社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

モデル生成に基づくJavaScriptプログラムの型検査系

大久保弘崇† 山本晋一郎† 坂部 俊樹†† 稲垣 康善†

† 愛知県立大学 情報科学部
〒 480-1198 愛知県愛知郡長久手町大字熊張字茨ヶ廻間 1522–3
†† 名古屋大学大学院 情報科学研究科
〒 464-8603 名古屋市千種区不老町
E-mail: †{ohkubo,yamamoto,inagaki}@ist.aichi-pu.ac.jp, ††sakabe@is.nagoya-u.ac.jp

あらまし JavaScript プログラムに対する CASE ツール・プラットフォームを利用した型推論システムについて述べる．従来，型推論システムを定式化するには，構文規則に対応させて推論規則を生成していた．プラットフォームが汎用にマークアップしたプログラムの XML 木を入力とする本システムでは,『部分木のパターンマッチングとアクションによる生成』に拡張しなければならない．この拡張により，形式的な手法を直接適用して，現実的な言語を対象とした型推論システムが構成できることを示す．型推論システムの後段では，モデル生成法に基づく定理証明系により生成された推論規則を満たすモデルである型割当を求める．

キーワード JavaScript，型推論系，XML，定理証明系，モデル生成法

A JavaScript Type Checker based on Model Generation Theorem Prover

Hirotaka OHKUBO†, Shinichiro YAMAMOTO†, Toshiki SAKABE††, and Yasuyoshi INAGAKI†

† Aichi Prefectural University, Faculty of Information Science and Technology
1522–3, Ibaragabasama, Kumabari, Nagakute-cho, Aichi
†† Nagoya University, Graduate School of Information Science
Furo-cho, Chikusa-ku, Nagoya, Aichi
E-mail: †{ohkubo,yamamoto,inagaki}@ist.aichi-pu.ac.jp, ††sakabe@is.nagoya-u.ac.jp

Abstract This paper introduces a type inference system for JavaScript, based on a CASE tool platform. In traditional methodology, type inference systems are formalized on their specialized syntax and inference rules corresponded to word generation rules. To formalize a type inference system on the general-purpose syntax, this correnspondence must be extended to between matching over abstract syntax tree and inference rules. With this extension, we show that a type inference system based on formal approach can be applicable to practical programming languages.

Key words JavaScript, type inference system, XML, model generation theorem prover

1. はじめに

本研究は，広く用いられているクライアントサイドのスクリプト JavaScript を対象として，型推論規則を素直な形で定理証明系上に実現することにより，JavaScript プログラムの信頼性を向上することを目標としている．Web アプリケーションにおけるクライアントサイドスクリプト言語として登場した JavaScript は，XUL [7] や Konfabulator [8] に見られるようにアプリケーション記述言語などへその用途を広げて普及している．より重要な場面で使用されるにつれ，信頼性を保証する手段が要求されてきている．JavaScript は型のないオブジェクト指向言語であるため，型検査を行うことにより一定の信頼性を保証することができる．

JavaScript を対象とした型体系として BabyJ^T 型推論系が提案されている [3]．ここでは，推論規則の集合として型推論系が提示されている．型検査系を実装する際，型推論規則が定める型割当を求める手続き的プログラムを作成する方法が一般的であるが，我々は定理証明系を用いてこの型推論系を JavaScript プログラムに直接適用することを試みた [1]．モデル生成法に基づく定理証明系 JavaMGTP [4] を用いて推論を直接実行させるために，推論手続き中の曖昧な部分を改良し，探索空間を規定する型のドメインの表現および，BabyJ^T 型推論規則をこの表

現を用いて JavaMGTP 文法で記述する方法を与えた．

次いで [2] において，フロントエンドの自動化と連携させて型推論体系を拡張することにより，関数の引数及び局所変数の扱いを強化した．正しい型を推論するための**型推論規則**だけでは不完全なモデルを排除できないことに対処するため，型の不整合を検出して不完全なモデル候補を棄却させる**型検査規則**の追加が必要であることを示した．

