TypeScript の型システム

佐藤周行

2022 年 5 月 3 日

1　はじめに

本稿は、大学院講義「プログラミング言語処理系」中で説明する型システムを特に TypeScript を例にとって説明したものである。最初に型システムの一般論を述べ、その後で、TypeScript の型システムの説明を行う。このような文脈で説明するので、ソフトウェア工学の見地からの説明が主となる。計算論や意味論から型を説明することはほとんどしないし、型理論一般についてはもちろん触れない。理論面で興味をもつような読者は教科書（例えば [2]）を読みましょう。

本稿の記述は、O'Reilly の [1] を参考にし、さらに TypeScript `tsc 4.5.5` でテストしている。

2　TypeScript

TypeScript は、JavaScript にアノテーションと呼ばれる型情報を付加してできるプログラミング言語である。文法は、アノテーションを除けば JavaScript のものを採用している。つまり TypeScript のアノテーションを除去すれば、JavaScript のプログラムになる。この自明な埋め込みによって TypeScript $\subset$ JavaScript と考えることができる。しかし、この逆は成立しない。JavaScript のプログラムが TypeScript の型（アノテーション）をつけられるわけではない（図 1 参照）

処理系としての TypeScript (`tsc`) は、JavaScript へ source to source transformation を行う。出力された JavaScript プログラムは JavaScript としてそのまま実行可能である。

3　型システム

定義 1 (型 (Type)) 型 *term* と、式 *term* が *Syntax* として定義されている言語を考える。式 *term* e が型 *term* T をもつ $(e : T)$ 二項関係が定義される。型関係と言うこともある。

定義 2 (型推論 (Type Inference)) $e : T$ を成立させる関係を、型関係の（複数の）条件に対して結論を対応させることで定義する。この対応を型推論規則と言い、全体として一つの型関係を

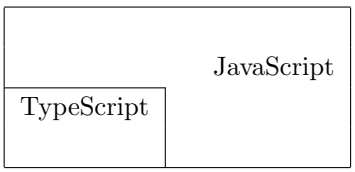

図 1: JavaScript と TypeScript

導出することを型推論という。条件、$e_0 : T_0, \cdots, e_n : T_n (n \geq -1)$ に対して結論 $e : T$ が対応することを以下のように書く。

$$\frac{e_0 : T_0 \cdots e_n : T_n}{e : T}$$

例 1 以下、*term* の文法については *TypeScript* で考える。以下は、*TypeScript* の正しい型推論規則である。

1.
$$\frac{(n \text{ is a number expression })}{n : \text{number}}$$

2.
$$\frac{e_0 : T_0 \quad e_1 : T_1}{\{e_0, e_1\} : [T_0, T_1]}$$

以下は正しい型推論になる。

$$\frac{1 : \text{number} \quad 11 : \text{number}}{\{1, 11\} : [\text{number}, \text{number}]}$$

例 2 *TypeScript* において、以下は正しい型推論である。

$$\frac{x : \text{any}}{x : \text{number}}$$

型は、素朴には型関係を満たす式 term の（集合論的な意味での）集合と解釈される。ただし、そうすると、例えば Generics を使って`<T>(T):T` のような型をもつ関数の「集合」が存在するかということは常に問題になる。

問題 1 型`<T>(T):T` をもち得る式 *term* を一つ構成せよ。

例えば、[3] は、F とよばれる二階型システムに素朴に集合論的な解釈を与えると矛盾が生じることを指摘した論文である（今読み返すと、さまざまな隠れた仮定があって、結論が正しいかどうか少し怪しい）。

定義 3 (型の意味 (Semantics of Types)) 型に対して、集合（集合論的）やオブジェクト（圏論的）などの数学的な実体を与えることを「意味を与える」と言う。特に、集合論的な意味を与えるならば、型 T は集合 $[\![T]\!]$ と解釈され、$e : T$ は関係 $[\![e]\!] \in [\![T]\!]$ と解釈される。

以後、TypeScript には、集合論的な意味を与える（それで十分である）。言語処理系に興味をもつならば、これ以上のことは別分野の話であると割り切るのがよろしい。

このセクションの前半では syntactic に、後半では semantic な議論をした。二つを合わせてその言語の型システムという。

4　型システムのソフトウェア工学的な意義

型システムが満たすべき性質として、以下は基本的なものである。準同型というのは、式の評価と型推論の間に準同型が成り立つということである。

定義 4 (準同型) 式 e が、評価されて e' になるとする。このとき $[\![e]\!] = [\![e']\!]$. $e : T$ が成立するならば $e' : T$.

