プログラミング言語 Julia における型推論を用いたバグ検出器について

門脇　宗平

2020 年 1 月 18 日

概要

TODO

1　はじめに

プログラミング言語 Julia は，Python のような動的型付けの高水準言語が提供するような柔軟な記述性と，C のような静的型付けの低水準言語を書くことで得られるようなパフォーマンスを両立することを目指して作られた，オープンソースの汎用言語である．

非常に簡潔なシンタックスを用いて書かれたジェネリックな関数は，型推論や LLVM フレームワークを用いた JIT コンパイルにより最適化される．プログラマはプロトタイプに用いたスクリプトのまま良好なパフォーマンスを得ることができるため，”two-language-problem”[1)]を根本的に解決することができる言語として注目されており，特に科学計算の場面を中心に既に多くのユーザーを獲得している．またレイヤーの抽象度を問わず一貫してシンプルな仕組みを用いるその設計思想は，結果として開発の容易さと拡張性の高さをもたらしており，2012 年に発表された比較的新しい言語であるにも関わらず，既にそのコミュニティは大きな広がりを見せている．

一方で，Julia プログラムに含まれるエラーを実行する前に検出するのは難しい．というのも，Julia はあくまで動的型付けの言語であり，その言語機能として Julia プログラムの安全性を何らかの形で静的に保証する機能は備わっていないからである．今後 Julia がさらに普及し，Julia で書かれたソフトウェアの規模が大きくになるにつれ，この問題がさらに深刻になり得ることは，産業ソフトウェアとして現在広く使われている他の動的言語における同様の問題意識を鑑みれば明らかであるが，現行のエコシステムにおいては例えば単なる typo の検出でさえも正確に行うことは簡単ではない．

このような問題意識の下，本論文では，Julia プログラムに対する静的解析手法の 1 つとして，Julia の型推論機能を用いてプログラムを実行せずにプログラムを検査する「型プロファイラ」を提案する．この型プロファイラは静的解析のための追加的な型アノテーションを必要とせず，素の Julia プログラムに対して解析を行うことができ，バグの検出やパフォーマンスの向上に有用な情報を引き出す．また，解析に用いられる型推論のルーチンは Julia の言語機能の核をなすものであり，それ自体が既に実用上の JIT コンパイルに耐え得るパフォーマンスを備えているため，結果として本論文で提案する型プロファイラもスケーラビリティを保ちつ

---

1) 科学計算の場面において，開発の初期段階のプロトタイピングにおいては，記述が容易な高水準の動的型付け言語を用いることが多いが，開発が進むにつれて，もともとパフォーマンスを念頭において設計されていない動的言語のままでは期待するパフォーマンスを得られなくなり，結果として，C や Fortran など低級の静的型付け言語にプログラムを書き直さなくてはならなくなることが多い．このプログラマの生産性と，プログラムの効率性の両立におけるジレンマを，Bezanson, Karpinski, Shah, Edelman ら（2012, p67) は”two langauge problem”[1] と表現している．

つ，十分に実用的なエラーの検査機能を実現することができた．

本論文では，その取り組みについて以下の構成で説明する．まず 2 節では，Julia の特徴と性質を概観し，ランタイムによらない静的解析器の必要性を確認した後，Julia 以外の動的型付け言語における静的解析の取り組みを紹介する．次の 3 節で Julia の型推論を用いた型プロファイリングによる静的解析器を提案し，まず Julia の言語機能として実装されている型推論システムについて確認した後，そのルーチンを内部的に再利用する型プロファイラの設計方針を説明し，最後に型プロファイラの性能について簡単な証明を行う． 4 節では型プロファイラの評価を行い，本論文で提案する型プロファイラが十分に実用的であることを示すとともに，現状で把握できている課題についても報告する．最後の 5 節で本論文をまとめ，本プロジェクトの今後について述べる．

2　バグ検出を考える上での Julia 言語の考察と先行研究

このセクションではまず Julia の重要な性質を概観し，その静的解析に求められる要件を確認する．その後，他の動的言語における静的解析によるエラーチェックの取り組みを紹介し，本論文において取りうるアプローチを検討する．

2.1　Julia 言語の特徴と静的解析

2.1.1　generic function

Julia は generic function（総称関数）に対する code specialization と code selection により，動的な多相性を保ちつつ，良好なパフォーマンスを得ている．

