

令和 6 年度 卒業論文

漸進的型付けの意味における健全性 に焦点を当てた TypeScript チェッカ

A TypeScript checker with a focus on soundness in the

context of gradual typing

鈴鹿工業高等専門学校

電子情報工学科

青山研究室

糸川 優太朗

指導教員: 青山 敏弘

令和 7 年 1 月 17 日

概要

ここに要旨を書く！

目次

第1章	はじめに	1
第2章	概要	2
2.1	TypeScript	2
2.2	Gradual Typing	3
2.3	Siek と Taha の体系の型安全性	3
2.4	Criteria for Gradual Typing	5
2.5	TypeScript の型システム	6
2.6	TypeScript と Criteria for Gradual Typing	7
2.7	Safe TypeScript における対応	8
2.8	decaf が実現する静的解析	9
第3章	decaf の実装	10
3.1	型検査	10
3.1.1	decaf が型を検査する手順	11
第4章	実験	13
4.1	概要	13
4.2	実験内容	13
4.2.1	simple.tsx	13
4.2.2	middle.tsx	14
4.2.3	complex.tsx	15
4.2.4	very_complex.tsx	15
4.3	考察	16
4.4	decaf の改善点	17
第5章	まとめ	18
参考文献		20

第1章 はじめに

ソフトウェア開発が大規模化する状況において、コード品質の担保や開発速度の維持は重要な課題である。静的型付けの手法はこの課題に対処するための手段として広く用いられている。Web開発においては長い間JavaScriptが主要な開発言語として用いられてきたが、近年の大規模化するWeb開発においては静的手法の欠如が問題となった。

TypeScriptは、動的型付けシステムであるJavaScriptに静的型システムによる検査を部分的に導入できるプログラミング言語であり、gradual typingシステムであるとされる。Siekmannらは、動的型付けと静的型付けの融合を図るシステム全般が無闇にgradual typingを名乗ることを疑問視して、gradual typingシステムが満たすべき基準を提案した。

本研究では、この基準に照らしてTypeScriptがgradual typingの条件を満たすようなコンパイラを実装する。

第 2 章 概要

2.1 TypeScript

TypeScript は、Microsoft により開発されているプログラミング言語である。TypeScript は ECMAScript 5[1] を拡張した構文を持ち、既存の構文に型アノテーションを追加できる。また、それを補助するものとして、型エイリアスや関数オーバーロードを宣言する機能も用意されている。TypeScript コンパイラは型アノテーション及び型推論機構を基に TypeScript プログラムを静的に検査し、プログラムのある種の誤りを検出する機能を持つ。

TypeScript の最も大きな特徴のひとつは `any` 型の存在だ。ある値が `any` 型を持つ場合、その値に対する一切の操作は TypeScript コンパイラによるチェックの対象とならない。特に、潜在的に危険な操作であっても `any` 型を用いることで型検査を成功させることができる。これは時として、型情報と矛盾する挙動を実行時に引き起こしたり、実行時エラーという結果に繋がったりする。

TypeScript に `any` 型が存在する理由は、型をオプショナルなものとするためである。ここで、オプショナルというのは、プログラム全体を型検査しなくてもよいということを指していると考えられる。実際、TypeScript ではあらゆる型アノテーションが省略可能だ。TypeScript コンパイラは原則として、型アノテーションが提供されているところや、型アノテーションが無くても型を推論可能なところのみを型検査の対象とする。

このような「型アノテーションが存在するところのみを型検査の対象とする」という挙動を、静的型付け¹⁾の枠組みで表現するための道具が `any` 型である。TypeScript コンパイラは型アノテーションが存在しない変数を `any` 型として扱うことで、その変数が関わる部分の型検査を実質的に無効化する。そもそも、TypeScript コンパイラの型推論においては、変数の型は原則として宣言時に決定される。これは Hindley-Milner 型推論 [2] のような、変数の使われ方から型を推論できる方法とは大きく異なる特徴である。関数の引数も例外ではなく、一部の場合²⁾を除いて、引数の型は明示的に宣言しなければならない³⁾。