一般に、型は式と比べて単純な構成をもっている。準同型定理が成り立つならば、以下が成り立つだろう。

定理 1 e に対して $e : T$ なる型 T がないとする。このとき、$[\![e]\!]$ は（普通の意味で）意味をもたない。

例 3 `1+"str"`は、*TypeScript* では対応する型をもたないが、*JavaScript* では、評価が成功して文字列`"1str"`になる。で、後者は、プログラミング的に望ましいことだろうか？

この定理の基本的なことは、$e : T$ という静的な（実行前の）関係が、評価という動的なものを拘束することである。

この意味で、実行前に「式に型が対応するか」を確認することは大きな意味をもつ。特に、それ以上評価が進まない式（正規形）に型がつけられると、評価前の式に型がつけば、その式を評価していくことによって、正規形になって計算が終了することが強く期待される。

定義 5 (型チェック) 式 e が、対応する T とともに与えられたとき、$e : T$ が成立するかどうかを推論することを型チェックという。

もちろん、型システムには限界がある。型として「計算が有限回で停止する」に対応するものがとれるはずもない。

問題 2 なぜか？

しかし、`(x) => {return x}`に $< T > (T) : T$ の型がつかず、$(number) : number$ や $(string) : string$ と、型ごとに別の関数型が割り当てられるのも困る。generics とその前身である polymorphism は、このようにして研究の対象になった。

型システムが、意図しない結果を出す式に型を与えないということを前提に、できるだけ広い範囲の式に型をつけられるようなものであれば、デバッグを含め、プログラミングの生産性を格段に高めるだろう。JavaScript に対して TypeScript が開発されたのには、このようなソフトウェア工学的な意味があったのである。

実際、型付けが動的に行われるような JavaScript や Python では、大規模なプログラムを開発するときに細心の注意が必要である。特にモジュールをインポートするときには、関数の引数の制約に気を配らなければならない。このようにして、生産性は削られていくのである。

同様に、型付けができるように Python を制限したものに RPython[4] がある。RPython については、VM の構成と最適化のところで、別の観点から触れることにする。

なお、[5, 6] のように、一定のモデルをつくり、機械学習を使って型をつける試みもあるが、はっきり言って悪乗りである。そうやってつけた型がソフトウェア工学的にどのような意味があるというのか、著者らは一度考えてみるべきである。

5 TypeScript の型システム

5.1 基本型と literal

TypeScript では、JavaScript の boolean, number と string を基本型としてそのまま採用する。その他に TypeScript では bigint, symbol, object, function が基本型として定義される。また、undefined が型として認識される。TypeScript のエラーメッセージを借りれば、

```
TS2367: This condition will always return 'false' since the types '"string" | "number" |
 "bigint" | "boolean" | "symbol" | "undefined" | "object" | "function"'
and '"unknown"' have no overlap.
```

5.1.1 any その他

さらに、以下の「型」表現を考える。

any wildcard として用いる。any が出てきた時点で、型の演算は抑止される。通常の意味で型付けができないような計算を表現する。例えば、以下の例で ggg は TypeScript で any に型付け可能になり、JavaScript に変換後実行されて 33,4 を出力する。

```
var g:any = 3;
var gg:any = [3,4];
var ggg = g+gg;
```

また、以下の例では ggg は any に型付けされ、number には型付けされない。

```
var g:any = 3;
var gg:number = 4;
var ggg = g+gg;
```

unkown unknown は、型が不明であることを明示的に表現するときに用いる。この型をつけた式では、JavaScript の意味での値の比較や `typeof` 演算などの限定された演算が可能である。

undefined JavaScript の値 undefined に対応する型。`undefined:undefined` のみが許される。

null JavaScript の値 null に対応する型。`null:null` のみが許される。

any と unknown は、型推論を抑止するために明示的に用いられる。
また、undefined や null は対応する値のチェックをするのに有用である。さらに、Widening でも any にエスカレートするのに用いることができる。

