F#を用いたJavaScriptの型検査器の実装

田村 健介　倉光 君郎

近年の Web アプリケーションの開発に用いられる代表的な言語として JavaScript がある. しかし, JavaScript は動的型付け言語であるため, 静的に型エラーを検知することができないといった問題がある. そこで, 我々は JavaScript を F#に変換し, F#の型システムの上で JavaScript の型検査を行う手法を提案する.F#は強い静的型付け言語であり, JavaScript と比較して型付けが厳格である. これを利用し, JavaScript を F#の型システムで検証することで, 型エラーの検出が期待できる. 本論文では, JavaScript から F#への変換アルゴリズムについて説明し, 変換の限界について述べる. また, 本研究によって型エラーが検出できた例を提示し, 実装の評価を行う.

JavaScript is a standard language in developing modern Web applications. However, JavaScript has a problem. The problem is that a static checker for JavaScript is poor, because JavaScript is a dynamically language. We propose a practical implementation for static checker. We convert JavaScript to F#, and check JavaScript on F#'s type system. F# is a strong statically language, and its type system is strong than JavaScript's. By using F#'s type system, we can expect finding type errors from JavaScript. In this paper, we describe the conversion algorithm from JavaScript to F#, and the limits of conversion. We show examples that could be detected type error by this study, and evaluate the implementation.

1　はじめに

近年, Web アプリケーションの開発において, JavaScript は主要な開発言語である [4]. JavaScript は C 言語のような構文, 動的型付け, プロトタイプベースのオブジェクト指向や第一級関数などの特徴を持つ.JavaScript を用いる開発者は動的型付けによる柔軟なプログラミングを行っている. しかし, それと同時に動的型付けであるために, 静的検査によるエラー検証が困難であり, 保守性を保証することができない.

JavaScript のような型付けの弱いコードから型情報を静的に推論する検査器の構築において, 型の再構築や型推論は主要な技術である. これまで, 多くの研究者や開発者が, JavaScript のコンテキスト上で, 型の再構築に関する研究を行ってきた. これらは JavaScript の実行速度を向上させるための最適化を目的に行われ, 部分的な型の再構築によって多くの成果が得られた. しかし, エラー検証を目的とした実用的な静的検査器の実現には 2 つの障害が存在する.1 つ目は, JavaScript 全体の型システムの仕様は難しいということにある. これにより, JavaScript の型システムに対して型推論を実装することは容易ではない.2 つ目は, JavaScript の型システムは柔軟な性質によって成り立っており, 検査器を用いるには適さないことにある. 例えば, 図 1 において, 関数呼び出しである `f("hello")` は JavaScript の型システムに則っているが, 開発者が想定する挙動とは異なる挙動をする可能性がある.

我々は実用的な型検査器の実現手法を新たに提案する. その手法とは, JavaScript のコードを外部の静的型付け言語の型システムで評価するものである. つまり, 提案する型検査器は JavaScript のコードを外部の静的型付け言語に変換し, 変換後の言語のコンパイ

Implementation of the type checker for JavaScript using F#.

Kensuke Tamura, Kimio Kuramitsu, 横浜国立大学大学院工学府, Graduate School of Engineering Yokohama National University.

```
function f(n) {
  return n+1;
}
console.log(f("hello"));
```

図 1　型エラーが内包されている可能性のある JavaScript のコード例

ラを型検査器として利用することで JavaScript の型検査を実現する. この手法には以下のような点で望ましい.

- 外部言語の型推論器を用いることで, 型推論器開発にまつわる煩雑さを避けることができる
- 外部言語の信頼できる型システムの上で検査するため, 型検査の信頼性を保つことができる
- 型検査に利用する型システムが明確である
- 外部言語がよいプログラミングモデルを提供するなら, そのモデルにしたがって信頼できるコーディングが行える

一方, 外部言語にとって型エラーとなるコードは, JavaScript にとって正しいコードであっても型エラーとなる. これは, 汎用的な JavaScript の検査器としては問題点であるが, より優れたモデルによるコード品質管理としては, 我々の目的に合致する.

