Vol.2018-SE-199 No.20
2018/7/19

TypeScriptにおけるインタフェースの生成

中村 晋太[1,a)]　丸山 勝久[1,b)]

概要：構造的な型付けを採用した TypeScript では，クラスやインタフェースの持つプロパティやメソッドのシグネチャが構造的に一致するかどうかに基づき型を特定する．さらに，TypeScript では，型推論の機構が導入されており，変数や関数の宣言において型注釈を省略することができる．このような構造的な型付けと型推論の採用によりソースコードの柔軟さと簡潔さが向上する反面，型注釈の過度な省略は静的な型検査による誤りの検出を阻害することがある．そこで，本論文では，関数の引数において型注釈が省略されたオブジェクトに対して，関数内部でそのオブジェクトがどのようにアクセスされているかを解析した結果からオブジェクトの型を特定し，その型を表現するインタフェース宣言を生成する手法を提案する．生成されたインタフェースは，オブジェクトを格納する引数の型として割り当てられる．この手法を TypeScript のプログラミング環境に導入することで，プログラマが型を割り当てる手間を増加させずに，静的な型検査の効果を高めることが期待できる．

Generating Interfaces in TypeScript Programming Language

Shinta Namamura[1,a)]　Katsuhisa Maruyama[1,b)]

1. はじめに

オブジェクト指向プログラミング言語では，データとその操作をセットにしたオブジェクトという抽象データ型を用いてプログラミングを行う．オブジェクトの内部構造であるデータはプロパティ，操作はメソッドとも呼ばれる．一度設計されたオブジェクトはクライアントコードで利用されるときに，新たにプロパティやメソッド（これらをまとめてメンバと呼ぶ）を追加することで拡張できる．このように拡張されたオブジェクトは，拡張元のオブジェクトを利用する場面で安全に実行することができる [1]．なぜなら，拡張されたオブジェクトは拡張元のオブジェクトのメンバをすべて備えているからである．

いま，メソッド `m():stirng` のみを持つオブジェクト A に対して，プロパティ `p:number` を追加したオブジェクト B を考える．オブジェクトの型を，そのオブジェクトの持つメンバの集合で定義すると，A の型 T_A と B の型 T_B はそれぞれの以下のように表現できる．

$T_A = \{$ `m():string` $\}$

$T_B = \{$ `m():string`, `p:number` $\}$

`string` は文字列型，`number` は数値型を指す．この例において，B は A のメンバをすべて備えているので，型 T_B を持つ B は A の代わりとして利用することができる．いいかえれば，型 T_A のオブジェクトが要求される場面で，型 T_B のオブジェクトを利用できる．このとき，T_B を T_A の派生型と呼ぶ．

本論文では，JavaScript の上位プログラミング言語[*1]として，近年 Web アプリケーション開発での利用が増えている TypeScript [2] を扱う．TypeScript では，変数，式，関数などに対して，静的な型をプログラマが指定することができる．ここでは，型を指定する記述を型注釈と呼ぶ．さらに，TypeScript は型推論の機構を備えているので，型注釈を省略することも可能である．リテラルの型やプログラマが付与した型注釈に基づき，コンパイル時に型が特定され，型検査が行われる．コンパイラは，型制約を満

[1] 立命館大学
Ritsumeikan Uniersity, Kusatsu, 525–8577, Japan
[a)] nakamura@fse.cs.ritsumei.ac.jp
[b)] maru@cs.ritsumei.ac.jp

[*1] TypeScript のソースコードをコンパイルすることで JavaScript のソースコードが生成される．

たさない演算に対して，型エラーを報告する．Gao らは，JavaScript で記述されたバグを含むソースコードに対して型注釈を付与することで，15%程度のバグが静的な型検査により防げることを実験により示している [3].

ここで，TypeScript が採用している構造的な型付け（structual subtyping）では，前述の T_A と T_B に対して明示的に型の派生関係を決定する必要はなく，型の持つメンバの構造に基づき，T_B が T_A の派生型と特定される [4], [5], [6]．このような型付けは，型の派生関係を柔軟に指定できるようにしつつ，型の構造から静的な型検査が可能な点で注目されている．たとえば，Malayeri と Aldrich は，構造的な型付けの機構を持たないプログラミング言語 Java で記述されたオープンソースプロジェクトのソースコードに対して調査を行い，構造的な型付けの導入潜在的な有用性を示している [7].

