プログラミング 11−14
(1997. 1. 24)

手続き型言語での再帰の除去について

北川 拓* 渡辺 坦**
電気通信大学 情報工学科
〒182 東京都調布市調布ヶ丘 1-5-1
E-mail: *taku-ki@kankou.cs.uec.ac.jp **tan@cs.uec.ac.jp

あらまし 再帰呼び出しを用いたプログラムは、同じ処理を再帰を用いずに書いた場合と比べて実行するのに時間とメモリを多く必要とする。そのため、再帰呼び出しに対する最適化が望まれるが、それの実装されているコンパイラはわずかしか無い。本稿では、線形再帰を末尾再帰に変換して最適化する方法を提案し、実装を行なった。この方式は Arsac と Kodratoff によって提案された方法を改良したもので、彼らの方法では末尾再帰の変換に発見的方法を用いているが、それを本稿では探索を実装に適したアルゴリズミックな手順で実現している。
キーワード 再帰プログラム、再帰除去、コンパイラ、最適化

Removal of recursive call in procedural language

Taku Kitagawa Tan Watanabe
Department of Computer Science,
University of Electro-Communications
Chofugaoka 1-5-1, Chofu, Tokyo 182, JAPAN
E-mail: taku-ki@kankou.cs.uec.ac.jp tan@cs.uec.ac.jp

Abstract Execution overhead of recursive call is high and it is desirable to improve its object code. However, only a few compilers do it. We propose a technique of object code optimization for recursive calls and implemented it in our complier. In this technique, liner recursion is transformed to tail recursion by using Arsac and Kodratoff's method and then transformed to loop. Arsac and Kodratoff use heuristic method in finding tail recursion function, but we propose an algorithmic method suitable for computer.
Key words **recursive program,recursion removal,complier,optimization**

1 はじめに

コンパイラで行なうオブジェクト最適化としては、共通部分式の削除やループの展開、命令強度の削減などの方法がよく知られている。これらの方法は大きな効果が得られることもあって、今までに多くの研究が行なわれ、現在のほとんどのコンパイラに組み込まれている。しかし、再帰の効率化については、理論的研究は種々行なわれているが、実際のコンパイラに組み込まれている例は少ない。再帰的呼び出しは実行時間やメモリ使用量の点でオーバーヘッドが大きいので、手続き型言語に対して再帰の最適化を実装し、既存の最適化に上乗せすることで、より高度な最適化を行なうことを検討した。

2 再帰に対する最適化

再帰手続きを対象に行なう最適化の方法としては、関数の末尾に再帰がある場合、それを繰り返しに変換して再帰を除去する方法が良く知られている [1]。それができない場合には次のような方法が考えられる。

1. スタックを用いて再帰を繰り返しにする。
2. 特定の再帰的関数に対して、同等の働きをするが再帰ではない代替関数を用意し、置き換える。すなわち、あらかじめ使用頻度の高そうな再帰関数をいくつか選び、その関数と同等な計算を行なう関数を作成しておき、プログラムにその関数が出現したならば置き換える。
3. 再帰呼び出しの部分に自分自身を展開して呼び出しを削減する。[2]
4. Arsac らの提案している、一般化による再帰の除去 [4][3] をする。

これらのうち、第 1 の方法は、データを保存しておくスタックを管理する必要があるので、高速化はあまり期待できず、また確保したスタックの大きさによっては本来実行が可能な場合でも実行不可能になることがある。たとえば、階乗計算の関数をこの方法で書き換えた時、データを保存しておくスタックの大きさを呼び出し 20 回分確保したとする。この時に 21 の階乗を計算しようとすると、データ保存用のスタックがあふれて計算できない。

第 2 の方法は速度の高速化は期待が持てるが、適用できる機会が少なく、プログラム中の関数が置き換えの対象になるものであるか否かの判断も容易でない。

第 3 の方法は全ての再帰に適用可能であり、呼ばれる頻度が高ければそれだけ高速化にも期待ができる。この方法では再帰的に呼び出す回数は 2 回展開すれば 1/2、3 回展開すれば 1/3 になる。

