

関数プログラムのコンパイラにおける 型情報を用いた効率的な共有解析の実現

尾上能之 金子敬一 武市正人

東京大学 工学部 計数工学科

概要

共有解析は、プログラムの中に出現する同じ部分式をまとめることによって、計算の重複を避けるための手法である。これは実行時間の短縮や消費されるメモリの節約などにつながる。従来から、手続き型言語で書かれたプログラムをコンパイルする過程において共通部分式を削除する方法は知られている。これは式を二分木で表現し、任意の二つの部分木の同等性を調べていく、というものである。この手法を簡潔なパスに分解し、関数型言語で書かれたプログラムを変換するための手法を示す。また型情報を用いることによって共有解析自体にかかる時間を短縮させる方法も示す。

Effective implementation of sharing analysis for compilers of functional programs using types

Yoshiyuki Onoue Keiichi Kaneko Masato Takeichi

Department of Mathematical Engineering and Information Physics,
Faculty of Engineering, University of Tokyo

Abstract

Sharing analysis eliminates common sub-expressions in a program to avoid repeated computation of the same expressions. This reduces evaluation time and saves memory consumption. Techniques for eliminating common sub-expressions of programs in procedural languages are well-known. These methods deal with expression trees and check identity of any sub-trees. We extend these methods for functional languages, and show how to transform functional programs using simple passes. Finally we propose a method to decrease time of sharing analysis using type information.

1 はじめに

手続き型言語で書かれたプログラムを効率良いコードにコンパイルする手法は、以前から研究されてきた [1][2]。そのコード最適化に関する手法として、ループ最適化やデータフロー解析などと共に、局所最適化の一種として基本ブロック (basic block) に対する共通部分式の削除 (common sub-expression elimination) がある。基本ブロックとは、初め以外のところへ飛び越してくることはなく、終わり以外から飛び越していくこともないコード列のことである。これは基本ブロックの中に出現する同一の部分式を一つにまとめることによって、重複した計算を避けるというものである。この操作により、実行時間の短縮や消費されるメモリの節約、さらにコンパイルコードが短くなる、などの向上が見込まれる。

しかし手続き型言語では、同一の部分式であってもそれが副作用を伴う場合、一つにまとめることができない。例えば、

$$y = f(x) + f(x);$$

という式が出現しても、関数 $f()$ が大域変数を操作している場合などを考えると

$$\begin{aligned} z &= f(x); \\ y &= z + z; \end{aligned}$$

とは簡単には変換できない。したがって削除の対象となるのは副作用のない事が分かっている四則演算や配列の参照などで、ユーザの定義した手続きや関数までこの方法だけで調べるのは不可能である。ゆえに基本ブロック内の共通部分式のみが重複削除の対象となる。

ここで対象とする関数型言語は、副作用を許さないで参照透過性 (referential transparency) という性質を持つ。これは手続き型言語におけるような状態遷移という概念を持たず、同一の式は常に同じ値を表わすことを意味する。この性質により、関数プログラムは文法的に同一の式はすべてまとめ上げることが可能で、共有解析が活かされるものであるといえる。

Syntactic Domains

$b \in \text{Bas}$	basic values
$x \in \text{Ide}$	identifiers
$e \in \text{Exp}$	expressions

Abstract Syntax

```
e ::= b | x | e e | \x x...x->e
      | let x=e;...; x=e in e
      | letrec x=e;...; x=e in e
```

図 1: 関数プログラム

本稿では、従来から手続き型言語に対して行なわれていた共通部分式の重複削除の方法を関数型言語に応用し、検索の鍵として型情報を用いることによって、共有解析自体の効率を上げる実験について報告する。

また共有解析の中で極大自由式 (maximum free expression) を検出できる点を利用して、遅延評価系で完全遅延性を実現するためのプログラム変換算法を埋め込む手法も提案されている [7]。これに対応して共有解析の算法に手を加えることによって、完全遅延のプログラムへの変換も容易に行なうことができることを示す。

2 関数プログラム

本稿では図 1 に示すような関数プログラムを対象とする。式は定数、変数、複合式、ラムダ式、局所定義式のいずれかに分類される。局所定義式は、さらに再帰的な letrec 式と非再帰的な let 式に分かれる。

