プログラミング 4－3
（1995．11．30）

項書換えシステムに基づくソフトウェア自動合成システムの実現

友部 実、佐藤明良、山之内徹
tomobe@swl.cl.nec.co.jp, a-sato@swl.cl.nec.co.jp, yamano@swl.cl.nec.co.jp
NEC C&C 研究所
〒216 川崎市宮前区宮崎 4-1-1

項書換えシステムに基づくソフトウェア自動合成システム SOFTEXSHELL のアーキテクチャと実現方式について報告する。本稿ではSOFTEXSHELL で用いた項書換えシステムの実現方法として、直接実行方式と、抽象項書換え機械の改良方式とその最適化手法について提案し、それぞれの手法の評価を行った。直接実行方式では書換え戦略を自由に選択可能な反面、実行時のリデックスの探索にコストがかかる。抽象項書換え機械による実装では書換え戦略を最左最内戦略に限定することにより、スタックマシンによる実装が可能になり、最適化の手法と併せて、直接実行方式に比べ、5-20 倍高速化可能なことがわかった。

Implementation of Software Synthesis Shell using Term Rewriting System

Minoru Tomobe, Akiyoshi Sato, Toru Yamanouchi
NEC C&C Research Lab.
4-1-1, Miyazaki, Miyamae-ku, Kawasaki, Kanagawa, 216 Japan

In this report, we describe the architecture and implementation of software synthesis shell SOFTEXSHELL. SOFTEXSHELL uses a many-sorted term rewriting system (TRS) for its transformation mechanism. We propose two implementation techniques of many-sorted TRS, direct execution method and reduction with abstract reduction machine (ARM). In direct execution method, the left-hand sides of rewriting rule are compiled into a target language function, and the runtime routine searches redex and applies compiled rules. In this method, reduction strategy is not limited, However it requires considerable redex searching cost. In ARM, reduction strategy is limited to left most inner most strategy, However this limitation enables efficient implementaion with stack machine architecture. With some optimization techniques, the execution speed of ARM is 5-20 times faster than direct execution method.

1 はじめに

項書換えシステム (Term Rewriting System 以下 TRS) に基づくソフトウェア自動合成システム SOFTEXSHELL のアーキテクチャと実現方式について報告する。SOFTEXSHELL は変換規則と入力仕様形式から特定の領域のジェネレータを生成するシステム [1] であり、変換の実行時のメカニズムとして TRS を用いる。TRS を用いることにより、変換規則の記述をパターンマッチを用いて簡単に記述できる、書換え規則が参照透明な為、変換のデバッグが容易などの利点をもつ反面、計算モデルが単純な為、複雑な処理を行おうとすると、自動合成に要する書き換え回数が膨大になり、高速な変換系が要求されるという問題点がある。本稿ではこのような問題点を解決するため、TRS の実装方式についていくつか検討を行い、高速に書換えが可能な TRS の実行方式を提案する。

2 項書き換えシステムによるソフトウェア自動合成システム

2.1 SOFTEXSHELL の概要

図 1に SOFTEXSHELL のシステムの概要を示す。SOFTEXSHELL は対象とする領域に特化したジェネレータを開発する為の自動合成シェルであり、入力仕様の記述形式と、対象とするプログラムを合成する為の変換規則を入力することにより、ジェネレータを生成する。

図 2に SOFTEXSHELL を用いたジェネレータの変換のイメージを示す。SOFTEXSHELL は入力仕様ならびに変換規則を記述するための言語として DSL 言語を提供する。DSL 言語によってユーザが定義した入力仕様はパーザによって構文解析が行われ構文木として表現される。SOFTEXSHELL では、この構文木に DSL 言語によって記述された変換規則を適用することによって変換を行い、最終的にユーザの意図するプログラムを合成する。SOFTEXSHELL における変換では書き換え規則は幾つかのモジュールと呼ばれる書き換え規則の集合に分割される。変換は対象となる構文木にモジュールを適用し、そのモジュールに属する書き換え規則の適用の結果得られた正規形に対して、さらに別のモジュールを繰り返し適用するという過程を経て行われる。