本稿では，CASE ツール・プラットフォームが提供する XML 木から必要な型推論規則を抽出する方法を示す．形式的に型推論系を定義する際には，独自の文脈自由文法を定義し，その文法の生成ルールと一対一対応で型推論規則を定義する方法が多く用いられる．BabyJ^T の型推論系の定義もこの方法を用いている．これらの文法は言語本来の文法とは異なり，またサブセットであるため，型推論系の機能を実用的なレベルに拡張する際に妨げとなる．言語のフルセットの XML 木を対象とすることでこの制限が排除される．その反面，CASE ツールのために汎用に設計された XML 木はノード単体では型を推論できるだけの意味を持たず，あるパターンをもつ部分木を型推論規則生成の対象とする必要がある．XML 木からパターンにマッチする部分木を取り出す方法には XPath [10] を用いる．取り出したパターンに対するアクションとして型推論規則を生成する構成は，従来の独自文法による定式化の拡張とみなせる．

2. BabyJ 言語とその型推論系 BabyJ^T

本稿で用いる型体系は，BabyJ^T [3] を基にしたものである．我々はこの型推論系を JavaMGTP で直接実行することを試みている [1] が，この際に体系の厳密化を行った．本節では文献 [1] で修正した BabyJ^T 型体系について概説する．

2.1 BabyJ

BabyJ 言語は JavaScript 言語のサブセットである．機能を本質的な部分に限定し，その制限した文法の上でプログラムに対する型推論系の形式化，および型付けされたプログラムの Java プログラムへの変換手法について議論している．BabyJ の文法を図 1 に示す．型推論系に影響する JavaScirpt からの本質的な制限は，関数の引数は唯一 `x`，関数本体の局所変数は唯一 `y`，値としての関数はオブジェクトでない，メンバの削除はできない，文字列の eval ができない，の 5 点である．

2.2 BabyJ^T

BabyJ プログラムに型を付与したものを BabyJ^T と呼ぶ．BabyJ^T では，関数は `new` 演算子とともに用いられるコンストラクタ関数，(大域) 関数呼び出しをされる大域関数，そのいずれでもないメンバ関数に区分され，区分外の形で呼び出されることはないと仮定する．

オブジェクトの型を区別するために，生成に用いたコンストラクタ関数の名前をそのオブジェクトの型として用いる．これをオブジェクト型と呼ぶ．メンバ関数がメンバ関数として動作するためには，コンストラクタなどでまずメンバに代入される必要がある．このように，メンバ関数は値として扱われるため，型をつける必要がある．関数型は，this の型，引数の型，返り値の型の 3 項組と定義される．this はオブジェクトであること

```
P ∈ Program   ::= D *
B ∈ Body      ::= var y; e
D ∈ FuncDecl  ::= function f (x) { B }

e ∈ Exp       ::= this              レシーバ
                | f                 関数
                | x                 引数
                | y                 局所変数
                | new f(e)          オブジェクト生成
                | null              空
                | e;e               逐次実行
                | e.m(e)            メンバ関数呼び出し
                | e.m               属性読み出し
                | f(e)              大域関数呼び出し
                | v=e               代入

v ∈ Var       ::= x | y | e.m
f ∈ FuncID
m ∈ MemberID
```

図 1　BabyJ の文法

から先頭の要素は必ずオブジェクト型になるが，後二者には，別の関数型が入ってもよい．

BabyJ^T の型体系には多相性はない．全ての部分式，全ての関数の this・引数・局所変数・返り値，全てのオブジェクト型と全ての属性名の組，に対して，プログラム全体で一意にオブジェクト型あるいは関数型を割り当てる．関数からその実行時の this・引数・局所変数・返り値の型への対応づけを写像 $\mathcal{P}(f) = (ts, t_{\mathrm{arg}}, t_{\mathrm{local}}, t_{\mathrm{ret}})$, オブジェクト型からそのメンバの型への対応づけを写像 $\mathcal{D}(ts, m) = t$ とする．BabyJ^T の型推論規則を図 2 に示す．ここで Γ は変数 `this`, `x`, `y` とその型との対応づけである．`noobject` は大域関数の実行環境の this に何も割り当てられないことを示す特別な型である．プログラムの全ての部分式に対して，推論規則により型が割り当てられるような $\mathcal{P}, \mathcal{D}$ の対応付けを，制約の解消によって決定する．型推論の結果は $\mathcal{P}, \mathcal{D}$ および部分式への型割り当てであり，$\mathcal{P}$ は関数の引数および局所変数への型割り当てを，$\mathcal{D}$ はコンストラクタが作るオブジェクトの各属性の型を与える．

