

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 27 日

概要

1 はじめに

プログラミング言語 Julia[1] 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.

簡潔なシンタックスを用いて記述された関数は強力な多相性を持ち, 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. プログラム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 [2]. また, 言語のコア機能の実装からユーザコードに至るまでレイヤーを問わず一貫した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, そのコミュニティは既に大きな広がりを見せている.

一方で, Julia はあくまで動的型付けの言語であり, Julia プログラムの型安全性は静的に保証されないが, 型エラーを実行する前に検出しそのプログラム品質を高めようとする取り組みはこれまでの所試みられていない. 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なるにつれ, この問題がさらに深刻になり得ることは, 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 [4], 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 2.2.3.

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い, 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ことを目指している. 型プロファイラの実装には型推論ルーチンが必要となる. 今回の取り組みでは, Julia の核をなす言語機能として JIT コンパイル目的に使用されている型推論のルーチンを用いることで, 抽象解釈を用いたプログラム解析に

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. このプログラムの生産性と, プログラムの効率性の両立におけるジレンマを, Bezanson 他は "two language problem" [3, 67] と表現している.

において問題となりやすいスケーラビリティを保ちつつ、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の簡単な命題の証明を行う。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 静的な型エラー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

数ある Julia の特徴の中でも、以下の 2 点は Julia プログラムの静的な型解析を考える上で重要である。

1. generic function を通じた多相性と最適化
2. 強力なメタプログラミング機能（マクロの頻繁な使用）

この小節では generic function とメタプログラミングという 2 点から Julia の性質を概観し、Julia プログラムを型レベルで静的解析する必要性と、その際にどのような困難を生じ得るかをコード例と共に紹介する。セクション 4 において、それらのコード例は型プロファイラを用いて解析される。

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 [5]。

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、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、その呼び出し内で生じる関数（正確にはメソッド）呼び出しのコンパイルタイムにおける決定や、constant propagation, function inlining などの標準的な最適化が行われた後、LLVM フレームワーク [6] 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}`（1 は配列の次元が 1 次元であることを示す）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.])` はまた別に最適化されたコードを生成する。

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（これをメソッドと呼ぶ）が適用され得る引数型組の集合を記述することができる。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²⁾」もの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）。

```
1 sum(a) = sum(identity, a)
2 sum(a::AbstractArray; dims=:) = _sum(a, dims)
3 sum(a::AbstractArray{Bool}) = count(a)
4
5 sum((1, 0, 1))           # => 2
6 sum([1, 0, 1])           # => 2
7 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 `Bool` 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³⁾、次のような `sum` を呼び出す関数 `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⁴⁾。

```
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、その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付けなくてはならない場面はほとんどない⁵⁾。

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⁶⁾、またそのそれぞれの振る舞いは実行時の型に応じて決定され

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。

³⁾ 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。

⁴⁾ 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: <https://discourse.julialang.org/t/type-annotation-make-jit-compile-faster/31906/2>

⁵⁾ 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 [7]。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。

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おり、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。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 ARGS is 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起因す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定義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ため、それぞれの関数や変数の定義だけを見て行うシンタックスレベルの解析では、このような型エラーの検出はできないからである。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 intra-procedural 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

6)

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。

generic function 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、静的解析のために型アノテーションをつけることにより、元の generic function が持っている多相性が損なわれることがあってはならない。

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⁷⁾。

```
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（いわゆる「Lisp スタイル」の）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 の数理最適化分野のエコシステム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^[8] 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。

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⁸⁾、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^[5]。

⁷⁾ <https://github.com/JuliaLang/julia/blob/a7cd97a293df20b0f05c5ad864556938d91dcdea/base/reducedim.jl#L648-L659> より抜粋。

⁸⁾ こうしたパースタイムにおけるマクロ展開（セクション 2.1.3 参照）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。

