象機械コードに置き換えると不要な処理を多く含んでしまう可能性がある。そのようなプログラムをコンパイラにとって最適な形に変形する手法が「ソースプログラムの変形」である。本節ではそのうちの、今回実装した最適化手法であるファクタリングと決定性変換を主に説明する。

ファクタリング

ファクタリングは、Most Specific Generalization（以下 MSG）という操作に基づく変換である。MSG は単一化の逆操作であり、2 つの TFS をとってより一般的な TFS を計算する。そしてそのような TFS の中で最も特殊なものを返す（図 6）。

ファクタリングは、ある述語に対する複数の確定節に対し、それぞれの引数から MSG を計算し、その結果である TFS のルートが 1 つ以上の素性を持つならば、それらの節を一つの節にまとめる操作である。MSG を計算することにより、引数の共通部分を取り出すことができる。

図 7 は、ファクタリングの具体例である。①のように、2 つの節の引数がともに「素性 AGE を持つ型 person」で、AGE の値のみが異なっているとき、②のように、別の述語（p1）を作って AGE の中身だけをそこで判定させ、残りの差異のない部分については一回の単一化で済むようにできる。このようにして複雑な TFS との単一化の回数を減少させることができる。

```
①fib(0,X) :- X=1.
②fib(1,X) :- X=1.
③fib(N,X) :- N > 1, N1 is N – 1, N2 is N – 2,
             fib(N1,X1), fib(N2,X2), X is X1 + X2.
```

呼び出し元で fib の第 1 引数が 0 なら①の節を、1 なら②の節を、それ以外の自然数なら③の節を呼び出す。このとき、呼び出す節は一意に定まる。但し、第 1 引数が変数のときは①②③を順に呼ぶ。

図 8　一意な呼び出しができる例

決定性変換

決定性変換は、失敗することがコンパイル時に判明しているバックトラックを省く最適化である。ファクタリングは、この決定性変換の精度を向上させるのにも役立つ。

決定性変換においてコンパイラは、ある述語に対して複数の節があり、どの節を呼び出すかが引数から限定できる場合（図 8）に、バックトラックによる探索を分岐命令に置き換える。つまり、引数を見て適切な節のみを呼び出し、バックトラックを行わないようにする。LiLFeS コンパイラでは、引数の TFS におけるルートの型を見て節の分類を行なっている。

現在の決定性変換の実装では、ルートの型しか見ていないので、「ルートの型が同じで他が異なる」という場合に対応できない。しかし、ファクタリングによって引数の TFS を分解すれば、型の異なる部分だけを抜き出すことができ、決定性変換を行える部分が増えることになる。

3.2. 単一化コードの特化

最適化のもう一つの方法は、コンパイル結果の単一化コードをソースプログラム中の状況に応じて特化することである。プログラムを解析することで、単一化する変数の型や素性に関する情報を割り出し、遅い汎用の単一化コードのかわりに特化した速いコードを生成する、あるいは、不必要な単一化を省く、などの最適化を行う。

例として、図 9 のようなプログラムの断片を考える。この述語は、述語 get_lexicon を呼び出して単語 Word に対応する語彙を Lex に取得したあと、その TFS の BFORM 素性を単一化によって BaseForm に取り出すものである。この単一化の部分は、通常のコンパイルでは、変数 Lex が lexicon

```
get_baseform(Word, BaseForm)  :-
        get_lexicon(Word, Lex),
        Lex = (lexicon & BFORM:BaseForm).
```

図 9 サンプルプログラム

```
get_baseform(Word, BaseForm) :-
        Word = word型       }  ①get_baseform
        Lex = 未使用        }  呼び出し直後の
        BaseForm = ⊥        }  変数の下界

get_lexicon(Word, Lex),        ②get_lexicon
        Word = word型       }  実行直後
        Lex = lexicon型     }  の変数の
        BaseForm = ⊥        }  下界

Lex = (lexicon & BFORM: BaseForm).
        Word = word型       }  ③単一化
        Lex = lexicon型     }  実行直後の
        BaseForm = string型 }  変数の下界
```

図 10 データフロー解析結果

型を持つかどうかの判定や、BFORM 素性の場所の計算などの複雑な処理を多く含むが、例えば述語 get_lexicon が必ず lexicon 型を返すことがわかっていれば、処理を単純化できる。

ソースプログラム中の変数の情報を得るための手法が、**データフロー解析**である。データフロー解析は、ソースプログラムに対して抽象解釈を行い、ソースプログラムの各確定節の各リテラルに対し、出現する全ての変数の「取りうる TFS の下界」（つまり、取りうるどの TFS よりも一般的な TFS）を求める操作である。図 10 は解析結果の例である。

データフロー解析の結果を単一化コードに反映させるため、今回は以下の 2 つの最適化手法を実装した。

型の単一化コードの特化

一般的な型の単一化コードは、任意の型階層に対応するため、複数の条件分岐を含んだ複雑なものになっている。しかし、型の下限などの情報がある場合には、それに応じてコードを特化し、より単純な処理に置き換えることができる。図 10 の変数 Lex との単一化で、型 lexicon を単一化する部分では、変数 Lex の下限が型 lexicon と一致しているので、単一化コードそのものが不要になる。

