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

京都大学　総合人間学部　認知情報学系専攻

2020 年 2 月 3 日

概要

本論文はプログラミング言語 Julia で書かれたプログラムを抽象解釈することにより，プログラム中に含まれるバグを静的に検出する型プロファイラを提案する．

Julia は，高水準な動的型付け言語が提供する簡潔で柔軟な記述性を保ったまま，低水準な静的型付け言語で得られるパフォーマンスを達成することを目指して開発された比較的新しいプログラミング言語であり，科学計算の場面を中心に既に多くのユーザを獲得しているが，一方で Julia プログラムの型安全性を静的に保障する手法はいまだ確立されていない．

本論文で提案する型プロファイラは，実装に Julia の JIT コンパイルに用いられている型推論のルーチンを再利用することで，抽象解釈によるプログラム解析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，実用的なエラー検出能力を達成することができた．この型プロファイラは，素の Julia プログラムに対してそのまま用いることができるため，既存のスクリプトやパッケージに対して気軽に導入することができ，今後 Julia プログラムの型安全性を向上させるうえでの足がかりとなることが期待される．

目次

1 はじめに 3

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 4
2.1 Julia 言語の特徴と静的解析 4
2.1.1 generic function 4
2.1.2 メタプログラミング 7
Julia の実行プロセスと中間表現 8
2.2 先行研究 9
2.2.1 追加的な型シグネチャによる型検査 9
2.2.2 抽象解釈による型プロファイリング 10
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 11

3 型プロファイラの設計 13
3.1 Why ”type profiler” ? 13
3.2 Julia の型推論システム 13
3.2.1 アルゴリズム 14
3.2.2 停止性とヒューリスティック 16
3.2.3 correctness 16
3.3 TypeProfiler.jl の設計と性質 16
3.3.1 設計 16
3.3.2 性質 17

4 評価と今後の課題 17
4.1 実験と比較 17
4.2 今後の課題 17
4.2.1 built-in function のプロファイリング 17
4.2.2 マクロ展開に対するエラー検出 17

5 まとめ 18

謝辞

Julia で書かれたプログラムに対して既存の型推論ルーチンを再利用し型プロファイリングを行うアプローチについての着想をいただき，また本プロジェクトを通じてとても温かく指導していただいた指導教員の櫻川貴司准教授に感謝いたします．また，本論文に関して様々に有益なコメントをいただいた全ての方々へ感謝の意を表します．

1　はじめに

本論文は，プログラミング言語 Julia で記述されたプログラムを型レベルで抽象解釈することにより，プログラム中に含まれるバグを静的に検出する解析器を提案する．

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．プログラマはプロトタイピング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) 2)] 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 [5]．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 [2]，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になるにつれ，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 [6]，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，プログラムを型レベルで抽象解釈することにより解析を行う「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検出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

---

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は記述が容易な高水準の動的型付け言語が好まれるが，開発が進みプログラムの効率性が求められるにつれて，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，結局 C や Fortran など効率的に動作する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いて生じるジレンマを，Bezanson 他は”two langauge problem”[2, 3] と表現している．

2) Numpy[4] に代表される，高級な動的言語のインターフェースを通じて内部的に C や Fortran で書かれたルーチンを呼び出すような”two-tiered architecture”の試みは，two-language-problem に対する 1 つのアプローチであるが，以下のような問題点を持っている [2].

- 並列プログラミングにおいて複雑性が増大する
- 常に vectorization が求められ，不必要な一時オブジェクトの生成を避けることができない
- 複数言語間のインタフェースで生じるオーバヘッドにより，プログラム全体を最適化することが難しい
- 動作原理が複雑であり，結果としてユーザが内部を理解しそのシステムの向上に貢献することを妨げる

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

この節ではまず 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

Julia は **generic function（総称関数）ベースのオブジェクト指向言語**である[3]．数ある Julia の特徴の中でも，特に以下の 2 点は Julia プログラムに対する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．4 節において，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 [7].