```

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

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 programming 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、プログラムの可読性を著しく下げてしまうこともある。しかも、メタプログラミングされたコードを含むプログラムを静的に解析することは難しい。というのは、`eval` のように実行時の値がなければ原理的に解析が不可能なメタプログラミング機能が存在する他、マクロを含むプログラムの正確な意味を得るためには、マクロ展開を行った後のプログラム（セクション 2.1.3 参照）に対する解析を行う必要があるが、マクロ展開にかかるプロセスは静的解析の実装における 1 つの障壁になりうる 2.2.3 からである。

2.1.3 Julia の実行プロセスと中間表現

最後に、以降の説明を容易にするため、Julia プログラムが実行されるプロセスを大まかにまとめ、各実行段階におけるプログラムの中間表現を指す用語を導入する [9, 10].

以下の操作は全て、実行されるプログラムにおけるそれぞれのコード片（Julia プログラムの実行における単位となる各コード部分）に対して行われるものである。

1. コード片をパースし **surface AST**⁹⁾を得る。
2. モジュールの使用や型の定義などの”`toplevel action`”はインタープリタにより実行される。以下は関数呼び出しなど”`toplevel action`”ではない場合の実行プロセスとする。
3. surface AST に含まれるマクロを全て展開（マクロ展開）し、さらに **lowering**¹⁰⁾を行い **lowered form**を得る。
4. 簡単なヒューリスティック¹¹⁾を用いて lowered form をインタープリタにより実行するか、JIT コンパイルを行うかを決定する。以下は JIT コンパイルされる場合のプロセスとする。
5. lowered form に対して型推論と最適化を行い、**typed lowered form**を得る
6. type lowered form から LLVM 命令列を生成する
7. LLVM が native code を生成する
8. native code が実行される

⁹⁾ ”AST”は”Abstract Syntax Tree”（抽象構文木）の略。

¹⁰⁾ ここで”lowering”とは、型推論などの解析がしやすいように、マクロ展開後の surface AST を SSA 形式を取る lowered form（または単に IR, Intermediate Representation と呼ぶ）にさらに変換することを指す。

¹¹⁾ <https://github.com/JuliaLang/julia/blob/master/src/toplevel.c#L588-L819>

2.2 先行研究

プログラムの堅牢性よりも記述性を優先し、duck-typing により強力な柔軟なアドホック多相性を得るという思想は、他のいくつかの動的型付け言語においても核となるパラダイムであり、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、それぞれの方式におけるトレードオフを考える。

また、最後に現行の Julia エコシステムにおける静的解析によるバグ検出の試みの例として、StaticLint.jl を紹介し、本論文の取り組みと比較する上で重要な相違点を説明する。

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 [4]。

検査において多相性は、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 を使用する¹²⁾ことで表され、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。この方式を採用する多くのシステムは、静的に型が決定できない場合、動的型¹³⁾を表す特殊な型を導入し検査を続ける。動的型が導入された場合、型システムは通常何らかの警告を出すか、以降その動的型に対する操作についての検査は行われないため、動的型を許容する限り、この方式で導入される型システムは健全ではない。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¹⁴⁾の他、TypeScript のように元の言語との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 (Python) [11], Steep (Ruby) [12], TypeScript (JavaScript) [13] など、既に商用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として intra-procedural に行われるため、実用的なパフォーマンスを得や

¹²⁾ 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部分型関係を判断する。

¹³⁾ ここで「動的型」は、TypeScript における **any** 型のような、静的に型を決定できない場合に型システムあるいはプログラマが導入し、以降型システムはその型に対する操作について型検査をスキップするような、特殊な型を指す。漸進的型付けにおいては、動的型を導入することで、既存のコードに対する型付けを徐々に行うことができるようになるため、ほとんどのシステムで導入されている。

¹⁴⁾ 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。

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 [14]。また同様の理由から編集集中のコードに対してリアルタイムに型検査を行うことで、補完や推論された型情報のフィードバックなどのプログラマ支援に応用することも可能である [15, 13]。