ここで，generic function は同一の名前を持ち多相的な振る舞いをする関数であり，generic function は実行時にある引数を伴ってはじめて呼ばれたタイミングでコンパイル（これを JIT コンパイル，または単に JIT とよぶ）される．JIT の過程においては，まず引数の型情報を用いてその関数呼び出しに対する型推論が行われ，その関数内部で呼び出す関数（正確にはメソッド）の決定や，constant propagation, function inling などの標準的な最適化が行われた後，LLVM フレームワーク [2] を用いた最適化が行われ，最終的に個々のプラットフォームに対応したマシンコードが生成され，実行される．生成されたコードはキャッシュされ，次回以降の同じ引数型組を伴う呼び出しで使用される．この generic function のそれぞれの引数型組を伴う呼び出しに対する最適化のプロセスを code specialization と呼ぶ．例えば，Julia に標準で備わっている `sum` 関数の呼び出し `sum([1, 0, 1])` は `Array{Int,1}` という引数型に対し最適化されたマシンコードを生成し，引数型 `Array{Float64,1}` を伴う呼び出し `sum([1., 0., 1.])` はまた別のコードを生成する．

また，1 つの generic function は複数の実装を持つことができ，プログラマは generic function の定義時に引数に型アノテーションを付けることで，その実装が適用され得る引数型組の集合を記述し，その引数に対し適切な実装（これをメソッドと呼ぶ）を与えることができる．generic function が呼ばれた時，そのメソッドの中から（全ての引数の実行時の型を考慮した上で）その引数型組に対し最も「特化した」もの[2)]が選択され実行される．この dynamic multiple dispatch によるメソッドの選択を code selection と呼ぶ．

具体例として，`sum` 関数に対する code selection を考えてみる．Julia v1.5 においては `sum` 関数には標準ライブラリを含めて 14 個のメソッドが与えられている．そのうちの 3 つは以下のように実装されている．

```
1 sum(a) = sum(identity, a)
```

2) 「特化した」（specialized）メソッドの定義が実は曖昧であることに触れておく．

```
sum(a::AbstractArray; dims=:) = _sum(a, dims)
sum(a::AbstractArray{Bool}) = count(a)

sum((1, 0, 1))            # => 2
sum([1, 0, 1])            # => 2
sum(BitArray([1, 0, 1])) # => 2
```

1 行目のメソッドが最も generic な場合を扱う実装であり，2 行目のメソッドが generic な配列型に対する実装，そして 3 行目のメソッドが boolean 型の要素を持つ配列に特化した実装となっている．5 行目の呼び出しは `(1, 0, 1)` の型 `Tuple{Int,Int,Int}` に合うメソッドシグネチャが `sum(a)` の他に存在しないため，1 行目のメソッドに dispatch される．次に，6 行目の呼び出しは一般的な配列型の引数に特化した 2 行目のメソッドに dispatch される．最後に 7 行目の呼び出しは，特に `BitArray` 型の配列の効率的なメモリレイアウトを利用する `count` 関数を呼び出す 3 行目のメソッドに dispatch される．

ここで注意するべきなのは，型アノテーションは基本的にメソッドの dispatch をコントロールする目的のためだけに用いられるということである．例えば，`sum(a::AbstractArray; dims=:)` は和を取る配列の次元を指定するキーワード引数 `dims` を受け取るため，最も generic な `sum(a)` とは異なるメソッドである必要があるが[1)]，`sum` を呼び出す次のような関数 `add_sum` を定義する場合，型アノテーションの有無に関わらず呼び出し時の引数型に応じた code specialization が行われるため，型アノテーションを付けることはパフォーマンスの向上には繋がらず，むしろいたずらにこの関数が持ちうる多相性を損なってしまうことになる．[2)]

```
add_sum(a) = a .+ sum(a)
```

この例からも分かるように，様々なユースケースにおいて効率的に動く generic function を用意するために言語のコア機能やパッケージの開発において型アノテーションを行うことはあっても，エンドユーザがパフォーマンスを得るためにプログラムにアノテーションをしなくてはならない場面はほとんどない．[3)]

以上のような言語設計により，プログラマは演算子や関数の複雑な振る舞いを generic function のメソッドとして自然にプログラミングすることができ[4)]，またその呼び出しは実行時の型に対して最適化されるため，科学計算の動的な性質に対応しつつかつ効率的に実行されるプログラムを書くことができる．

一方で Julia のこうした強力な多相性は duck typing により暗黙的にもたらされるため，バグの温床にもなりうる．というのも，Julia のプログラミングパターンにおいては，良くも悪くもプログラマは generic function がある種 out-of-box に動くことを想定しつつプログラミングをすることになるからである．Julia の言語実装の大部分は自身で記述されているため，言語のコア機能から各パッケージに至るまで様々なレイヤーのコードが多相的にプログラムされている．この Julia の洗練された言語設計によりそれらのコードは多くの

