

遅延評価系の関数型言語の静的なゴミ集めについて

田中 聡, 寺島 元章

qtanaka@cs.uec.ac.jp, terashim@cs.uec.ac.jp

電気通信大学 情報工学科

概要

関数型言語では動的なメモリ管理が必須であり、不要な記憶領域を回収するいろいろな方法が考えられてきた。本稿では遅延評価の関数型言語に対する静的な解析を用いて、翻訳時にデータ構造の共有性の解析を行ない、不要となったリストセルを即時回収／再利用する方法を提案する。また他の解析と組み合わせることによってより多くのリストセルを回収できることを示す。

Compile-time garbage collection for lazy functional languages

Satoshi TANAKA, Motoaki TERASHIMA

qtanaka@cs.uec.ac.jp, terashim@cs.uec.ac.jp

Department of Computer Science

The University of Electro-Communications

Abstract

Dynamic memory management is an important technique for functional languages. There is many techniques to reclaim unnecessary storage cells. This paper describes a method to analyze lazy functional programs at compile-time by static analyses and to reclaim/reuse immediately the garbage cells. We discuss that more garbage cells can be reclaimed by combining with other static analyses.

1 はじめに

純粋な関数型言語では破壊的代入がないためにデータ構造を多用する傾向にある。この問題を解決するために従来の処理系ではゴミ集めと呼ばれる自動的な記憶管理ルーチンに一括して記憶領域管理を任せていた。しかしこれが処理系の実行効率が落ちたり、本来の仕事を中断させる原因となっている。

しかし近年プログラムの静的解析の技術が上がるにしたがって多くの情報が翻訳時に集めることができるようになってきた。[2] では引数を値渡しする関数型言語で翻訳時にリストセルを回収／再利用するアルゴリズムが提案されている。本論文ではこのアルゴリズムを遅延評価の関数型言語のプログラムに対して適用できるように変更し、その際問題となる点について明らかにする。また他の解析による最適化を併用することによっていくつかの問題が解決されることを示す。

まず2節で対象となる関数型プログラムについて定義し、次の3節で静的解析について説明する。4節では[2] のアルゴリズムを遅延評価の関数型プログラムに対して拡張する。またその際に問題となる点について明らかにし、それを解決する方法について述べる。5節では他の仕事の比較と今後の課題について述べる。

2 対象プログラム

まずこの節では解析の対象となる遅延評価の関数型言語のプログラムについて説明する。

2.1 データ型

データ型は整数、真偽値などの原始型とそれらを要素として持つことのできるリストを考える。

リストはデータの順序付けられたつながりを表現するデータ構造である。リストは二つの要素からなるセルと呼ばれるデータ構造によって実現される。それぞれの要素は head 部、tail 部と呼ばれる。空リストは $[]$ で表し、空でないリストは $[1, 2, 3]$ のように表す。リスト x の先頭に一つ要素 a を付け加えるには関数 $cons$ を使って $cons(a, x)$ とする。またこのリストを特に $[a|x]$ と表す。またリスト x の head 部を取り出すには $head(x)$ とし、リスト x の tail 部を取り出すには $tail(x)$ とすればよい。

2.2 関数型言語

関数型言語は状態に依存せずに関数の値が引数の値だけから決定されるという特徴を持つ。またデータ構造に対する破壊的操作を行なう代入文などを持たない。今回高階関数については考えない。関数は常に十分な数の引数が与えられるものとし、関数の引数に関数自体を渡すことはない。

文法は次の定義によって与えられる。

```
program ::= { $f(x_1, \dots, x_n) = expression$ } *      (プログラム)
expression ::= constant      (定数)
               | variable    (変数)
               | if  $e_1$  then  $e_2$  else  $e_3$       (条件式)
               |  $f(e_1, \dots, e_2)$       (関数適用)
constant ::= 整数、真偽値
variable ::=  $x, y, z, \dots$ 
```

