モデル生成に基づく JavaScript プログラムの型検査系の構築について

大久保 弘崇* 山本 晋一郎 * 坂部 俊樹† 稲垣 康善 *

概要

本論文では，JavaScript プログラムに対する静的な型検査系の構築について述べる．オブジェクトと関数からなる単純な型体系 BabyJ^T を，JavaScript の主要な構文を扱えるように拡張し，型検査系の形式的な定義を与え型検査のための論理式を構文に対応して抽出する．抽出された論理式は，定理自動証明系により充足可能性が判断される．形式的に定義される提案手法を，直接機械実行する形に実装し，実際に JavaScript プログラムの型検査が行えることを示した．JavaScript 構文から論理式を導出するには，Sapid の XML 形式と XSLT 技術を用いた．また定理自動証明系には MGTP を用いた．

1 まえがき

1.1 はじめに

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 [*1]，Windows の HTML Application [*2]，MacOS X の Dashboard [*3] 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，また WWW の分野では AJAX の普及により JavaScript プログラムの複雑さが増大する一方で，より重要な場面で使用されるにつれ，その信頼性の向上が要求されるようになっている．

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 Anderson らによって提案されている [1]．そこでは，型推論系が推論規則の集合として提示されている．このような推論系を基礎にして型検査系を実装する際には，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，著者らはモデル生成法に基づく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 [2]．その目的は，形式的手法を現実的な問題に直接適用する可能性を追求し，またそのために必要な定理証明系の機能や言語体系を明らかにすることであった．定理証明系にはモデル生成法に基づく JavaMGTP[3] を用いた．モデル生成法によれば，推論規則を充足するモデルが自動的に取得できる．型推論規則を充足するモデルは矛盾のない型割当であるので，証明の成功は型検査の視点からそのプログラムに矛盾のないことを意味しており，この意味でプログラムの妥当性を保証する．さらには，この型割当は，JavaScript に対するソフトウェア理解支援などの CASE ツールとして応用できると考えられる．

しかしながら [1, 2] の論文では，JavaScript の構文のうち非常に限定的な部分しか扱っていないこと，型検査を行う基礎である型の体系が非常に単純なものであること，の二つの問題点がある．そこで，本論文では，構文の対応を現実的な範囲に拡大し，それに対して，JavaScript プログラムから論理式導出の機械実行可能な形式的な枠組みを示し，これに基づいて型検査系の実装を与える．

1.2 考え方

本論文で提案する型検査系構成の考え方の要点を述べておく。本提案システムは，JavaScript に対する静的型検査システムであり，その特徴は型検査系を形式的な手法で定義することである．すなわち，言語の意味に基づいて，型を導出する推論規則が構文規則

* 愛知県立大学 情報科学部
† 名古屋大学大学院 情報科学研究科

[*1] http://www.mozilla.org/projects/xul/
[*2] http://msdn.microsoft.com/
[*3] http://www.apple.com/jp/macosx/features/dashboard/

に沿って定義される．構文から論理式を導出する仕組みの実装には，JavaScript プログラムの XML 化表現である JSX-model を利用する．XML に対してパターンマッチを行い，論理式を導出する規則の記述形式には，XSLT を利用する．さらに，導出した論理式に基づいて，モデル生成に基づく定理証明系を用いて型推論を機械実行する．矛盾なく型が推論されたとき，プログラムへの正しい型割当が得られる．ここでは定理証明系に MGTP を用いた．MGTP はモデル生成法に基づく推論系であり，horn 節に限定されない否定を扱える一階述語論理の推論系である．モデル生成法に基づく推論では，証明を与えることは論理式を満たすモデルを与えることあるので，MGTP によるモデルの探索はプログラムへの正しい型割当の探索とみなせる．定理証明系として MGTP を用いるので，JSX-model から導出される論理式は MGTP 論理式である．したがって，JSX-model からそれに対応する MGTP 論理式を導出する XSLT を記述することが，本提案システムの形式的かつ機械実行可能な型推論系を実装するための中心的作業となる．

モデル生成法に基づく推論系を採用することにより，ドメインを有限にし，その要素が列挙できなければならないという制限が課せられる．型推論を行う上でのドメインとはプログラムの型である．BabyJ^T 型体系おける関数型の空間は無限の要素を持つが，有限の要素で近似して扱う必要がある (6.1 節)．