1) あるいは静的型システム

2) contextual type により型が推論可能な場合など

3) 多相関数も明示的に型引数を宣言しなければならない

それにも関わらず変数や引数の型アノテーションが省略された場合、TypeScript コンパイラはその変数の型の情報を得られない状態となる。TypeScript はこのような場合にその変数の型を `any` とすることで解決するというデザインを採用している。それ以外にも、変数の型が推論できないさまざまな場面において `any` 型が当てはめられる。

この特徴により、TypeScript は JavaScript からの移行を支援している。そもそも、ただの JavaScript プログラムは、型アノテーションが完全に省略された TypeScript とみなすことができる。この状態から徐々に型アノテーションを追加することによって、TypeScript による型検査が行われる部分を段階的に増やすことが可能だ。この段階において、TypeScript プログラムは型検査が行われる部分と行われない部分が混在している状態となる。

2.2 Gradual Typing

Gradual typing [3] は、Siek と Taha が 2006 年に提案した、静的型付けと動的型付けを融合させる手法のひとつである。静的型付けと動的型付けは異なる利点と欠点を持つものとして共存してきたものであり、両方の手法を取り入れる方法も古くから模索されてきた。Gradual Typing ではこれら 2 つの方式をひとつのプログラムの中に共存させることができる。特に、プログラマがアノテーションを用いて、プログラムのどこに静的な検査を適用できるという特徴を持つようなシステムが Gradual Typing と呼ばれる。当該論文では、特に関数型言語に対して Gradual Typing の要件を満たす型システムを提案している。

Siek と Taha の型システムでは、通常の型に加えて `?型` が導入されてる。`?型` が与えられた変数は静的なチェックが行われない。その点で、これは静的型検査の対象としないというアノテーションを表す型と見なすことができるものであり、TypeScript の `any` 型に相当する。

2.3 Siek と Taha の体系の型安全性

Gradual Typing においては、プログラムの正しさを静的に保証することは当然ながらできない。型によって静的な検査を無効化した隙に誤りが入りこむかもしれないからである。Siek と Taha の論文の序盤から例を引用する。

$((\lambda (x) (\text{succ } x)) \#t)$

これは彼らが提案した計算体系 $\lambda^?$ の式だが、TypeScript に直すと以下に相当する。

Listing 2.1: TypeScript における例

```
1 const succ = (x: number) => x + 1;
2
3 ((x: any) => succ(x))(true)
```

2.1 の式は、実行すると `true` という `boolean` 型の値が `succ` 関数に渡される。`succ` は `number` 型の引数を渡すべきなので、これは誤ったプログラムということになる。

ただし、上の式は型システムによる静的検査をくぐり抜ける。これは、`x` という `?型`⁴⁾ を持つ変数を経由しているからである。変数 `x` に値が入った時点で、型システム上ではその値が本来何型であるかという情報は消えている。また、`?型` はどのような型としても使用できることから、`number` 型を受け取る関数 `succ` に `x` を渡しても問題ない。

Siek と Taha の体系 $\lambda_{\rightarrow}^?$ においては上記のように誤ったプログラムはどのような結果になるだろうか。実はこの体系ではランタイムに誤りを検知する。これは、キャストの情報を値に保存するセマンティクスによって実現される。

Gradual typing においては `?型` の存在が注目されるが、それに加えてランタイムのチェックを合わせてひとつの理論であるということは強調するに値する。

この体系では、プログラムの意味は別の体系 $\lambda_{\rightarrow}^{<\tau>}$ のプログラムへの変換を通して理解される。上記のプログラムは以下のように変換される。

$((\lambda (x : ?) (\text{succ } <\text{number}>x)) <?>\#t)$

元々のプログラムとの違いは、`<number>x` や`<?>`のように値の前に型が書かれている部分がある点である。このような式はキャスト式と呼ばれる。このプログラムでは、「`boolean` 型の値を `?型` として使う」や「`?型` の値を `number` 型として使う」といった操作がキャスト式によって明示的に表現されている。