関数プログラムに対する共有解析も手続き型プログラムに対する共通部分式の削除と同様に、現在注目している部分木が既に出現したかどうかを調べることによって共通部分式を検出する。

ここで注意すべき点は、手続き型と関数型のプログラムに対する二分木表現は異なるものを用いている、ということである。手続き型プログラムに対する共有解析では、式に対する二分木の表現として、葉に変数や定数、内部節 (internal node) に演算子を対応させるものが広く使われてきた (図 2 (a) 参照)。この表現法では内部の節に演算子が対応しているので、これを検索の鍵として用いることが

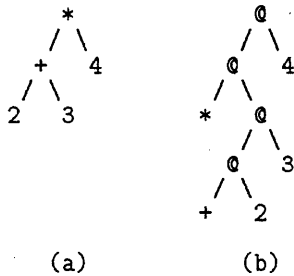


図 2: $(2 + 3) * 4$ の二分木表現

できる。

一方、関数型言語ではカーリー化 (currying) に対応させるため、これとは異なる木の構造を用いるのが便利である。葉には定数や組み込み関数や変数を、内部節には関数を引数に適用した式に対応させる (図 2 (b) 参照)。ここで 0 は適用節 (application node) を表わす。ラムダ式は左の部分木として「ラムダ変数」、右の部分木に本体の式に対応させるような節にする。また局所定義式については特に木構造は仮定しない。これは各局所定義の右辺と本体については木構造を作り他の部分と同一性を調べるが、局所定義式同士の共有は考えないからである。この構造によると内部の節には何も情報が付加されないで、これを素直に実装すると、木の内部節の個数を N としたとき $O(N^2)$ の比較が必要になる。以下では、部分木に固有な情報を与えておき、これを検索の鍵として用いることによって、この手間を軽減させる手法を考察する。

3 共有解析の算法

関数プログラムに対する共有解析は、以下の処理を順次施すことによって実現される。すべての処理はボトムアップに行ない、ここでは各節に対する動作を説明する。

3.1 一変数のラムダ式への分割

ここでは多変数のラムダ式を、入れ子にした一変数のラムダ式に分割する。これは完全遅延評価のための変換は各ラムダ変数を個別に扱う方が効率の良いプログラムを生成できるためである。

3.2 レベルの割り当て

各ラムダ変数はレベル、すなわちその外側のラムダ抽象の入れ子の数によって識別することができる。部分式のレベルが 1 であるということは、その式が依存しているラムダ変数のレベルが高々 1 ということで、すべての部分式に対してそのレベルを計算することによって、最外からみてどこまでのラムダ変数に依存しているかを決定できる。これは極大自由式の検出に必要であると同時に、共有性の検出の際に節に付加する情報としても利用できる。

ここからは式を二分木の構造としてとらえ、それぞれのパスにおいて各節に情報を注釈付けしていく。レベルを求めるためには、まず各部分式に対して自由変数 (free variable) の集合を求める。自由変数とは、その式の外側で値が定まる変数である。その後、自由変数の集合からレベルを求め節に割り当てる。レベルは定数を 0、組み込みの変数を 1 とし、最も外側のラムダ変数には常にレベル 2 を割り当てるものとする。

3.3 節変数の割り当て

二分木のすべての葉と内部節に固有な変数を割り当てる。これを節変数と呼ぶことにする。共有解析をした際に、既に出てきた節 a に対して節 b が共通であることが判明したとき、 $\text{letrec } a = \text{expr in } \dots$ と新しく局所定義を作り、 b の出現は a で置き換える。

3.4 変数名の付け替え

局所定義式に対して、局所定義の右辺にあてられた節変数を新たに局所変数名として名前を付け替える。これにより局所変数名と局所定義の右辺の式の節変数が一致すると共に、局所変数名はすべて一意なものになる。3.6 節では、すべての局所定義 (共有された部分式から生成されたものも含む) は一度別の場所に保存し、ラムダ式の外側に越えない所までまとめて掃き出すので、局所変数名は相異なるものしておく必要がある。

