成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.53 No.2 (2016)pp.21-24

関数型プログラミング言語における遅延評価機構

高野 保真[*1]

Lazy Evaluation of Functional Programming Languages

Yasunao TAKANO[*1]

ABSTRACT：Lazy evaluation is an evaluation strategy in programming languages. Lazy evaluation delays the evaluation of an expression until its value is needed. With lazy evaluation, there are some advantages. One of the advantages enables to avoid unnecessary evaluations. Another advantage enables to help programmers write clear programs. In case of linked list, programmers can divide functions into two kinds, generation of list and consume of list. In this paper, we introduce an overview of lazy evaluation and our continuous research for lazy functional programming languages.

Keywords：Programming languages, Functional programming, Lazy evaluation

(Received October 21, 2016)

1．はじめに

複雑なソフトウェアを開発するために，プログラミング言語が提供する抽象化機構は，ますます重要になってきている．その点に注目して，プログラムのモジュール化や記述性を重視した関数型プログラミング言語の研究が進み，関数型プログラミング言語を用いてプログラミングを記述する**関数プログラミング**が盛んになっている．

関数プログラミングでは，処理を関数という単位に分けて定義し，関数を組み合わせることで，より高い抽象度でプログラムを記述する．以下に，関数プログラミングの主な特徴を挙げる．

- 宣言的な記述
  数式のように，関数を宣言的に記述する．つまり，計算機が何をするかという手続きを記述するのではなく，プログラムで解く問題の性質を記述する．
- 副作用のない関数
  変数への代入のような副作用がなく，関数は参照透明（referencially transparent）である．
- 抽象データ型による処理
  処理の対象を抽象データ型として，計算機上での実際の表現を隠蔽して扱う．

このような特徴から，関数プログラミングは，**遅延評価**（**lazy evaluation**）と相性がよい．

遅延評価とは，値が実際に必要になるまで計算を遅らせる評価戦略である．必要になった値から計算するため，最終結果を求めるためには不要な計算を除去することができる．どの値が必要となるかという判断を処理系に任せることができるため，プログラマが計算の順序に頼ったプログラムを記述する必要がなくなるという利点がある．

本報告では，関数型プログラミング言語おける遅延評価について紹介するとともに，我々がこれまで行ってきた研究について概要を述べる．筆者のこれまでの仕事を総括する論文である性質上，自分の論文を多く引用している点はご容赦いただきたい．また，それぞれの研究の詳細については，参考文献を参照されたい．

2．遅延評価

2．1　遅延評価とは

遅延評価は，式の評価をその値が必要になるまで遅らせるプログラミング言語の評価戦略である．基本的な遅延の対象は，関数やデータ構成子の引数であり，遅延を表す構文を導入する場合もある．たとえば，引数が遅延

[*1]：理工学部情報科学科助教 (yasunao-takano@st.seikei.ac.jp)

の対象であるとき，ある関数適用の引数に与えられる式の計算は遅延され，遅延された計算は，その値が実際に必要になったときに初めて処理される．

プログラムの実行を要求駆動的に進めることができるため，主に以下のような利点がある．

- 無駄な計算を除去することによる実行効率の向上
- 宣言的な記述を用いることによる記述性の向上

まず，一つ目の項目に関して，最終結果に寄与しない無駄な計算を省くことができれば，C などの手続き型言語に代表される先行評価の言語に比べて，潜在的には実行効率が良い．たとえば，以下のように定義される単純な関数 firstを考える．

```
first :: Int → Int → Int
first a b = a
```

この関数は a と b という 2 つの Int 型の引数を受けとり，第一引数を返す．つまり，第二引数は結果に関係ない．遅延評価に基づく言語では，関数の実引数は遅延の対象であるので，first 0 (heavy 1) という呼び出しがあったときには，実引数の 0 とheavy 1 はすぐには計算されない．ここで，heavy 1 の計算が非常に時間のかかるときには，計算せずに済ませることができ，遅延評価を採用したことによる効果が大きくなる．

次に，二つ目の項目に関して，プログラムの実行順序を処理系に任せることができることにより，プログラムには処理の手順を記述するのではなく，そのプログラムの性質を記述することが可能となる．そのため，プログラムは宣言的になり，モジュール化を促進することができる[1]．特に，この特徴は抽象データ構造を扱う際に有効である．つまり，あるデータ構造に対して，そのデータを生成するプログラム記述と，そのデータを読み進めるプログラム記述を分離でき，プログラムの見通しがよくなる．