- [4] で触れられているように、型検査に用いられる型シグネチャ自体がそのライブラリの API のある種のドキュメントとして機能しうる。
- 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 の型検査器はそれ自身 mypy の型アノテーションが施された Python コードで記述されており、その型情報を用いて専用のコンパイラで CPython の C 拡張モジュールへコンパイルすることで、元のプログラムと比較して約 4 倍のパフォーマンスの向上を達成している [14]。

次に、この方式のデメリットを考える。まず、最大のデメリットは、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 点になるだろう。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、型の記述にかかる冗長性を受け入れなくてはならない。次に、漸進的型付けの性質上、健全な型システムを期待することが難しいということが挙げられる。動的型の存在の他に、そもそも型システムを与える型シグネチャ自体がプログラマ自身によって与えられることが多く、それ自体が誤りを含んでいる場合、型検査に意味がなくなってしまう。後者に関しては、後述する型プロファイリングなど、素の動的型付け言語プログラムからその型情報を自動生成する手法を組み合わせることで改善できる可能性がある¹⁵⁾。

2.2.2 抽象解釈による型プロファイリング

「型プロファイリング」は素の動的型付け言語で記述されたプログラムを受け取り、プログラムの実行を型レベルに抽象化した上でトレース（抽象解釈）し、その過程で発見した型エラーやトレースされた型情報そのものを報告するような手法を指す。

プログラムをそのまま実行するのではなくあくまで型レベルに抽象化した上で解釈することにより、プログラム中の実際の実行における制御フローでは到達できない部分の解析や、終了しないプログラムやプログラムが予期しない副作用を伴うプログラムなどのプロファイリングも可能であり、プログラムを実行するよりも有用な情報を引き出し、バグの発見や、出力された型情報をコード理解に役立てることができるなどの恩恵が期待される¹⁵⁾。解析の正確性と実行速度は抽象化の程度に依存し、トレードオフをなす。

ここで注意すべきなのは、プログラムの抽象解釈は inter-procedural に行われるということである。つまり、関数やメソッドを定義するプログラムを入力とするだけでは解析を行うことができず、それらを起動するエントリポイントとなるトップレベルのスクリプトも含めて与えられることにより解析を行うことが可能となる。

また、追加的な型シグネチャによる型検査 2.2.1 とは異なり、この方式の先行事例は比較的少なく、ruby-type-profiler [16] や”Type Analysis For JavaScript” [17] などがあるが、いまだ広く実用に用いられているものはないように思われる。

この方式における利点は次の 1 点に集約される。つまり、型プロファイラは追加的な型注釈を必要とせずに行われるため、動的言語本来のプログラミングスタイルを保ったまま、何らかの型解析を行うことができるということである。

¹⁵⁾ コード理解の応用例として、遠藤らは型プロファイラの出力を上述の漸進的型付けによる型検査で必要となる型シグネチャとして利用できる可能性を指摘している [4]。

一方で抽象解釈に伴う困難は様々に存在する．ここでは `ruby-type-profiler` [4, 16] で報告されているものを中心に紹介する．これらの問題点は後述の `TypeProfiler.jl` の設計 3 においても重要となる．

■ **トップレベルスクリプトの必要性** 上述したように型プロファイリングにはそれぞれの関数やメソッドを起動するトップレベルスクリプトが必要である．

一般に型解析がもたらすプログラムの品質保証はパッケージやライブラリの開発の場面において必要とされることが多く，テストコードをトップレベルスクリプトとして用いることができるものの，そのコードが（抽象解釈する上で）到達しない関数やメソッドは解析することができない．それらの引数として擬似的な入力を与えることで解析を行い，タイポや未定義エラーを検出できる可能性があるものの，解析の正確性は大きく下がらざるを得ない．

■ **false positive な報告** 前述したように，抽象解釈は実際にプログラムを実行するよりも有用な情報を引き出せる場合もあるが，それとは逆に実際のプログラム実行では起こりえないエラーを報告してしまう場合がある．