2.2 多ソート項書換えシステム

ここでは SOFTEXSHELL で用いる多ソート項書換えシステムの定義と、SOFTEXSHELL で用いるモジュールの定義を行う。

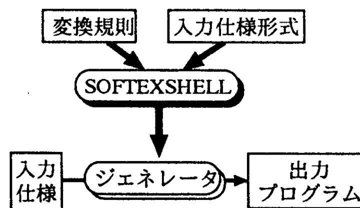

図 1: SOFTEXSHELL のシステム構成

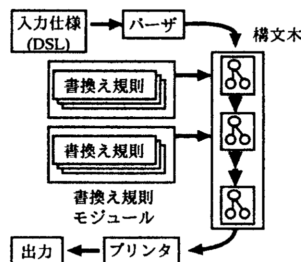

図 2: 変換の過程

定義 2.1(多ソート指標) 多ソート指標 Σ とはソート記号集合 S と関数記号集合 F の組 $\Sigma \stackrel{\mathrm{def}}{=} < S, F >$ である。ただし、$\forall f \in F; f_{s_1 \ldots s_n s}; n \geq 0; s_1, \ldots s_n, s \in S$ である。ここで、ソート列 $s_1 \ldots s_n$ は関数記号 f の定義域、s は f の値域を示す。□

定義 2.2(変数) 変数集合 V は、ソート記号 $s \in S$ を持つ変数 v_s の集合である。□

定義 2.3(多ソート項) 多ソート指標を $\Sigma =< S, F >$、変数集合を V とするとき、ソート s の多ソート項集合 $T_s(\Sigma, V)$ は以下のように定義される。:

1. $v_s \in V, s \in S$ ならば $v_s \in T_s(\Sigma, V)$
2. $f_{s_1 \ldots s_n s} \in F;\ t_1 \in T_{s_1}(\Sigma, V), \ldots, t_n \in T_{s_n}(\Sigma, V)$ ならば $f(t_1, \ldots, t_n) \in T_s(\Sigma, V)$

多ソート項集合 $T(\Sigma, V)$ とは $T(\Sigma, V) \stackrel{\mathrm{def}}{=} \bigcup_{s \in S} T_s(\Sigma, V)$ である。□

定義 2.4(多ソート書換え規則) 多ソート項集合を $T(\Sigma, V)$ とし $l, r \in T(\Sigma, V)$ かつ r に現れる変数は l に現れるとするとき、$l \rightarrow r$ を多ソート書換え規則という。□

定義 2.5(モジュール) 多ソート項集合 $T(\Sigma, V)$ 上の書換え規則集合 $R = \{l_i \rightarrow r_i \mid l_i, r_i \in T(\Sigma, V), i = 1, \ldots, n\}$ をモジュールと呼ぶ。□

SOFTEXSHELL では、上に定義した多ソート項のうち、特に $T(\Sigma, \phi)$ であるような変数を含まない基礎項に対する書換えを行う。また書換え規則の左辺は非線

形な項を含み、書換え規則の線形性は規定しない。また同一モジュール内の書換え規則の重なりを許し、書換え規則の重なりがあった場合、書換え規則の出現順序順に適用が行われる。

定義 2.5(構成子項) 多ソート書換え規則集合 $R = \{l_i \to r_i \mid l_i, r_i \in T(\Sigma, V), i = 1, \ldots, n\}$ があったとき、各書換え規則の左辺が $l_i = f_i(t_{i_1}, \ldots, t_{i_n})$ で表されるとする。このとき、$\Sigma_c =< F - \{f_i \mid i = 1, \ldots, n\}, S >$ とし、このような多ソート指標 Σ_c から構成される多ソート項集合 $T_c(\Sigma_c, \phi)$ の要素を構成子項と呼ぶ。□

3 直接実行方式による項書換えシステムの実現

項書換え規則のコンパイル手法として、書換え規則を LISP 等の言語にコンパイルする手法 [2][5] がある。これらの実現では書換え規則に対応する関数の実行によって書換えが実行されるが、正規化戦略を自由に選択することはできない。

