

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 JavaScript コンパイラと
TypeScript チェッカ

A JavaScript compiler and 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 倫太郎

指導教員: 青山 敏弘

<日付>

目次

第 1 章	型検査の仕様	1
1.1	変数の宣言	1
1.2	変数への代入	1
1.3	変数への参照	2
1.4	変数への再代入	2
1.5	存在しない変数への参照	3
1.6	変数の宣言前の代入	3
1.7	存在しない変数への代入	3
1.8	重複して宣言された変数	4
1.9	変数のシャドーイング	4
1.10	未初期化の変数	5
1.11	存在しないプロパティへの参照	5
第 2 章	decaf の実装	6
2.1	型検査	6
2.1.1	decaf が型を検査する手順	7

第 1 章 型検査の仕様

それぞれの機能に対する型検査の仕様を示す。例として，decaf と TypeScript における型検査の結果を示す。

1.1 変数の宣言

変数名の後にコロン (:) を付け、その後に型を指定することで変数の型を指定できる。変数の型が指定されていない場合、その変数は `any` 型として扱われる。1.1 のコードを例にすると、decaf では 2 は 2 型であり、`number` 型の部分集合であるため、`x` は `number` 型として扱われる。しかし、`y` は `string` 型であるため、エラーが発生する。同様に、`z` は `object` 型であるため、エラーが発生する。diagnostics は 1.2 に示す。

Listing 1.1: 変数宣言の例

```
1  const x: number = 2
2  const y: string = 2
3  const z: object = 4
```

Listing 1.2: decaf の diagnostics

```
1  Type 2 is not assignable to type string
2  Type 4 is not assignable to type object
```

Listing 1.3: tsc の diagnostics

```
1  Type 'number' is not assignable to type '
    string'.
2  Type 'number' is not assignable to type '
    object'.
```

1.2 変数への代入

変数に代入する値の型が変数の型と一致しない場合、エラーが発生する。1.4 のコードを例にすると、decaf では `x` は `number` 型であるため、`"hello world"` は `number` 型として扱われる。しかし、`number` 型に `"hello world"` 型を代入できないため、エラーが発生する。diagnostics は 1.5 に示す。

Listing 1.4: 変数への代入の例

```
1  let x: number = 3
2  x = "hello world"
```

Listing 1.5: decaf の diagnostics

```
1   Type "hello world" is not assignable to  
    type number
```

Listing 1.6: tsc の diagnostics

```
1   Type 'string' is not assignable to type '  
    number'.
```

1.3 変数への参照

変数への参照の型が変数の型と一致しない場合、エラーが発生する。1.7 のコードを例にすると、decaf では `a` は `3` 型であるため、`b` は `3` 型として扱われる。しかし、`3` 型に `string` 型を代入できないため、エラーが発生する。diagnostics は 1.8 に示す。

Listing 1.7: 変数への参照の例

```
1   const a = 3  
2   const b: string = a
```

Listing 1.8: decaf の diagnostics

```
1   Type 3 is not assignable to type string
```

Listing 1.9: tsc の diagnostics

```
1   Type 'number' is not assignable to type '  
    string'.
```

1.4 変数への再代入

変数への再代入の型が変数の型と一致しない場合、エラーが発生する。1.10 のコードを例にすると、decaf では `a` は `2` 型であるため、`"hello world"` は `2` 型として扱われる。しかし、`2` 型に `"hello world"` 型を再代入できないため、エラーが発生する。diagnostics は 1.11 に示す。

Listing 1.10: 変数への再代入の例

```
1   let a = 2  
2   a = "hello world"  
3   a satisfies number
```

Listing 1.11: decaf の diagnostics

```
1   Expected number, found "hello world"
```

Listing 1.12: tsc の diagnostics

```
1   Type 'string' is not assignable to type '  
    number'.
```

1.5 存在しない変数への参照

存在しない変数への参照がある場合、エラーが発生する。1.13 のコードを例にすると、`nexists` は存在しないため、エラーが発生する。diagnostics は 1.14 に示す。

Listing 1.13: 存在しない変数への参照の例

```
1  const exists = 2;
2  nexists;
```

Listing 1.14: decaf の diagnostics

```
1  Could not find variable 'nexists' in scope
```

Listing 1.15: tsc の diagnostics

```
1  Cannot find name 'nexists'. Did you
    mean 'exists'?
```

1.6 変数の宣言前の代入

変数の宣言前に代入がある場合、エラーが発生する。1.16 のコードを例にすると、decaf では `a` は宣言される前に代入されているため、エラーが発生する。

Listing 1.16: 変数の宣言前の代入の例

```
1  a = 3;
2  let a = 2;
```

Listing 1.17: decaf の diagnostics

```
1  Cannot assign to 'a' before declaration
```