5.1.2 literal

literal は、値一つからなる型である。例えば、以下のように定義する。

```
var g: 7 = 7;
var gg:7 = 3+4;
```

上の例で、g は型付け可能であるが、gg はエラーが出る。右辺の 3+4 は number に型付けられ、それから Narrowing されないからである。
このような不便さは残しつつ、literal は、値の世界と型の世界が交わることを可能にする。C や Fortran のような従来型のプログラミング言語では許されなかったこの手のことを、assertion を型チェックの形で表現することを可能にした。具体的には、後述する enum や、union に対して一段突っ込んだ型推論を可能にしている。ここについては Refinement のところで詳述する。

5.1.3 Union と Intersection

型のもつメンバーの和と積が定義される。

```
type TwoD = {x:number, y:number}
type ThreeD = {x:number, y:number, z:number}
type TwoOrThreeD = TwoD|ThreeD
type TwoAndThreeD = TwoD & ThreeD
```

特に重要なのが、literal の和を取った型であり、普通に使われる。例を下にあげる。

```
type nodetype = 'a' | 'ol' | 'ul'
type strOrnum = string | number
```

5.1.4 typeof と instanceof

JavaScript にもある typeof と instanceof は、計算中に型の情報を利用するのに有用である。伝統的な型推論では、式 e の意味 $[\![e]\!]$ は、型の世界とは独立に与えられることになっていたが、型の情報が式の計算中に利用可能になった。例えば、以下のような Refinement を考えれば、この重要さはよくわかる。

```
type Unit = 'cm' | 'in'
let units: Unit[] = {'cm', 'in'}

type Width = {unit: Unit, val: number}

function generalizedlen(w: number | string | undefined): Width|null {
   if (w == null) { return null}
   if (typeof w === 'number) { return {unit: 'cm', val: w}}
   let u = parseUnit(w)
   if (u) {return {unit:u, val: parseFloat(w)}}
   return null
}
```

ここでは、上から下に、w の型の候補がひとつづつつぶされていくのがわかり、そこに typeof が本質的な役割を果たしているのがわかる。

5.1.5 型ガードと条件型

TypeScript では、式の計算と同じように、型も計算の対象になる。特に、型ガードは、式が特定の型をもつかどうかをチェックし、その後の計算にその型情報を付加する。特に、和で型が定義されていて、一方の型のメソッドを使いたいときには、型ガードを使わないと型チェックが通らない。

さらに、条件型が定義されていて、型の包含関係を判断して型をスイッチすることができる。例えば下のように。

```
type ISString<T> = T extends string? true: false
```

typeof, instanceof とこれらを組み合わせ、if, switch と併用することで、計算のフローの中で型情報を流通させ、型推論において型に関する条件を追加するることができる。

5.2 関数型

JavaScript では、関数は関数型 function をもつだけであるが、TypeScript では、call signature という形で、引数の型と結果の型を指定する。例えば、以下は TypeScript で正しいプログラムである。

```
function sum(a:number, b:number) {
   return a+b;
}
```

call signature として (a:number, b:number) => number をもつ。

関数にどのような signature を付けることを可能にするかは、その言語の型体系を特徴づける critical point になる。

5.2.1 引数と結果の型

引数には、optional なもの、可変個の引数を取るものがある。前者は?を後ろにつける。後者は... を前につける。

```
function sum1(a:number, b:number) {
   return a+b;
}

function sum2(a:number, b:number, c?:number) {
   if (c===undefined) {
      return a+b;
   } else {
      return a+b+c;
   }
}

function sum3(a:number, b:number, ...c:number[]) {
return a+b+c.reduce((total,n)=>total+n, 0);
}
```

これからわかるように、関数型における TypeScript の型チェックは、call signature チェックとして提供されている。関数の本体では、引数に与えた型を前提として型推論が行われる。関数本体では、返り値の型が推論されているはずであるので、特に指定しなくとも、よい。実際、上の例で d.ts ファイルを見ると次のようになっている。

```
declare function sum1(a: number, b: number): number;
declare function sum2(a: number, b: number, c?: number): number;
declare function sum3(a: number, b: number, ...c: number[]): number;
```