SOFTEXSHELL で用いる直接実行方式では書換え規則のパターンマッチの部分を書換え規則コンパイラによって直接実行可能な C 言語にコンパイルし、これと書換え対象のリデックスからこれらの関数を呼び出す実行時処理系を組み合わせて多ソート項書換えシステムを実現する。この為、実行時に自由な書換え戦略を選択することができる。

3.1 書換え規則のコンパイル

書換え規則はモジュール単位で書換え規則の最外の関数記号単位に C の関数にコンパイルされる。直接実行方式では項表現はヒープ上のツリーとして表現される。また書換え対象を基礎項に限定しているため、書換え規則の単一化は単純なパターンマッチとして実現される。図 3に書換え規則とコンパイルされた書換え規則の C 関数を示す。この例では書換え規則の 5-7 行目で項 foo を項 bar に書換える規則を定義している。書換え規則の左辺は C 関数の if 文として展開され、ヒープ上のツリーによって表現された書換え対象と左辺のパターンマッチを行う (4-7 行目)。左辺に現れる変数はソートの検査を行った後代入される (8 行目)。左辺に同一の変数が出現する場合は Equal 関数によって同一性をチェックする (10 行目)。パターンマッチに成功すると、書換え規則の右辺にしたがってヒープ上にツリーを構成する (11 行目)。

3.2 実行時処理系

実行時処理系はヒープ上のツリーとして表現された書換え対象から指定された書換え戦略に従って、リデックスを選択し、選択されたリデックスの関数記号

書換え規則
```
 1: sortdef s;
 2: termdef 's('s,'s) foo;
 2: termdef 's('s) bar;
 4: vardef 's x,y;
 5: relation r_1 eql (
 6:     foo(bar(x),x),bar(x)
 7: )
```

C にコンパイルされた書換え規則
```
 1: int R_foo(term **spec,term **addr) {
 2:   term *var_120 /* x */ ;
 3:   term *tmp_0,*tmp_1,*tmp_2,*tmp_3;
 4:   if((tmp_0 = *spec) &&
 5:      (tmp_1 = tmp_0->u.car) &&
 6:      (tmp_1->u.functor == 115) &&
                  /* 115 = 'bar'*/
 7:      (tmp_2 = tmp_1->cdr) &&
 8:      Bind(var_120,5234,tmp_2) &&
                  /*5234 = 's'*/
 9:      (tmp_3 = tmp_0->cdr) &&
10:      Equal(tmp3,var_120)) {
11:         *addr = make_cell(115,1,
                cons(var_120,NULL));
12:         return(TRUE);
13:      }
14: return(NULL);
15: }
```

図 3: 書換え規則のコンパイル例

に相当する書換え規則の適用を行う。書換え規則はあらかじめ C 関数としてコンパイルされており、実行時処理系はこの関数をリデックスを引数として呼び出すことにより書換えを実行する。最左最外戦略の場合、書換えが成功した後、再び書換え対象の最左最外部から再びリデックスの選択を行い、書換え可能な部分がなくなるまで、これを繰り返す。最左最内戦略の場合、書き換えた結果生成された項を再び書き換え対象として評価し、最左最内部分から正規形を求め書き換えを繰り返す。

3.3 構成子項マーキングによる高速化

一般に項書換えシステムで書換え実行中に書換え対象が大きくなった際、その中からリデックスを選択するコストが大きくなる。SOFTEXSHELL における変換は、書き換え規則の集合をモジュールとして順次適用していくことにより行われる。この為、書換え対象はあるモジュールに対して構成子項だけで構成される部分が多く含まれる。この部分をあらかじめリデックスの選択の対象から外しておくことにより、実行時に選択するリデックスの範囲を大幅に減少することが可能になり変換の高速化を行うことができる。実行時処理系はあるモジュールに属する書換え規則の適用を行う前に、書き換え対象の中で構成子項だけで構成されている項にあらかじめマークをつけ、書換えを実行し、正規形が得られたところで、これらのマーク