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，実行時にある引数型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 [8] 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**[4]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．

---

[3] 通常のオブジェクト指向言語だと，メッセージを受け取るメインのオブジェクトがあり，それによってメッセージの処理方法が定まる．一方 Julia の場合は，引数全部のオブジェクトの型によって，generic function のどの実装（メソッド）が実行されるかが決まる点（2.1.1 参照）が異なる．また Julia のオブジェクト指向は，Java や Ruby などの言語にみられるようなクラスベースのオブジェクト指向ではないことにも注意されたい．

[4] ただし，全ての引数型に対して code specialization が行われるわけではない．ヒューリスティック的にある種の引数型に対する code specialization を制限することで，過剰な code generation（2.1.2）が行われてしまうことを避けている．ヒューリスティックの具体的な内容については [9, p. 11] が詳しい．

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[5]を考慮した上で，その引数型組に対し最も「特化した（”specific”[6]な）」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[7]，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[8]．

```
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

[5] Julia の実行時において，全てのオブジェクトの型は **concrete type**（**具体型**）である．一方，**abstract type**（**抽象型**）は各 concrete type の関係性を記述し型システムの階層性を表現する上での（type lattice の node としての）役割だけを持ち，実行時にインスタンス化されることはない．

[6] この「最も”specific”なメソッド」の選択は Julia の言語機能の中でも特に重要なものの 1 つであり，generic function のメソッドテーブルを引数型組の”specificity”によりソートすることにより自然に実装することができる．しかし，一方の引数型組がもう一方の subtype である場合など”specificity”が明らかな場合だけでなく，Julia の型システム内で表現しうる**全ての**引数型組の組み合わせに対して適用できる”specificity”は自明ではない．そのアルゴリズムは現在のところ経験的に実装されており，その形式は informal にしか与えられていない [7].

[7] 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．

[8] 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`

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[9)]．

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[10)]，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[11)]，`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

9) 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 [10]．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．

10) 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．

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-procedual 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 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[12)]．

```
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[11] 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

---

11) Julia は現在のところ逆方向の型推論（”backwards type flow”）を行うことができない [2]．コンテナ型の変数のパラメータ型を変数宣言以降のコードの操作から推論することはできず，この場合 `ints` には `[]` の型 `Array{Any,1}` がそのまま与えられる．

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

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[13]，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 [7]．

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，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（小々節 2.2.3 参照）からである．

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 [12, 13]．

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**[14] 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**[15] を行い **lowered form** を得る．
4. 簡単なヒューリスティック[16] 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

[13] こうしたパースタイムにおけるマクロ展開（Julia の実行プロセス参照）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．コードを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．

[14] ”AST”は”Abstract Syntax Tree”（抽象構文木）の略．

[15] 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 [14] 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．

[16] `https://github.com/JuliaLang/julia/blob/master/src/toplevel.c#L588-L819`

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

特に 5 から 7 のプロセスを指して，**code generation** と呼ぶ．

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 により強力で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，**漸進的型付け** [15] の考えに基づき，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 [6]．

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[17]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**[18] 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が，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[19] 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

---

[17] 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．

[18] 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．漸進的型付けの考えに基づく型システムにおいては，動的型を導入することで，静的な型検査を部分的に無視し，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動的型を導入する．

[19] 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．

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） [16]，Steep（Ruby） [17]，TypeScript（JavaScript） [18] 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 [19]．また同様の理由から編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 [20, 18]．
- [6] 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 [19]．

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，静的な型付けを意識し型シグネチャを書かかなくてはならない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[20]．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て用いることで改善できる可能性がある[21]．

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で**抽象解釈** [23, 24] 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マ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

[20] もっとも，型推論を用いる場合プログラマは必要最低限の型を与えればよいことや，型シグネチャ自体がある種のドキュメンテーションとしての役割を果たし得ることを考えると，型の記述に伴う冗長性はそこまで大きな問題としてみなされない場合もあるだろう．

その意味で，静的な型検査に伴う最大のデメリットは，静的な型付けを意識しなくてはならないことそのものであると考えることもできる [21]．プログラムの型検査を通すために，プログラマの思考方式は良くも悪くも変化させられ，また必要に応じて型検査を通すためだけのテクニックを覚えなくてはならない（”Expression Problem”[22] のように，型システムが自然かつ明快に表現することができない問題が存在する）．漸進的型付けは，前述の動的型を許すことによりこの問題にアプローチしているとも言えるが，後述するように動的型の存在自体が漸進的型付けにおける問題点であるとみなすこともできる．

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[21)]．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 や JavaScript での取り組み [25, 26, 27] 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．

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 [6, 25] 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 節）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．

■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る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，抽象解釈中の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 [28] 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 2 月 3 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 （Julia の実行プロセスと中間表現参照）の段階の

---

21) 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 [6].

