関数型言語の実行系についての覚書

数理システム技術レポート

2012 年 8 月 31 日

概 要

これは関数型言語の実行系に関するサーベイである。本レポートでいう関数型言語とは遅延評価による純関数型言語を意味する。関数型言語ははラムダ計算、型理論、カテゴリー理論等の理論的な背景を基に設計されてきたため、理論的には美しい。

一方でそれを効率的に通常のマシン上で実行するためのしくみ - 実行系は極めて複雑でドロ臭い。関数型言語というと、その理論的な美しさにのみ目を向けがちであるが、実際に関数型言語で効率的なプログラムを作成する場合、そういった美しいことを影で実現している実行系の理解も重要である本レポートは、効率的に正規順序評価を行うための実行系の発展について時系列順にまとめた。

1 はじめに

本レポートでいう関数型言語とは以下の二つで特徴付けられる。

1. 参照透過性 (referential transparency)

   破壊的な代入がなく、変数は書き換えられない。

2. 正規順評価 (normal-ordr evaluation)

   遅延評価とも言う。式の外側から内側に評価する。正確な定義はラムダ計算を参考のこと。

3. 強い型付け (strong typing) と型推論

   コンパイル時に型エラーが起きなければ、実行時に型エラーが決して起きないという性質。 正確な定義は [cardelli-85.pdf] にある。型推論は型を明記しなくても処理系が推論する機構であり、Hindley-Milner スタイルの型システム [milner-78.pdf] に基づく。

3. の強い型付けは、本レポートのテーマである実行系とは全く関係がない。実際 1. と 2. を満たし、型付けについては Lisp と同様の動的型付けによる関数型言語も存在する (e.g. SASL)。また、3. の強い型付けに基づくが、1. と 2. を満たさない言語もあり、今でも広く使われている (e.g. ML, Caml)。

言語の特徴として最も際立っているのが 1. の参照透過性である。この参照透過性を守る場合、通常の言語における評価順序 (作用的順序) では、ほとんど実用的なプログラムは書けないことが知られている。一方で、 ラムダ計算的な観点から「正しい」正規順評価を使えば既存のプログラミング言語で出来ること「以上」のさまざまな楽しいプログラミングが可能である [hughes-89.pdf] ことが知られている。この意味で、条件 1. と 2. はセットである。すなわち、参照透過性に拘る限り正規順評価によらざるをえない。

2　ラムダ計算

Landin はラムダ計算とプログラミング言語 (Algol60) の関連について最初に言及した [landin-65a.pdf]。しかし当時ラムダ計算がプログラミング言語の汎用的なモデルとして利用できるかどうかは疑問視されていた。それは、ラムダ式を数学的な意味で関数と解釈出来る空間が存在するかどうかという問題であった。この問題は Scott により解決され [Scott 71]、これから表示的意味論という分野が発展した。

与えられたプログラムに表示的意味を与えるということは、簡単に言えばその与えられたプログラムをラムダ計算という言語で書き換えることに他ならない。そして、このような研究を続けて行けば「元の言語がラムダ計算ならこの作業は要らない」そして、「ラムダ計算ベースのプログラミング言語を作ってしまおう」という発想が自然に出てくる。

このように、ラムダ計算 [Church 41] とコンビネータ理論 [Curry 58] は関数型言語の理論的な背景であり、特にチャーチロッサの定理 (評価結果は一意) および正規化定理 (最外リデックスからの簡約が理論的に良い) は関数型言語の存在を正当化する上でも、またその実行系のデザインにおいても重要である。

ここで一つ強調しておくことがある。ラムダ計算のリダクションを行う実行系を作ること自体は非常に簡単である。特に Lisp のような記号処理向け言語であれば、それは学生の演習問題程度である (Lisp で 200 行程度)。そして、本レポートでいう実行系とはまさにそれを指す。

難しいのは、効率的な実行系を作ることである。実際ラムダ計算の (数学的に美しく簡潔な) 定義どうりにコーディングしたプログラムは、動かす前からその非効率さが分かるほど酷い。関数型言語のインプリメンターたちが 30 年以上も努力してきている問題は、まさにその酷いプログラムをより早く動かすということである。

3　グラフリダクションのアイディア

上記の素朴なラムダ計算実行系を書いてみればすぐわかる非効率な部分がある。例えば以下の一見無駄な簡約列を考える。

$$(\lambda x.x * x)(1+1) \to (1+1) * (1+1) \to 2 * (1+1) \to 2 * 2 \to 4$$

明らかに $1+1$ を先に簡約しておく方が良さそうなものだが、ここでは正規順評価を考えているので、それは出来ない。これが関数型言語の実行系を難しくしている原因である。