またラムダ変数については、そのラムダ式

の内側に存在するラムダ式の入れ子の深さの最大値に応じて名前を付け替えることによって、特別な場合のラムダ式の共有を調べることができるようにする。

3.5 共通部分式の検出

本節では、内部節と節変数は一対一に対応するので、節変数も節と呼ぶことにする。

本節の処理は、与えられた式の中で共通部分式を検出し、同一の式が既に出ていた節（以下、共有節と呼ぶ）と、その元となる節（被共有節と呼ぶ）の環境を返す。このために、既に調べた部分式を登録しておくための表を用いる。共有性をみるためには、注目している節と左右の子の節の三つ組を表の中に保存しておけば良い。この表の中のすべてと現在注目している節を比較するのは無駄が多く、内部節の総数を N とすると $O(N^2)$ の手間がかかる。そこで各節に固有な情報を、検索の鍵として用いることによって、候補を絞ることをする。ここでは 3.2 節で節毎にレベルを割り当てているのでこれを利用する。以下節 n に対応する式 e に対する場合分けで説明する。

3.5.1 定数または変数の処理

式 e が定数 b または変数 x の場合は、表は変更せずに現在の節を返す。

3.5.2 複合式の処理

式 e が複合式 $e_0 e_1$ の場合は、 e_0 と e_1 を順に処理し、その結果の n_0 と n_1 という節を左右の子としてもつような節を表が保持しているか否か調べる。保持していれば対応する節 n' を返し、後に複合式の節 n' でこの節 n を書き換えられるように環境を更新する。もし保持していなければ (n_0, n, n_1) という三つ組を表に登録し節 n を返す。

3.5.3 ラムダ式の処理

式 e がラムダ式 $\lambda x. e_0$ の場合は、まず e_0 を処理する。その結果を右の子、 x にラムダ変数だというタグを付けた節を左の子として表を検索する。以後は複合式に対する処理

と同様である。この処理により e_0 が x に依存しないような特殊なラムダ式の共有性は判別できるが、一般のラムダ式は今の所共有性を検出できない。

3.5.4 局所定義式の処理

式 e が局所定義式の場合、再帰的か非再帰的かを問わず同一の処理を行なう。各局所定義の右辺について逐次的にこの処理をし、環境と表を更新していく。最後に本体を処理し、本体に対応する節を返す。

3.6 共通部分式の局所定義化

前節で求めた環境を用い、共有している部分式を新たに局所定義として作り出す。共有性の判明により新しくできたものも含め局所定義はすべて、一度別の場所に保存しておく。このとき局所定義の右辺の式のレベルに応じて分類して保存され、レベルの対応するラムダ式の内側で局所定義式として展開される。各節に対して、被共有節が現われた時は節変数を局所変数として新たに局所定義を作り、別の場所に保存する。共有節が現われた時は、対応する被共有節で置換する。

4 型を用いた共有解析

4.1 型推論

ここで対象にする関数型言語は、強い型決め (strong typing) が可能であるものとする。強い型決めとは、式の値がそれを構成している式の値にのみ依存するのと同様に、型も構成要素である式の型によって決まるということである。この考え方により、型を適切に割り当てることのできない式は正しく定義されているものとは認められず、入力段階でエラーとなる。従って強い型決めのもとで型推論を行なうことによって、計算機は多くの論理的な誤りを発見することができるので、構造のきちんとしたプログラムの設計に役立つ。また実行時に式の意味を誤って解釈することなくなるので、言語の実装が容易になるという利点もある。

型推論のアルゴリズムは [3] の第 8,9 章のものをういた。このアルゴリズムは Milner の型付けに基づいており、多様型 (polymorphic type) にも対応している。

4.2 型を用いた共通部分式の検出

3.5 節の共通部分式の検出の際に型情報を用いて効率の良い検索をすることを考える。そのためにはまず前節の型推論を用いて、各節に型を注釈付けする。そして節変数の三つ組を表に登録するとき、レベルと型の二つで分類することによって、共有の対象となる候補を大きく絞ることができる。

すべての節に対して左右の子に一意となるような節変数を付けているので、その二つの変数名からハッシュ値を計算し表への登録・検索に用いても同様の効果が得られる。この方法に対し型情報を用いる場合の利点は、

- 型推論によりプログラムの論理的な誤りを検出できる
- 表検索だけが目的のハッシュ関数を考えなくても良い
- 可換・結合法則なども考慮した共有を調べるとき利用できるかもしれない