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．

```
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

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[22]．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[23]，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[24]．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．

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[25]．

---

[22] 2020 年 2 月 3 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 v3.0.0 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`

[23] `Foo` が abstract type であることに注意せよ．

[24] 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`

[25] 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`

3　型プロファイラの設計

この節では，まず本論文の取り組みとして型プロファイリングによる解析手法を採用する理由を前節の先行研究（2.2）を踏まえて説明する．次に，Julia の型推論システムについて，そのアルゴリズムや停止性などに関する基本的な性質を説明する．というのは，本論文で提案する型プロファイラの設計方針や性質は，結局その抽象解釈のメインルーチンとなる Julia の型推論システムにより決まるからである．そして最後に型プロファイラの実装である TypeProfiler.jl の設計とその性質を説明する．

3.1　Why "type profiler" ?

Julia プログラムに対する静的な型エラー解析の手法として，本論文で抽象解釈による型プロファイリングを採用した理由は次の 3 点である．

1. Julia の言語機能として実装されている型推論システムを再利用することができる．
   Julia の型推論は実用上の JIT コンパイルに耐え得る正確性とスケーラビリティをすでに備えており，また今後も発展していくことが期待される．この型推論システムを再利用することで，抽象解釈を用いた型プロファイリング（2.2.2）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，型プロファイラを比較的容易に実装できる．
2. Julia 言語のプログラミングパターンと，追加的な型アノテーションによる検査方式との相性が良くないと思われる．
3. 型プロファイラは，素の Julia プログラムに対して解析を行うことができるため，気軽に導入を試すことができる．そのためまだ静的な型エラー解析が広く行われていない現行のエコシステムにおいても受け入れやすいと思われる．将来的な Julia プログラムに対する静的な型エラー解析の必要性を考える上での，足がかりと捉えることもできる．

2 点目に関しては，本論文執筆時点ではまだ理解が未整理であるが，ここでは現状における理解を説明しておく．先述したように，ソースプログラムに対する型アノテーションは既に Julia の言語機能として存在し，型アノテーションの導入自体は自然に行うことができる．一方で，Julia プログラムにおいて型アノテーションは generic function の dispatch をコントロールする上で非常に重要な意味を持っており，code selection のメカニズムにおいて型アノテーションは Julia プログラムの多相性を大きく左右し[26)]，プログラムの意味論に根本的な影響を与える．そのため，intra-procedural な解析が可能な程度に引数型を限定できるような形で静的な型エラー検査を目的とした追加的な型アノテーションを与える場合，Julia プログラムが本来持ちうる多相性を保つのは難しく，Julia の言語機能を大きく損なってしまう可能性が高い．

3.2　Julia の型推論システム

Julia の実行過程においては，メソッド呼び出し時，与えられた引数組型に対するそのメソッドの code specialization がまだ行われていない場合（つまりそのメソッドの，その引数型組に対する最適化されたコードのキャッシュがまだ存在しない場合），呼び出しの最適化を目的として型推論が行われ，型推論の結果に基

[26)] 型アノテーションを付けないことは，`Any` 型のアノテーションを施すことと同義である．関数の引数に型アノテーションを施さないことは最も generic な場合に対応するメソッドを定義することを意味し，引数に型アノテーションを施すことは一般にそのメソッドの多相性を低下させる．このことから，プログラマは型アノテーションにより Julia プログラムの多相性を調整していると考えることもできる．

づいて最適化が行われる（Julia の実行プロセス参照）.

Julia のプログラムの式や変数の型は，プログラムを型レベルで抽象解釈することで推論される．Julia の型推論に用いられている解析手法は元々，[23, 24] で提唱されたものであり，Bezanson らによる論文 [7, 9, 2] ではこの手法のことを指して，**data-flow analysis** という用語を用いており，以降本論文においても使用する．

Julia の型推論は，ML などの静的型付け言語における型推論とは根本的に異なる．静的型付け言語においては通常，unification ベースの型推論 [29] が用いられ，コンパイル時におけるプログラムの正しさの証明の一環としてソースレベルの型システムからでは自明に得られないプログラムの性質を発見するために推論が行われるのに対して，Julia の型推論は，もとより code specialization によるパフォーマンスの向上の一環として，プログラムの性質を「できる限りの範囲で詳細に [7]」調べるために行われる．

data-flow analysis は以下のような性質を持つ．

1. 推論が柔軟である．例えば，変数は（明示的なキャストをせずとも）プログラムの各時点において異なる型を持つことができ，また推論により型は必ずしも決定されなくてもよいなど，より動的型付けに即した性質を備えている．
2. 型システムが表現可能な型の数が有限でない場合においても，widening と呼ばれる手法を用いることでアルゴリズムの停止性を保つことができる．また，ヒューリスティックを用いて推論中の型をナイーブな実装よりも早くより抽象的な型へ遷移させることで，推論の収束を早めることができる．