本研究は, 外部言語として F#[5] を用いる.F#は, OCaml をベースに設計された関数型プログラミング言語である.F#を選ぶのは, いくつかのよい理由がある.F#は, よく設計された関数型プログラミング言語で, 強力な型推論を備えている. また, .NET のインターフェースから型情報が修得しやすい.

2　問題と解決

我々の提案する型検査器は JavaScript のソースコードを F#に変換し, これを F#の実行系で動作させることで型検査を行う. この実行結果にコンパイルエラーが存在する場合は, F#の型システムを違反したソースコードであるとして, 型エラーが存在すると判断する.F#の型システムを違反した記述を含む JavaScript のソースコード例を図**??**に示す.

図 2 において, 関数 dev() によって返される値は

```
function dev() {
   var a = 10;
   var b = " 30px ";
   return a / b;
}
```

図 2　F#の型システムを違反している JavaScript のソースコード例

NaN(Not a Number) となり, この実行結果は多くの場合において期待する結果ではない. さらに, JavaScript では NaN を返す動作はエラーではないため, 関数 dev() の実行後も処理は続く. これにより, 連鎖的に期待しない結果が生じ, 最終的に出力される結果からエラーの原因を突き止めることは非常に困難になる.

そこで, 図 2 のソースコードを例に提案した型検査器でどのようにしてエラーを検知するか説明する. 図 2 のソースコードを F#に変換したものを図 3 に示す.

```
type ScopeOf_dev = class
  member self.a = ref None
  member self.b = ref None
  member self.dev =
    self.a := Some(10)
    self.b := Some(" 30px ")
    ref(Some((!(self.a)).Value / (!(self.b)).Value))
  end
```

図 3　F#への変換例

さらに, これを実行して得られる出力を図 4 に示す.

```
Error 4: Number / String is error
```

図 4　型エラー情報の出力例

F#の型システムでは, 異なる型の四則演算は許されていない. そのため, 図 2 の入力から型検査器は, 異なる型の除算に対してエラーを出力している. これにより, プログラマは予期しない結果が発生する原因を発見することができる.

3　JavaScript と F#

本研究では, 外部言語への変換によって JavaScript の型検査を実現する. 我々は, この外部言語として F#を選択するが, JavaScript と F#の言語の特徴は

大きく異なる. そのため, JavaScript のソースコードは, F#のソースコードに容易に変換できるものではない. 本節では, 言語間の違いを考慮した対応付けについて述べる.

3.1 値

JavaScript の値と F#の値の対応は, JavaScript の型を F#の型に置き換えて扱うことで実現する. 表 1 に本研究における, JavaScript の型と F#の型の対応表を示す.

表 1 JavaScript と F#の型の対応表

JavaScript の型	F#の型
Number	float
Boolean	boolean
String	string

JavaScript の Number 型は ECMAScript 標準仕様 [3] によれば, 「倍精度 64 ビット形式による IEEE754 値」となっている. 一方で, F#の float 型は「64 ビット浮動小数点型」となっており, 表現可能な値の範囲が等しいため, 等価に扱うことが可能である. しかし, インデックスの値としての Number 型の値など, 整数として振る舞わなければならない場合がある. そこで, このような場合では, F#の int 型に型変換して対応する.

JavaScript の Boolean 型は true か false のいずれかの定数を表す型である. これは, F#の boolean 型においても, 同様の振る舞いをするため, これらは等価に扱える.

JavaScript の String 型は文字列を値に取り, 最大サイズは仕様として明確に定まっておらず, 処理系に依存している. 一方で, F#の string 型は JavaScript と同様に文字列を値に取り, 最大サイズは 2GB までとなっている. 最大サイズはいずれの言語でも一般の実用に耐えるため, これらは等価に扱えるものとする.

また, 値は F#のオプション型で覆って扱う. これは, JavaScript の null などの値が存在しないことを表す値を表現するためである.