遅延評価には多くの利点がある一方で，以下のような欠点も挙げられる．

- 処理を手続き的に追うことは難しい
- 代入などの副作用のあるプログラミング言語と相性が悪い
- プログラミング言語処理系を実現する際に効率のよい実装が難しい

2．2　遅延評価に基づく関数型プログラミング言語

遅延評価に基づく関数型プログラミング言語の歴史は古く，70 年代後半から 80 年代にかけて研究が始まった[2]．1985 年に，Research Software 社のプログラミング言語 Miranda© [3] が現在の関数プログラミングの基礎となる特徴を持った言語であった．Miranda の文法は，等式による関数の定義，パターンマッチング，リスト内包表記（list comprehensions）などの要素により，記述の簡潔さを考えたものであった．

そのMiranda の流れを受けた Haskell [4] という言語が現在最も広く利用されている遅延評価に基づく関数型プログラミング言語である．Haskell のプログラミング言語としての特徴は，遅延評価に加えて，代数的データ構造を用いた高い記述性，静的型付けと型推論による高い信頼性など，多くの理論的背景を持った機能を積極的に採用していることである．

当初は研究的側面が強い言語であったが，最近では，実用的な場面でも Haskell が用いられるようになってきた．たとえば，金融取引に Haskell を用いた例 [5] やシステムの管理に Haskell を用いた例 [6] などがあり，幅広い分野に渡って Haskell が使われはじめている．

3．Improving Sequence

前節で述べた遅延評価の記述性における利点を生かす研究として，我々は Improving Sequence (IS) [7] と呼ばれるデータ構造を利用し，探索問題を記述する研究を行ってきた．

IS とは，ある順序関係の意味において単調で，徐々に真の値に近づいていく近似値の列を表現するデータ構造である．IS 上の近似値が全順序関係の意味において単調であることにより，真の値による結果と同じ結果をある時点での近似値で求めることができ，その場合には，その時点以降の真の値へと至る計算を枝刈りすることができる．

たとえば，文字列 "seikei university" の長さを 1 文字ずつ順に前から数える場合を考える．このとき，まず ‘s’ という文字を数えたときには，この文字列は少なくとも 1 の長さがあると分かるため，1 を全体の長さの近似値と考えることができる．そのため IS は 1 という近似値と残りの文字列 "eikei university" を組とするデータ構造となる．その IS に続けて次の ‘e’ を読めば少なくとも 2 の長さがあると分かり近似値は実際の長さに近づく．つまり，近似値である部分的な長さは < の関係のもとで単調に増加する．この性質を利用すれば，もし上の文字列が 5 より大きいか知りたいときには “seikei” というところまで文字の長さを数えれば，少なくとも 6 文字あり，そこから単調に増えていくだけであるため，残りの文字列の長さを数える必要がない．つまり，すべての文字列を数えずに計算を枝刈りして結果を求めることが

できるのである．

以上の文字列の長さを比較する例は，最も単純な例であるが，同様の考え方で IS を用いることで，巡回セールスマン問題や編集距離や 8 パズルを解くプログラム，探索木を用いたオセロのプログラムなどが簡潔に記述することができる．

IS に関する研究の中でも筆者は，IS を効率よく処理できるプログラミング言語処理系の設計と実装 [8] を中心に研究してきた．論文 [8] においては，プログラムのコンパイル時に IS の単調性を解析したうえで，コード生成をするプログラミング言語処理系を設計し実装した．この研究で得られた着想が，次節で述べる遅延データ構造の遅延に必要なメモリ割当量の削減が可能であるというアイデアのきっかけとなった．

4．遅延評価に基づく関数型プログラミング言語におけるメモリ割当量の削減

遅延評価においては，計算の遅延に必要となるオブジェクト（遅延オブジェクト，以下**サンク**と呼ぶ）を処理系内に生成する．式を評価する代わりにサンクを割り当てるが，このことは時間的・空間的にコストがかかる操作である．たとえ遅延評価により結果に寄与しない計算を除去できるとしても，プログラムの実行時のオーバヘッドが大きくなってしまうという問題点がある．そのため，効率的に遅延評価を行う言語処理系においては，サンクの生成を抑制するために，様々な静的解析手法が開発されてきた．

我々はサンクの削減を目指した一手法として，リストのように線形に定義される再帰的データ構造に注目し，既存のサンクを再利用してサンクの生成を抑える手法（以下，**Thunk Recycling**と呼ぶ）について研究してきた．

Thunk Recycling の目的は，すでに割り当てられているサンクの内容を破壊的に更新して再利用し，新たな遅延オブジェクトの生成を抑えることである．Thunk Recycling を用いれば，たとえば，代表的な再帰的データ構造である線形リストでは，後続のリストの遅延に必要となるサンクを再利用することができる．