を外す。これを各モジュール毎に繰り返し実行することにより、各モジュールによって書換えが可能な部分のみをリデックスの選択範囲とすることができる。

3.4 直接実行方式の限界

直接実行方式では書換え戦略を自由に選択できる反面、リダクションを実行する際に、多くのリデックスの探索を行う必要があり、書換え規則のコンパイルだけでは実行速度の向上には限界があった。

SOFTEXSHELL による変換では書換え戦略として最左最外戦略と最左最内戦略を用いている。これまで開発を行ったジェネレータの書換え規則を分析すると、多くの場合条件分岐等のスペシャルフォームを用いることにより戦略を最左最内戦略に固定することが可能であることがわかった。書換え戦略を最左最内戦略に固定することにより、従来の関数型言語の実装技術を用いることが可能になり、より効率の良い書換えを実現することができる。

4 抽象項書換え機械による実現

項書換えシステムの実現手法として、抽象項書換え機械 (以下 ARM) [4] による方式がある。ARM では書換え戦略を最左最内戦略に限定し、リダクションを関数型言語の実装に多く用いられるスタックマシンを用いて行う。この結果リデックスの探索を大幅に削減することが可能になり、高速な書換えを実現できる。本節では ARM の動作原理について述べたあと、SOFTEXSHELL で用いる ARM の実装方式と最適化手法について説明する。

4.1 抽象項書換え機械

ARM の簡単な動作原理を図 4にしめす。ARM では二つのスタック (制御スタックと引数スタック) と一つのヒープ領域を持つ。書換え対象となる項は制御スタック上に置かれ (1)、スタック上の項に対応する書き換え規則が存在する場合には、その規則による書き換えが行われ、その結果が再び制御スタック上に積まれる (4)。また書換え規則が存在しない場合には、関数記号のアリティに応じて引数スタックより値が取られ、その結果構成された項をヒープ領域上に確保し、その結果が引数スタック上に置かれる (2,3,5)。最終的に制御スタックに書換え対象となる項が存在しなくなった時に書換えを終了する (5)。ARM ではこのような動作を行うスタックマシン上に、パターンマッチを行う select,check, 書換え結果の生成を行う proceed といった基本的な命令セットを用意し項書換えシステムを実現している。

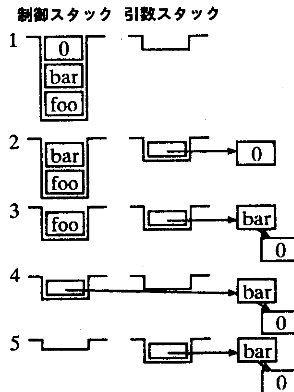

図 4: 抽象項書換え機械の動作

4.2 ARM の実装と最適化手法

4.2.1 弁別ネットによるパターンマッチ

ARM では書換え規則のパターンマッチに select 命令を用いる。select 命令はパターンマッチを行う書換え規則の左辺に現れる項の関数記号に応じて場合分けを行い、パターンマッチに成功した場合 proceed 命令によって右辺に対応した項を生成する。この select 命令の実現手法としては [4] による C 言語への直接コンパイル、また [3] の Flatterm を用いた弁別ネットによる実現 [6] が提案されている。SOFTEXSHELL の ARM による実装では [6] と同様に select 文を弁別ネットにより実現し、弁別ネット上での最適化を行う。SOFTEXSHELL ではモジュール単位で実行時の実現を選択することが可能であり、インタプリタによる実行では弁別ネットを用いてパターンマッチを行い、コンパイラによる実行では弁別ネットをパターンマッチを行う C 関数にコンパイルすることにより、実行時の性能を向上する。

4.2.2 インクリメンタルな弁別ネットの生成

[6] で指摘されているように、[3] による弁別ネットの実現では、左辺の全てのシンボルを並べたテーブルをノードとしているため、メモリ効率が悪い。SOFTEXSHELL の実装では弁別ネットの生成は書換え規則のモジュール単位で行う。SOFTEXSHELL では書換え規則の重なりがあった場合、ルールの出現順序順に適用を行う為、モジュール内のルールを出現順序順からインクリメンタルに弁別ネットに展開す