関数適用の項の f にはプリミティブ関数の $cons, head, tail, eq, +$ など含まれる。

2.3 遅延評価

本論文では遅延評価によって実行される関数型プログラムについて解析を行なう。遅延評価とは、関数の適用を行なう時に実引数を評価せずに関数閉包のまま引き渡し、その値が必要になった時に始めて評価を行なう方法のことをいう。例えばリストセルを生成する時には関数 *cons* の実引数を評価せずに、それらを関数閉包としてリストセルに格納する。この評価が遅延された式は関数 *head*, *tail* などによってその値を取り出して使用しようとした時に始めて評価される。この遅延評価によって無限リストを使ったストリーム計算を行なったり、無駄な式の評価を避けることができる。また実引数を評価してから関数に渡す方法は値渡しによる評価といわれる。[2] ではこの値渡しによる評価を仮定している。

3 静的解析

プログラムによって生成されたデータ構造がいつ不要になるか解析するためには、与えられたデータ構造が式を評価した結果に含まれるか、他の式から参照されていないか、などを調べる必要がある。遅延評価を行なう関数型言語では式を評価し終った結果にデータ構造が含まれている場合だけでなく、データ構造を含んだ式が遅延評価され持ち運ばれる可能性が出てくる。

関数型言語にこのような解析を行なうにはいろいろな方法が存在する。ここでは解析の対象をより簡単な性質に対応させ、検出したい性質に対応した有限の抽象領域を与えて、その上で最小浮動点計算を行なうことによってプログラムの近似的な性質を得る方法について説明する。これは抽象解釈と呼ばれる方法で [2] でも用いられている解析技術である。

[2] では次のような抽象領域 P が与えられている。

$$P = \{0, [0|0], [0|1], [1|0], [1|1], 1\}$$

この抽象領域の 0, 1 はプログラムのどんな性質を考えているかに従って意味が与えられる。例えばリストの必要性を考える場合には、1-リストの全体が必要、0-リストを必要としない、[1|0]-リストの head 部だけ必要、となる。また領域の大きさが有限でないと再帰関数に対する再小不動点計算が停止しないので、head 部の一部でも必要な場合には [1|0] を用いる。このため求められる性質は真の性質に対して近似的なものとなっている。

例えば head 部のみ必要なリスト同士 (性質は [1|0]) を *cons* したリストの性質は [1|1] と近似的に表す。また 2 箇所が必要なリストについては要素ごとに boolean or することによって合成された必要性が得られる。($[a|b]$ かつ $[c|d]$ なら $[(a \cup b)|(c \cup d)]$) となる。

4 アルゴリズム

この節では [2] で定義された翻訳時のゴミ集めのアルゴリズムを、遅延評価の関数型プログラムに対して解析を行なうように変更する。

まず始めに [2] で用いられているアルゴリズムについて説明する。

4.1 [2] のアルゴリズム

[2] では引数を左から右の順で評価して値渡しする関数型言語を対象とし、[1] で用いられた表示的意味論による formal semantics を拡張したものを使ってアルゴリズムを定義している。この formal semantics ではプログラムの通常の意味としての値だけではなく、リスト構造を実現するための内部記憶についても表現し、意味付けできるようになっている。

そこでは以下に述べるような方法によって翻訳時のゴミ集めを実現している。Formal semantics では関数に対して各引数ごとに次の二つの情報が与えられる。

- すでに評価済みの式と共有している部分
- まだ評価の済んでない式を評価する時に必要な部分

この二つの情報を引数ごとに環境として持つ。環境には引数名と共有性や必要性を表すリストパターン (3 節の抽象領域 P の要素) が組になっている (それぞれ she (共有性)、 use (必要性) とする)。この環境を持ち運ぶことによって関数や引数間で情報を伝搬している。