具体例として以下のような Ruby プログラムを考える．

```
1 def foo(n)
2   if n < 10
3     return n
4   else
5     return "over 10"
6   end
7 end
8
9 def bar(n)
10  if n < 10
11    return foo(n) + 1
12  else
13    return foo(n) + "+1"
14  end
15 end
16
17 bar(1) # here ruby-type-profiler ends up reporting potential errors
```

2 行目の条件と 10 行目の条件は同一の `n` に対して等価であるため，`n` が `Integer` であれば，11 行目の `foo(n)` の返り値型は `Integer` であり，13 行目の `foo(n)` の返り値型は `String` となるため，このプログラムはエラーなく終了する．

一方 `ruby-type-profiler` は 2 行目および 10 行目の分岐において，`n` の型レベルの情報（この場合であれば `Integer`）しかもたないため，分岐の条件を正確に評価することができず，抽象解釈の状態を単純に分岐させる．つまり，11 行目および 13 行目の `foo(n)` の返り値型は `Integer` または `String` になりうるとして解析するため，それぞれの行で `Integer` 型と `String` 型の変数を受け取るメソッド呼び出し `+` が存在しないこと

をエラーとして報告してしまう¹⁶⁾。

ただ、こうした false positive なエラー報告は、いわば静的解析の保守性に起因するものであり、型プロファイリングに限らずその他の静的なプログラム解析に共通する問題点ではある。

■抽象化の程度とスケーラビリティ 一般に型プロファイリングのような inter-procedural な解析手法はスケーラビリティに問題を持つ。具体的には、プログラムの抽象解釈では型を一意に決定できない場合、union 型などを導入し解釈を続行するが、条件分岐が続いた場合解析すべき状態の数が増大し、解析に時間がかかってしまう可能性がある。また、型として（その要素型がネストしうる）コンテナ型が存在する場合、そのままでは存在しうる型が有限にならず、抽象解釈のアルゴリズムの停止性を保証できない。

これらの問題は、何らかのヒューリスティックを用いて、状態や抽象値の数を制限することで対処することが多いが、状態がより抽象化される分だけ解析の正確性は失われてしまうため、解析の正確性とスケーラビリティはトレードオフの関係をなしている。実用的に必要な解析の正確性を保ちつつ、同時に実用に耐え得るスケーラビリティも保てるように抽象化の程度を調整する必要がある。

2.2.3 現行のエコシステムにおけるエラーチェッカー: StaticLint.jl

StaticLint.jl [18] は Julia プログラムに対する静的な解析を行いエラーを検出する、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る。

Julia プログラム中に含まれる、シンタックスレベルでの解析により検出可能なエラーを、2020 年 1 月 27 日現在の Julia エコシステムにおいて最も正確に検出することができ、また、編集中のコードに対してリアルタイムに linting した結果をフィードバックすることができる程度のパフォーマンスを備えている。

ここで、シンタックスレベルの解析とは、surface AST 2.1.3 の段階のプログラム表現から得られる情報を用いた解析を意味し、つまり StaticLint.jl は実行プロセス 3 以降のプロセスに相当する解析は行わない。

解析をシンタックスレベルに留めることで、編集中のコードに対する linter として機能し得るパフォーマンスを獲得している一方、以下で例にあげるような型レベルのエラーやマクロ展開に伴うエラーは検知できない。

■シンタックスレベルの解析で検知できないエラー シンタックスレベルでの解析では検知できないエラーは様々あり、以下のような非常に簡単な typo も、型レベルでの解析を行わない場合、検出できない。

このコードでは 6 行目に typo が含まれており、実行すると 11 行目において F2 型の変数にフィールド bar が定義されていないことに起因するエラーが発生する。しかし、StaticLint.jl はこのプログラムに含まれるエラーを報告しない¹⁷⁾。というのは、関数 get_bar の定義だけを見るとその引数 foo として実際にどのような型の変数が渡されるのか決定できず¹⁸⁾、また上述したパフォーマンス上の理由から、エラーが発生する呼び出し get_bar(F2(1)) までも含めて解析する inter-procedural なプログラム解釈は行えないからである。