上のプログラムを実行すると、変数 `x` に`<?>\#t` が入るので、`<number>x` は`<number><?>\#t` という値になる。`#t` が `boolean` 型であることに留意すると、ここで `boolean` 型から `number` 型へのキャストが発生していることが明らかになる。これは誤りなので、ここでランタイムエラーが発生する。

当該論文では、彼らの体系の“型安全性”の証明が与えられている。`?型` の存在により型の誤りがランタイムに発生することは避けられないが、それは全て上述の CastError としてキャッチされる。ここでの型安全性は、上述のランタイム機構をすり抜ける型の誤りが発生しないという意味で用いられている。実際、上記でインフォーマルに説明した通り、型アノテーションに反する値がランタイムに発生することはない。

4) `any` 型

2.4 Criteria for Gradual Typing

前述の論文以降 gradual typing という言葉は知名度を増したが、その結果として gradual typing という言葉が何を指しているのか曖昧になり、静的型付けと動的型付けを融合させる試みが無秩序に gradual typing を名乗るという問題があった。Siek ら [4] の 2015 年の論文はこの問題を指摘し、gradual typing を名乗るシステムの条件を整理した。

Gradual typing システムが満たすべき条件は以下の通りだ（論文の Theorem 1 から 5）。2006 年のオリジナルの体系はこれらの条件を全て満たすことが示されている。

1. 静的型付けの体系を内包する。すなわち、全ての型がアノテートされている⁵⁾ プログラムは、ただの静的型付けシステムと同じ挙動をする⁶⁾。
2. 動的型付けの体系を内包する。すなわち、型アノテーションが無い⁷⁾ プログラムはただの動的型付けシステムと同じ挙動をする。また、任意の式の型が?である。
3. 健全性。前節で述べたように、ランタイムにキャッチできない型の誤りは発生しない。
4. Blame-Subtyping Theorem. $T_1 <: T_2$ ならば、 T_1 から T_2 へのキャストはランタイム型エラーの原因とならない。
5. Gradual Guarantee. すなわち、静的型検査に成功するプログラムから型アノテーションを減らしても静的型検査は成功し、型も変わらない。また、プログラムから型アノテーションを減らしても動作が変わらない⁸⁾。

なお、4. に出てくる部分型関係は以下のように定義されている（前述の論文 [4] から引用）。この論文では今まで?型と呼んでいたものが*型となっている。なお、*というのは基本型または $* \rightarrow *$ である。

$$B <: B \quad * <: *$$

$$\frac{T <: G}{T <: *} \quad \frac{S_1 <: T_1 \quad T_2 <: S_2}{T_1 \rightarrow T_2 <: S_1 \rightarrow S_2}$$

$$T <: T$$

この定義からは $number \rightarrow *$ や $* \rightarrow number <: number \rightarrow number$ 、また $* \rightarrow number <:$

5) ?型を用いない

6) 静的検査の結果も、実行結果も同じである

7) 全ての型が?である

8) ただし、キャッチできるランタイム型エラーが減る可能性はある

- * などが成り立つ。一方, $number \rightarrow * <: *$ のようなものは成り立たない。
- 部分型関係における * の扱いは注目に値する。特に, $* <: T$ となる T は * だけである。
- * は静的解析において型エラーの原因にはならないが、ランタイム型エラーの原因となる、それゆえ、特に上述の定理 4 の観点からは他の型の部分型とはなれない。

2.5 TypeScript の型システム

TypeScript の型システムは、完全に静的なシチュエーションにおいても健全性を持たないことが広く知られている。

静的型検査の側面において、TypeScript の any 型は Siek と Taha の gradual typing 型システムにおける ? 型と同じ特徴を持つ。すなわち、any 型の値は他の任意の型の値が必要な場面において使用可能である。また、any 型が求められる場面においても任意の型の値を any 型として使用できる。

一方で、TypeScript のランタイムの挙動は $\lambda?_x$ とは大きく異なる。具体的には、ランタイムの型チェックを行われない。そもそも、TypeScript には型の情報によってランタイムの挙動が変化しないという大原則があるため、2.3 で説明したような型情報ベースのランタイム型チェックは趣旨に沿わない。このようなデザインを取っている理由として、型システムによるランタイムのオーバーヘッドを避けることが第一に挙げられる。また、型によるランタイムの挙動への影響を排除することで、JavaScript ユーザーから見て TypeScript のコンパイル処理の透明性を向上するという目的があると推測される。