---

1) キーワード引数は dispatch において考慮されないことに注意．

2) また型アノテーションにより，JIT コンパイルの速度面でのパフォーマンスが向上することもない．

3) 型アノテーションが code selection 以外の目的で用いられる場面としては，プログラム中の型推論で型が決定されない部分に型アノテーションをつけることで，コンパイラがコードを最適化する上でのヒントを与えるというものがある [3]．コンパイラはアノテーションされた型情報を使ってコード生成を行い，実行時にはアノテーションされた型と実際の型が一致することを確かめるアサーションが行われる．

4)

Mathematics is the art of giving the same name to different things.

という，Jules-Henri Poincaré の有名な警句に表されるように，数学において同一の名前の演算子は使用される文脈により多様かつ複雑な振る舞いをみせる．Julia のプログラミングパターンにおいては，その名前から「連想される」振る舞いを行う限り様々な実装（メソッド）を同一の generic function に追加するが，そのそれぞれのメソッドが実行時の引数の型に応じて動的に呼ばれることで，そうした数学の複雑な振る舞いを自然にプログラミングすることができる．

場合でプログラマの期待通りに動くものの，そうでない場合に発生するエラーをプログラマが事前に予想することは難しい．

例えば，以下のようなコマンドライン引数（文字列型の配列型）を整数にパースし，それらの和を返す関数 parse_sum があるとする．

```
function parse_sum(args)
  ints = []
  for arg in args
    push!(ints, parse(Int, arg))
  end
  return sum(ints)
end

parse_sum(ARGS) # error when empty
```

Listing 1: poorly typed code

このプログラムは一見するとバグが含まれていないように見え，実際に ARGS の長さが 1 以上の時は正常に動作する．しかし，ints の型が Array{Any,1} であるため，ARGS が空の配列である場合に，４行目の sum(ints) の内部における関数呼び出し zero(T)（T は sum の引数の要素型，この場合は Any）で zero(::Type{Any}) という関数呼び出しのシグネチャが dispatch され得るメソッドが存在しないことによるエラーが発生する．この場合，zero をオーバーロードし zero(::Type{Any}) に対応するメソッドを追加するか，ints = Int[] として ints の要素型まで指定して宣言することでエラーは避けられるものの，sum が引数の要素型に制限を設けず多相的に定義されている以上，sum が generic な配列型に対して動作すると期待するプログラマがこうしたエラーを事前に予測することは簡単ではない．

また，こうしたエラーを実行時前に静的に検査することも容易ではない．というのは，一般に Julia の generic function の挙動は，その呼び出し時に与えられた引数型に従って決定されるからである．型アノテーションを付けることでメソッドの引数の型が静的に決定される場合，静的型付けの言語で用いられるような標準的な型解析を行うことができるが，上述したように Julia の型アノテーションはメソッドの dispatch をコントロールするためにあくまで補助的に用いられるものであり，プログラムに闇雲にアノテーションを付けることは，そのプログラムの多相性の損失にも繋がるため，静的な型検査のためだけに型アノテーションを強制することはできない．

2.1.2 メタプログラミング

また，Julia はそのメタプログラミング機能によっても特徴付けられる．

例えば，上述の sum(::AbstractArray; dims = :) は実際にはメタプログラミングを用いて，他の同様の構造を持つ関数とまとめて同時に定義されている．[5)]

```
for (fname, _fname, op) in [(:sum,     :_sum,     :add_sum), (:prod,     :_prod,
↪   :mul_prod),
```

[5)] https://github.com/JuliaLang/julia/blob/a7cd97a293df20b0f05c5ad864556938d91dcdea/base/reducedim.jl#L648-L659 より抜粋．

```
                                (:maximum, :_maximum, :max),     (:minimum, :_minimum,
                                ↪  :min)]
    @eval begin
        # User-facing methods with keyword arguments
        @inline ($fname)(a::AbstractArray; dims=:) = ($_fname)(a, dims)
        @inline ($fname)(f, a::AbstractArray; dims=:) = ($_fname)(f, a, dims)

        ...
    end
end
```