3.2 オブジェクト

JavaScript のオブジェクトは, F#のクラスと対応する. オブジェクトのプロパティをクラスのメンバとして扱う. 同じコンストラクタ関数から生成されたオブジェクトは, 同じクラスのインスタンスとして扱い, JSON [2] 形式のオブジェクトは, それぞれ固有のクラスを生成し, 複製されたオブジェクトは同じクラスのインスタンスとして扱う. また, JavaScript では動的にプロパティを追加, 削除できる. 本研究では, オブジェクトの型付けを structural subtyping として扱うので, プロパティの削除は考えない. 動的なプロパティの追加は, 解析時に先読みし, オブジェクト定義時に動的に追加されたプロパティも定義する.

3.3 関数

JavaScript では関数は第 1 級オブジェクトである. そのため, オブジェクトと同様の操作や参照が可能である. そこで, JavaScript の関数をオブジェクトと同様に F#のクラスに置き換える. 生成するクラスは関数が行う処理とローカル変数をメンバとして持ち, さらにプロパティもメンバとして持つ. このようにすることで, F#においてもオブジェクトのように扱うことが可能となる. また, ローカル変数をメンバに持つことで, 外部からのローカル変数の参照が容易になり, 型情報の取得が容易になるといった利点がある.

3.4 制御構文

JavaScript では式と文の両方が存在するが, F#では式のみ存在する. ここで, JavaScript の主な構文を以下に挙げる.

- do…while
- for
- if…else
- switch
- while

if…else 構文は F#においても, if…else 構文として扱う. ここで, 式全体として返す値の型を統一するために以下のような処理を行う.then 節のみ存在する場合, else 節に then 節を複製して型の統一を図る. それ以外の場合では, 各節の最後に意味を持たない同一の

式を挿入し, 型を統一する.

switch はF#の match に置き換えられる.switch の分岐条件となる式は match の分岐条件の式に対応し, case がとる値は with に続くパターンと対応する. 各パターンの処理によって返される値の型も統一される必要があるので, 各パターンの処理の最後に意味を持たない同一の式を挿入する.

また, 本研究の解析手法は静的なものであり, 条件文による実行パスの分岐や, ループ文による実行パスのループなどは, 型情報に影響を与えないものとして扱う. do…while, while はループの条件文を抽出し, F#の if 文に置き換える. for 文はカウンタ変数を 0 から 1 までインクリメントするループに置き換える.JavaScript の for 文のループ条件式はループで行われる処理の先頭に挿入する. また, 1 回のループ処理が行われた後に実行される文はループ処理の最後に挿入する.

4 F#変換

JavaScript から F#への変換は言語間の仕様の違いなどにより, 等価に変換することは難しい. 本研究の目的は, 等価な変換ではなく, 変数や関数の型情報を抽出することにある. 従って, 本研究では変換前の JavaScript のプログラムと変換後の F#のプログラムで同じ動作は保証せず, 型情報に影響を与えるコードの変換に重点を置く.

これらの方針により, 実際に変換されたコードは無限ループなどの挙動をする可能性を含んでおり, 実行することは危険である. そのため, 変換したコードは実行されずに型推論のみ行われるように工夫する必要がある. そこで, まず JavaScript を F#への変換に適した構造に書き換えてから, F#のコードに変換する.

本節では, JavaScript の書き換えと F#へのコード変換のアルゴリズムについて説明する.

4.1 JavaScript の情報抽出

JavaScript の情報の抽出は, 木を探索して行う. 必要となる情報は以下に挙げるものである.

- 関数定義
- オブジェクト定義
- 変数定義
- プロパティの追加

変数定義には未定義の変数に対する代入文も含む. これらの情報は, JavaScript の関数とオブジェクトを F#のクラスに変換する用途と, JavaScript の変数や関数を参照する記述を F#のクラスのメンバを参照する記述に修正する用途で使用される.