Listing 1.18: tsc の diagnostics

```
1  Block-scoped variable 'a' used before its
    declaration.
```

1.7 存在しない変数への代入

存在しない変数への代入がある場合、エラーが発生する。1.19 のコードを例にすると、`doesNotExist` は存在しないため、エラーが発生する。diagnostics は 1.20 に示す。

Listing 1.19: 存在しない変数への代入の例

```
1  doesNotExist = 3;
```

Listing 1.20: decaf の diagnostics

```
1  Could not find variable 'nexists' in scope
```

Listing 1.21: tsc の diagnostics

```
1  Cannot find name 'nexists'.
```

1.8 重複して宣言された変数

変数が重複して宣言された場合、エラーが発生する。1.22 のコードを例にすると、decaf では `a` は 2 で宣言されているため、`a` は 2 である。しかし、`a` は 3 で再宣言されているため、エラーが発生する。ここで、`{ }` で囲まれた部分はスコープを示しているため、エラーにならない。diagnostics は 1.23 に示す。

Listing 1.22: 重複して宣言された変数の例

```
1  const a = 2
2  {
3      const a = 3;
4      a satisfies 3;
5  }
6  a satisfies 2;
7  const a = 3;
```

Listing 1.24: tsc の diagnostics

Listing 1.23: decaf の diagnostics

```
1  Cannot redeclare variable 'a'
```

```
1  Cannot redeclare block-scoped variable 'a'.
2  Cannot redeclare block-scoped variable 'a'.
```

1.9 変数のシャドーイング

変数のシャドーイングがある場合、エラーが発生する。1.25 のコードを例にすると、decaf では `a` は 2 で宣言されているため、`a` は 2 である。`{ }` で囲まれた部分はスコープを示しているため、`a` は 3 である。しかし、`satisfies` で `a` が 2 であることを示しているため、エラーが発生する。diagnostics は 1.26 に示す。

Listing 1.25: 変数のシャドーイングの例

```
1  const a = 2
2  {
3      const a = 3;
4      a satisfies 2;
5  }
```

Listing 1.26: decaf の diagnostics

```
1 Expected 2, found 3
```

Listing 1.27: tsc の diagnostics

```
1 Type '3' does not satisfy the expected
   type '2'.
```

1.10 未初期化の変数

未初期化の変数は `undefined` 型として扱われる。未初期化の変数に型が指定されている場合、エラーが発生する。1.28 のコードを例にすると、decaf では `b` は未初期化のため、`undefined` 型として扱われる。しかし、`undefined` 型に `string` 型を期待できないため、エラーが発生する。diagnostics は 1.29 に示す。

Listing 1.28: 未初期化の変数の例

```
1 let b;
2 b satisfies string;
```

Listing 1.29: decaf の diagnostics

```
1 Expected string, found undefined
```

Listing 1.30: tsc の diagnostics

```
1 // エラーにならない
```

1.11 存在しないプロパティへの参照

存在しないプロパティへの参照がある場合、エラーが発生する。1.31 のコードを例にすると、decaf では `my_obj` は `{ a: 3 }` 型であるため、`a` は `3` 型として扱われる。しかし、`b` は存在しないため、エラーが発生する。diagnostics は 1.32 に示す。

Listing 1.31: 存在しないプロパティの例

```
1 let my_obj = { a: 3 }
2 const a = my_obj.a
3 const b = my_obj.b
```

Listing 1.32: decaf の diagnostics

```
1 No property 'b' on { a: 3 }
```

Listing 1.33: tsc の diagnostics

```
1 Property 'b' does not exist on type '{ a:
   number; }'.
```

第 2 章 decaf の実装

decaf は Rust を用いて実装された。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 2.1: 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 info
```

2.1 型検査

decaf は任意の TypeScript(`.tsx?`) ファイルを入力として受け取り、型検査する。2.2 ように実行すると、標準出力として 2.3 のような結果が得られる。

Listing 2.2: decaf の型検査

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

Listing 2.3: 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 Diagnostics: 446
8 Types: 5780
9 Lines: 2239
10 Cache read: 285.954µs
11 FS read: 169.096µs
12 Parsed in: 8.294198ms
13 Checked in: 4.887375ms
14 Reporting: 204.832µs
```

2.1.1 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, AST¹⁾を生成する.
 - (a) ここで変換される AST は ESTree²⁾ や swc³⁾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) これを decaf では TypeID と呼んでいる.
3. TypeID を用いて型検査を行う

1) 抽象構文木のこと

2) ECMAScript の抽象構文木の実質的な標準

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