また，Thunk Recycling は，サンクの内容を破壊的に書き換えることにより，矛盾が生じないようにする機構も合わせ持つ．具体的には，コンパイル時の静的なプログラム変換により，再利用するサンクへの参照が単一であるようにする．

我々は，以下のような Thunk Recycling に関する一連の研究をこれまで行ってきた．

- Thunk Recycling を提案した．[9]
- Thunk Recycling の形式的な定義を与え，その正しさを証明した．ここでいう正しさとは，Thunk Recycling を適用したプログラムが，適用前のプログラムと同じふるまいをして同じ結果を導くことをいう．[10]
- Haskell の代表的な処理系である Glasgow Haskell Compiler（GHC）上に Thunk Recycling を設計し，実装した．[11]

ベンチマークプログラムによる実験により，再利用のおよぼす影響を確認した．図 1 と 図 2 に結果を示す．

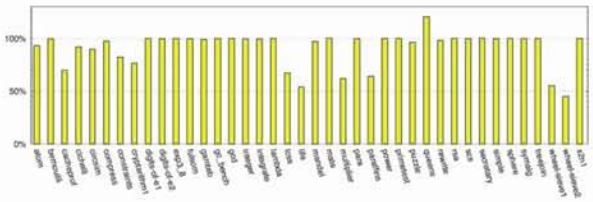

図 1：総メモリ割当量

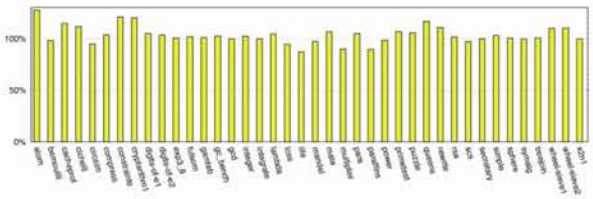

図 2：実行時間

横軸にベンチマーク、縦軸に Thunk Recyclingの導入前の GHC を 100% としたときの値をとっている．総メモリ割当量を見ると多くのベンチマークで効果がみられ，最も効果があるものでは導入前の半分以下の総メモリ割当で実行できた．一方，実行時間の観点から見ると，適用するプログラムを選ぶものであった．

6．むすび

本報告では我々がこれまで行ってきた遅延評価に基づく関数型プログラミング言語に関する取り組みについて紹介した．遅延評価は潜在的な利点が数多くある一方で，まだ課題の多い技術である．そのため，今後のさらなる研究により改善を目指していくべき研究課題である．

参考文献

[1] J. Hughes. Why Functional Programming Matters. The Computer Journal, Vol. 32, No. 2, pp. 98–107, 1989.

[2] P. Hudak, J. Huges, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. Proc. the 3rd Conference on History of Programming Languages (HOPL-III), pp. 1–55. ACM, 2007.

[3] D.A. Turner. Miranda: a Non-strict Functional Language with Polymorphic Types. Proc. the 2nd ACM Conference on Functional Programming Languages and Com- puter Architecture (FPCA 1985), pp. 1–16. ACM, 1985.

[4] S. Peyton Jones. Haskell 98: Introduction. Journal of Functional Programming, Vol. 13, pp. i–6, 2003.

[5] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. Journal of Functional Programming, Vol. 19, No. 1, pp. 27–45, 2009.

[6] I. Pop. Experience report: Haskell as a reagent: results and observations on the use of Haskell in a python project. In Proc. the International Conference on Functional Programming, pp. 369–374. ACM, 2010.

[7] H. Iwasaki, T. Morimoto, and Y. Takano. Pruning with Improving Sequences in Lazy Functional Programs. Higher-Order and Symbolic Computation, Vol. 24, No. 4, pp. 281–309, 2011.

[8] 高野保真, 岩崎英哉. Improving Sequence を第一級の対象とする Scheme コンパイラ. 第 8 回プログラミングおよびプログラミング言語ワークショップ, pp. 153–162. 日本ソフトウェア科学会, 2006.

[9] 高野保真, 岩崎英哉, 鵜川始陽. Glasgow Haskell Compiler における再帰的データ構 造のための遅延オブジェクトの再利用. 情報処理学会論文誌 プログラミング, Vol. 5, No. 2, pp. 67–78, 2012.

[10] Y.Takano, H. Iwasaki. Thunk Recycling for Lazy Functional Languages: Operational Semantics and Correctness. Proc. 30th ACM/SIGAPP Symposium on Applied Computing, pp.2079-2086, Salamanca, Spain. 2015.

[11] 高野保真, 岩崎英哉, 佐藤重幸. Glasgow Haskell Compiler 上の遅延オブジェクトの再利用手法の設計と実装. コンピュータソフトウェア, Vol. 32, No. 1, pp. 253–287, 2015.