素性位置計算の削減

LiLFeS では、素性構造の型によって素性の種類が制限されているため、ヒープ上の素性の位置は型に対して一定である。一般的な単一化コードでは、実行時の型から素性が格納されている相対番地を計算しているが、コンパイル時に得られた型情報を使うと、一部の相対番地はコンパイル時に決定することができる。

これらの最適化は、複雑な (ノード数の多い) 素性構造に対してより効果的に、実行時間やプログラ

ムサイズを押さえることができる。HPSG などの自然言語処理では、ノード数の比較的多い (50 から 100 以上) 素性構造を扱うことが多いので、これらの最適化が有効に働くと考えられる。

4. 実験結果

表 1 は、LiLFeS の 2 つのバージョンと、他の TFS 処理システムについて、実行速度を測定した結果である。ALE システム[8]に付属する文法を使って 14 語の文 (134 通りのパーズが可能) をパーズさせたもの (「ALE 文法」) と、HPSG に似たごく簡単なパーザに 9 語の文 (曖昧性なし) を 1000 回パーズさせたもの (「簡単なパーザ」) について測定している。実行環境は、Sun UltraSparc 1/167MHz (メモリ=128MB) である。

新バージョンについては、最適化のあり/なしの両方を測っている (opt と表記されているのが最適化を伴ったものである。ただし、決定性変換は旧 LiLFeS でも実装されていた最適化なので、今回の実験では「最適化なし」のほうにも実装してある)。既存のシステムは、ALE[8]と ProFIT[9]について調べた。いずれも SICStus Prolog 上で実行している。SICStus は、WAM エミュレーションとネイティブコードの 2 つのバージョンで試した (表では、それぞれ emulation, native と表記されている)。

なお、新 LiLFeS の現在のバージョンでは、ソースプログラムを C 言語コードに変換し、最後に C コンパイラにかけるという方式をとっている

「ALE 文法」については、メモリ容量の問題から、完全には実験できなかった。最適化なしの場合で未実験とあるのは、メモリが不足しコンパイルに失敗したものである。また、最適化ありの場合でも、やはりメモリ容量の問題で C コンパイラの最適化がかけられなかった。

表 1 実行時間の比較 (単位=秒)

	単純なパーザ	ALE 文法
新 LiLFeS(opt)	1.20	1.61
新 LiLFeS	2.00	未実験
旧 LiLFeS	5.70	2.56
ALE(emulation)	225.60	37.71
ALE(native)	67.05	26.69
ProFIT(emulation)	2.94	8.08
ProFIT(native)	1.48	9.78

単純なパーザでは、前バージョンの 4 倍以上の高速化を実現している。ALE 文法ではそれほど高速化していないが、これは C コンパイラの最適化をかけることができなかった為だと思われる。それでも既存のパーザの 5 倍の速度を達成している。

最適化もある程度効果を示し、全体で約 4 割の高速化になっている。ただし、今回ファクタリングは速度の向上に結びつかなかったので除外してある。これは「決定性変換」の実装がまだ不完全であることも影響していると考えられる。「決定性変換」を完全に実装すれば、ファクタリングの効果も現れてくると思われる。

5. 結論と今後の課題

LiLFeS コンパイラを BAM ベースのものに実装し直したことにより、実行速度は簡単なパージングプログラムで 4 倍以上に上がった。また、今回論じた最適化は、約 1.7 倍の高速化を実現した。

今後の課題は、この新しい LiLFeS の処理系を完成させ、その上で動く自然言語処理システムを実装していくことである。また、現在 C 言語コードを経由して生成しているネイティブコードを直接生成するようにして、さらなる高速化を目指すことを考えている。最適化については、現在まだ実装が不十分な最適化がいくつかあるので、その実装を進めていく。現在第 1 引数しか見ていない決定性変換をすべての引数で行うことや、現在ほとんど最適化されていない変数どうしの単一化についても最適化を行うことなどが考えられる。

参考文献

[1] Van Roy: Can Logic Programming Execute as Fast as Imperative Programming? Technical Report CSD-90-600, University of California, Berkeley. (1990)

[2] MAKINO, TORISAWA and TSUJII : LiLFeS - Practical Unification-Based Programming System for Typed Feature Structures. In *Natural Language Pacific Rim Symposium '97.* (1997)

[3] 光石、鳥澤、辻井 : HPSG を用いた統語解析のための統計モデル. 言語処理学会第 4 回年次大会発表論文集 (1998)

[4] 二宮、鳥澤、辻井 : 並列 HPSG パーザー. 言語処理学会第 4 回年次大会発表論文集 (1998)

[5] 宮尾、鳥澤、建石、辻井 : 実用的な HPSG 文法のための二つの手法 : 型の Combining と選言的素性構造の Packing. 言語処理学会第 4 回年次大会発表論文集 (1998)

[6] Aït-Kaci : Warren's Abstract Machine: A Tutorial Reconstruction. The MIT Press. (1991)

[7] Carpenter : The Logic of Typed Feature Structures. Cambridge University Press. (1992)

[8] Carpenter and Penn : ALE 2.0 User's Guide. Carnegie Mellon University Laboratory for Computational Linguistics, Technical Report (1994)

[9] Erbach : ProFIT – Prolog with Features, Inheritance and Templates. In *IWPT 95*, pages 59-70 (1995)