これらの定義式の情報の抽出方法について述べる. あるノードを起点として, そのノード以下に存在するノードを再帰的に下降して探索することを, ここでは, スコープ探索と定義する. スコープ探索は次のような動作をする.

1. スコープ探索ごとにインスタンスを作成し, 固有の記憶域を持つ
2. 変数定義を検出し, 検出したノードを記憶する
3. 関数定義かオブジェクト定義に到達すると, そのノードを記憶し, そのノード以下の探索は行わない
4. スコープ探索が終了したとき, 記憶したノードの中に関数定義かオブジェクト定義のノードがあった場合, そのノードを起点に新たにスコープ探索を行う

まず, スコープ探索の起点を木の頂点のノードに設定して, 探索を開始する. これにより, トップレベルのスコープ探索を含む, 各関数, オブジェクトごとのスコープ探索が実行され, F#のクラスを生成するブロックごとに情報を抽出, 記憶できる.

4.2 JavaScript の書き換え

JavaScript のコードを書き換えるとき, 主に書き換える点は以下に挙げる点である.

- 関数とオブジェクト
- 関数の返り値
- 制御構文

JavaScript の関数, オブジェクトは F#のクラスに対応させるため, いずれも JavaScript のオブジェクトに書き換える.

オブジェクトの書き換えについて述べる. オブジェクトの定義は JSON 形式の定義であれば, 書き換えを行わない.JSON 形式の定義以外であれば, JSON 形

式の定義に書き換える. このとき, 動的に追加されたプロパティもオブジェクトの定義に追加する.

次に, 関数の書き換えについて述べる.JavaScript の関数定義は JSON 形式のオブジェクトの定義に書き換え, プロパティに関数が行う処理とローカル変数を定義する.

JavaScript の関数の返り値は, 複数の返り値が存在する場合に型の統一を図るために、プロパティに返り値として新たに変数を定義し、return 文をこの変数への代入文に書き換える.

JavaScript の制御構文の F#変換は第 3.4 項で説明した通りだが, for 文, while 文, if 文は JavaScript の書き換えの段階で対応するものに書き換える.

まず, for 文はカウンタ変数を 0 から 1 までインクリメントするループに置き換える. また, JavaScript の for 文のループ条件式はループで行われる処理の先頭に挿入する. また, 1 回のループ処理が行われた後に実行される文はループ処理の最後に挿入する.

次に while 文の書き換えについて説明する.while 文はループの条件文を用いた if 文に置き換える. なお, else 節は then 節の処理を複製したものを挿入する.

JavaScript の if 文の置き換えについて述べる.if 文は各節の処理の最後には意味のない式を挿入し, then 節しか存在しない場合は else 節に then 節を複製する.

4.3 F#のコード生成

F#への変換は先の項で述べた JavaScript の書き換えにより, 対応する構文に書き換え可能な記述となっている. 従って, 第 3 節で述べた対応に従って変換を行えば良い.

ただし, 第 4.1 項で検出したスコープはすべてトップレベルでクラス定義を行う. さらにローカル変数をクラスのメンバとして定義するとき, オプション型の None を初期値として定義する. メンバの初期値に値が存在しないことを示す None を用いることで, その後の命令によって型を決定することができる.

5 事例研究

本節では, 型エラーが含まれている JavaScript のソースコードを入力として, 得られた結果の例を入力と共にいくつか提示する.

まず, 異なる型の値を 1 つの演算に用いた例を示す. この例の入力を図 5 に示す.

```
function add10(x){
  var y = 10;
  return x + y;
}

function run(){
  var z = "hello";
  add10(z);
}
```

図 5 異なる型の演算を含む入力

図 5 のソースコードは, 関数 *add10* の return 文で, 引数 x と数値 10 を格納した変数 y の加算を行っている. これにより, F#の型システム上では引数 x は数値であると推論される. しかし, 関数 *run* で行なわれている関数 *add10* の呼び出しでは, 文字列”hello”を格納した変数 z が引数に用いられている. 従って, 異なる型の値同士の演算が行われるため, 型エラーとして検出されることが期待される.