第 4 の方法では、末尾再帰でない場合も、ある条件が満たされれば、再帰呼び出しを繰り返しに変換する。再帰呼び出しが行なわれなくなることで再帰が必要としているスタックを削減できる。

第 4 の方法は、それが適用できる場合には他の方法のような欠点がなく、こちらの方法を適用した方がより高速化すると思われる。そこで本研究では第 4 の方法を試みる。

3 一般化による再帰の除去法

この方法は線形再帰 (自分自身を呼び出す回数が高々一回であるような再帰関数) の関数 f に対して、再帰が関数末尾にあり、適当な引数を与えれば f と等しくなる関数 g で f を置き換えたのち、末尾再帰を繰り返しに変換することによって、再帰の除去を行なう方法である。この方法では、generalize,unfolding,folding の 3 つの手続きを用いて、目的とする関数 g を決定する。

generalize(一般化) は f の定義式の各辺のうち f を含んでいる辺の構文木を作成し、根から f が現れるまでのパスを比較し、一致していなければ式の意味が変わらないように注意して演算子を挿入して、根から f に至る全てのパス上に現れる演算子を等しくする。

例えば、

$$f(x) = \begin{cases} x * f(x-1) & (x > 0) \\ 1 & (x = 0) \end{cases}$$

. とするとき、f を含んでいるのは左辺の f(x) と右辺の x*f(x-1) である。それぞれを構文木であらわすと図 1のようになる。根から f の呼び出しまでの演算子の列は、左の木が<>、右の木が<*>であり、一致していない。この場合は左側に*が不足しているので、意味が変わらないようにしながら左側の木に*を挿入すると、図 2のようになる。

generalize によって得られた木全てに共通している部分を残し、共通していない部分は新しい変数に置き換えた木が目的の関数 g となる (図 3)。例の場合、2 つの木で共通している部分は根の*とその右の子の f であるので、この 2 つは残しておく。共通でない x と x-1,1 と x はそれぞれ変数 x,y に置き換える。その結果、この例の場合は

g(x,y)=y*f(x)

となる。

このとき g はまだ末尾再帰になっていないので、unfolding,folding を用いて末尾再帰にする。

unfolding では、g の f による定義式に f の漸化式を代入する。(図 4) g(x,y)=y*f(x) を f(x)=x*f(x-1) を用いて unfolding すると、

g(x,y)=y*(x*f(x-1))

となる。

folding は unfolding により得られた式を g の f による定義にあてはめ、g の漸化式を得る。この例では y*x*f(x-1) を，g(x,t)=y*f(x) にあてはめて、

g(x,y)=g(x-1,y*x)

とする。もしも g がまた末尾再帰でないならば、g の再帰的な呼び出しよりも後に行なわれる演算がある。しかし上の式では、x-1,y*x ともに g の引数であるため、g の呼び出しよりも前に評価されるので、g はすでに末尾再帰となっている。

4 実装する上での問題およびその解決法

一般化による再帰の除去を実際に実装する上で、以下のようなことが問題となる。

1. generalize,unfolding,folding の 3 つの手続きのうち、folding は目的のパターンにあてはまる漸化式を求めるという発見的な動作を含んでおり、そのまま機械的に取り扱うことができない。

2. この方式が対象としているのは、定義内容が一つの式の形で表現できる関数となっているが、一般的に手続き型言語で書かれているプログラムでは、関数は式であるとは限らず、いくつかの文の集まりで書かれている。

3. 関数に副作用が含まれている場合問題がおきる。

このうち、まず 1 番に対して考察する。構文木の根から f の呼び出しまでのパスに注目すると、folding は、unfolding によってパス上に挿入された演算子を変形によってパス以外の部分に追い出して、パスの部分を unfolding を行なう前と同じ形に変形させていると見ることができる。そこで、呼び出しまでのパスの長さが短くなるような変換、たとえば、a*(b*f(x)) を (a*b)*f(x) に置き換える変換 (図 5) や、a-(b+f(x)) を (a-b)-f(x) に置き換える変換 (図 6) などをいくつか組み合わせて適用し、folding の代用をする。

