

令和 6 年度 卒業論文

静的解析と実行時性能に焦点を当て
た TypeScript チェッカ

A TypeScript checker with a focus on static analysis and

runtime performance

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
第3章	decaf の実装	3
3.1	型検査	3
3.1.1	decaf が型を検査する手順	4
3.1.2	型と集合	5
第4章	実験	6
4.1	概要	6
4.2	実験内容	6
4.2.1	simple.tsx	6
4.2.2	middle.tsx	7
4.2.3	complex.tsx	8
4.2.4	very_complex.tsx	8
4.3	考察	9
4.4	decaf の改善点	10
第5章	まとめ	11

第1章 はじめに

第 2 章 概要

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
 - ii. 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  }
```

3.1.2 型と集合

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が分からぬ。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。3.1.2 で述べた通り、decaf の型検査に用いられるTypeID は、不可分であり本来はペースを並列で行える。だが、現在の実装ではペ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での、decaf の型検査速度を向上させることができるものだろう。

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。