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について

門脇　宗平

2020 年 1 月 20 日

概要

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，高水準の汎用言語であり，オープンソースで開発されている．

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 [2]，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抽象解釈することにより検査する「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，型レベルのエラー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

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．

ティを保ちつつ，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 [3] 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計 14 個のメソッドが与えられている．そのうちの 3 つを抜粋する（読みやすさのためやや改変している）．

---

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) = sum(identity, a)
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，`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．[3)]

```
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．[4)]

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[5)]，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

---

2) キーワード引数は dispatch において考慮されないため，位置引数の型に対する dispatch を用いる必要がある．

3) 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．

4) 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 [4]．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．

5)

> 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．

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 である）において， zero(::Type{Any}) という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，2 行目を 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出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，静的な型解析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．

メタプログラミングは Julia プログラムのいたるところで用いられる．例えば，上述の sum(a::AbstractArray; dims = :) メソッドも実際にはメタプログラミングを用いて，同様の構造を持つ他の関数とまとめて定義されている．[6)]

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

```
for (fname, _fname, op) in [(:sum,     :_sum,     :add_sum), (:prod,    :_prod,
↪   :mul_prod),
                            (: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動的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，プログラマが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をデータ構造として扱い操作することができるという点で，**より**強力であり，いわゆる「Lisp スタイル」のマクロを提供している．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[5] は独自のマクロを効果的に定義し，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．

```
1 model = Model(with_optimizer(GLPK.Optimizer))
2
3 @variable(model, 0 <= x <= 2) # x will automatically be defined
4 @variable(model, 0 <= y <= 30) # y will automatically be defined
5
6 @objective(model, Max, 5x + 3y)
7 @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[7)]，冗長なコードをより簡潔に記述できる他，科学計算の各分野におけるドメイン特化的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 [6].

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

[7)] 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 のメカニズムを用いた非常に自然な言語機能として提供されている．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．

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を静的に解析することは難しい．例えば，listing 2 の 3 行目の`@variable` マクロの引数式 `0 <= x <= 2` だけを見ると，まだ定義されていない変数 `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，この引数式はそのまま評価されず，`@variable` マクロの展開時に式のそれぞれの形式に対応して JuMP.jl の内部的なデータとして変数 `x` を定義するようなプログラムが生成されるため，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，マクロ展開のためには何らかのインタプリタが必要になり，静的解析の実装における 1 つの障壁になりうる．[8)]

2.2 先行研究

以上のような，プログラムの堅牢性よりも記述性を優先し，duck-typing により強力で柔軟なアドホック多相性を得るという思想は，いくつかの動的型付け言語の核をなすパラダイムであり，そのトレードオフとして生じるプログラムの品質保証に伴う困難さについてはそれらの言語においても同様の問題意識が存在する．

そうした動的言語で書かれたプログラムに対する型レベルでの解析取り組みとしては，これまで大きく以下の 2 つのアプローチが取られてきた．

1. プログラムに追加的な型シグネチャを与えることにより，型検査を行う方針
2. 素のプログラムに対し抽象解釈を行い，型を「プロファイリング」する方針

以下ではそれぞれのアプローチについて，適宜既存の具体的なプロジェクトについて言及しつつ，その方式におけるトレードオフや問題点を考える．