などが挙げられる。一方、不利な点は、ハッシュ関数を用いる場合に比べ、型推論は非常に重い処理なので、型の検査を不要とする場合はかえって時間がかかることである。

5 完全遅延評価に適した共有解析

5.1 完全遅延評価への変換

遅延評価系で完全遅延評価を実現するためのプログラム変換算法は幾つか提案されている。その何れも基本的な操作は、入力式に対して極大自由式を検出して、それが局所定義や関数引数となるように式を変形すること、および局所定義が自由である限りラムダ式の外側に移動すること、からなる。この結果、その中の変数すべてが束縛されたときに直ちに極大自由式も束縛されるので、変換後のプログラムを遅延評価すれば、もとのプログラム

を完全遅延評価することになる。このような完全遅延評価用のプログラム変換算法を、一般的な共有解析の算法に対して埋め込む手法が [7] で提案されている。

5.2 ラムダ巻き上げを埋め込んだ共有解析

共有解析の算法中にラムダ巻き上げを埋め込むには 3.5 節の共通部分式の検出の処理だけを変更すれば良い。ここで極大自由式に対する節を、実際は共有されていなくても被共有節として環境に登録することによって 3.6 節で局所定義を生成し、適切なレベルまでラムダ式の外側に巻き上げられることになる。ある自由式 e が極大自由式となるのは、複合式 $e\ e'$ または $e' e$ において e' が束縛されている時か、ラムダ式の本体、局所定義式の局所定義の右辺または本体に出現するときである。また複合式 $e_0\ e_1$ において e_0, e_1 の両方が自由であってもレベルが異なれば、レベルの低い方を極大自由式と同等に扱うことにする。

6 評価

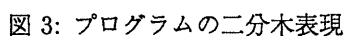
6.1 簡単な例

例として、リスト l 、自然数 n 、述語 p をとり、 l の要素のうち p を満たすものを、先頭から n 個集めてできるリストを返す関数 f_0 を考えよう。

```
f0 = \l->\n->\p->
  if(or(eq n 0)(null l)) nil
  (if(p(head l))
    (cons(head l)(f0(tail l)(sub n 1)p))
    (f0(tail l)n p))
```

この関数 f_0 を二分木で表現すると図 3 のようになる。内部節に対して (i, j) とあるのは、 ξ_i が節に対する固有の節変数、 j がその節のレベルであることを意味する。また各節と型の対応は表 1 に示す。

表 1 より共有が無い場合、レベルだけによる分類では $(1+\cdots+8)+(1+\cdots+10)+(1+\cdots+7) = 119$ 回の比較が必要である。一方、型情報も用いた分類では $3 \times 1 + 2 \times (1 + 2) +$



レベル	1	2	3	4
[a]→Int→	f			
(a→Bool)→[a]				
Int→(a→Bool)		14,22,28		
→[a]				
(a→Bool)→[a]			17,23,27	
[a]		13,21		18,19,24,25,26
[a]→[a]		12	7	20
[a]→[a]→[a]			6	10
Bool		4	2,5	9
Bool→Bool			3	
Int→Bool			1	
a		8,11		
Int			16	
Int→Int			15	
計	1	9	11	8

関数	1 回目	2 回目
F_0	366	365
F_1	361	330
F_2	361	314

表 2: 変換した関数の簡約段数

$(1 + 2 + 3 + 4) = 19$ 回の比較で済む。この関数の例では二箇所共有しており、レベルだけによる分類では 102 回、型情報も用いた分類では 18 回の検査が必要であった。

この関数 f_0 に、共有解析を適用すると以下の関数 f_1 を得る。

```
f1 = \l->letrec ξ8=head l;
      ξ14=f1(tail l) in
  \n->\p->if(or(eq n 0)(null l)) nil
      (if(p ξ8)
        (cons ξ8(ξ14(sub n 1)p))
        (ξ14 n p)))
```

またこの関数 f_0 に、ラムダ巻き上げを埋め込んだ共有解析を適用すると以下の関数 f_2 を得る。

```
f2 = \l->letrec ξ4=null l; ξ8=head l;
      ξ12=cons ξ8; ξ14=f2(tail l) in
  \n->letrec ξ7=if(or(eq n 0)ξ4)nil;
      ξ17=ξ14(sub n 1);
      ξ23=ξ14 n in
  \p->ξ7(if(p ξ8)(ξ12(ξ17 p))(ξ23 p))
```