この情報は $f(e_1, \dots, e_n)$ という関数適用を解析する場合に次のように用いられる。それぞれの引数 e_i に対し

- すでに評価された引数の結果と引数 e_i が共有している部分を解析する。これをこの関数適用の外側ですでに評価した式に対する解析の結果と (後者はこの関数適用の意味を表す意味関数に引数として与えられている) 合成して she_i を作る。
- この関数適用のまだ評価していない引数を評価する時に必要なリストで、この引数 e_i に含まれる部分を解析する。これをこの関数適用の外でまだ評価されていない式に対して (後者はこの関数適用の意味を表す意味関数に引数として与えられている) 同様の解析を行なったものと合成して use_i を作る。
- 上の she_i, use_i と前の引数 e_{i-1} を評価した後の内部記憶の状態を用いて e_i を解析する。

を行なう。

これを関数 f に与えられた実引数 e_1 から e_n まで (左から右へ) 順に繰り返す。

こうしてできた環境 she, use と内部記憶の状態によって f 本体に対する解析を続行する。2 節で定義した他のプログラム構成要素についても同様な解析が行なわれる。

4.2 アルゴリズムの変更

上のアルゴリズムを遅延評価の関数型プログラムに対して適用できるようにする。遅延評価を行なう場合には引数は評価されずに関数に渡されるので次のような変更を行なえばよい。

- 共有性を表す環境 she は外側の式から与えられたものをそのまま使う。
- 必要性を表す環境 use は他の引数すべて (まだ評価されていないので) を評価する時に必要なものを使う。

このようにして解析を行なえば遅延評価の関数型プログラムに対しても安全に静的なゴミ集めを行なえる。しかしこの方法は関数閉包が評価の時期について最悪の場合を見積もっている。つまり関数閉包はいつまでたっても評価されずにそのまま存在し続ける場合を想定している。これでは一つの値が二つ以上の引数に現れただけで、それは永遠に共有され続けるものとされてしまう。実引数間 (あるいはその部分式) でどのような順序で評価が起こるか知ることができればこの問題は解決できる。しかし、関数の実行によって実引数に対して起こる要求駆動の関係を調べるのは非常に困難である。

そこで本論文では引数の必須性解析を用いた最適化を用いてこの問題を解決する。多くの引数は関数閉包として渡されても関数が評価されれば要求駆動によってすぐに評価されることになる。この場合最適化要求駆動によって関数適用が行なわれるようにし、引数の評価をあらかじめ行なってしまえばよい。最適化要求駆動による評価とは、プログラムに出てくる関数に対して引数の必須性の解析を行ない、その値が必須である引数に対しては先行評価を行ない、評価の結果求められた値を関数に対して渡してやることをいう。遅延評価の関数型言語の処理系では関数閉包に対する要求駆動の伝搬が実行時のオーバーヘッドになるために最適化要求駆動による最適化が行なわれことが多い。

この最適化を用いれば、必ず評価される引数は関数適用時に評価されるため、この引数に対する共有性と必要性をより詳しく解析できるようになる。この変更を行なったアルゴリズムを部分的に Appendix A に付ける。このアルゴリズムによって遅延評価の関数型プログラムを安全にまたより詳しく解析することができる。

4.3 問題点と解決法

上で述べた遅延評価の関数型プログラムに対する解析は次のような不都合がある。

```
append x y = if null(x) then y else cons(head(x), append(tail(x), y));  
reverse x = append(reverse(tail(x)), cons(head(x), []));
```

のようにプログラムが定義されているとする。

まず始めに値渡しの評価系で実行する場合を考える。関数 *append* を呼び出す時には、引数 *x* を評価する時点で一番先頭のリストセルが分解される。引数の評価が終ると関数が呼び出されリストセルを生成する関数 *cons* が実行される。渡された引数 *x* の先頭のリストセルは他には使われていないので、関数 *append* では分解だけが行なわれる。この時、*append* に共有されていないリストを渡していれば、渡された引数 *x* の先頭のリストを破壊的に変更することによってリストセルを消費せずに *append* を実行することが可能である。関数 *reverse* についても同様なことがいえる。