2 JavaScript プログラムの型検査

本章では，本論文で提案する，形式的手法を用いて JavaScript プログラムに対して型を付与する手法の概略を説明する．

2.1 型体系

型体系には，BabyJ^T を拡張した単純なものを用いる．この型体系では型の多相性は扱わない．型はオブジェクト型と関数型に大別される．あるオブジェクトが new 演算子によって生成されるとき，new の引数として呼び出されるコンストラクタ関数名をそのオブジェクトの型とする．すなわち，オブジェクト型の要素はコンストラクタ関数名である．アリティ n の関数に対する関数型は，型の n+2 項組である．その内訳は，n 個の引数と返り値，そして this の型である．

BabyJ^T 型体系からの拡張として，undefined 型と Number 型を追加する．undefined 型の唯一の値は void である．Number 型は数値全体を含む型である．暗黙の型変換の問題を回避しつつ，基本型を型体系に導入するためにこのようにした．

2.2 型の割当

プログラム全体に矛盾なく型が割り当てられたとき，そのプログラムは型が正しいと判定する．型を割り当てる要素は (部分) 式，変数，オブジェクトの属性，関数の環境，である．オブジェクトの属性への型割当とは，オブジェクト型の要素と属性名の組に対して型を割り当てることである．関数の環境とは，関数が評価されるときの this，返り値，全ての仮引数，全ての内部変数，である．著者らは論文 [2] において，BabyJ^T 型体系の問題点の一つとして関数のアリティ及び局所変数がそれぞれ 1 個に固定されている点を挙げた．本論文ではこの制限を外し，任意個数に拡張している．

2.3 論理式の導出

JavaScript の構文要素ごとに，型割当対象が満たすべき関係を論理式で記述する。

まず，プログラム中の関数宣言に対して環境を設定する．各構文要素に対して，環境や部分要素への型割当に基づき，対象の要素に割り当てる型を導く論理式を導出する．これを**型推論規則**という．

型推論規則とならんで，プログラムの型安全性を確認する論理式も同様に導出する必要がある．例えば，オブジェクトの属性をアクセスする式に対して，属性をアクセスされる式がオブジェクト型でなければならないという条件を検査する．これを**型検査規則**と呼ぶ．型の正しくないプログラムに対する，型推論規則のみからなる規則集合での推論の実行は，型に矛盾のない部分だけの不完全な型割当による充足という結果を導く．型検査規則を追加することにより，このような不完全な解を矛盾として検出できる．このことは，すでに著者らの論文 [2] で示されている．

2.4 型推論の実行

導出された論理式は MGTP の文法に従っており，MGTP 処理系により推論が実行できる．モデル生成法では，与えられた論理式を満たすモデルを発見することにより証明を行う．プログラムの型に誤りがある場合，関数の環境をどのように設定しても矛盾なく

型を割り当てる方法がなく，推論規則を充足するモデルは存在しない．プログラムの型が正しい場合，その推論規則を満たすモデルには，プログラムの各要素への矛盾のない型割当が含まれる．モデルからこの型割当を容易に取り出すことができる．

3 サブセット言語

JavaScript の標準の仕様書である ECMAScript Language Specification[4] では，65 種類の構文要素が定義されている．現実問題に適用する型検査系に求められる機能を検討し，本論文ではそれらのうち 20 種類の構文要素に制限したサブセット言語を設定し，これに対応する型検査系を構築する．

3.1 構文上の制限

型推論系を単純化するため，例外処理に関する構文は除外した．制御構文については if 文の扱いのみを示す。他の制御構文についても，同様に扱える．省略記法などの，対応方法が自明な構文についても省略した．オブジェクトの属性アクセスはドット記法によるもののみ扱い，括弧記法によるものは扱わない．また，配列は扱わない．関数定義は，名前つきの定義のみを扱う．

3.2 意味上の制限

関数は全て名前を持ち，また使用方法がコンストラクタ，メンバ関数，大域関数のいずれかに限定されると仮定する．関数の使用方法の分類は，new 演算子とともに使用されていればコンストラクタ，大域呼び出しされていれば大域関数，そのいずれでもない場合にメンバ関数と判断する．JavaScript では値としての関数もオブジェクトであるが，本手法では関数をオブジェクトとして扱うプログラムは対象としない．文字列型と数値型の自動変換などの，暗黙の強制型変換も考慮しない．