ここで、グラフリダクションというアイディアが登場する [Wadsworth 71]。何のことはない、全ての部分式について、それを保持するノードを用意し、リダクションの際に式そのものではなく、それを保持するノードを渡す。

$$p = (1+1); (\lambda x.x * x)p \to p = (1+1); p * p \to p = 2; p * p \to 4$$

そして、引数に渡された式の一つが簡約される場合は、それらを一括管理しているノードが保持する式を簡約後の式に「破壊的に」置き換えればよい。これで無駄な簡約が防げるというわけである。

簡約すべきラムダ式は、各部分式に対して一度評価した式を保持するためのノードが付加されたグラフとなり、これを簡約していくので「グラフリダクション」と呼ばれる。 その後、このアイディアに基づいた Lisp への適用が提案された [henderson-76.pdf]。しかしグラフリダクションの効率的な実装は困難でるためか、この時点でのグラフリダクションベースの処理系は存在しなかった。

実際、この雑な説明を見ただけで、グラフリダクションにより確かに無駄な計算は無くなるが、そのための余分なコストが馬鹿にならないことは容易に見て取れる。例えば上記の例で部分式を保持するノードをヒープに用意したり、そのスロットにアクセスしたり、評価結果で書き換えたりと、あまりにも無駄が多すぎる。この程度であれば $(1+1)$ を二度計算する方が遥かにマシである。

以後、関数型言語の実行系のテーマは、いかにしてグラフリダクションを無駄なく高速に行うかという流れとなる。

4　グラフリダクションマシンの最初の実装

Turner はグラフリダクションに基づいた処理系を実装した [turner-79.pdf]。グラフリダクションにおいては、関数定義をコンビネータ (閉じたラムダ式) に分解する必要がある。Turner はコンビネータ理論 [Curry 58] の結果を使い、与えられた関数を S-K コンビネータベースのある固定された個数のコンビネータに分解した。当時の時代背景もあり、この固定されたコンビネータを高速に実行するリダクションマシンを考えていたらしい。

5　Turner のアプローチへの疑惑〜スーパーコンビネータ

コンビネータ理論に基づいた turner のアプローチはたしかにエレガントだが実行単位の粒度が細かすぎ、特殊なハードなしでは高速な実行は難しい。しかし、グラフリダクションを行うには S-K コンビネータに拘る理由はない。という発想から、Hughes は任意のプログラムをスーパーコンビネータに分解し、グラフリダクションを行う方式を提案した [Hughes 83]。

「スーパーコンビネータ」とは、それ自体閉じたラムダ式であり、さらにその全ての部分式も閉じたラムダ式である。また、与えられたプログラムをスーパーコンビネータに分解することを「ラムダリフティング」という。これ以降のほとんど全ての実装はラムダリフティングとスーパーコンビネータに基づくものとなる。

6　汎用マシンにおける高速実行〜 G-マシン

Augustsson は汎用マシンでも効率的にグラフリダクションを行う G-マシンを提案し、実装した [Augustsson 87]。簡単に言えば、各スーパーコンビネータをグラフとして持つ代わりに、グラフを作成するターゲットマシンのコードを使うことで、効率的にコンビネータのインスタンスを作成するというアイディアである。これ以降のほとんど全ての実装は G-マシンに基づくものとなる。

[Jones 87] はこの時点までの手法をまとめた本である。

7　G-マシンの改良〜 spineless G-マシン

グラフリダクションでは、再評価を避けるために、グラフのリデックスノードを評価後、その結果でリデックスノードを破壊的に更新する。[burn-88.pdf] は以下の考察により、G-マシンを効率化した。

1. ほとんどのリデックスノードは共有されていない
2. 共有されないなら、グラフを破壊的に更新しなくてもよい
3. 実際、破壊的な更新はヒープアクセスであり非効率
4. 静的なプログラム解析により、更新不要なリデックスノードの評価は破壊更新のない簡単な方法で評価して結果を返せばよい

破壊更新の際に使われるスタックを spine (背骨) と呼ぶ。これが不要 (な場合がある) というとで、これを spineless G-マシンと呼ぶ。

8　さらなる G-マシンの改良〜 STG マシン

Peyton Jones は GHC (Glasgow Haskell Compiler) の実装に向けて、それまで知られていた高速化手法を盛り込んだ STG-マシンを設計した [jones-89.pdf] - [jones-92.pdf]。