上記の理由から、any 型の存在に起因する型のミスマッチは、 $\lambda?_x$ における CastError のような形でキャッチされるのではなく、別の予期せぬ形で現れることとなる。これは、TypeScript の型システムでは、gradual typing の意味での健全性が失われていることを意味する。

以降でもランタイム型エラーという語を用いるが、これは gradual typing システムによるランタイムのチェックによって検出されるものを指す。実際の TypeScript では、型のミスマッチは呼びだそうとしたメソッドが存在しないことによる実行時エラーや、undefined や null に対するプロパティアクセスしたことによる実行時エラーなど、その結果はさまざまな形で現れる。以降では、これらは前述のランタイム型エラーとは区別し、予期せぬ結果と呼んでいる。ランタイム型エラーはプログラムの動的な部分のチェックが正しく行われた結果として現れるのに対し、予期せぬ結果は、システムの健全性が失われた結果として現れるものである。

2.6 TypeScript と Criteria for Gradual Typing

TypeScript の型システムの性質を Siek らの criteria for gradual typing に照らし合わせてみる。

1 (静的型付けの体系を内包する) と 2 (動的型付けの体系を内包する) については成り立つ。というよりも、1 と 2 は変なセマンティクスを持つ体系を除外するための条件であると考えられるので、これらの条件は今回の設定ではあまり意味を持たない。1 に関しては TypeScript と比較するための静的なシステムが必要だが、それは `any` を取り除いた TypeScript そのものである。

2 については比較対象の動的システムは JavaScript となるが、上述の性質から 2 が成り立つことは明らかである。ただし、この条件を TypeScript に当てはめる際には注意すべき点がある。本来の TypeScript では「任意の式の型が `any` 型である」は満たされない。例えば次のプログラムにおいて変数 `v` は `number` 型である。

```
1 const x: any = 123;
2 const y: any = 456;
3 const v = x * y;
```

しかしながら、これは*演算子の挙動によるものである。Siek らが本来対象としているのが関数型言語であることを鑑みると、*のような計算も全て関数としてみなすのが適切である。Siek らの論文にも、2 の条件については組みこみ定数や関数も全て * 型として見なすものと定義されている。実際、このことをより忠実に反映した以下の TypeScript プログラムでは変数 `v` の型は `any` となる。

```
1 const mul: any = (x: any, y: any) => x * y;
2 const x: any = 123;
3 const y: any = 456;
4 const v = mul(x, y);
```

3 つ目の条件である健全性については、すでに議論した通り、明らかに満たされない。そもそもランタイムのチェックがまったく行われないからである。

4 についても状況設定から議論する必要がある。部分型からのキャストはランタイム型エラーの原因にならないという条件だが、そもそも全くランタイムの型チェックが行われない設定では意味のある主張ではない。一方で、その他の予期せぬ結果もここでのエラーに含めることも考えられる。ただし、その場合は健全性が失われていることから反例を作るのは簡単である。

最後の条件、gradual guarantee については、TypeScript の言葉で言い換えれば、型ア

ノテーションを減らすというのは型註釈を何らかの型から `any` に変えることを指す。ランタイムの動作については、前述の性質から型註釈が `any` に変わってもランタイムの動作が変わらないことは自明である。一方で、静的型検査については自明でない。

以上の 5 条件について議論した結果において、TypeScript が gradual typing の意味での健全性を失っていることが特徴として現れている。ランタイムの型チェックを行わないことは TypeScript の根本的な言語デザインであり、その点において gradual typing とは乖離していることが分かった。

2.7 Safe TypeScript における対応

前節では、TypeScript が gradual typing の意味において健全性を持たないことで gradual typing の基準を満たしていないことを指摘した。この問題に対応するアイデアのひとつが Safe TypeScript[5] である。これは、健全性を保ったバージョンの TypeScript として Microsoft Research により開発されたものである。