ここで第 1 引数を固定した関数 $F_i = f_i$ [1..10] ($i = 0, 1, 2$) に対して、`sumList ( $F_i$  10 even)` を評価した簡約段数は表 2 のようになる。

1 回目の簡約で F_0 より F_1, F_2 の方が 5 段少ないのは `head l` の計算が 5 回省かれたためである。2 回評価しているのは、固定された引数に対して計算が進行している様子を確かめるためで、 F_2 は可能な限りの部分計算 (partial computation) を進めたことに相当する。したがって F_1 より F_2 の方が 2 回目の簡約段数が減少している。

	検索回数	簡約段数 [$\times 10^6$]	消費セル数 [$\times 10^6$]
sa	5454	1.21	2.39
salh	5454	2.22	3.88
sat	577	5.63	7.40
satlh	577	6.64	8.89

表 3: 消費された資源

6.2 大きなプログラムの変換

ここでは [8] で紹介されていたベンチマーク用のプログラムの 1 つに対して、共有解析を行なってみた。対象としたのは、マスタースレーブ型のフリップフロップをシミュレートするプログラムで、元は関数型言語 HASKELL で書かれていたものを本稿での仕様に合わせ 124 行のプログラムに変換して実験した。具体的には以下の 4 通りの変換を行ない、部分式の表に対する検索回数とその変換に要した簡約段数と消費セル数を調べた。

sa 型情報を用いずラムダ巻き上げもしない
salh 型情報を用いずラムダ巻き上げはする
sat 型情報を用いてラムダ巻き上げはしない
satlh 型情報を用いてラムダ巻き上げはする

この表 3 より、共有部分式の検出のとき既に調べた部分式との比較の回数は、レベルに加え型を用いることによって、約 10 % に減らすことができた。その反面、簡約段数や消費セル数は 3 ~ 5 倍になっており、型推論の処理が非常に重いことを示している。

7 おわりに

関数プログラムに対する共有解析の算法を、簡潔なパスに分解し、さらに共有性の検出の際に型情報を用いることによって表検索の手間を軽減させた。また大きなプログラムを実際に変換することにより、その有効性を検証した。

今後の課題としては、型情報を用いたより高度な変換が考えられる。これは和や積などに対して可換・結合などの法則を考慮して共有性を調べるということである。

また Milner の型付けでは、多様型の性質により同じ形の式でも型が違ふことがありうる。これを型の間に半順序を入れるなどして、検索の鍵として対応させることが考えられる。

```
pair ((\x->cons x nil)1) ((\y->cons y nil)True)
      ↓
letrec f = \z-> cons z nil in pair (f 1) (f True)
```

以上の例では型が $\text{Int} \rightarrow [\text{Int}]$ と $\text{Bool} \rightarrow [\text{Bool}]$ の2つのラムダ式をまとめて型が $a \rightarrow [a]$ のラムダ式にしている。

謝辞

本研究の一部は文部省科学研究費補助金(奨励研究 (A) 05780231: 完全遅延評価による関数プログラムの部分計算)の援助による。

参考文献

- [1] A. V. Aho, J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.
- [2] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley. 1986.
- [3] S. L. Peyton Jones. *The Implementation of Functional Programming Language*. Prentice-Hall. 1987.
- [4] S. L. Peyton Jones, D. Lester. A Modular Fully-lazy Lambda Lifter in HASKELL. *Software—Practice and Experience*, Vol. 21(5), pp. 479–506, 1991.
- [5] R. Bird, P. Wadler. *Introduction to Functional Programming*. Prentice-Hall. 1988. (武市正人訳: 関数プログラミング, 近代科学社. 1991.)
- [6] M. Takeichi. Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs. *New Generation Computing*, Vol. 5, pp. 377–391, 1988.
- [7] K. Kaneko, Y. Onoue, M. Takeichi. Sharing Analysis of Functional Programs for Fully Lazy Evaluation. *Technical Report, METR 93-10*. Faculty of Engineering, University of Tokyo.
- [8] P. H. Hartel, K. G. Langendoen. Benchmarking implementations of lazy functional languages. *ACM Conf. FPCA '93*, pp. 341–349.