しかし遅延評価でこの関数を実行する場合には、*cons* はどちらの引数も評価せずに関数閉包のままリストセルに格納する。したがって引数 *x* の先頭のリストセルが分解されずに関数 *reverse*, *append* を評価した結果に含まれてしまう。

このため関数 *reverse*, *append* で引数の関数閉包に含まれたリストセルを回収するためには、引数 *x* を分解する二つの式 *head(x)*, *tail(x)* の評価される順序を知ることができなければならない。そうすれば後から評価される式で引数 *x* の先頭リストセルを回収することが可能である。しかし、評価が遅延されているリストの一部分がいつ評価されるかを知るのは難しいし、多くの場合はどちらともいえない。

このようなセルの回収を行なうことができるように、関数 *append* に対して次のようなプログラム変換をおこなう。

```
append x y = if null(x)  
  then y  
  else let  h = lazy_head(x);  
           t = lazy_tail(x);  
           in cons(h, append(t, y));
```

(*lazy_head*, *lazy_tail* はリストの head 部、tail 部を評価することなく取り出す関数である)

この変換を行なえば関数 *append* は条件式 *null(x)* を評価した結果が *false* ならば引数 *x* を分解してやることができる。また引数 *x* は条件式 *null(x)* を評価する段階ですでに参照されているので、この変換を行なうことで関数 *append* の non-strict 性 (関数に未定義の値を渡しても必ずしも未定義になるとは限らない) が損なわれることはない。未定義の値が引数 *x* 渡された場合には変換以前／以後どちらの関数も値は未定義となる。ただしこの場合分解されたリストの要素に対して参照が発生するとは限らないので、リストの分解には要素の評価を引き起こさない *lazy_head*, *lazy_tail* を使う必要がある。また引数に対する参照が必ずしも発生しない場合には、関数の non-strict 性を損なうのでこの変換を行なってはならない。

以上の分析は必須性解析によって容易に行なうことができる。また *reverse* に対しても関数間に渡って解析を行えば、関数 *append* の性質からリストセルの回収ができるプログラムに

変換できることが分かる。この変換によって関数 *append, reverse* では引数 x に渡されたリストが共有されていない場合、そのリストのセルを 100% 即時再利用してやることができる。

以上のように必須性解析を用いれば、このプログラム変換によって関数の性質が変わることはない。また関数型言語では、引数に渡される構造をあらかじめ分解することはよく行なわれる。多くの言語にパターンマッチ渡しの機能があるためである。この場合、プログラムに対して解析を行なくてもあらかじめ分解が行なわれるプログラムとなっている。

5 まとめ

本論文では [2] のアルゴリズムを遅延評価の関数型プログラムに対して拡張する方法について述べ、その時に問題となる点について明らかにし、またその問題を解決する方法について述べた。それによって関数に対して必須性の解析を用いてプログラム変換を行なうことによって、遅延評価の関数型プログラムに対しても翻訳時の静的なゴミ集めが有効に働くことが分かった。

同様の仕事としては次のようなものがある。Hudak による配列に対する Hoare の quick sort に対する解析 [1]。この解析の結果、一回も配列のコピーを行わずにプログラムを実行している。鶴岡らによる計算経路解析を用いた配列に対する破壊的変更の副作用解析 [4]。計算経路解析を用いているために値の共有性などにより強力な解析力を持つ。

今後の課題として次のことがあげられる。

- 要求の伝搬による関数閉包が評価される時期について、より詳しく解析する。現在関数閉包になった式についてはいつ評価されるか分からないという立場をとっている。評価される時期が分かれば共有性の解析をより詳しく行なうことができ、さらに多くのセルを回収できるようになる可能性がある。
- データの生成と死滅が行なわれる時期について解析をし、セルを生成した関数よりデータ構造そのものの方が寿命が短い時にはスタックに領域をとってセルの寿命の違いによる記憶領域のフラグメンテーションを抑える。また generation scavenging [3] などのデータ構造の寿命によって管理する方法を変える実行時支援のゴミ集めに対して有効な情報を与えることもできる。
- 高階関数や双方向リストやリングなど自己再帰なデータも扱えるようにする。