2 番に対しては、関数が複数の分の列として書かれていても、一つの式の形で表現可能なも

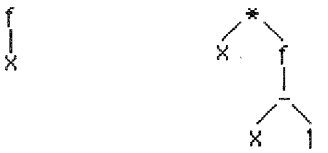


図 1: 与えられている木

のであれば、コピー伝播などによって一つの式で表現されているのと同じ解析木に変換する。

副作用を含んでいる位置が再帰呼び出しよりも後に現れていれば、この方法は全く使えないが、前であれば評価順序を壊さないようにして、この方法を適用することができる。しかし、関数 g を構文木で表現した時のパスに*と+が含まれており、なおかつ根から順に見ていった時、*よりも前に+が現れることがない場合は、folding を上記の方法で行なうことができない。例えば g(x,y,z)=(y+f(x))*z は演算子/を新たに導入しないと folding が行なえない ((y+a*f(x))*z は (y/a+f(x))*a*z になるため)。

整理するとこの方法を適用できる条件は次の通りとなる。

- 線形再帰 (自分自身を呼び出す回数が高々1回) である。
- 副作用を含んでいない。
- 関数が一つの式の形で表現されている。

5 実装

根から関数呼び出しまでのパスを短くする変換の組合せにより、folding を具体化した。その実装は次のように行なった。

パスを短くする変形の規則を収めたデータベースを作り、入力プログラムとそのデータベースのデータとのパターンマッチングにより、適用する規則を決定して変形を行なうようにする。これは、ルールをプログラムに直接記述するよりもデータベースを利用する方が後から規則の追加が容易になること、また判定を行なっている部分をまとめることで信頼性が高くなると期待されることによる。

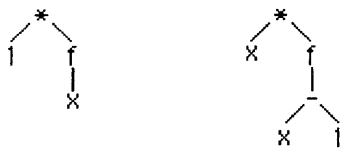


図 2: generalize を行なってできた木

また鏡像になっている変換規則 (たとえば図7) は、一つにまとめず別のデータとして扱う。その方が、データ量は増えるが、処理部が簡素になり、信頼性が向上する。逆向きのデータを分けることにより、変形を行なった部分以外を極力変化させないようにする。このことで関数に副作用が含まれている場合の適用可能性を向上させる。

個々の変換によって必ず呼び出しまでのパスの長さが短くなるので、変換が有限回で終了する (目的とする結果に到達する、または変換が行なえなくなる)。このことは処理が無限ループに陥らないことを示し、コンパイラに組み込む上で都合が良い。

実装したのは、関数が次のような条件を満たす場合についてであり、これらに対しては、末尾再帰への変換を経由して再帰を繰り返しに変換することができる。

- 関数の中で用いている演算子が+, −, *の3つである。または、用いている演算子が and,or,not の3つである。
- 再帰呼び出しよりも後に評価されるべき部分に副作用が含まれていない。
- 関数が一つの式で表現されている。

6 変換例

関数呼び出しまでのパスを短くする変換の組合せで folding を行なっている例として、

g(x,y,z,a)=y+(a+(1+f(x-1)*x)*z

を次の式に folding する。

g(x,y,z,a)=y+(a+f(x))*z

もとの式を木で表したのが図8である。この木に対して図11に示す変換規則を根の部分から適用すると図9のようになる。このときの根から関数呼び出しまでのパスは、folding の目的とする形と一致していないので、次に図11の変換を行なう。それにより図10の木が得られる。こ