構造的な型付けと型推論を活用することでプログラマが型注釈を付与する負担が軽減される反面，型注釈の過度な省略は静的な型検査の効果を減少させることがある．たとえば，関数やメソッド[*2]の引数に対して型注釈を省略した場合，その引数の型は any 型（どのような型の値でも代入できる型）となり，その引数が関数やメソッド内部で参照されていても型検査の対象とならない．いま，あるオブジェクトが関数やメソッドの呼出しにおいて型注釈のない引数として渡され，関数やメソッドの内部でそのオブジェクトにアクセスしている場面を考える．このとき，オブジェクトを格納している引数の型は any 型である．このような状況において，そのオブジェクトが持たないメソッドへのアクセスが発生していたとしても，それを静的に検査することはできず，実行時エラーとなる[*3]．型検査により，このような実行時エラーを回避するためには，引数に対する型注釈をプログラマが記述する必要がある．しかしながら，関数やメソッドの記述途中において構造的な型付けを意識し，適切な型注釈を付けることはプログラマにとって面倒な作業である．

本論文では，型注釈が付与されていない関数やメソッドの引数に対して，構造的な型付けに基づき，その引数の型を表現したインタフェース宣言を自動的に生成する手法を提案する．具体的には，引数が格納しているオブジェクトの持つメンバの中で実際にアクセスされているものを抽出し，それらのシグネチャを持つインタフェース宣言を新たに定義する．その上で，そのインタフェースをオブジェクトを格納する引数の型として割り当てる．さらに，本論文では，このようなインタフェース宣言を生成する手法を Visual Studio Code に組み込んだ実装を提案する．

[*2] クラスやインタフェースの内部で定義される関数がメソッドである．

[*3] any 型のオブジェクトに対して，それが持たないプロパティにアクセスした場合は，実行時エラーとならず，プロパティの値は undefined と扱われる．

提案手法を TypeScript のプログラミング環境に導入することで，プログラマが引数に対する型注釈を省略しても自動的に型注釈が補完され，その引数に対する型検査が実行されるようになる．このことは，プログラマの負担を増加させずに静的な型検査の機会を増加させ，将来的な誤りの混入を防ぐことにつながる．また，引数に型注釈を付与することで，関数やメソッド内部における引数の利用方法（引数が格納しているオブジェクトにおいてアクセスが可能なメンバ）をプログラマが把握しやすくなり，保守性の向上も見込める．特に，提案手法は，型注釈を引数に付与するだけで，もとのソースコードの（外部的）挙動を変えないことから，型注釈の付与を目的とした（TypeScript 向け）リファクタリングといえる．

以下，2 節では，提案手法が対象とするソースコードと生成されるインタフェース宣言の例を述べる．3 節では，提案手法がインタフェース宣言を生成する手順を説明する．4 節では，提案手法の実装と制限を述べる．最後に，5 節でまとめと今後の課題について述べる．

2. 型検査とインタフェース宣言の生成例

提案手法は，TypeScript において，オブジェクト型の引数を対象とする．プリミティブ型，共用型（union type），交差型 (intersection type)，ジェネリック型（generic typer），関数型の引数は対象としない．オブジェクト型は，オブジェクトの持つメンバを列挙したクラスあるいはインタフェースを用意することでプログラマが自由に定義可能である．インタフェースはメンバのシグネチャだけを集めたものである．これに対して，クラスはメンバのシグネチャだけでなく実装も提供する．

図 1 に，引数に対する型注釈を省略したソースコード sample1.ts を示す．クラス `C` には，メソッド `m1(string)` が定義されており，そのクラスから生成されたオブジェクトが関数 `f` の引数 `arg` として渡されている．この例では，`f` の `arg` の型は指定されておらず，any 型となる．これにより，コンパイル時に型検査は実行されず，コンパイルエラーは発生しない．さらに，`f` の呼出しにおいて，実行時に `arg` に格納されるオブジェクトは `m1(string)` を持つため，実行時エラーも発生しない．その結果，期待通りコンソールに “`Hello`” と表示される．