Tagless の意味はグラフノードの種別をタグとして入れて、操作を選択する代わりに操作へのポインタを埋め込むというアイディア (後に疑惑発生)。

Hughes のラムダリフティングに対する疑惑: たしかに自由変数を無くす変換をすれば一切自由変数を気にしなくて良いが、大きなブログラフでは引数が増えて効率が悪い。STG ではラムダリフティングなしでスーパーコンビネータを表現する。

[Jones 92] はこの時点までの手法をまとめた本である。

9　STG-マシンへの疑惑

STG の tagless アプローチに対する疑惑 [hammond-94.pdf]、[marlow-07.pdf]。STG の push/enter アプローチに対する疑惑 [marlow-06.pdf]。クロージャの破壊的更新に対する疑惑 [Marlow 07]、破壊的更新の代わりにインダイレクションを使う。

参考文献 (年代順)

[Church 41] Alonzo Church, "The Calculi of Lambda Conversion," Princeton University Press, 1941.

[Curry 58] H. B. Curry and R. Feys, "Combinatory Logic," North Holland, 1985.

[landin-65a.pdf] P. J. Landin, "A Correspondence between ALGOL 60 and Church's Lambda-notation: part I," Commun. ACM, vol.8, no.2, pp.89–101, 1965.

[Scott 71] Scott, D. and Strachey, C., "Toward a mathematical semantics for computer languages," Proc. of the Symposium on Computers and Automata, 1971.

[Wadsworth 71] C. P. Wadsworth, "Semantics and Pragmatics of the λ-calculus," PhD Thesis, Oxford University, 1971.

[henderson-76.pdf] Henderson, Peter and Morris,Jr., James H. "A lazy evaluator," POPL, 3rd, pp.95–103, 1976.

[milner-78.pdf] Robin Milner, "A theory of type polymorphism in programming," Journal of Computer and System Sciences, vol.17, pp.348–375, 1978.

[turner-79.pdf] D. A. Turner, "A New Implementation Technique for Applicative Languages," Software Practice and Experience, vol.9, no.1, pp.31–49, 1979.

[Hughes 83] J. Hughes, "The Design and Implementation of Programming Language," PhD Thesis, Oxford University, 1983

[cardelli-85.pdf] Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol.17, no.4, pp.471–522, 1985.

[Augustsson 87] L. Augustsson, "Compiling Lazy Functional Languages," phD Thesis, Chalmers University, 1987.

[Jones 87] Peyton Jones, S. L., *The implementation of functional programming languages,* Prentice Hall, 1987.

[fairbairn-87.pdf] Jon Fairbairn and Stuart Wray, "Tim: A simple, lazy abstract machine to execute supercombinators," LNCS, vol.274, pp.34–45, 1987.

[burn-88.pdf] Burn, G.L., Peyton Jones, S.L. and Robson, J.D., "The spineless G-machine," Proc. ACM, Conf. on Lisp and Functional Programming, pp.244–258, 1988.

[hughes-89.pdf] Hughes, J., "Why functional programming matters," Computer Journal, vol.32, no.2, pp.98–107, 1989.

[jones-89.pdf] Peyton Jones, S. L. and J.Salkild, "The spineless tagless G-machine," Proc. ACM Conf. on Functl. Prog. Langs. and C. Arch. , pp.184–201, 1989.

[jones-92.pdf] Peyton Jones, S. L., "Implementing lazy functional languages on stock hardware: The spineless tagless G-machine," Journal of Functional Programming, vol.2, no.2, pp.127–202, 1992.

[Jones 92] Peyton Jones, S. L., *Implementing functional languages: a tutorial,* Prentice Hall, 1992.

[hammond-94.pdf] Kevin Hammond, "The Spineless Tagless G-Machine - NOT!," `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8854`

[marlow-06.pdf] Simon Marlow and Simon L. Peyton Jones, "Making a fast curry: push/enter vs. eval/apply for higher-order languages," Journal of Functional Programming, vol.16, pp.415–449, 2006.

[marlow-07.pdf] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones, "Faster laziness using dynamic pointer tagging," ICFP, 2007.

[Marlow 07] Simon Marlow, "Indirection vs. in-place update of closures," `http://www.mail-archive.com/glasgow-haskell-users@haskell.org/msg12241.html`