3.3 サブセット言語の構文要素

以上より，サブセット言語は，表 2 に示す式に関する 12 種の構文要素，表 3 に示す文に関する 6 種の構文要素，さらに関数定義とプログラムの計 20 種の構文要素からなるものとなる．

前の論文 [2] から拡張した点は，関数の引数および局所変数がそれぞれ一つという制限をなくし任意個の引数，局所変数を持てるようになったこと，式だけでなく文を扱うこと，プログラムが関数定義だけでなく文も含むこと，である．

```
function f1(x) { var y;
  y = 1; return this.i }
function c1(x) { var y;
  this.i = x; this.m = f1; y = 3; return y }
function g1(x) { var y;
  x = 2; y = new c1(4); return y.m(5) }
g1(3);
```

図 1　JavaScript サンプルプログラム

3.4 サンプルプログラム

3.1，3.2 節のサブセット言語の条件を満たす JavaScript プログラムの例を図 1 に示す．3 つの関数宣言 f1,c1,g1 があり，それぞれメンバ関数，コンストラクタ，大域関数として使用されている．また，大域関数 g1 が定数 3 を引数に渡されて呼び出されている．

4 サブセット言語に対する型推論

本章では，サブセット言語に対する型推論系を形式的に定義する．論理式の表現形式は MGTP 構文であるので，まずその構文を説明する．定義に必要な項および述語の定義を行い，構文規則に対応させて導出するべき論理式について説明する．

4.1 MGTP

MGTP[5] は，モデル生成法に基づく一階述語論理の定理証明系である．MGTP の入力は

```
a1, a2, …, am -> b1; b2; …; bn.
```

という節形式の規則の集合である．これは「$a_1 \wedge a_2 \wedge \cdots \wedge a_m$ が成立するとき $b_1 \vee b_2 \vee \cdots \vee b_n$ 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，後件のいずれのリテラルも否定できる．述語の引数に関して，変数は大文字で，定数を含む関数記号は小文字で表記する．前件, 後件は 0 個でもよい．前件が 0 個のものを正節と呼び後件が恒真であることを，後件が 0 個のものを負節と呼び前件が成立したとき矛盾することを表す．略記法として，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．

表 1　述語と関数記号の定義

要素	意味
関数記号	
コンストラクタ関数名	オブジェクト型の構成子
ft(TO,TR,TA1,…)	関数型の構成子
this(FN), return(FN), arg1(FN), …, local1(FN), …	型割当対象である関数 FN の環境の要素
member(CO,M)	型割当対象であるオブジェクト型 CO のメンバ M
述語	
arity(FT,N)	関数型 FT のアリティは N
ta(O,T)	型割当対象 O に型 T を割当
typesafe(S)	文 S は型安全
objecttype(T)	型 T はオブジェクト型
hastype(O)	要素 O は型をもつ
eq(S,T)	型 S,T は同一

4.2　述語および項の定義

オブジェクト型の要素はコンストラクタ関数名である．これは，ある JavaScript プログラムを考えたとき要素が定まる有限の集合である．関数型は型の任意長のタプルであり，this の型が TO, 返り値の型が TR, 引数の型が順に TA1,TA2,…である関数型を ft(TO,TR,TA1,TA2,…) と表す．TO はオブジェクト型でなければならないが，他の型は関数型でもよいので，関数型の要素は無限に存在する．関数型はアリティをもつ．ある関数型 FT がアリティ N であることを述語 arity(FT,N) で表す．型を割り当てる対象は，部分式，環境の各項目，オブジェクト型の属性である．これらの型割り当て対象がある項 O で表わされているとき，O に対して型 T が割り当てられることを述語 ta(O,T) で表す．部分式を表す項は後で定義する．関数の環境は，this, 返り値, 引数, 局所変数からなる．名前が FN である関数の環境の項目を順に，this(FN), return(FN), arg1(FN), …, local1(FN), … という項で表す．コンストラクタ名 CO で生成されるオブジェクトの属性 M を member(CO,M) という項で表す．文 S が型安全であることを述語 typesafe(S) で表す．以上を表 1 にまとめる．