また実際の程度の回収／再利用が行なえるかを詳しく調べるために、処理系を作成してより経験を深めることが必要と思われる。

References

- [1] Hudak P. "a semantic model of reference counting and its abstraction". In Hankin C. Abramsky S., editor, "Abstract interpretation of declarative languages". Ellis Horwood, 1987.
- [2] Simon B. Jones., Daniel Le Métayer. "compile-time garbage collection by sharing analysis". In *Proceedings of the ACM Symposium on Lisp and Functional Programming, 1989*, pages pp.54-74, 1989.
- [3] David Ungar. "generation scavenging: A non-disruptive high performance storage reclamation algorithm". *SIGSOFT/SIGPLAN*, pages 1157-1167, April 1984.
- [4] 鶴岡行雄 and 小野諭. "関数型言語の並列要求駆動型計算における副作用". 情報処理学会論文誌 Vol.30 No.12, pages pp.1538-1546, 1989.

Appendix A

関数適用に関する部分だけ抜き出してある。

$$\begin{aligned}
 & \mathcal{R}'[f(e_1, \dots, e_n)] \text{ strict fve bve she use st free allo} = \\
 & \text{let} \\
 & \quad \vdots \\
 & \quad she_i = [((she\ x) \cup_{j \in Strict(f)[1, \dots, i-1]} T[e_j][x]1)/x] \\
 & \quad use_i = [((use\ x) \cup_{j \in Lazy(f)[1, \dots, i-1, i+1, \dots, n-1]} \mathcal{N}[e_j][x]1)/x] \\
 & \quad (v_i, st_i, free_i, allo_i) = \mathcal{R}'[e_i] \text{ strict}(f, i) \text{ fve bve she}_i \text{ use}_i \text{ st}_{i-1} \text{ free}_{i-1} \text{ allo}_{i-1} \\
 & \quad \vdots \\
 & \text{in } (fve[f])\ v_1 \dots v_n\ st_n \\
 & \quad (S[e_1][((she\ x) \cup_{i \in Strict(f)[2, \dots, n]} T[e_i][x]1)/s]) \\
 & \quad \vdots \\
 & \quad (S[e_n][((she\ x) \cup_{i \in Strict(f)[1, \dots, n-1]} T[e_i][x]1)/s]) \\
 & \quad (S_t[e_1][((use\ x) \cup_{i \in Lazy(f)[2, \dots, n]} /x)] \dots \\
 & \quad (S_t[e_n][((use\ x) \cup_{i \in Lazy(f)[1, \dots, n-1]} /x)] \text{ free}_n \text{ allo}_n
 \end{aligned}$$

$\mathcal{R}'$ は第一引数に渡された式の意味を与える意味関数。
各引数の意味は次の通り。

- strict* : この式が先行評価されるかどうかを表すフラグ。
- fve* : 関数名と関数の実体を結び付ける環境。
- bve* : 変数名と変数の実体を結び付ける環境。
- she* : 式の間のリスト共有パターンを登録する環境。
- use* : 式を評価する時のリスト依存パターンを登録する環境。
- st* : 内部記憶 (リスト構造を保持する)。
- free* : 内部記憶の中で不要なものの集合。
- allo* : 内部記憶の中で有効なものの集合。

また n 引数の関数 f に対して

$Strict(f)[1, \dots, n]$ は必須引数のインデックスの集合を、

$Lazy(f)[1, \dots, n]$ は必須でない引数のインデックス集合を与えるものとし、

$strict(f, i)$ は f の i 番めの引数が必須ならば 1、そうでなければ 0 を返すものとする。