図 2 は，図 1 のソースコードにおいて，クラス `C` のメソッドの名前を `m2` と間違えて記述したソースコードである．図 1 のソースコードと同様に，引数 `arg` に対して型検査は実行されず，コンパイルエラーは発生しない．一方，`f` の呼出しにおいて，実行時に `arg` に格納されるオブジェクトは `m1(string)` を持たないので，`f` の内部において `m1(string)` を呼び出している部分で実行時エラーが発生する．

このような実行時エラーを回避するためには，関数 `f` の

```
class C {
  m1(str:string) { console.log(str); }
}

function f(arg) { arg.m1('Hello'); }
f(new C());
```

図 1　ソースコード sample1.ts（引数に対する型検査が実行されないが正しく動作する）

```
class C {
  m2(str:string) { console.log(str); }
}

function f(arg) { arg.m1('Hello'); }
f(new C());
```

図 2　ソースコード sample2.ts（引数に対する型検査が実行されず実行時エラーとなる）

```
interface I {
  m1(str:string);
}

class C {
  m1(str:string) { console.log(str); }
}

function f(arg:I) { arg.m1('Hello'); }
f(new C());
```

図 3　ソースコード sample3.ts（引数に対する型検査が実行された上で正しく動作する）

引数 arg に対して型注釈を付与し，コンパイル時に呼出しメソッドの存在を確認する型検査が実行されるようにしておくことが重要である．図 1 のソースコードに対して，型注釈を付与したソースコードを**図 3** に示す．

この例では，関数 f の引数 arg の型を直接的に C と指定せず，f の内部でアクセスされるメソッド m1(string) を持つインタフェース I を新たに定義し，それを arg の型として指定している．C と I のメンバのシグネチャの集合が同一であるため，構造的な型付けに基づき，実際に arg に渡されるオブジェクトの型 C は，f が要求するオブジェクトの型 I の派生型と判断される．よって，引数の受け渡しで型の不整合は起こらず，コンパイルエラーは発生しない．さらに，C から生成されたオブジェクトはメソッド m1(string) を持つので，実行時エラーも発生しない．たとえ図 1 と図 3 の実行結果が同じであったとしても，型注釈を付与することで型検査が実行されている点が重要である．

最後に，図 3 のソースコードに対して，図 2 と同様の誤りを混入させたソースコードを**図 4** に示す．この例で

```
interface I {
  m1(str:string);
}

class C {
  m2(str:string) { console.log(str); }
}

function f(arg:I) { arg.m1('Hello'); }
f(new C());
```

図 4　引数に対する型検査が実行され，コンパイルエラーとなるソースコード

は，実際に関数 f の引数 arg に渡されるオブジェクトの型 C と，f が要求するオブジェクトの型 I との間に不整合が検出され，コンパイルエラーが発生する．これにより，プログラマは，コンパイル時にメソッド名の書き間違いに気づくことができる．

このように，誤りの早期発見，さらには安全にプログラムを実行可能という観点から，関数やメソッドの引数には型注釈を付与することが望ましい．提案手法では，プログラマが記述した図 1 のソースコードを解析することで，関数 f の引数 arg のオブジェクトが持つべきメンバ（図 1 のソースコードではメソッド m1(string)）を特定する．その上で，必要なメンバを持つインタフェース I の宣言を自動的に生成し，それを arg の型として指定した図 3 のソースコードに書き換える．つまり，提案手法は，図 1 のソースコードを入力とし，それを図 3 のソースコードに変換して出力する．

3. インタフェース宣言の自動生成手法

提案手法は，次に示す 5 つの手順により，インタフェース宣言を生成する．

Step 1　引数を介してアクセスされるメンバの収集
Step 2　引数として渡されるオブジェクトの決定
Step 2　シグネチャの取得
Step 4　シグネチャの選択
Step 5　インタフェース宣言の生成と型注釈の付与