3. 型推論の実装と言語仕様とを切り離すことが可能である．例えば Julia においては，推論の correctness を保つ限り，型推論の結果によって Julia プログラムの意味論が変わることはなく，ただそのパフォーマンスが変わるだけである．このことはつまり，型推論のルーチンを改善しようとするときに言語仕様の変更を気にする必要がないことを意味し，結果としてコンパイラの開発スピードを早めることができる．

以下では，まずこの Julia の型推論システムの基本的なアルゴリズムを説明する．その次にヒューリスティックを導入することによりこのアルゴリズムの停止性が与えられることを説明し，最後に Julia の型推論ルーチンの推論結果の correctness について言及する．

3.2.1 アルゴリズム

Julia の実行プロセスで述べたように，型推論は generic function がある引数型組を伴って初めて呼ばれたときに引き起こされ，Julia プログラムを **SSA 形式（Static Single Assignment Form）**[14] に変換した lowered form に対して行われる．この SSA 形式は surface AST とは異なり，木構造のデータ構造ではなく，関数呼び出しや条件分岐など「ある 1 つのことを行う」**statement** の連なりである．SSA 形式においては，各変数は使用される前に必ず定義され，また一度しか代入されない[27]上，control flow はネストせず，分岐先と対応する”basic block” への goto として表現される．こうした性質から SSA 形式は，定数伝播（constant propagation）やデッドコード削除（dead code elimination）等といった最適化を行うための分析に適している．以下，単に”statement”という場合，この SSA 形式の statement を意味する．

Julia の型推論は，[30] で提案された graph free な最大不動点計算のフレームワークを相互再帰関数も扱えるように拡張したアルゴリズムにより実装されている．基本となるアイディアは，それぞれの statement の副作用を決定し，その型情報を，その statement から control flow により到達可能な全ての statement に伝播させ，プログラムの各時点おける状態をトレースするというものである．

[27] ここで，SSA 形式中の「変数」は元のプログラム中の変数と一対一に対応しない．元のプログラム中における変数の再代入は，SSA 形式においては新たな変数への代入として変換される．

以下にその基本的なアルゴリズムを説明する [31, 2].

Julia の型推論は，1 つの関数呼び出し内におけるプロセス（local type inference）と，複数の関数呼び出しにまたがって行われるプロセス（interprocedural type inference）の 2 つに分けて説明することができる[28)].

■local type inference

W を次に推論されるべき statement の集合とする.

まず，推論が始まる前に，推論されるメソッド内の状態を以下のように初期化する.

- メソッドのエントリポイントとなる statement を W に追加する
- メソッドの引数それぞれの型は推論されている型をそのままセットする
- 全てのローカル変数を `Bottom` 型[29)]にセットする
- メソッドの返り値型を `Bottom` 型[29)] にセットする

その後，推論は以下のプロセスを W が空になるまで繰り返す.

1. W に残っている statement を 1 つ取り出す（以下”currrent statement”と呼ぶ）
2. current statement の副作用を決定する．例えば代入は変数の型を変化させる．もし current statement が関数呼び出しである場合，その関数の返り値型を推論する（このサブルーチンについては後述する）．推論を再帰的に行うことにより，すべての statement の型を導出することができる
3. current statement から到達可能な**全ての** statement （以下そのそれぞれを”next statement”と呼ぶ）を W に追加し，そのそれぞれに対し current statement の型情報を伝播させる．このプロセスは 2 つの statement の変数の型を **type merge** させることで行われる．無限ループを避けるため，current statement との type merge により next statement に含まれるいずれかの変数の型が変化する場合にのみ，next statement は W に追加される．この type merge において，アルゴリズムの収束を保証するため，widening 3.2.2 と呼ばれる手法が用いられる.

current statement が関数呼び出しである場合，以下のサブルーチンにより関数呼び出しの返り値型が推論される.

- generic function の呼び出しである場合，まず呼び出しの引数組型とマッチするメソッドを求める[30)]．マッチするメソッドは，呼び出し引数組型とメソッドシグネチャの型の type intersection （greatest lower bound，または交差型）が `Bottom` 型とはならないものとして求まる．マッチしたメソッドのそれぞれに対して以上と同様の型推論を行い，全ての推論結果を merge することで，その generic function の呼び出しの返り値型を求める.
- 呼び出される関数が generic function でない場合，それは built-in function の呼び出しである．built-in function の数は限られており，その返り値型は hard code された型遷移関数により計算される.

T をこのサブルーチンの型遷移関数とし，f を generic function，t_{arg} を推論された引数組型，s と g をそれぞれ f のメソッドの引数のシグネチャと実装とすると，generic function の呼び出しに対するサブルーチンは以下のように表される.

$$T(f, t_{arg}) = \bigsqcup_{(s,g)\in f} T(g, t_{arg} \sqcap s)$$

[28)] 正確には，この 2 つとは別に，comprehension（内包表記）に対する推論のサブルーチンがある種の special case として存在する [31].

[29)] `Bottom` 型に関する説明.

ここで，⊔，⊓ はそれぞれ type intersection と type merge に対応する演算である．

■interprocedural type inference