■マクロを含むコードの解析 同様にパフォーマンス上の理由から StaticLint.jl は完全なマクロ展開をサポートしていない¹⁹⁾。マクロ展開ではマクロが使われている場所とマクロの定義式の両方を参照しなくてはならず、ここでも inter-procedural なプログラム解釈が必要となる。

¹⁶⁾ Ruby 2.6.5 を用いて 2020 年 1 月 27 日での ruby-type-profiler で動作確認を行った: <https://github.com/mame/ruby-type-profiler/tree/31b43adfbf0320b820dbd2c3a766efb2825072a4>

¹⁷⁾ 2020 年 1 月 27 日時点での Julia レポジトリの master ブランチからビルドした Julia を用いて、StaticLint.jl が linter として使用されている julia-vscode v0.13.1 上で動作確認を行った: <https://github.com/julia-vscode/julia-vscode>

¹⁸⁾ Foo が abstract type であることに注意せよ。

¹⁹⁾ パッケージ著者と私信にて確認:

<https://github.com/julia-vscode/StaticLint.jl/pull/64#issuecomment-575801768>

```

1 abstract type Foo end
2 struct F1 <: Foo
3     bar
4 end
5 struct F2 <: Foo
6     baz # typo
7 end
8 get_bar(foo::Foo) = foo.bar
9
10 get_bar(F1(1)) # no error
11 get_bar(F2(1)) # error because of undefined field `bar`

```

Listing 3: typo

現状では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²⁰⁾.

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

4.2 今後の課題

5 まとめ

参考文献

- [1] The Julia Language. <https://julialang.org/>. (Accessed on 2020-01-22).

²⁰⁾ 例えば, listing 2 の 3 行目の `@variable` マクロは, 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` を参照しているように見えるが, 実際にはマクロ展開時に, 引数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 (この場合は `x`) を定義するようなコードを生成するためもちろんエラーは起きない. 一方 StaticLint.jl はマクロ展開をサポートしていないため, v3.0.0 まではこの場合に未定義変数の参照エラーを報告されていたが, 現在は対応されている: <https://github.com/julia-vscode/StaticLint.jl/pull/72>

- [2] Julia Computing. Newsletter January 2020 - Julia Computing. <https://juliacomputing.com/blog/2020/01/06/january-newsletter.html>. (Accessed on 2020-01-22).
- [3] 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.
- [4] 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/pp12019.pdf>, 2019.
- [5] Jeff Bezanson. *Abstraction in Technical Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2015.
- [6] 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.
- [7] *Julia Documentation - Performance Tips - Annotate values taken from untyped locations*.
- [8] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, Vol. 59, No. 2, pp. 295–320, 2017.
- [9] *Julia Documentation - Developer Documentation - Eval of Julia code*. (Accessed on 2020-01-27).
- [10] *Julia Documentation - Developer Documentation - Julia ASTs*. (Accessed on 2020-01-27).
- [11] mypy - Optional Static Typing for Python. <http://www.mypy-lang.org/>. (Accessed on 2020-01-19).
- [12] Soutaro Matsumoto. Steep - Gradual Typing for Ruby. <https://github.com/soutaro/steep>. (Accessed on 2020-01-19).
- [13] Microsoft. TypeScript - JavaScript that scales. <https://www.typescriptlang.org/>. (Accessed on 2020-01-19).
- [14] 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).
- [15] FaceBook. Flow - Adds static typing to JavaScript to improve developer productivity and code quality. <https://github.com/facebook/flow>. (Accessed on 2020-01-19).
- [16] ruby-type-profiler - an experimental type-level ruby interpreter for testing and understanding ruby code. <https://github.com/mame/ruby-type-profiler>. (Accessed on 2020-01-21).
- [17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pp. 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [18] julia vscode. StaticLint.jl - Static Code Analysis for Julia. <https://github.com/julia-vscode/StaticLint.jl>. (Accessed on 2020-01-24).