4.3　型規則

次に式，文，関数定義，プログラムに対して型を導出する推論規則について述べる．

4.3.1　式

式の型は，その部分式の型に基づいて定まる．例えば，grouping operator が出現しているとき，すなわち括弧でくくられた部分式が出現しているとき，その型は括弧内の式の型と同じになる．部分式のない式は，環境やオブジェクトの属性型などからその型が定まる．例えば，関数 f の環境において this の型が T であるならば，その関数内部の this の出現の型は T である．この原理に基づき，式の型を導く論理式は表 2 のようになる．型推論規則に相当する論理式は，出現 o に対する型割当 ta(o,T) を結論として導く形をしている．

型検査規則を必要とする構文も存在する．例えば，addition に関して，この式が Number 型を持つと結論する前に，その部分式がともに Number 型であることを確認しなければならない．部分式 exp1 に Number でない型が割り当てられたとき, 型検査規則が矛盾を導く．型検査規則なしでは，ta(o,Number) は導かれないだけで矛盾が起きず，プログラムに型の誤りがあるにもかかわらず，論理式を満たすモデルを許す．型検査規則の詳細に関しては，表 2 の脚注に示した．

4.3.2　文

文に対する論理式は，その出現が型安全であることを推論する．複合文は，部分文全てが型安全であるときに型安全である．例えば，二つの文 stm1,stm2 からなる block statement は，stm1,stm2 がともに型安全であるとき型安全である．式を含む文は，式に正しく型が推論されるとき型安全である．例えば expression statement は，その式に型が割り当てられることが表 2 の論理式で推論されるとき型安全である．この原理に基づき，文の型安全を推論する論理式は表 3 のようになる．

4.3.3　関数定義とプログラム

関数定義が与えられたとき，この関数のアリティおよび局所変数の数が定まる。それら全てに型が割り当てられ，矛盾なく環境が定まることが必要である．関数名を f, アリティを n, 局所変数の数を m とすると，以下の論理式を生成する．

表 2　式の型推論規則

ECMA 名称	JavaScript 擬似コード	導出する推論規則
this	this	`ta(this(f),T) -> ta(o,T).`
identifier reference	k 番目の局所変数 k 番目の仮引数	`ta(local`k`(f),T) -> ta(o,T).` `ta(arg`k`(f),T) -> ta(o,T).`
literal reference	数値表現	`-> ta(o,Number).`
grouping operator	(exp1)	`ta(o1,T) -> ta(o,T).`

property accessors “.”	obj.prop	`ta(o0,T)`
		` -> objecttype(T), hastype(member(T,prop)).` (1)
		`ta(o0,T), ta(member(T,prop),S) -> ta(o,S).`

new operator	new con(exp1,exp2,⋯,exp$k$)	`ta(arg1(con),T1), ta(o1,S1) -> eq(T1,S1).` (2) (同様の検査規則を k 個導出) `ta(o1,S1), ta(o2,S2), ⋯ -> ta(o,con).`

function calls
及び argument lists
・メンバ関数呼び出し
	obj.mem(exp1,exp2,⋯,exp$k$)	`ta(o0,T)`
		` -> objecttype(T), hastype(member(T,mem)).` (3)
		`ta(o0,T),ta(member(T,mem),S) -> arity(S,`k`).` (4)
		`ta(o0,T),ta(member(T,mem),ft(TO,TR,TA1,⋯,TA`k`))`
		` -> eq(T,TO).` (5)
		`ta(o0,T),ta(member(T,mem),ft(TO,TR,TA1,⋯,TA`k`)),`
		` ta(o1,T1) -> eq(TA1,T1).` (2)
		(同様の検査規則を k 個導出)
		`ta(o0,T),ta(member(T,mem),ft(TO,TR,TA1,⋯,TA`k`)),`
		` ta(o1,T1),⋯ -> ta(o,TR).`

・大域関数呼び出し	glo(exp1,exp2,⋯,exp$k$)	`ta(arg1(glo),TA1), ta(o1,T1) -> eq(TA1,T1).` (2)
		(引数の数だけ検査規則を導出)
		`ta(return(glo,TR)),ta(o1,S1), ta(o2,S2), ⋯`
		` -> ta(o,TR).`

function expressions	fun	`ta(this(fun),TO), ta(return(fun),TR),` ` ta(arg1(fun),TA1), ⋯ -> ta(o,ft(TO,TR,TA1⋯)).`
void	void exp1	`ta(o1,T) -> ta(o,undefined).`

addition	exp1 + exp2	`ta(o1,T), ta(o2,S)`
		` -> eq(T,Number), eq(S,Number).` (6)
		`ta(o1,Number), ta(o2,Number) -> ta(o,Number).`

assignment	v = exp1	`ta(arg`k`(f),T), ta(o1,S) -> eq(T,S), ta(o,T).`
		(v が k 番目の引数のとき)
		`ta(local`k`(f),T), ta(o1,S) -> eq(T,S), ta(o,T).`
		(v が k 番目の局所変数のとき)