の木の呼び出しまでのパスはfoldingの目標と一致しており、

$$g(x,y,z,a)=g(x-1,y+a*z,x*z,1)$$

が得られた。

7 まとめ

Arsacの一般化による再帰の除去の実装を行なった。その際にArsacの方法ではアルゴリズムの明らかでない発見的方法を含んでいたfoldingを、アルゴリズム的に実装する方法を提案した。副作用を含んでいる場合や、まだ変換規則が判っていない演算子を含んでいる場合など、適用が行なえない場合が残っている。今後の課題として、無作為に実用的なプログラムを選びだしてこの最適化がどれだけの効果を持つかを調べることがある。また副作用を含む場合などうまく実現できていない場合への対処や、適用できる演算子を増やすことがあげられる。

最後に、多くの有益な御違憲を頂いた電気通信大学情報工学科の鈴木貢助手に深く感謝致します。

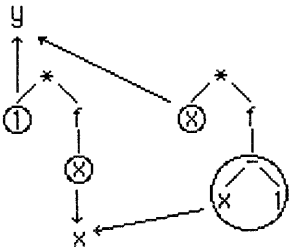
参考文献

[1] David F.Bacon,Susan L.Graham, and Oliver J.Sharp:
"Compiler Transformations for High-Performance Computing" ACM Computing Surveys, Vol.26, No.4, pp.345-420. Dec 1994.

[2] J.Darlingtton and R.M.Burstall:
"A System which Automatically Improves Programs" Acta Informatica 6,pp.41-60.(1976)

[3] R.M.Burstall and John Darlington:
"A Transformation System for Developing Recursive Programs" J.ACM Vol.24, No.1. pp.46-67. Jan 1977.

[4] J.Arsac and Y.Kodratoff:
"Some Techniques for Recursion Removal from Recursive Functions." ACM TOPLAS Vol.4,No.2,pp.295-322. Apr 1982.



全ての木で共通となっていない部分は
変数に置き換える

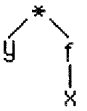


図 3: 関数の置き換え

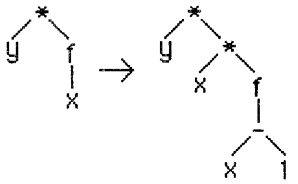


図 4: f(x)=x*f(x-1) による unfolding

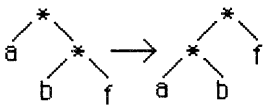


図 5: 変換規則の例 1

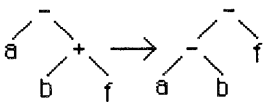


図 6: 変換規則の例 2

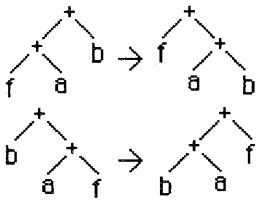


図 7: 形が鏡像となっている 2 つの変形規則

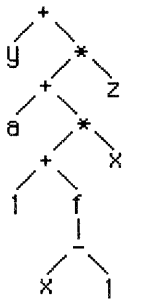


図 8: folding を行なう前の式

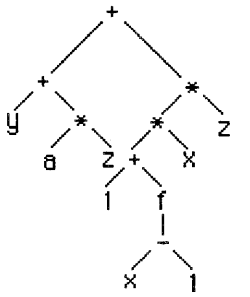


図 9: a+(c+f)*b=a+c*b+f*b を用いて変換した式

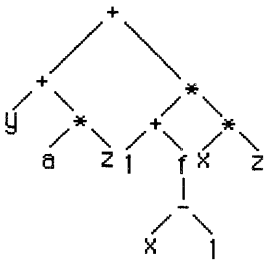


図 10: folding が行なわれた式

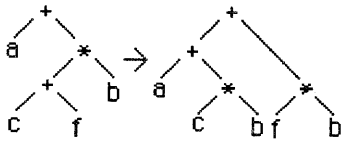


図 11: a+(c+f)*b=a+c*b+f*b

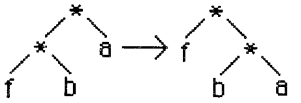


図 12: f*b*a=f*(b*a)

変形規則