る。インクリメンタルに弁別ネットを構成することにより、ネットワークに現れる関数記号は[3]の手法に比較して、モジュール内に現れる書換え規則の関数記号に限定される為ネットワークのサイズを縮小することができる(図5)。

4.2.3 弁別ネットの最適化

全ての規則について弁別ネットが構成されると、SOFTEXSHELLでは次に弁別ネットの最適化を行う。弁別ネットで同一関数記号で規則間の重なりが無い場合、再帰規則となっているものを弁別ネットワーク上で上位にもってくることによって、パターンマッチを行う際に、出現頻度の高い部分を先に処理を行い、パターンマッチのコストを減少させるといった最適化の処理を行う(図6)。

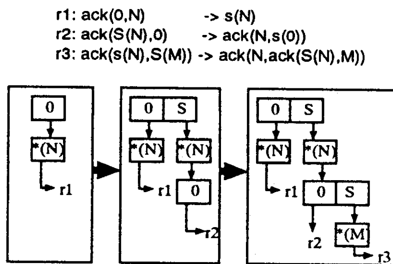

図 5: 弁別ネットの生成

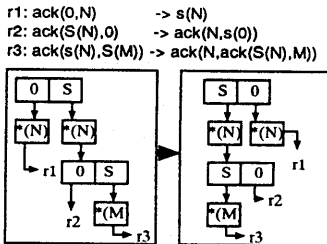

図 6: 弁別ネットの最適化

4.2.4 項生成の最適化処理

ARMでは書換え規則の左辺のパターンマッチに成功すると、対応する右辺の項を制御スタック上に生成することにより書換えを実現する。SOFTEXSHELLでは右辺を生成する際に以下のような最適化処理を行う。

- 右辺の項の中で構成子項だけで構成される項があった場合、これらの項をヒープ上にあらかじめ構成するようなコードを生成することにより、制御スタック、引数スタック間の値の受け渡しの回数を削減する。
  図7の書換え規則r1では右辺より生成される項は全て構成子項となる為、制御スタック上で評価する必要が無い為、直接引数スタック上に生成される。

- 右辺が生成する項が末端再帰となる場合は、再び同一規則でのパターンマッチを行うコードを生成することにより、制御スタックへのアクセス回数を削減する。
  図7の書換え規則r2では構成子項は全て引数スタックへ積まれ、関数記号ackを制御スタック上に置かず、再び関数記号ackに関するパターンマッチを行う。

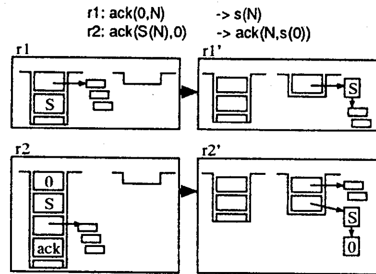

図 7: 書換え規則の右辺の生成

5 評価

直接実行方式による実装と、ARMに基づく実装方式のプロトタイプに対して、それぞれ最適化手法を適用した場合について実行速度の評価を行った。(表1,2,3) ここではアッカーマン関数を項書換え規則で実現した例と、SOFTEXSHELLを用いて実際に開発した13モジュール、360の書換え規則からなるジェネレータの変換規則に対して評価を行った。

またARMによる実装では

1. 弁別ネットをC関数にコンパイルし、右辺を直接C関数により生成する方式。

2. インタプリタによる弁別ネットを用いたパターンマッチと、右辺を生成するインタプリタコードを実行する方式。

の二種類について評価した。

表1, 表2の結果からわかるように、ARMによる実装では直接実行方式による実装に比べ、書換え戦略を最左最内戦略に限定したことによりスタックマシンによる実装が可能になり、リデックスの検索コストと