3. JavaMGTP

JavaMGTP は，モデル生成法に基づく定理証明系である．値域限定性という制限をもつが，ホーン節に限らない一階述語論理の問題一般を扱える．特に，否定を含む推論を効率よく実行できる．JavaMGTP の入力は `a1,a2,···,an -> b1;b2;···;bn.` という形の規則の並びである．これは「a1,a2,···,an がすべて成立するとき b1,b2,···,bn のいずれかが成立する」と読む．リテラルの前に `-` を置くと否定となる．前件，後件のいずれも否定できる．前件, 後件は 0 個でもよい．前者は正節と呼びリテラルが恒真であることを，後者は負節と呼び前件が成立したとき矛盾することを表す．前件が共通する複数の規則の後件をコンマで続けて記述することができる．

評価述語という特徴的な機能がある．`{ }` で囲って数値演算

(Var)

$$\frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})\quad \Gamma \vdash \mathtt{y} : \Gamma(\mathtt{y})\quad \Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this})}$$

(Null)

$$\frac{}{\Gamma \vdash \mathtt{null} : t}$$

(Seq)

$$\frac{\Gamma \vdash \mathsf{e}_1 : t' \quad \Gamma \vdash \mathsf{e}_2 : t}{\Gamma \vdash \mathsf{e}_1;\mathsf{e}_2 : t}$$

(New-cons)

$$\frac{\mathsf{f}\text{ はコンストラクタ} \quad \mathcal{P}(\mathsf{f}) = (\mathsf{f}, t_a, t_l, t_r) \quad \Gamma \vdash \mathsf{e} : t_a}{\Gamma \vdash \mathtt{new}\ \mathsf{f}(\mathsf{e}) : \mathsf{f}}$$

(Member-sel)

$$\frac{\Gamma \vdash \mathsf{e} : t' \quad \mathcal{D}(t', \mathsf{m}) = t}{\Gamma \vdash \mathsf{e}.\mathsf{m} : t}$$

(Var-ass)

$$\frac{\Gamma \vdash \mathsf{e} : t \quad \Gamma(\mathsf{v}) = t}{\Gamma \vdash \mathsf{v} = \mathsf{e} : t}$$

(Member-ass)

$$\frac{\Gamma \vdash \mathsf{e1} : t' \quad \Gamma \vdash \mathsf{e2} : t \quad \mathcal{D}(t', \mathsf{m}) = t}{\Gamma \vdash \mathsf{e1}.\mathsf{m} = \mathsf{e2} : t}$$

(Member-call)

$$\frac{\Gamma \vdash \mathsf{e1} : t_o \quad \Gamma \vdash \mathsf{e2} : t_a \quad \mathcal{D}(t', \mathsf{m}) = (t_o, t_a, t_r)}{\Gamma \vdash \mathsf{e1}.\mathsf{m}(\mathsf{e2}) : t_r}$$

(Global call)

$$\frac{\mathsf{f}\text{ は大域関数} \quad \Gamma \vdash \mathsf{e2} : t_a \quad \mathcal{P}(\mathsf{f}) = (\mathsf{noobject}, t_a, t_l, t_r)}{\Gamma \vdash \mathsf{f}(\mathsf{e}) : t_r}$$

(Member-func)

$$\frac{\mathsf{f}\text{ はメンバ関数} \quad \mathcal{P}(\mathsf{f}) = (t_o, t_a, t_l, t_r)}{\Gamma \vdash \mathsf{f} : (t_o, t_a, t_r)}$$

図 2　BabyJ^T の型推論規則

や，ユーザ定義の Java クラスメソッドの呼び出しが行える．この機能を用いて，項の上の演算をユーザサイドで拡張できる．また，JavaMGTP のクラスを取り込むことで，一般の Java アプリケーションから JavaMGTP の推論機能を利用することができる．CASE ツールプラットフォームと連携した実際的な応用を考える際，これらの特徴は大きな利点となる．