図 5 から得られた出力を図 6 に示す.

```
Error 8: This expression was expected to have type
Number but here has type String
```

図 6 型エラー情報の出力例

出力結果において, *Error 8* となっている記述の *8* は入力の行番号を指している. 得られた結果から, Number 型の演算に String 型が用いられていることが検出できた.

次に構造の異なるオブジェクトを同一の引数に用いた入力例を示す.

図 7 のソースコードは, 関数 *printUser* の引数に, 異なる構造のオブジェクト *user* と *country* を用いている. 関数 *run* でオブジェクト *user* を引数に取り, この段階で関数 *printUser* は引数にオブジェクト *user* をとると推論される. 従って, 次の命令でオブジェクト *country* を引数に適用した段階で型エラーとして検出されることが期待される.

```
var user = {
  "name": "",
  "age": 22
}

var country = {
  "name": "japan",
  "language": "ja"
}

function printUser(user){
  console.log(user.name);
  console.log(user.age);
}

function run(){
  printUser(user);
  printUser(country);
}
```

図 7　異なる型の演算を含む入力

図 7 から得られた出力を図 8 に示す.

```
Error 18: This expression was expected to have type
ObjectOfuser but here has type ObjectOfcountry
```

図 8　型エラー情報の出力例

出力結果から異なる構造のオブジェクトが同じ扱いをされていたことが検出できた.

6　静的解析で困難な点

JavaScript は動的言語であるため, 静的解析では解析できる情報に限度がある. 例えば, JavaScript の this の指すオブジェクトは呼び出し方によって異なる. 静的解析ではプログラムの実行フローを追うことはできない. そのため, this を考慮した解析を行うことは困難である. そのほかにも, eval 関数も静的解析では解析することは難しい.eval 関数の実行コードは String 型の値であり, この String には変数の値を組み込むこともできる. 静的解析では変数の値を得ることはできないので, これを解析することは不可能である.

7　関連研究

TypeScript [1] は Microsoft 社が開発した JavaScript に静的型付けとクラスベースのオブジェクト指向を加えたスーパーセットの言語である.TypeScript はプログラマが変数や関数に任意に型を指定することで, TypeScript のコンパイラが型エラーを検出する. 型指定にはすべての型を許容する *any* を用意しており, これによって動的処理も表現することが可能となっている. しかし, 動的処理については型検査を行わず, プログラマは型検査を適用する部分と適用しない部分を任意に定める. これによって, JavaScript 特有の柔軟な表現力を失うことなく, 型検査の部分適用を実現している.

8　結論

本論文では, 外部言語への変換を用いた JavaScript の型検査を提案し, いくつかの簡単な型エラーの検出例を示した. 型推論の実装の面において, いくらかの利点を見出すことができた. しかし, 実用的な JavaScript のソースコードは動的な処理を利用しているものもあり, 第 6 節で説明したようにそのようなソースコードに提案した型検査を適用することは困難であると考えられる.

謝辞 本論文の初期の版について議論していただいた本多峻氏 (横浜国立大), 須藤建氏 (横浜国立大) に感謝する.

参 考 文 献

[1] Bierman, G., Abadi, M., and Torgersen, M.: Understanding TypeScript, *ECOOP Object-Oriented Programming*, Vol. 8586(2014), pp. 257–281.

[2] Ecma International: *ECMA-404 The JSON Data Interchange Standard*, October 2013.

[3] Ecma International: *ECMA-262 ECMAScript 2015 Language Specification*, June 2015.

[4] Lebresne, S., Richards, G., Östlund, J., Wrigstad, T., and Vitek, J.: Understanding the Dynamics of JavaScript, *STOP '09 Proceedings for the 1st workshop on Script to Program Evolution*, 2009, pp. 30–33.

[5] Smith, C.: *Programming F# 3.0, 2nd Edition*, O'Reilly Media, 2012.