本節では，上記の手順の詳細を順番に説明する．その際，2 節で紹介した例ではクラスが 1 つのメソッドしか持っておらず，詳細を説明するのに不適切である．そこで，説明に適した別の例題を用意する．

プログラマが記述しているソースコード print.ts を**図 5** に示す．また，そのソースコードから利用される可能性のあるクラス群が記述されたソースコード magazine.ts を**図 6** に示す．print.ts を見ると，関数 print の引数 m に格納されるオブジェクトは，名前が title のフィールドと名前が getPrice のメソッドを持つことが要求されていることがわかる．クラス Magazine は，

```
import { Magazine,
         MonthlyMagazine,
         SpecialMagazine }
  from "./magazine";

function print(m) {
  console.log(m.title);
  console.log(m.getPrice());
}

function print2(m2) {
  console.log(m2.title);
}
```

図 5　ソースコード print.ts

プロパティとして title:string と price:number を持つ．MonthlyMagazine と SpecialMagazine は Managine を継承したクラスである．MonthlyMagazine には，プロパティとして volume:string と pages:number，メソッドとして getPrice():string が追加されている．また，SpecialMagazine には，プロパティとして volume:number，メソッドとして getPrice():number が追加されている．

3.1　引数を介してアクセスされるメンバの収集

プログラマが記述したソースコードに対して，型注釈が付与されていない引数を探し，関数やメソッドの内部でアクセスしているメンバの名前を収集する．print.ts において，対象となる引数は関数 print の m と関数 print2 の m2 の 2 つである．print あるいは print2 において，引数 m および m2 を介してアクセスされるメンバの名前の集合をそれぞれ A_m と A_{m2} とすると，次のようになる．

$A_m = \{$ title, getPrice $\}$

$A_{m2} = \{$ title $\}$

ここでは，型注釈の付与対象として，プログラマが print の m を選択したとする．

3.2　引数として渡されるオブジェクトの探索

Step 1 において，型注釈の付与対象の引数を介してアクセスされるメンバが判明したので，引数として渡される可能性のあるオブジェクトを探索する．そのために，候補となるオブジェクトが持つメンバを，対象の引数が存在するソースファイルと，そのファイルでインポートしているソースファイルまたはライブラリから収集する．ただし，アクセス修飾子により，対象の引数が宣言されている箇所からアクセスできないようになっている（private，および，親クラスに存在しない protected）メンバは収集対象から除外する．

この例では，print.ts と magazine.ts がオブジェクトを探索する対象のソースファイルとなり，クラス Magazine，MonthlyMagazine，SpecialMagazine から生成されるオブジェクトが，print の引数 m に渡される候補となる．それぞれのメンバの名前の集合 M_m，M_{mm}，M_{sm} を取得すると，以下のようになる．

```
export class Magazine {
  title:string;
  price:number;
  constructor(t:string, p:number) {
    this.title = t;
    this.price = p;
  }
}

export class MonthlyMagazine
       extends Magazine {
  volume:string;
  pages:number;
  constructor(t:string, p:number,
              v:string, pp:number) {
    super(t, p);
    this.volume = v;
    this.pages = pp;
  }
  getPrice():string {
    return this.price + '-yen';
  }
}

export class SpecialMagazine
       extends Magazine {
  volume:number;
  constructor(t:string, p:number,
              v:number) {
    super(t, p);
    this.volume = v;
  }
  getPrice():number {
    return this.price;
  }
}
```

図 6　ソースコード magazine.ts

$M_m = \{$ title, price $\}$

$M_{mm} = M_m \cup \{$ volume, pages, getPrice $\}$

$M_{sm} = M_m \cup \{$ volume, getPrice $\}$

TypeScript では，クラスだけでなく，インタフェースやオブジェクト型リテラルにおいて，そのメンバを定義することが可能である．よって，実際には，クラスだけでなく，これら 3 つの構文要素に関係するすべてのオブジェクトが探索対象となる．この例では，インタフェースとオブジェクト型リテラルは現れない．