なお，ここではシンタックスレベルで得られる情報を用いて行う解析方式については触れない．本論文の取り組みとして，シンタックスレベルではなく型レベルの情報をもちいてバグ検出を試みる理由については 3 で述べる．

2.2.1 追加的な型シグネチャによる型検査

この方式では，たとえ動的型付けの言語で記述されたプログラムであっても，定義が完了したクラスや関数においてはある種の静的なシグネチャが存在することを期待し，その型シグネチャをプログラマ自身に記述させることで，ライブラリの実装と使用（呼び出し）がそのシグネチャと矛盾していないか検査を行う [2]．多相性は，部分型多相やパラメータ多相の他，duck-typing に相当するものとして structural typing が使用されるが[9)]，いずれの場合もプログラマが与えるシグネチャにより明示的に導入される． 静的に型が決定できない場合，動的型を表す特殊な型を導入し検査を続ける．以降その動的型に対する操作についての検査は行わ

---

8) マクロの存在が静的解析の実装を難しくする例として，StaticLint.jl での事例がある．StaticLint.jl は主にエディタの Linting 機能のバックエンドとして使用されることを想定したパッケージであり，現行のエコシステムにおいて最も正確に Julia プログラムに含まれるシンタックスレベルのエラーを検出することができるが，パフォーマンス上の理由から完全なマクロ展開をサポートしていない [7]．マクロ展開に必要なプログラム解釈のプロセスは，解析の複雑度を上げ，結果として編集中のプログラムをリアルタイムで（on-the-fly に）解析可能なパフォーマンスを保つことを困難にする（例えばユーザが既に定義してあるマクロの定義を変更した場合，正確な解析のためにはもう一度プログラム全体を解釈しなおす必要がある）．そのため，現状では listing 2 の`@variable` 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のみをそれぞれ個別に special case することで部分的に対応している：e.g. `https://github.com/julia-vscode/StaticLint.jl/pull/72`

9) structural subtyping の表現方法として，mypy は”Protocol”と呼ばれるある種のインターフェースを明示的に指定させるが，Steep はクラスシグネチャのメソッド集合の包含関係から自動的に部分型関係を判断する．

れないため型安全性は保障されない．型検査器は動的型が導入された場合通常何らかの警告を出すが，そのレベルについてはそれぞれのプロジェクトにより異なる．型シグネチャの記述方法としては，プログラムの実行には関与しない形で追加的に与える方法[10)]の他，TypeScript [8] のように元の言語との前方互換性を保たない拡張言語において型に関する記述を言語の標準機能として行えるようにするというものがある． この検査方式を採用したプロジェクトとしては数多くのプロジェクトが存在し，mypy [9]（Python），Steep [10]，TypeScript（JavaScript）など，既に産業ソフトウェアの場面で実際に広く運用されているものもある．

この方式を採用することのメリットは様々ある．ここではその中でも重要と思われるものを述べる．

- おおよその安全な型システムを得ることができる．
- 型検査はクラスや関数を単位に行われるため，実用的なパフォーマンスを得やすい．具体例として，mypy プロジェクトは incremental checking など様々なエンジニアリングを経て Dropbox 社の 400 万行ものコードベースに対し数分で検査可能なパフォーマンスを得ている [11]．また同様の理由から編集中のコードに対して型検査をリアルタイムに行うことで，補完や推論された型情報のフィードバックなど，プログラマにとって有用な情報を引き出すことも可能である．
- 遠藤，松本宗太郎，上野，住井，松本行弘ら（2019）[2] が述べるように，型検査自体に用いられる型シグネチャはそのライブラリのある種の API として機能しうる．
- 型アノテーションを元のプログラムのパフォーマンスの向上に用いることが可能な場合がある．具体例として，mypy の型検査器は Python で書かれたコードを専用のコンパイラでコンパイルすることで，約 4 倍のパフォーマンスの向上を得ている．

一方で，この方式のデメリットは，型シグネチャを書かせることによるプログラマ負担という 1 点に絞られるだろう．つまり，動的言語はそもそも（おおよそ）その動的機能がもたらす柔軟な挙動と簡潔な記述性を目指して設計されているにも関わらず，この方式においてはプログラマは静的な型付けを意識しつつ，型の記述にかかる手間を受け入れなくてはならず，また型の記述はプログラムを冗長にしその簡潔性を損ない得る．

2.2.2　抽象解釈による型プロファイリング

3　型プロファイラの設計方針

3.1　Why "type profiler" ?

3.2　Julia の型推論システム

3.3　TypeProfiler.jl の設計

4　評価と今後の課題

4.1　実験と比較

- これまでに紹介したコードに対するレポート
- self profiling ?
- ruby-type-profiler との比較

[10)] 例えば，mypy は Python3 の標準機能である docstring を用いてプログラム中にシグネチャを記述するが，Steep はプログラム本体とは別のファイルを用意しそこにシグネチャを記述する．どちらの場合も，型シグネチャはそれぞれの言語との前方互換性を保つ形で与えられるため，型検査システムを導入したプログラムは元の言語処理系でそのまま動かすことができる．

4.2 課題

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] 遠藤侑介, 松本宗太郎, 上野雄大, 住井英二郎, 松本行弘. Progress report: Ruby 3 における静的型解析の実現に向けて. `https://github.com/mame/ruby-type-profiler/blob/master/doc/ppl2019.pdf`, 2019.

[3] 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.

[4] *Julia Documentation - Performance Tips - Annotate values taken from untyped locations.*

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

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

[7] Zac Nugent. add support for Turing.model macro by ZacLN - Pull Request #64 - julia-vscode/StaticLint.jl. `https://github.com/julia-vscode/StaticLint.jl/pull/64#issuecomment-575801768`. (Accessed on 2020-01-19).

[8] Microsoft. TypeScript - JavaScript that scales. `https://www.typescriptlang.org/`. (Accessed on 2020-01-19).

[9] mypy - Optional Static Typing for Python. `http://www.mypy-lang.org/`. (Accessed on 2020-01-19).

[10] Soutaro Matsumoto. Steep - Gradual Typing for Ruby. `https://github.com/soutaro/steep`. (Accessed on 2020-01-19).

[11] Our journey to type checking 4 million lines of python — dropbox tech blog. `https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/`. (Accessed on 2020-01-20).