- o：注目している式の出現を表す項
- f：出現 o が含まれている関数
- o0, o1, o2：部分式 obj, exp1, exp2 の出現を表す項

(1) obj がオブジェクト型であること，その型がメンバ prop を持つことを検査
(2) 実引数の型と仮引数の型が同一であることを検査
(3) obj がオブジェクト型であること，その型がメンバ mem を持つことを検査
(4) obj.mem がアリティ k の関数型であることを検査
(5) メンバ関数の this の型が callee の型と同一であることを検査
(6) 左右の部分式がともに Number 型であること検査

表 3　文の型推論規則

ECMA 名称	JavaScript 疑似コード	導出する推論規則
block statement	{ stm1, stm2 }	`typesafe(s1), typesafe(s2) -> typesafe(s).`
variable decl	var i, j	(局所変数の番号を決定する)
empty	(空文)	`-> typesafe(s).`
expression	exp	`ta(o,T) -> typesafe(s).`
if statement	if (exp) stm1 else stm2	`ta(o,T),typesafe(s1),typesafe(s2) -> typesafe(s).`
return	return exp	`ta(return(f),T), ta(o,S) -> eq(T,S), typesafe(s).` (1)

- s：注目している文の出現を表す項
- s1, s2：文 stm1, stm2 の出現を表す項
- o：式 exp の出現を表す項
- f：出現 s が含まれている関数

(1) 環境の定義する関数の返り値と式 exp が同じ型であることを検査

```
-> hastype(arg1(f)), …, hastype(argn(f)).
-> hastype(local1(f)),
…, hastype(localm(f)).
-> hastype(this(f)).
ta(this(f),T) -> objecttype(T).
-> hastype(return(f)).
```

ただし，大域関数については this には値が入らないので，f が大域関数のときには 3,4 行目は `-> ta(this(f),undefined).` となる．

プログラムは関数定義と文の集まりである．それら全てに対して上記の論理式を導出する．これら全てが矛盾なく成立するとき，プログラムは型が正しいといえる．また，論理式を満たすモデルは，プログラムに対する型割当を与える．

4.4　型検査系の健全性

以上により与えた型検査系は文献 [1, 2] の拡張であり，同様の健全性を持つ．すなわち，型安全であると結論されたプログラムの実行は，オブジェクトの存在しない属性へのアクセスを起こさない，ただし，オブジェクトが null であるときを除く．

二種類のエラーについて考える必要がある．一つは，プログラムが型安全と判定されたにもかかわらず，実行時例外が発生する場合である．これは，健全性の定義の例外事項，オブジェクトが null である場合が相当する．端的には，BabyJ^T 型体系が注目する事柄は，オブジェクトへの属性アクセスが存在する属性か否かのみであるため，その属性が初期化されていない場合の実行時例外を検出できない．この問題は BabyJ^T の本質的な限界であり，動的な検査を必要とする．

もう一つは，実行に問題のないプログラムが型安全でないと判定される場合である．これは例えば，一つの変数へプログラムの箇所によって異なる型のオブジェクトを格納しているときに発生する．型推論系を拡張して，変数の生存判定を行い，このようなプログラムも型安全と判断できるようにすることもできよう．また，BabyJ^T の意味でこのようなプログラムはよくないスタイルであると考えることもできる．

4.5　型推論の駆動

モデル生成に基づく推論では，ドメインを有限にし，またその要素が列挙できなければならない．ここで扱う論理式に含まれる変数のドメインはプログラムの型である．そこで，型の要素が t1,t2,…,tn と列挙されたと仮定する．ある型割当対象 o に型が割り当てられるという述語 hastype(o) は，以下のように定義される．

```
hastype(o) -> ta(o,t1);ta(o,t2);
…;ta(o,tn).
ta(o,t1) -> -ta(o,t2), …,-ta(o,tn).
 :
ta(o,tn) -> -ta(o,t1), …,-ta(o,t(n-1)).
```