4. Sapid と JSX-Model

Sapid [5], [6] は，細粒度のソフトウェア・リポジトリに基づいた CASE ツール・プラットフォームである．さまざまな言語を処理対象にしているが，この中に JavaScript に対する XML リポジトリである JSX-model がある.JSX-model では，ECMAScript 3rd Edition の文法規則によるプログラムを 7 種類の終端要素と 6 種類の非終端要素よってモデル化する．JSX-model は元のプログラムテキストの内容をテキスト要素として完全に含み，マークアップにより情報を追加しているという点で細粒度である．図 3 にタグとその意味を示す．

解析器は Java で実装されている．JavaScript ソースプログラムを入力として，JSX-model によりタグ付けされた XML ファイルを出力する．各要素にはユニークな `id` 属性が付けられている．また，解析器はシンボルの参照を解決する機能を持ち，`ident` 要素の `refid` 属性にその識別子の宣言部の `id` が設

非終端要素

`Program`	プログラム全体 (XML のルート要素)
`FunDec`	関数宣言
`Param`	仮引数宣言
`VarDec`	変数宣言
`Stmt`	文
`Expr`	式

終端要素

`ident`	識別子
`literal`	定数表記
`comment`	コメント
`kw`	予約語
`op`	演算子
`sp`	空白文字
`nl`	改行

文要素の sort 属性

`Block`, `Empty`, `Expr`, `If`, `For`, `While`, `DoWhile`, `Continue`, `Break`, `Return`, `With`, `Labelled`, `Switch`, `Throw`, `Try`

式要素の sort 属性

`FunCall`, `ArrayAccess`, `Allocation`, `Paren`, `VarRef`, `Literal`, `This`, `Empty`, `Argument`, `Void`, `TypeOf`, `Delete`, `StrictEqual`, `StrictNotEqual`, `Assign`，各種整数演算，演算を伴う代入，各種論理演算

図 3　JSX-model の要素

定されている．

5. JSX-Model からの型推論

BabyJ^T の型推論規則は，BabyJ の構文規則に一対一対応させて構成されている．逆にいえば，BabyJ の構文規則は BabyJ^T 型推論規則の構成を考慮して設計されたものである．具体的には，型推論規則の生成手続きは，BabyJ プログラムの抽象構文木について，全てのノードに対して一つずつ型推論規則を出力する．このとき，推論規則の種類はノードの種類のみで決定される．

これに対して，JSX-model の DOM tree は ECMA Script の文法に沿って構成される．すなわち，JSX-model の DOM node にから，BabyJ^T のように型推論規則を生成することはできない．このため，JSX-model を入力として型推論規則を生成する手続きは，BabyJ^T の抽象構文木のノードと対応する DOM tree のパターンを考え，これにマッチする部分木に対して型推論規則を出力する．

型検査を行うことのみが目的であれば，目的に合った構文規則に基づくパーザを利用して型検査系を構成すれば充分である．汎用の CASE ツール・プラットフォームを利用したため，型検査系の構成が複雑化している．しかし，同じプラットフォームを用いる他の CASE ツールに対して本型検査系の検査結果を受け渡すなど，汎用プラットフォームを用いることによる利点は大きい．

5.1 型 変 数

型を割り当てる対象には型変数を対応づける必要がある．BabyJ^T を発展させた [1], [2] では，全ての部分式を出現で識別し，これに型変数を対応づけた．JSX-model では，全てのノー

ドに識別子属性 (id) が設定されている．識別子はユニークであることが保証されていること，また，JavaScript プログラム中の識別子の参照に対して，その識別子の定義の箇所の id 値が refid 属性によって与えられるため，型変数を対応づける対象に id 属性を選択する．

関数 F の中の擬変数 this の型，返り値の型には this(F), ret(F) を対応づける．これは，以前の写像 $\mathcal{P}$ に代わるものである．コンストラクタ C のオブジェクトの属性 m の型には，dee(C,m) を対応づける．これは，以前の写像 $\mathcal{D}$ に対応するものである．