ここで例えば、a:string と宣言すると、演算子+の型チェックにひっかけることができる。しかし、a:any などと宣言しなおすと、結果の型が any になって型チェックを通ってしまう。さらに

```
function sum1(a:any, b:number):number {
   return a+b;
}
```

とすると、結果の型は number になるのだが、例えば sum1('aaa', 1) が TypeScript の型チェックを通ってしまう。any の扱いの危険さはここでも露呈する。

関数が呼び出されると、関数の実引数が call signature の各引数に対してチェックされる。call signature が、optional な引数や可変数個の引数を扱えることは、実用主義的なソフトウェア工学の点で大きな意味をもつことになる。

なお、関数名に*をつけると、その関数が generator になることを示し、型が IterableIterator になる。

5.2.2 Polymorphism

TypeScript は、型に Generics を導入している。Generics は C++をはじめ型パラメタを導入したシステマティックな Polymorphism である。典型例を下にあげる。

```
function filter<T>(arr:T[],f:(item:T)=>boolean):T[] {
   let result = []
   for (let i=0; i<arr.length,i++) {
      let item = arr[i]
      if (f(item)) {
         result.push(item)
      }
   }
   return result
}
```

型パラメタ T を先頭につけることで、predicative polymorphism といわれる体系の一種を導入していることになる[1]。

型パラメタには制限をつけることができる。後述する型階層を反映する `extends` をつけて、型パラメタのとる範囲を制限することができる。これも Generics で普通に考えられてきた Bounded Polymorphism の実装の一つである。

5.3 Class と Interface

TypeScript は、オブジェクト指向でクラスが扱える。クラスをどう型付けするかは関数と並んで中心的な話題の一つである。JavaScript では、オブジェクトは Object 型をもつとされるだけであったが、TypeScript ではもちろんそれだけではない。クラスとその中のメソッドをチェックできることで、JavaScript のメソッドの実行時エラーは、早期にキャッチできることになった。

とりあえず、TypeScript での記述と、対応する JavaScript のプログラムをあげてみる。

```
*** TypeScript ***

class Position {
  constructor(
     private x:number, private y:number
     ){}
  diag(this:Position) {
      return this.x-this.y
  }
```

[1] predicative でないと、F を未だにやっている人をはじめとした数理論理学者が殴りこんでくるので注意。

```
}

var x = new Position(1,2)
x.diag()

*** JavaScript ***
var Position = /** @class */ (function () {
    function Position(x, y) {
        this.x = x;
        this.y = y;
    }
    Position.prototype.diag = function () {
        return this.x - this.y;
    };
    return Position;
}());
var x = new Position(1, 2);
console.log(x.diag());

*** declarations ***
declare class Position {
    private x;
    private y;
    constructor(x: number, y: number);
    diag(this: Position): number;
}
declare var x: Position;
```

実は、Curry-Howard Isomorphism を起源とする初期の型理論からの飛躍をもたらしたのは、クラスとそれに関連する継承その他の概念の解析であった。クラスのもつ豊かな世界は、型に関する研究を飛躍的に複雑にし、実用的にした。

5.3.1 interface

クラスは、メンバーとメソッドのリストからなる。対応するクラスの型は、メンバー名とその型、メソッド名とその型とする。これを shape という。interface には private なものを指定しない。

interface は、implements によって、メンバーやメソッドを具体的に定義することができる。さらに、extends によって、クラスを継承によって拡張することができる。

```
interface Pos {
   diag():number
   }

interface Pos2 extends Pos {
   xcoord(): number
}
```

表 1: TypeScript の型階層

				any			
↓	↓	↓	↓	↓	↓	↓	↓
string	number	bigint	boolean	symbol	undefined	object	function
↓	↓			↓		↓	
string enum	number enum			unique symbol		array	
						↓	
						tuple	

```
class Position implements Pos {
  constructor(
     protected x:number, private y:number
     ){}
  diag(this:Position) {
      return this.x-this.y
  }
}

class Position2 implements Pos2 {
  constructor(
     private x:number, private y:number
     ){}
  diag(this:Position2) {
      return this.x - this.y
  }
  xcoord(this:Position2) {
      return this.x
  }
}
```