いま，関数 *func* の引数 *arg* として渡されるオブジェク

ト *obj* は，*func* の内部においてアクセスされるすべてのメンバを持つ必要がある．よって，*obj* の持つメンバの集合 M_{obj} と，*arg* を介して *func* の内部でアクセスされるすべてのメンバの集合 M_{arg} には，以下の関係が成立する必要がある．

$M_{arg} \subseteq M_{obj}$

図 5 と図 6 の例では，$A_m \not\subseteq M_m$，$A_m \subseteq M_{mm}$，$A_m \subseteq M_{sm}$ となるので，クラス `MonthlyMagazine` あるいは `SpecialMagazine` から生成されたオブジェクトが関数 `print` における引数 `m` の候補となる．

3.3 シグネチャの取得

インタフェース宣言を生成するためには，メンバの名前だけではなく，そのシグネチャを取得する必要がある．このために，**Step 2** で特定したオブジェクトのシグネチャを利用する．

この例では，クラス `MonthlyMagazine` あるいは `SpecialMagazine` から生成されたオブジェクトが型注釈を付与する引数に渡される可能性を持つので，それぞれのシグネチャの集合 S_{mm} と S_{sm} を取得すると，以下にようになる．

$S_{mm} =$ { `title:string`, `price:number`, `volume:string`, `pages:number`, `getPrice():string` }

$S_{sm} =$ { `title:string`, `price:number`, `volume:number`, `getPrice():number` }

ここで，引数として渡されるオブジェクトにおいて実際にアクセスされるのは，A_m に含まれるメンバだけである．よって，A_m に含まれないプロパティを取り除くと，以下のようになる．

$S'_{mm} =$ { `title:string`, `getPrice():string` }

$S'_{sm} =$ { `title:string`, `getProce():number` }

以上より，関数 `print` の引数 `m` に付与する型のシグネチャは，S'_{mm} あるいは S'_{sm} となる．

3.4 シグネチャの選択

引数として渡されるオブジェクトの候補の種類（オブジェクトを生成するクラスなど）が複数存在する場合，引数の型のシグネチャが一意に決まらないことがある．この場合，プログラマが適切なシグネチャ集合を選択することになる．

Step 3 で取得したシグネチャの集合 S'_{mm} と S'_{sm} において，プロパティ `title` のシグネチャは同じである．一方，メソッド `getPrice()` については戻り値の型が異なるため，そのシグネチャは一意に決まらない．この場合，プログラマは，同じ名前でシグネチャの異なる複数のメンバから適切なものを選択することになる．

この例では，S'_{mm} に含まれる `getPrice():string` と，S'_{sm} に含まれる `getProce():number` が選択の候補となる．いま，プログラマが `getPrice():string` を選択したとすると，型注釈の付与対象である引数のシグネチャの集合は，以下のようになる．

$S_{arg} =$ { `title:string`, `getPrice():string` }

```
import { Magazine,
         MonthlyMagazine,
         SpecialMagazine }
  from "./magazine";

function print(m:IMagazine) {
  console.log(m.title);
  console.log(m.getPrice());
}

function print2(m2) {
  console.log(m2.title);
}

interface IMagazine {
  title:string;
  getPrice():string;
}
```

図 7 生成されたインタフェース宣言と型注釈が扶養された引数を含むソースコード print-revised.ts

3.5 インタフェース宣言の生成と型注釈の付与

Step 4 により，シグネチャの集合が一意に決定されたので，その集合に含まれるすべてのメンバを持つインタフェース `IMagazine` の宣言を生成する．さらに，このインタフェースの名前を型注釈として，関数 `print` の引数 `m` に付与する．生成されたインタフェース `IMagazine` と型注釈が付与された引数 `m` を含むソースコードを**図 7** に示す．

図 7 に示すソースコード print-revised.ts と図 6 に示すソースコード magazine.ts を見るとわかるように，生成されたインタフェース `IMagazine` のシグネチャ集合 S_{arg} と，`print` の `m` に渡されることが想定されているクラス `MonthlyMagazine` のオブジェクトのシグネチャ集合 S_{mm} との間には，以下の関係が成立している．