5.2 前 処 理

型推論手続きは，JSX-model によりタグ付けされた XML ファイルを入力として受け取る．木のパターンマッチによる推論規則抽出の前段階として，あらかじめいくつかの情報の取り出しをしておく必要がある．

5.2.1 関数の抽出と区分

本型推論系では，関数が重要な役割をもつ．手続きで最初にすることは，プログラム中の関数を列挙することである．これは，XPath 式 "/Program/FunDec" によって取り出すことができる．以下，ident 要素で示される関数名の id 属性を，関数の識別子とする．関数を，その使用法によってコンストラクタ・大域関数・メンバ関数に区分する．

5.2.2 型ドメインの生成

BabyJ^T の型体系を継承するので，コンストラクタをオブジェクト型の構成子とする．関数型は，引数が 2 以上の型構成子 ft による "ft(tt,tr,t1,t2,…)" の形の任意の項である．ここで，tt は this，tr は返り値，t1,t2,… は引数の型である．tt はオブジェクト型，その他は関数型を含む任意の型をとる．関数型は無限に生成できるが，モデル生成法により型推論を行うためには，制約解を探索するドメインを有限サイズで厳密に規定する必要がある．引数の数については，入力プログラムに存在する関数のものだけでよい．また，関数型構成子の入れ子を制限することで，型の空間を有限に近似する．JavaScript の利用目的からみて，極端な入れ子の段数をもつ関数型の重要性は低いと考えられる．

これら全ての型について，それが型構成子であるという述語 type(T) を満たすという公理を出力する．また，型の等価性 eq(S,T) を，組み合わせを尽くして宣言する．

5.2.3 各関数の型の宣言

抽出した関数全てについて，その型と引数の型や返り値の型の関係を定義する．例えば，id が idf，引数が 1 つでその id が ida であるメンバ関数について，

```
-> hut(this(idf)), hut(ret(idf)), hut(ida).
ta(this(idf),TF),ta(ret(idf,TR)),ta(ida,TA1)
-> ta(idf,ft(TF,TR,TA1)).
```

という推論規則を生成する．ここで，hut(id) は id に何かユニークな型が付いている (have unique type) という述語である．このとき，付けられた型は述語 ta(id,T) を真にする．このパターンにより，型変数を振ってモデルを探索する．

関数の引数および局所変数の扱いは, [2] では型ベクトルを用いていたためドメインの肥大化を招いたが，ここでは XML の id を用いて変数を特定する方法に変更し，抽出される推論規則と探索空間のサイズを抑えることかできた．

5.3 マッチングパターンと推論規則

ここでは，パターンにマッチする部分 DOM 木を探すために XPath を用いる．着目する部分木を得たあと，推論規則を生成するために必要なパラメータの抽出は DOM を用いてアクセスすることで行い，推論規則の生成は Java で行う．

以下，探し出す DOM 木のパターン，これにマッチする XPath 式，そこから生成する JavaMGTP 推論規則を列挙する．図 2 に対応する規則があるパターンについては，それを併記している．

a） 定数式 (Null を含む)

DOM Tree:

```
<Expr id="id" sort="Literal">
  <literal sort="lsort">
    定数表記
```

XPath:

```
//Expr[@sort='Literal']/literal/..
```

生成する推論規則:

```
-> ta(id,integer). (lsort=decimal, hex, octal のとき)
-> ta(id,string).  (lsort=string のとき)
-> ta(id,float).   (lsort=float のとき)
-> hut(id).        (lsort=unterminated のとき)
```

最後の場合は null のときである．

b） 式　　文

DOM Tree:

```
<Stmt id="id" sort="Expr">
  <Expr id="id1">
```

XPath:

```
//Stmt[@sort='Expr']
```

生成する推論規則:

```
ta(id1,T) -> ta(id,T).
```

引数を表す<Expr sort="Argument"><Expr> も同じ処理をする．

c） 変数参照 (Var)