さらに，再帰関数の呼び出しに対応するため，以上のプロセスを generic function の呼び出しにまたがって interprocedural に管理する，次のような手続きを考える必要がある．

推論の過程において，呼び出しが生じた generic function とその引数型組の組み合わせを保持しておく．同一の呼び出しが生じた場合，それらの呼び出し間で生じる全ての関数呼び出しを相互再帰関数呼び出しとして記録し，それらの返り値型がまだ完全に推論されていないことを示す *incomplete* タグを付与する．*incomplete* タグが付与されている全ての関数の返り値型が fix-point（不動点）に至るまで，この相互再帰関数呼び出しのサイクルに対して推論のプロセスを繰り返す[31)]．

3.2.2 停止性とヒューリスティック

上記のアルゴリズムを使用した data-flow analysis は以下の性質を満たす場合に，必ず収束し停止することが保証される [30, 31]．

単調性 2 つの型を merge したときに必ずより抽象的な型にならなければならない．また，data-flow analysis における関数呼び出しは単調でなければならない．つまり引数型が抽象的であるほど，推論される返り値型も抽象的でなければならない．

有限性 型システムにおいて表現され得る型が有限である．

有限性に関して，Julia の型システムは `Union` 型や `Tuple` 型，そしてコンテナ型の存在により，無限の型が作られ得る（型システムは高さが有限でない lattice として表される）ので，そのままではこの条件を満たさない．そこで，**widening** と呼ばれる手法を用いて型推論時における type lattice の高さが有限となるように強制する．この時，ヒューリスティックを用いて推論中の型をより早く抽象的な型に遷移させることで，型推論自体の収束を早めている．

3.2.3 correctness

[30] のアルゴリズムを用いた data-flow analysis には，推論された型について次の性質が成り立つ [9]．

定理 (correctness)**.** data-flow analysis により推論された型は，実行時においてありうる型を全て含んでいる．

3.3 TypeProfiler.jl の設計と性質

この節では本論文で提案する型プロファイラの実装である TypeProfiler.jl の設計と，その性質に関する説明を行う．

3.3.1 設計

実装は `https://github.com/aviatesk/TypeProfiler.jl` で公開している．

---

30) この時，引数それぞれの型は必ずしも concrete type ではないことに注意（Julia の型推論においては，widening により全ての型が concrete type とはならない）．そのため，マッチするメソッドは複数ある可能性がある．

31) interprocedural type inference の基本的のアイディアは以上のようであるが，特に相互再帰呼び出しのサイクルが複数存在する場合などで，実装により型推論のパフォーマンスは大きく左右される．Julia の型推論アルゴリズムにおいて interprocedural type inference が実際どのように実装され，技術的な課題点が克服されているかについては [31, 32] が詳しいが，ここでは詳細な説明は避ける．

3.3.2　性質

TypeProfiler.jl の性質として，3.2.3 を解釈しなおすことで，次の系を与えることができる．

系 (型プロファイラの型安全性)**.** 型プロファイラが公理として与える statement において，推論された型がその statement に対して型エラーを起こさないような型の集合により包含されている場合，その statement は型安全である．

4　評価と今後の課題

この節では TypeProfiler.jl の性能を評価し，また現状で把握している課題点についても報告する．

4.1　実験と比較

現時点では TypeProfiler.jl の性能の正確な評価までは行うことができていない．

4.2　今後の課題

4.2.1　built-in function のプロファイリング

Julia の型推論システムは，もともとエラー検出を目的に作られていないため，推論が built-in function の呼び出しに到達したとき，その呼び出しの引数型組が実行時エラーを起こし得るものであってもその情報を伝播させない場合がある．つまり，エラーが起きるか起きないか判断できないような引数型が与えられた場合，通常全ての実行時型を含む唯一の型である `Any` を返す．

この挙動は Julia の型推論がもっぱらパフォーマンスの向上を目的として行われていることを考えれば自然な挙動であるが，一方で TypeProfiler はエラーの検出を目的としているため，より保守的にエラーの可能性を報告する方が好ましい．

Julia の built-in function の数は数少ないとは言え，それらは手作業で実装される必要がある．まだプロファイリングが未実装の built-in function のサポートは必須の課題である．

4.2.2　マクロ展開に対するエラー検出

TypeProfiler.jl は，マクロを全て展開した後のプログラムに対して解析を行う．つまり，マクロ展開そのものに対する解析は行わない．

Julia のマクロ展開においてマクロの返り値となる式を計算する際，通常の関数呼び出しと同様の計算を行うことができるため，マクロ展開自体に対しても型プロファリングを行える方が望ましい[32)]．

しかし，Julia の code speicialization はマクロ呼び出しに対しては行われず，従って TypeProfiler.jl が利用する型推論ルーチン自体もマクロ展開に対しては利用できないため，現在の TypeProfiler.jl の設計ではマクロ展開に対する解析を実装することはできなかった．

マクロ展開に対するプロファイリングをどのように実装するべきかの検討は今後の課題である．

[32)] また，マクロ展開を抽象解釈したときに分かるのは，そのマクロの返り値となるプログラムの抽象値である．展開されたプログラムの抽象値と，そのマクロが使用されるプログラム部分との整合性を調べることにより，マクロが展開されるコンテクストに関するエラーを解析することも可能であると思われる．

5　まとめ

本プロジェクトでは Julia プログラムの型安全性を向上させる手法として，Julia の JIT に使用されている型推論ルーチンを再利用してプログラムを抽象解釈し，データ型エラーを静的に検出する型プロファイラを提案し，その実装を行った．

参考文献

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