Safe TypeScript が健全性を得るために行なった変更は大きく分けると 2 つある。1 つは型システムへの変更により静的検査における健全性を確保することであり、もう 1 つはランタイム型チェックの導入により gradual typing の意味での健全性を確保することである。実際の論文では、ランタイム型チェックのオーバーヘッドを削減するための工夫が述べられており、それらによってランタイム型チェックのオーバーヘッドが 15%程度に抑えられたことが報告されている。

ランタイム型チェックの一例が下の図に現れている（当該論文 [5] から引用）。関数 `f` の返り値である `x.f` が本当にアノテーション通りの `number` 型であるかどうかランタイムにチェックするための `RT.check` という呼び出しが追加されていることが分かる。

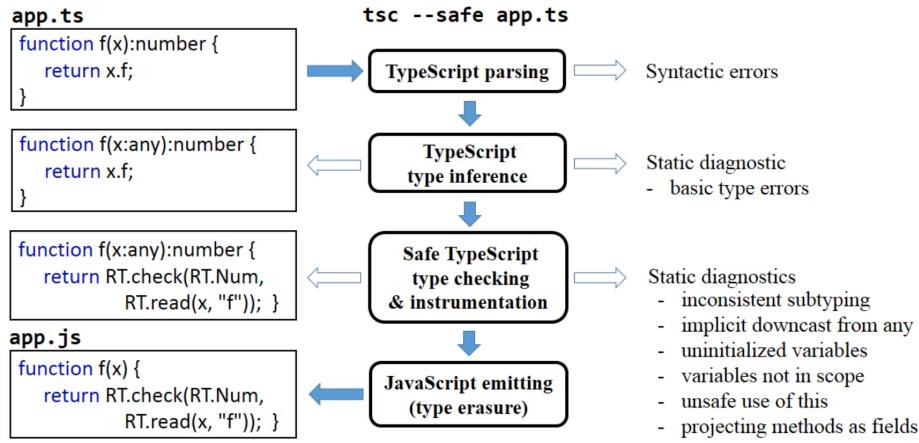


Figure 1: Architecture of Safe TypeScript

図 2.1: Safe TypeScript におけるランタイム型チェックの例

Safe TypeScript は明確に gradual type system を名乗っている。実際、論文で主張されている通り Safe TypeScript は gradual typing の意味での健全性を備えている。

しかし、Safe TypeScript ではオーバーヘッドを減らす様々な工夫をしてもなお、15%のオーバーヘッドが存在する。

2.8 decaf が実現する静的解析

decaf は TypeScript の型システムを拡張し、静的解析することで gradual typing の意味での健全性を保つことを目指している。

decaf は型情報を用いたフロー解析をすることで、プログラムの挙動を静的に解析する。この解析により、型情報に反する挙動がプログラムに含まれている場合、それを検出できる。この検出はランタイムにおける型エラーを検出することに相当する。

また、decaf では型エラーを静的解析時に検出することで、実行時の挙動をオーバーヘッド無しで保証できる。

第3章 decaf の実装

decaf は Rust を用いて実装された。TypeScript のパースは `oxc_parser`¹⁾を参考に実装した。また型の推論と検査は `tsc`²⁾を参考に実装した。

decaf は <https://github.com/re-taro/decaf> からインストールが可能である。Rust のパッケージマネージャである `cargo` があれば、以下のコマンドでインストールが可能である。

Listing 3.1: decaf のインストール

```
1 $ git clone git@github.com:re-taro/decaf.git
2 $ cd decaf
3 $ cargo install --path .
4 $ decaf --help
```

3.1 型検査

decaf は任意の TypeScript³⁾ ファイルを入力として受け取り、型検査する。

3.2 ように実行すると、標準出力として 3.3 のような結果が得られる。

Listing 3.2: decaf の型検査

```
1 $ decaf check <file>/ .tsx?$/
```

Listing 3.3: decaf の型検査結果

```
1 error:
2     └── all.tsx:726:3
3
4 726 | obj.prop2;
5   | ~~~~~ No property 'prop2' on { prop: 3, prop2: 6 } | { prop: 2 }
6
7 ---
8
9 Diagnostics: 446
```