1 行目は，後件の ta のいずれかは成り立つことを宣言する．2 行目以降により，o にある型 tk が割り当てられている (ta(o,tk) が真) ならば，それ以外の型は割り当てられない ($i \neq k$ ならば ta(o,ti) は偽) ことを表している．1 行目の論理式だけでは，一つの o に

複数の型を割り当てた場合も列挙してしまう．この論理式に対するモデルの探索は，o に対してあらゆる型割当の可能性を順に確認する動作になる．

4.3.3 節の定義により，関数宣言に対してその環境における型割当が必要であることを表す論理式が導かれている．定理証明系は，述語 hastype の定義により，関数の環境に対する全ての型割当に対して，文や式に対する型推論規則に矛盾しないモデルを探索する．この述語 hastype に対する探索が型推論全体を進行させる．型の誤りが含まれる場合は型検査規則により矛盾が導かれるので，全ての論理式を満たすモデルとは，プログラムに対する正しい型割当によるプログラムの型が正しいことの証明である．

5 JSX-model

前章で定めた論理式を導出するには，構文に対するパターンマッチを行う必要がある．

一つの方法として，JavaScript プログラムを構文解析してパターンマッチを行う手続き的プログラムを作成する方法がある．この方法は，形式的定義とプログラムによる表現の間に距離があり，正当性の確認作業や定義の変更をプログラムに反映させる作業が困難であるなどの問題がある．

もう一つの方法として，パターンマッチによる導出を直接記述し実行できる枠組みを使うことが考えられる．これは形式的定義を実行可能な形で表現してこれを実行させるため，手続き的プログラムによる方法の問題点から開放され，適応性の高い方法といえる．

XSLT は，XML に対するパターンマッチによる変換を記述する枠組みである．JavaScript プログラムの XML 表現には，CASE ツール・プラットフォーム Sapid[6, 7] の提供する JSX-model がある．JSX-model は JavaScript プログラムに対して構文および意味に基づくマークアップを行った XML 文書である．マークアップによる情報の追加は，CASE ツールの作成を支援することを目的にしており，本研究で必要なパターンマッチを XSLT で記述する対象として妥当な XML となっている．

5.1 XML 定義

JSX-model では ECMAScript 3rd Edition[4] の文法規則によるプログラムを 7 種類の終端要素と 6 種類の非終端要素によってモデル化している．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．表 4 に XML 要素とその意味を示す．非終端要素は他の要素だけを子として持ち，テキスト要素を持たない．終端要素は子に一つのテキスト要素のみを持つ．

JavaScript プログラムから JSX-model への変換器は Sapid のツール wapid.parser.js.MakeJSX として提供されている．これは 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．構文解析に基づくタグ付けの他に，要素毎にそれぞれを識別するための id が割り振られ，意味解析結果がその id を利用して付加される．

5.2 id 属性の扱い

JSX-model の各要素には一意の id 属性が付加される．この id により JavaScript プログラム中の任意の要素を特定することができる．また，解析器はシンボルの照応関係を特定してそれを次のように保存する．すなわち，変数などの参照を表す識別子の出現に対する `ident` 要素の `refid` 属性に，その識別子の宣言部の `ident` 要素の id 値が設定される．具体的には，スコープを理解して変数参照に正しい宣言部へのリンクがつけられる．プロパティ名，メソッド名の照応関係は動的に束縛されるものなので静的には解決できない．関数名には名前付きの関数宣言へのリンクがつけられる．本研究では，変数の定義と参照の対応付け，関数の宣言と呼び出しの対応付けを，等しい型をもつべき要素を発見するために使用する．

照応関係について例で説明する．図 1 のサンプルプログラムを JSX-model に変換すると，1 行目の変数宣言にある y は

```
<ident id="8">y</ident>
```

のように，2 行目の代入式にある y は

```
<ident id="12" refid="8">y<ident>
```

のようにタグ付けされる．後者の `refid="8"`は，その実体が `id="8"`である前者であることを示す．関数 c1 の中にある変数 y は `id="25"`となり，関数 f1 や g1 の y とは異なることが記録される．