$S_{arg} \subset S_{mm}$

構造的な型付けでは，このような関係が成立する場合，クラス `MonthlyMagazine`（から生成されたオブジェクトの型）は，インタフェース `IMagazine` の派生型とみなされる．よって，`print` における `m` の型を `IMagazine` で指定することでき，`m` を介したオブジェクトへのアクセスが実行時エラーを引き起こさないことが型検査により保証される．

```
import { Magazine,
         MonthlyMagazine,
         SpecialMagazine }
  from "./magazine";

function print(m:IMagazine) {
  console.log(m.title);
  console.log(m.getPrice());
}

function print2(m2:IMagazine2) {
  console.log(m2.title);
}

interface IMagazine {
  title:string;
  getPrice():string;
}

interface IMagazine2 {
  title:string;
}
```

図 8　生成されたインタフェース宣言と型注釈が扶養された引数を含むソースコード print-revised2.ts

逆に，m にクラス SpecialMagazine から生成したオブジェクトを渡したり，print の内部でクラス MonthlyMagazine のメンバに pages にアクセスしたりすると，コンパイル時に型エラーが報告される．

参考のため，提案手法より，関数 print2 の引数 m2 に対しても型注釈を付与したソースコードを図 8 に示す．

4. 実装

提案手法を，Microsoft Visual Studio Code の Extension として実装した．TypeScript のソースコードから抽象構文木を構築し，必要な構文要素を取り出す際には，Compiler API [8] を利用している．

4.1 実行手順

拡張された TypeScript プログラミング環境において，インタフェース宣言を生成する手順は以下のようになる．

(1) TypeScript のソースファイルをアクティブにした状態でコマンドパレットを開き，コマンド一覧から「Interface Generate」を選択する．

(2) アクティブなソースファイルにおいて，型が付与されていない引数の一覧が表示されるので，型を付与したい引数を選択する．図 9 に，プログラマが引数を選択している際のスクリーンショットを示す．ソースファイル print.ts に記述されている function print(m) と function print2(m2) が選択肢として表示されている．

(3) 選択した引数の型のプロパティのシグネチャが一意に決定できない場合，選択肢の中から適当なシグネチャを選択する．図 10 に，プログラマがシグネチャを選択している際のスクリーンショットを示す．同じ名前で戻り値の型が異なる 2 つのシグニチャ getPrice():string と getPrice():number が選択肢として表示されている．

(4) 生成するインタフェースの名前を入力する．

(5) 入力したインタフェースが引数の型注釈として付与される．同時に，インタフェース宣言のコードがクリップボードにコピーされるので，ソースファイルに貼り付ける．図 11 は，インタフェース宣言 IMagazine のコードを貼り付けた後のスナップショットである．関数 print の引数の型注釈として IMagagize が付与されている．

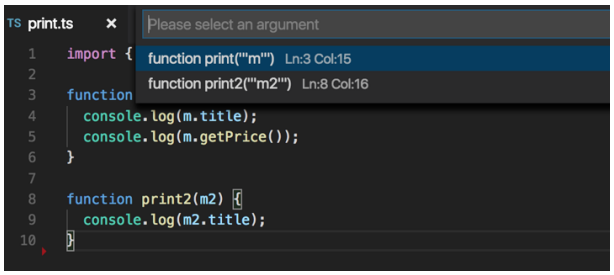

図 9　引数選択の様子

図 10　シグネチャ選択の様子

図 11　インタフェース宣言の生成結果

4.2 設計上および実装上の制限

本節では，提案手法において，現時点で対処していない設計上および実装上の制限を述べる．

(1) シグネチャの選択における制限

提案手法では，型注釈を付与する引数がアクセスしてい

るメンバの名前に基づき，その引数に割り当てる型のシグネチャを特定している．このような方法では，引数に渡されるオブジェクトが同じ名前でシグネチャが異なるメンバ（ここでは，シグネチャが衝突するメンバと呼ぶ）を持つと，生成するインタフェースのシグネチャを一意に決定できない．このような状況において，提案手法では，3.4 節で述べたように，プログラマが適切なシグネチャを選択する方針を採用している．