1) https://docs.rs/oxc_parser/latest/oxc_parser/

2) <https://github.com/microsoft/TypeScript/blob/v5.1.3/src/compiler/checker.ts>

3) .tsx を含む

```
10 Types: 5780
11 Lines: 2239
12 Cache read: 285.954µs
13 FS read: 169.096µs
14 Parsed in: 8.294198ms
15 Checked in: 4.887375ms
16 Reporting: 204.832µs
```

3.1.1 decaf が型を検査する手順

decaf が型検査する手順は以下の通りである。

1. 入力ファイルをパースし, AST を生成する.
 - (a) ここで変換される AST は ESTree⁴⁾ や swc⁵⁾ のものとは異なり, decaf 独自の AST である.
2. 生成された AST を decaf の型検査機が解釈しやすい形に変換する.
 - (a) この変換により, 型検査に他の言語で実装されているモダンなアルゴリズムを適用できるようになる.
 - (b) この形を decaf では TypeID と呼んでいる.
 - (c) この機構を decaf では変換機と呼んでいる.
3. 型検査する
 - (a) decaf の変換機はプログラムの値や構造を型として, 関数やブロックの振る舞いをイベントとしてエンコードする
 - i. 型やイベントは AST の走査中に生成され, decaf の型検査機に渡され, 使用される.
 - ii. decaf は型をその値が取りうる可能性の集合として捉える.
 - iii. 条件分岐やループに差し掛かった時, そのブロック内で取りうる型の集合を広げる.
 - 3.4 の例では, `response` の宣言時, 型は `Response` であり, プロパティ `data` の型は `any`⁶⁾ である.
 - 条件 `response.data instanceof Date` が真である場合, `response.data` の型は `Date` である.
 - 条件 `response.data instanceof Date` が偽である場合, `response.data` の型は `Date` 型の補集合を取った型である.

4) ECMAScript の抽象構文木の実質的な標準

5) <https://github.com/swc-project/swc>

6) ここで `any` 型は, 便宜上型の全集合とする.

- (b) AST を走査しながら型の集合を広げていくことで、実行時に起こりうるすべての可能性を型チェック時に考慮できる。
- i. 従来の TypeScript Compiler は分岐の評価は、実行されるかされないかのみを考慮している。
 - ii. decaf は分岐を非決定論的に扱い、すべての分岐を評価し⁷⁾型を拡大する。

Listing 3.4: 例

```
1  async function f(name: string) {  
2      const response = await fetch(`/post/${name}`).then(res => res.json());  
3      if(response.data instanceof Date) {  
4          return response.data;  
5      } else {  
6          return response.data;  
7      }  
8  }
```

7) 条件が自明で真である場合などは除く。

第 4 章 実験

4.1 概要

3.1.1 で decaf の型検査手法について述べた。述べられている通り、tsc と比べて decaf はより厳格な型システムを実現するために tsc よりも多くの経路を型検査する。そのため、decaf は tsc よりも型検査に時間がかかると予想される。

本章では、decaf の型検査の性能を評価する実験をする。具体的には、decaf のテストスイートとして用意している 350 件の構文¹⁾を使用して作成したソースコードを、decaf と tsc で型検査をし、その結果を比較する。

4.2 実験内容

decaf のテストスイートをソースコードに落とした simple.tsx と、それの 10 回、20 回、40 回結合した middle.tsx, complex.tsx, very_complex.tsx の計 4 つのファイルを用意した。それぞれに対して decaf と tsc で型検査をし、その結果を比較した。ベンチマークツールには mitata²⁾を使用した。

4.2.1 simple.tsx

4.1 に simple.tsx を decaf と tsc で型検査した結果を示す。

1) <https://github.com/re-taro/decaf/tree/main/checker/specification> の specification.md と staging.md