書換え関数呼び出しのコストを削減することができ、大幅に書換え速度が向上していることがわかる。

表1のARMコンパイラで最適化を行った評価結果からわかるように、末端再帰の最適化、パターンマッチの最適化を行うことにより、さらに書換え速度の向上を図れることがわかった。また表1のARMインタプリタによる評価結果からもわかるように、インタプリタによる実装でも、ARMの命令セットが単純な為、インタプリタの実現が簡単になり、これと弁別ネットを用いた高速なパターンマッチを行うことにより、直接C関数にコンパイルする方式に比べて数倍程度のオーバーヘッドでインタプリタを実現できることが判った。

表3の結果からわかるように、直接実行方式では構成子項をリデックスの探索対象からはずすことにより、2倍程度の高速化を図ることが出来る。表2の例ではアッカーマン関数が構成子項を持たない為、最適化の効果が現れていない。また、プログラムジェネレータにTRSを適用した場合は書換え対象がさらに大きく複雑になる為、大きく書換え速度が低下する。今回の実験ではARMによる実装がプロトタイプシステムであった為、ARMによるプログラムジェネレータの計測は行えなかったが、前述の実験結果よりかなりの速度向上が期待できる。

表 1: ARMによる実装(アッカーマン関数)

実現方式	変換時間(s)	書換え速度(rw/s)
ARM (コンパイラ)		
最適化あり	0.83	836000
最適化なし	1.39	500000
ARM (インタプリタ)		
最適化あり	2.56	271000
最適化なし	3.18	218000

表 2: 直接実行方式による実装(アッカーマン関数)

	変換時間(s)	書換え速度(rw/s)
最適化あり	17.0	40300
最適化なし	17.0	40300

表 3: 直接実行方式による実装(ジェネレータ)

	変換時間(s)	書換え速度(rw/s)
最適化あり	272	217
最適化なし	587	103

(SparcStaion20 実メモリ 32MB にて測定)

6 おわりに

項書換えシステムに基づくソフトウェア自動合成シェルSOFTEXSHELLの概要と、SOFTEXSHELLで用いた項書換えシステムの実現方法として直接実行方式とARMによる実装方式と最適化の手法について報告した。これらは表4に示す特徴をもち、ARMによる実装では直接実行方式に比べ、実行時のリデックスの探索時間が削減され、更に最適化手法を用いることにより、単純な例題では高速な書換えを実現できることが判った。

	直接実行方式	ARM
実現方式	パターンマッチ部のコンパイル	スタックマシン
書換え戦略	最左最外、最左最内	最左最内に限定
実行速度	低速	高速
高速化手法	構成子項の枝刈り	弁別ネットによるパターンマッチ ネットワークの最適化 右辺構成子項の最適化 末端再帰の最適化

表 4: SOFTEXSHELLの実装方式の比較

7 今後の課題

現在SOFTEXSHELLにARMを用いた実装方式のインプリメントを行っている。今後、この実装方式について、SOFTEXSHELLで用いた実例を元に書換え規則が多い場合や、書換え対象が大きくなった場合の変換速度に関して評価を行っていくと共に、他の最適化手法の検討ならびに実行時の効率の良いメモリ管理(GC)方式について検討を行っていきたい。

参考文献

[1] Yamanouchi,T.,Sato, A.,Tomobe M.,Takeuchi, H.,Takamura, J. and Watanabe, M. : Software Syntehsis Shell SOFTEX/S,*7thKBSEConf.*, 1992.

[2] Kaplan,S. : A Compiler for conditional term rewriting systems, *Proceedings of the First International Conference on Rewriting Techniques*, Vol.256 LNCS, pp. 25-41, 1987.

[3] Christian, J. : Flatterms, Discrimination Nets, and Fast Term Rewriting, *Journal of Automated Reasoning* Vol.10.,pp. 95-113, 1993.

[4] J. F. Th. Kamperman, H. R. Walters : ARM, Abstract Rewriting Machine, CWI, 1993.

[5] 戸村哲、二木厚吉：項書き換えシステムから Lisp プログラムへの変換系, 電子情報通信学会技術研究報告, SS86-9, pp. 15-20, 1986.

[6] 大原幸一、緒方和博、二木厚吉：項書換えシステムのための抽象機械の設計について, 情報処理学会第51回全国大会 pp. 5-41, 1995.