5.3.2 Generic Class

クラスにも Generic class が定義されている。考え方は関数の時と同じ。この場合は、Generics はテンプレートとしての役割が大きくなっていることが関数の場合と違う（個人の感想かもしれない)。

5.4 型の階層

TypeScript では、型の階層が定義されている。表 1 に図示する。

表 2: クラスの階層と array, function 内の階層

S	S	array<S>	S1	S2	S1 => S2
↓	↓	↓	↑	↓	↓
T (such that T extends S)	T	array<T>	T1	T2	T1 => T2

表 3: literal の階層

string	number	···
↓	↓	↓
string literal	number literal	

さらに、object は、クラスの階層をもつ。array や function は型構成子としても働く。以上を表 2 に示す。

literal 及び 型の和と積は階層としては [1] に書かれていないが、実際には、階層として理解される（表 3)。

5.5 型の Widening

TypeScript は、型推論においては、最も一般的な型を答えとして出す。典型的なのは literal の扱いで、literal と明示的に宣言しなければ、対応する型が割り当てられる。特に関数からの返り値は、関数を呼んだ文脈に従うことになる。[1] から例をあげる。

```
let a = 'x' //string
let b = 3   //number

let aa:'x' = 'x' //'x'
let bb: 3 = 3    // 3
```

5.6 Refinement

Refinement は、特に和の形をしている型に関係する推論において、計算のフローの中で if 文等で型に関する条件を付加することで、和の形の条件をフローの途中で外していく（より特定の型に絞り込む）ことをさす。

6 TypeScript の型システムの特徴 – 最後に

動的な型付けしか提供されていない JavaScript の preprocessor の形で TypeScript が提供され、そこでは静的な型付けが提供されていることをみた。従来の C, Fortran、さらに関数型言語の多くと異なり、TypeScript では、typeof, instanceof に見られるように、式 term と型 term が厳格に分離されておらず、さらに計算フロー中にも、refinement の形で型推論が進行する形になっている。

重要なことは、TypeScript は、これを symbolic execution の形で推論することである。例えば、literal 型にかけるものに式一般を許さず、文法的に literal なものに限ることで、symbolic execution と、その結果として静的型付けを可能にしている。型システムでのいろいろな制限は、symbolic execution が可能になる範囲で行うという方針に沿ったものである。

より重要なことは、型チェックが、ソフトウェア工学的なツールとしてうまく機能するように設計していることである。計算フローのなかで型の refinement ができるようになっていることはその典型である。また、クラスの階層を symbolic execution に利用できるようにしている。加えて、generics の扱いも、C++のように、計算を構成する部品一般に広げることをせずに、実質的に関数とクラスに限定していること、さらにクラスの generics はテンプレートとして使用することを想定するなど、理論的実際的な問題を回避している。

型体系は、数理論理学と深いつながりがあると指摘され、実際 1980–1990 年代の発展はそれを強い動機としてきた。しかし、21 世紀に入り、オブジェクト指向の普及により、クラス階層という複雑かつ興味深い対象を得たことで、ソフトウェア工学的に型システムの研究は大きな発展を見せるようになった。最後に指摘してこの稿の終わりとしたい。

参考文献

[1] Boris Cherny, Programming TypeScript, O'Reilly, 2019.

[2] B. C. Pierce, Types and Programming Languages, 2002.

[3] J. Reynolds, "Polymorphism is not set-theoretical," Proc. Int'l Symposium of Semantics of Data Types, LNCS???, 1984.

[4] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In Proceedings of the 2007 symposium on Dynamic languages (DLS '07), 2007, pp. 53 64.

[5] R. S. Malik, J. Patra and M. Pradel, "NL2Type: Inferring JavaScript Function Types from Natural Language Information," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 304-315.

[6] C. Boone, N. de Bruin, A. Langerak, F. Stelmach, "DLTPy: DEEP LEARNING TYPE INFERENCE OF PYTHON FUNCTION SIGNATURES USING NATURAL LANGUAGE CONTEXT," arXiv:1912:00680v1, 2019.