表 4　JSX-model の XML 要素

非終端要素

タグ	意味
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

タグ	意味
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

Stmt 要素の sort 属性がとる値

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

Expr 要素の sort 属性がとる値

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**

6　JSX-model からの MGTP 論理式への変換

JSX-model の XML 表現にマッチさせて，JavaScript プログラムの型推論規則を導出する具体的な方法について述べる．

6.1　型空間

4.5 節でも述べたように，モデル生成法に基づく推論では，ドメインを有限にし，その要素が列挙できなければならない．本提案手法における型のドメインは，オブジェクト型と関数型に分けられる．オブジェクト型の要素はコンストラクタ関数名であり，JavaScript プログラムを決めたとき，それらは有限であり列挙できる．一方，関数型の要素は無限に存在する．JavaScript では高階関数を扱えるのでこれを表現できるように関数型を帰納的に定義したためである．しかし，JavaScript という言語の用いられる場面から考えて，現実的な JavaScript プログラムでは関数型の入れ子は数段で十分対応できると仮定する．この仮定に基づき，関数型の入れ子の深さの制限段数をパラメータとして，関数型の集合を有限要素数の集合で近似して扱うことにする．

6.2　型割当対象要素

JavaScript プログラムの型を推論する論理式を生成するために，型を割り当てる対象を表す項を厳密に定義する必要がある．型割当対象要素とは 2.2 節で定義した式，変数，オブジェクトの属性，関数の環境，である．これらのうち，オブジェクトの属性と関数の環境については 4.2 節でその項表現を定義した．本節では，式および変数の項表現を定義する．また，型割当対象要素ではないが，述語 typesafe の引数である文の項表現も定義する．

6.2.1　式と文

JSX-model では文は Stmt 要素であり，全ての Stmt 要素には一意の id が割り振られている．これを文を表す項表現として用いる．同様に，式は Expr 要素であり，全ての Expr 要素には一意の id が割り振られている．これを式を表す項表現として用いる．

実際に MGTP 論理式に変換する際には，接頭辞として id をつけることで，整数値と解釈されないようにした．

6.2.2　変数

4.3 節，表 2，表 3 では変数は定義順に番号を振った項 arg$i$(f),local$j$(f) により表現するとした．しかし，XSLT では変数参照からこの番号を導くことが難しいため，5.2 節で述べた id を用いて宣言と実体の対応付けを行い，関数呼び出しの仮引数と実引数の対応付けを別途行うように導出する論理式に変更を加える．

変数および仮引数の参照の型は，その宣言への型割当から求める．表 2 では identifier reference および assiginment の型を項 arg$i$(f),local$j$(f) に割り当てられた型から導いていたが，これを表 5 のように属性 refid でリンクされた宣言部に割り当てられた型から導く形に変更する．

関数の環境は arg1(f),…,local1(f),… といった項に型を割り当てる．これらの項と表 5 の x とを関連づける必要がある．そのため関数定義に対して，4.3.3 の論理式に加え以下のような対応付け

表 5　式の型推論規則修正

ECMA 名称	JavaScript 疑似コード	導出する推論規則
identifier reference	v	`ta(x,T) -> ta(o,T).`
assignment	v = exp1	`ta(x,T), ta(o1,S) -> eq(T,S), ta(o,T).`

- 変数の識別子 v の `ident` タグの `rid` は x

の論理式も導出する．i 番目の仮引数宣言の id が x であるとき，`ta(arg`i`(f),T) -> ta(x,T).` とする．j 番目の局所変数宣言の id が y であるとき，`ta(local`j`(f),T) -> ta(y,T).` とする．

変数参照の出現 o に対して導出される論理式は，その変数が j 番目のとき

```
ta(localj(f),T) -> ta(o,T).
```

であった．本節の修正後に導出される論理式は，変数の refid が k であるとき

```
ta(idk,T) -> ta(o,T).
ta(localj(f),T) -> ta(idk,T).
```

となる．JSX-model の与える照応関係により，等しい結果を導く論理式が導出できている．

6.3　XSLT の構成

以上の定義に従い，JSX-model から，型推論を行う MGTP 論理式を導出する XSLT を構成する．これは，4.3 節で定義し前節で修正した構文から論理式を導出する規則を記述したものとなる．

XSLT の変換規則は，JavaScript 構文とマッチさせるための XPath 表現と，導出するべき論理式に含まれるパラメータを得るための XPath 表現から構成される．