こうした文字操作的なメタプログラミングは他の言語でもよく見られる機能であるが，Julia のメタプログラミング機能はより強力である．全ての Julia プログラムは木構造を持つ式であり，プログラマはマクロを通じてパースタイムにおいてあらゆるプログラムのデータ構造にアクセスし，操作することができる．上のプログラムにおける`@inline` など`@`から始まるコードがマクロに相当し，ここでは，`@inline` マクロは関数定義の式を受け取り，最適化の過程においてその関数を inlining するように促す「ヒント」を JIT コンパイラに与えるメタ情報を関数定義に追加する．言語のコア機能に限らず Julia のエコシステムにおいてもマクロは頻繁に用いられており，例えば Julia の数理最適化分野におけるデファクトスタンダードなパッケージである JuMP.jl[4] はマクロを効果的に利用し，以下のような明瞭で簡潔なモデリング記法を提供している．

```
model = Model(with_optimizer(GLPK.Optimizer))
@variable(model, 0 <= x <= 2) # x will automatically be defined
@variable(model, 0 <= y <= 30) # y will automatically be defined

@objective(model, Max, 5x + 3y)
@constraint(model, x + 5y <= 3)
```

Listing 2: code including macros

このように Julia のプログラミングパターンにおいては，いたるところでメタプログラミングが使用され[6)]，冗長なコードをより簡潔に記述できる他，ドメイン特化的なノーテーションを表現することが，科学計算の各分野における様々な記法を単一の汎用言語のシンタックスの中で表現することができる [5].

その一方で，一般にコードを生成するコード，つまりマクロ（あるいは staged programmming における code generator）はそれ自体がバグを含みやすく，またメタプログラミングは適切でない場面で使用された場合，コードの可読性を著しく下げてしまうこともある．しかしそれにも関わらず，メタプログラミングを含むコードに含まれるエラーを静的に解析することは難しい．というのも，`eval` など実行時の値を用いるため原理的に完全な静的解析が不可能なメタプログラミングが存在する他，マクロを含むコードについてもマク

6) こうしたパースタイムにおけるマクロ展開によるシンタックスレベルのメタプログラミングの他，関数のコンパイル時に引数の型情報を用いて新たにコードを生成しそれを実行する staged programming も可能である．Julia の staged programming は上述した generic function の dispatch のメカニズムを利用した非常に自然な言語機能として提供されており，コードの生成する関数自身がその他の generic function と同じように呼び出され，生成されたコードは通常の関数呼び出しと同じように JIT コンパイルによる最適化が行われた後，実行される．

ロを展開するまでその文法上の意味を決定できないからである．例えば，listing 2 の JuMP.jl の`@variable` マクロのその引数式だけを見ると，`0 <= x <= 2` の部分はまだ定義されていない変数を `x` を参照しているため未定義エラーを起こしてしまうように解析され得るが，実際には，その引数式はそのまま評価されず，`@variable` マクロ展開時に引数式のそれぞれの形式に対応してそれを JuMP.jl の内部的なデータに自動的に変換されたコードが実行されるため，エラーは生じず正常に動作する．この例からも分かるように Julia プログラムの正確な解析のためにはマクロ展開後のコードに対する解析を行う必要があるが，これはプログラムの静的解析が言語機能として行われない場合その完全な実装を困難にする 1 つの要因となる．[7)]

2.2　先行研究

3　型プロファイラの設計方針

4　評価と今後の課題

4.1　実験と比較

4.1.1　section 1 のエラー例に対するレポート

4.1.2　ruby-type-profiler との比較

1. builtin method
2. 再帰の扱い

5　まとめ

参考文献

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, Vol. 59, No. 1, pp. 65–98, 2017.

[2] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[3] JuliaLang Contributors. Annotate values taken from untyped locations. `https://docs.julialang.org/en/v1/manual/performance-tips/#Annotate-values-taken-from-untyped-locations-1`. (Accessed on 01/16/2020).

[4] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, Vol. 59, No. 2, pp. 295–320, 2017.

[5] Jeff Bezanson. *Abstraction in Technical Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2015.

[7)] Julia の現行のエコシステムにおいて，最も正確にシンタックスレベルのエラーを静的に検出するパッケージとして StaticLint.jl があるが，マクロ展開のためには Julia プログラムのインタプリタの再実装が必要となり，またプログラムの解釈にかかるプロセスが解析の複雑度を上げパフォーマンスを下げるため，listing 2 のようなシンタックス上の意味を大きく変えるマクロについては，JuMP.jl などエコシステムにおける重要度の高いパッケージのマクロをそれぞれ special case することで対応している: e.g. `https://github.com/julia-vscode/StaticLint.jl/pull/72`