[2] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, Vol. abs/1209.5145, , 2012.

[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] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: a structure for efficient numerical computation. *CoRR*, Vol. abs/1102.1523, , 2011.

[5] Julia Computing. Newsletter January 2020 - Julia Computing. `https://juliacomputing.com/blog/2020/01/06/january-newsletter.html`. (Accessed on 2020-01-22).

[6] 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.

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

[8] 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.

[9] Jeff Bezanson. Julia: An efficient dynamic language for technical computing. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2012.

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

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

[12] *Julia Documentation - Developer Documentation - Eval of Julia code*. (Accessed on 2020-01-27).

[13] *Julia Documentation - Developer Documentation - Julia ASTs*. (Accessed on 2020-01-27).

[14] Julia Documentation - Developer Documentation - Julia SSA-form IR. `https://docs.julialang.org/en/latest/devdocs/ssair/`. (Accessed on 2020-02-03).

[15] Jeremy Siek and Walid Taha. Gradual typing for functional languages. 01 2006.

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

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

[18] Microsoft. TypeScript - JavaScript that scales. `https://www.typescriptlang.org/`. (Accessed on

2020-01-19).

[19] 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).

[20] FaceBook. Flow - Adds static typing to JavaScript to improve developer productivity and code quality. https://github.com/facebook/flow. (Accessed on 2020-01-19).

[21] What is Gradual Typing — Jeremy Siek. https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/. (Accessed on 2020-02-03).

[22] Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pp. 238–252, 01 1977.

[24] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *J. ACM*, Vol. 27, No. 1, pp. 128–145, January 1980.

[25] 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).

[26] 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.

[27] Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. Towards type inference for javascript. Vol. 3586, 07 2005.

[28] julia vscode. StaticLint.jl - Static Code Analysis for Julia. https://github.com/julia-vscode/StaticLint.jl. (Accessed on 2020-01-24).

[29] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, No. 3, pp. 348 – 375, 1978.

[30] Markus Mohnen. A graph-free approach to data-flow analysis. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 46–61, Berlin, Heidelberg, 2002. Springer-Verlag.

[31] Jameson Nash. Inference Convergence Algorithm in Julia - Julia Computing. https://juliacomputing.com/blog/2016/04/04/inference-convergence.html, 2016. (Accessed on 2020-01-22).

[32] Jameson Nash. Inference Convergence Algorithm in Julia - Revisited - Julia Computing. https://juliacomputing.com/blog/2017/05/15/inference-converage2.html. (Accessed on 2020-01-30).