DOM Tree:

```
<Expr id="id" sort="VarRef">
  <ident refid="rid">
    変数名
```

XPath:

```
//Expr[@sort='VarRef']/ident/..
```

生成する推論規則:

```
ta(rid,T) -> ta(id,T).
```

変数は局所変数・関数の仮引数・関数名のいずれかである．いずれにおいても，その定義部でその型を決めるような推論規則を生成するため，そこで決められる型を式 id の型とすればよい．

d） 逐次実行 (Seq)

DOM Tree:

```
<Stmt id="id" sort="Block">
```

```
  <Stmt id="id1"><op>;</op>
   :
  <Stmt id="idn">
```
XPath:
```
//Stmt[@sort='Block']
```
生成する推論規則:
```
ta(id1,T1),…,ta(idn,Tn) -> ta(id,Tn).
```
e） メンバ参照 (Member-sel)

DOM Tree:
```
<Expr id="id" sort="VarRef">
  <Expr id="id1">
  <op>.</op>
  <ident>
    member
```
XPath:
```
//Expr[@sort='VarRef']/Expr/
following-sibling::op[string()='.']/
following-sibling::ident/..
```
生成する推論規則:
```
ta(id1,T) -> hut(dee(T,member)).
ta(id1,T), ta(dee(T,member),S) -> ta(id,S).
```
メンバ参照と変数参照はノードの属性だけでは区別できない．参照される属性には，型が割り当てられる必要があることを 1 行めの hut で記述している．

f） 代入 (Var-ass, Member-ass)

DOM Tree:
```
<Expr id="id" sort="Assign">
  <Expr id="id1">
  <op>=</op>
  <Expr id="id2">
```
XPath:
```
//Expr[@sort='Assign']
```
生成する推論規則:
```
ta(id2,T) -> ta(id1,T).
```
変数への代入とメンバへの代入は，ここでは区別する必要がない．左辺の変数の型は上記の変数参照あるいはメンバ参照で求められているので，ここで必要なことは右辺の型と等しいことを確認するだけでよい．型が等しくない場合はそのモデル候補は棄却される．このような性質の規則を型検査規則と呼ぶ．

g） オブジェクト生成 (New-cons)

DOM Tree:
```
<Expr id="id" sort="Allocation">
  <kw>new</kw>
  <Expr id="idc" sort="FunCall">
    <Expr id="id0" sort="VarRef">
      <ident refid="rid">
    <op>(</op>
    <Expr id="id1" sort="Argument">
     :
    <op>)</op>
```
ここで，rid はこの式を含む関数の局所変数・引数ではない．

XPath:
```
//Expr[@sort='Allocation']
```
生成する推論規則:
```
ta(id1,TP1),… -> ta(id0,ft(rid,rid,TP1,…)),
                 ta(id,rid).
```
h） 大域呼び出し (Global-call)

DOM Tree:
```
<Expr id="id" sort="FunCall">
  <Expr id="id0" sort="VarRef">
    <ident refid="rid">
  <op>(</op>
  <Expr id="id1" sort="Argument">
   :
  <op>)</op>
```
ただし，rid はこの式を含む関数の局所変数、引数ではないとする．また，この subtree の親ノードは<Expr sort="Allocation">ではない．

XPath:
```
//*[@sort!='Allocation']/Expr[@sort='FunCall']
```
生成する推論規則:
```
-> hut(id).
ta(id,TR), ta(id1,TP1), … ->
  ta(id0,ft(noobject, TR, TP1, …)).
```
i） メンバ呼び出し (Member-call)

DOM Tree:
```
<Expr id="id" sort="FunCall">
  <Expr id="idm" sort="VarRef">
    <Expr id="ido">
    <op>.</op>
    <ident>
      member
  <op>(</op>
  <Expr id="id1" sort="Argument">
   :
  <op>)</op>
```
XPath:
```
//Expr[@sort='FunCall']/Expr/Expr/
following-sibling::op[string()='.']/
following-sibling::ident/../..
```
生成する推論規則:
```
-> hut(id).
ta(id,TR), ta(ido,TO), ta(id1,TP1), … ->
  ta(dee(TO,member),ft(TO,TR, TP1, …)).
```