しかしながら，プログラマが引数に渡されるすべてのオブジェクトを把握していると考えることには無理がある．また，現在の実装では，シグネチャが衝突するメンバを 1 つずつプログラマが選択するようになっている．よって，不適切なシグネチャの集合が選択されると，引数に型注釈を付与したことにより，型エラーが発生する．さらには，引数として受け取るオブジェクトが一意に決まらないことを前提に，関数やメソッド内部のコードが記述されている可能性もある．この場合，シグネチャの集合を無理に一意に決定して型注釈を付与すると，型エラーが発生する．

このような問題を回避あるいは軽減するためには，ソースコード解析の精度を高め，引数に渡される可能性があるオブジェクトだけを，シグネチャ選択の際の絞り込みに用いればよい．また，そのオブジェクトのメンバへのアクセス時のコンテキストから，生成するインタフェースのシグネチャを絞り込むことが考えられる．たとえば，メンバがメソッドの場合は実引数の型を推論することで，実際にアクセスされているメソッドが絞り込める．ソースコード解析による情報を活用し，プログラマがシグネチャを選択する機会を減らすことで，型エラーを引き起こす型注釈の付与を回避できる可能性は高い．さらに，シグネチャの集合を一意に決定できない場合でも，型検査において複数の型のうち 1 つを選択できる共用型や，型を抽象化して扱うことが可能なジェネリック型を利用することで問題が解決できるかもしれない．

（2） データフロー解析に関する制限

提案手法では引数に関するデータフローを追跡せず，3.1 節で述べたように引数の名前だけに基づきオブジェクトのメンバに対するアクセスを抽出している．このため，関数またはメソッド内部で引数にエイリアスが発生し，別名を介してそのオブジェクトのメンバにアクセスがあったとしても，それを検出できない．さらに，引数に格納されたオブジェクトが再定義されていたとしても，それを検出していない．このような設計では，引数として渡されるオブジェクトにおいてアクセスされるメンバが正確に抽出できないことがある．結果として，不適切なシグネチャ集合を持つインタフェースが生成され，間違った型注釈が付与される可能性が残る．このような観点から，データフロー解析の導入は必須である．

5. おわりに

本論文では，TypeScript においてインタフェース宣言を自動生成する手法を提案し，その実装を示した．プログラマの負担を増やさずに，関数やメソッドの引数に対して型注釈が補完でき，型検査によるプログラムの安全性の向上が期待できる．提案手法の実装は，`https://github.com/topporo/ts_interface_generator` で公開している．

今後の課題として，ソースコード解析の精度の向上や引数に関するデータフロー解析の導入を検討することで，不適切なインタフェース宣言の生成を抑制する機構を提案手法に取り込む予定である．また，現在の実装では，引数として単純なオブジェクト型しか扱っていない．共用型，交差型、ジェネリック型，関数型のオブジェクトが引数として渡される場合への対応も今後の課題である．

謝辞 本研究をすすめるにあたり，有益なご意見を頂いた大分大学の紙名哲生氏に感謝いたします．本研究の一部は，科研費 (15H02685) の助成を受けたものである．

参考文献

[1] Liskov, B. H. and Wing, J. M.: A Behavioral Notion of Subtyping, TOPLAS, Vol. 16, No. 6, pp. 1811–1841 (1994)
[2] Microsoft, TypeScript Language Specification, https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.
[3] Gao, Z., Bird, C., and Barr, E. T.: To Type or Not to Type: Quantifying Detectable Bugs in JavaScript, Proc. ICSE '17, pp. 758–769 (2017).
[4] Cardelli, L.: Structural Subtyping and the Notion of Power Type, Proc. POPL '88, pp.70–79 (1988).
[5] Pierce, B. C.: Types and Programming Languages, MIT Press (2002).
[6] Fenton, S.: TypeScript 実践プログラミング, 翔泳社 (2015).
[7] Malayeri, D. and Aldrich, J.: Is Structural Subtyping Useful? An Empirical Study, Proc. ESOP '09, pp95–111 (2009).
[8] Using the Compiler API, https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API.