2) <https://github.com/evanwashere/mitata>



図 4.1: simple.tsx を decaf と tsc で型検査した結果

ベンチマーク結果を見ると、decaf は tsc の 14.01 倍の速度で型検査をしていることがわかる。

4.2.2 middle.tsx

4.2 に middle.tsx を decaf と tsc で型検査した結果を示す。



図 4.2: middle.tsx を decaf と tsc で型検査した結果

ベンチマーク結果を見ると、decaf は tsc の 1.99 倍の速度で型検査をしていることがわかる。

4.2.3 complex.tsx

4.3 に complex.tsx を decaf と tsc で型検査した結果を示す。



図 4.3: complex.tsx を decaf と tsc で型検査した結果

ベンチマーク結果を見ると、tsc は decaf の 1.31 倍の速度で型検査をしていることがわかる。

4.2.4 very_complex.tsx

4.4 に very_complex.tsx を decaf と tsc で型検査した結果を示す。



図 4.4: very_complex.tsx を decaf と tsc で型検査した結果

ベンチマーク結果を見ると、tsc は decaf の 3.57 倍の速度で型検査をしていることがわかる。

4.3 考察

各ベンチマークにおいて、常に decaf が tsc よりも速い結果とはならなかつた。そこで、型検査機ごとのベンチマークケースごとの速度を棒グラフにしたもののが以下である。



図 4.5: tsc の型検査速度



図 4.6: decaf の型検査速度

4.5 を見ると、ベンチマークケースごとの速度の平均が、272.67ms ~ 451.67ms の範囲にある。対して、4.6 を見ると、ベンチマークケースごとの速度の平均が、18.66ms ~ 1,590.00ms の範囲にある。結果として、プログラムのサイズが大きくなると decaf の型検査速度は遅くなることがわかった。

これは、decaf が tsc と比べてより多くの情報をプログラムから抽出し、より厳格な型システムを表現しているからだと考える。4.5 の数値の推移を見ると、プログラムを n としたとき、 $O(n)$ の時間で型検査していることがわかる。一方で、4.6 の数値の推移を見ると、プログラムを n としたとき、 $O(n^2)$ の時間で型検査していることがわかる。これらの計算量は、それぞれの型検査機のアルゴリズムによるものである。3.1.1 で述べた通り、decaf は tsc よりも多くの経路を型検査するため、より厳格な型システムを実現している。そのため、decaf は tsc と比較したときに巨大なプログラムに対しては遅いという結果になった。

一方で、4.2.1 のような標準的なケースにおいて、decaf は tsc よりも速い結果となつた。これは若干の結果論を含むが、4.2.1 のケースだけでもソースコードの長さは、2949 行もある。基本的にこの行数のコードが 1 つのファイルに書かれるのは技術負債として避けるべきである。そのため、decaf は tsc と比べてより厳格な型システムを実現しているが、標準的なケースにおいては tsc よりも速い結果となつたと言えるだろう。

4.4 decaf の改善点

まず、現在の TypeScript がサポートしている構文³⁾の全てに対応していない。そのいくつかは decaf の型の厳密性を実現するために実装をしないと決めているものもある。しかし、decaf の内部実装や仕様を熟知していないと、decaf の型検査が通らない理由は分からぬ。そのためより多くの構文をサポートすると同時に、分かりやすいエラー形式を提供することが望ましい。

また、3.1.1 で述べた通り、decaf は tsc よりも多くの経路を型検査するため、大きなプログラムに対しては遅いという問題がある。decaf の型検査に用いられるTypeID は、不可分であり本来はパースを並列で行える。だが、現在の実装ではパースを逐次で行っているため、TypeID の生成に時間がかかっている。そのため、TypeID の生成を並列で行うことによって、decaf の型検査速度を向上させることができるだろう。

3) 執筆時点での最新バージョンは v5.7.3 である

第5章　まとめ

謝辞

最後に、電子情報工学科教授の青山敏弘教授からは本研究についての適切なご指導を賜りました。感謝を申し上げます。

参考文献

- [1] Patrick Charollais. *ECMA-262, 5th edition, December 2009*. ECMA International, 12 2009.
- [2] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, No. 3, pp. 348–375, 1978.
- [3] Jeremy Siek and Walid Taha. Gradual typing for functional languages. 01 2006.
- [4] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Vol. 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [5] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *POPL ’15 Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 167–180, January 2015.