以上により，図 2 に相当する型推論規則が構成された．本アプローチでは，同様のパターンとアクションの対を追加定義することで，型推論系が対応する構文を増やすことができる．BabyJ^T では，文法定義の拡張からやり直す必要があることに比べ，一般性を保ち拡張性が高いといえる．

6. ま と め

CASE ツール・プラットフォームが提供する汎用のプログラムリポジトリを対象として，形式的なアプローチによる型推論システムを構成した．型を割り当てる対象が XML 木のノードではなく部分木として表現されるため，これをマッチングによって取り出し，マッチに対するアクションとして型推論規則を抽出する．型推論規則が抽出された後は，モデル生成法に基づく定理証明系により推論規則を満たすモデルが高速に探索される．このモデルは入力プログラムへの型割当である．型変数は XML の `id` 属性に対応づけられており，与えられた型割当は容易に入力の XML リポジトリに反映させることができる．これにより，プラットフォームで共通化したフォーマットに基づき CASE ツール間での連携が行える．

現実的な JavaScript 構文に対応できることを実際に示すため，さらにマッチとアクションの組を追加する必要がある．JavaScript プログラムの実例への適用と併せて，本方式の現実問題への適用を急ぎたい．

また，JavaScipt は単体で用いられる言語ではなく HTML や XUL といった他の文書に埋め込んで利用される．このとき，外側の文書により定義されるデータオブジェクトが JavaScript から可視になっている．これら外部オブジェクトを操作する JavaScript プログラムの信頼性を型検査によって保証するためには，本稿で JavaScirpt プログラムから型推論の公理を生成したように，同時に外側の文書を読み込んで必要な公理を生成する機能がフロントエンドに必要である．

BabyJ^T 型体系では，オブジェクト指向プログラムが本来持つべき型の多相性が全て排除されている．例えば，メンバ関数に対して，this の型を一意に定めるため，一つのメンバ関数を異なるコンストラクタから生成されるオブジェクト間で共用することさえできない．我々はクラスベースのオブジェクト指向言語に対する型検査アルゴリズムを提案している [9]．そこでは，クラスの集合をオブジェクトの型として，制約解消に基づく型推論を行う．BabyJ^T ではオブジェクト型はコンストラクタであるが，これをコンストラクタの集合に置き換えることで, [9] の手法を応用してオブジェクト指向プログラムの型の多相性を扱える型体系を構築する予定である．

文　献

[1] 大久保 弘崇, 山本 晋一郎, 坂部 俊樹, 稲垣 康善, “形式的手法に基づく JavaScript プログラムの型検査系の実現,” 信学技報 Vol.104, No.47, (ISSN 0913-5685), pp.13–18, April 2004.

[2] 大久保 弘崇, 山本 晋一郎, 坂部 俊樹, 稲垣 康善, “モデル生成に基づく JavaScript プログラム型検査のためのフロントエンド,” 信学技報 Vol.104, No.242, (ISSN 0913-5685), pp.41–46, August 2004.

[3] Christopher Anderson and Sophia Drossopoulou, “BabyJ: from object based to class based programming via types,” WOOD2003 Workshop on Object Oriented Developments, Warsaw, Poland, April 2003.

[4] 長谷川 隆三, 藤田 博, “Java によるモデル生成型定理証明系 MGTP の開発,” 情報処理学会論文誌, vol.41, No.6, pp.1791–1798, June 2000.

[5] 福安 直樹, 山本 晋一郎, 阿草 清滋, “細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid,” 情報処理学会論文誌, Vol.39, No.6, pp.1990–1998, June 1998.

[6] Sapid : Sophisticated APIs for CASE tool Development, `http://www.sapid.org/`

[7] XUL : XML User Interface Language, `http://www.mozilla.org/projects/xul/`

[8] Konfabulator, `http://www.konfabulator.com/`

[9] 大久保 弘崇, 坂部 俊樹, 稲垣 康善, “オブジェクト指向プログラムに対する Message Not Understood フォールト検知のための型検査アルゴリズム,” コンピュータソフトウェア, Vol.17, No.3, pp.63–76, 2000.

[10] XML Path Language(XPath) Version 1.0, `http://www.w3.org/TR/1999/REC-xpath-19991116`, W3C, 1999.