単純な例として grouping operator を採り上げる．JSX-model では，括弧式は Expr 要素でありその sort 属性が Paren となる．XPath ではこのような全てのノードは “`//Expr[@sort='Paren']`” と表せる．導出するべき論理式には，このノードの id が必要である．これを表す XPath は “`@id`” である．括弧式の Expr 要素は括弧内の部分式を表す Expr 要素 1 つを唯一の子要素として持つ．この子要素の id も導出するべき論理式に必要で，XPath では “`Expr/@id`” と表せる．以上より，grouping operator に関する論理式を導出する XSLT は次のようになる:

```
<xsl:for-each select="//Expr[@sort='Paren']">
   ta(id<xsl:value-of select="Expr/@id"/>,T)
      -&gt;
   ta(id<xsl:value-of select="@id"/>,T).
</xsl:for-each>
```

JSX-model のタグ付けの方針と，BabyJ^T 型体系の要求する意味単位のずれにより，XPath 表現に工夫が必要な導出規則がある．Property accessors は obj.prop という形の部分式であるが，この形だけで JSX-model にマッチさせるとメンバ関数呼び出しの obj.mem(exp1,exp2,…) の obj.mem の部分にもマッチしてしまい，意図しない論理式が導出されてしまう．このようなマッチの重なりがある場合には XPath の述部の表現を工夫する対処が必要になった．上の例では，メンバ関数呼び出しではない property accessors は，この式が最上位であるためにさらに親の要素は Expr ではないか，あるいは関数呼び出しでない式の部分式であるという条件を追加することで正しく指定できる．この条件を盛り込んだ，property accessors にマッチする XPath は

```
//*[name() != 'Expr' or @sort != 'FunCall']
/Expr[@sort='VarRef'][name(./*[1])='Expr']
[./*[2] = '.'][name(./*[3])='ident']
```

となる．同様の対処が identifier，new operator，member call，global call についても必要である．

全ての導出規則を XSLT で記述した結果，122 行の XSLT が得られた．ただし，XPath 表現が長大になる場合もあり，600 文字を越える行もあった．

7　実装と評価

JavaScript プログラムに対して型検査を行う実際の作業は以下のようになる．

1. JavaScript プログラムを変換器 wapid.parser.js.MakeJSX により JSX-model に変換する．

2. JSX-model に前章で構成した XSLT を適用して MGTP 論理式を導出する．
3. プログラム中のコンストラクタ関数と関数のアリティから，型ドメインを定義する．このとき関数型の入れ子の段数を指定する．
4. JavaMGTP を用いて，導出した MGTP 論理式の証明を実行する．

上の手順 (1),(2),(3) の実行時間はごくわずかである．手順 (4) の証明のためのモデル探索は，正解を発見するまで全ての組み合わせを試みるため大きな計算量が要求される．

図 1 のプログラムに対して，正節 480，負節 2，混合節 94 からなる論理式が導出された．これは型ドメインの宣言のための論理式を含む数である．関数型ドメインの入れ子は 1 段とした[*4]．この論理式を著者らの環境 (SunBlade 2000) で JavaMGTP による証明を実行した結果，約 157 秒の推論時間でモデルを発見した．このモデルには関数 f1,c1,g1 の環境，オブジェクト型 c1 のプロパティ i および m，全ての式への正しい型割当が含まれ，全ての文が typesafe であると判定された．得られたモデルの一部を図 2 に示す．関数 g1 のみ this の型が undefined，局所変数 y の型が c1 であり，オブジェクト型 c1 のプロパティ m には代入されるメンバ関数 f1 の型 ft(c1,number,number) が割り当てられている．JSX-model で割り振られている id で，16 は関数 f1 の this を，82 は最終行の関数 g1 の呼び出しをする文 (expression statement) を示す．式には正しい型が割り当てられ，文は typesafe であるとわかる．

```
ta(this(c1),c1)
ta(this(f1),c1)
ta(this(g1),undefined)

ta(local1(c1),number)
ta(local1(f1),number)
ta(local1(g1),c1)

ta(member(c1,i),number)
ta(member(c1,m),ft(c1,number,number))

ta(id16,c1)
typesafe(id82)
```

図 2　得られたモデル (一部)

8　検討

8.1　サブセット言語の妥当性

本論文で示した型検査系は 3 章で述べたとおり JavaScript 言語のサブセットである．文献 [4] で定義されている構文要素 65 種のうち，本論文で扱っていないものについて検討する．以下，文献 [4] の節番号を § 記号で示す．

組み込みデータタイプは Number のみを扱い、String と Boolean を扱っていない．String に対応するには Number と混合して演算を行った際の自動型変換に対応する必要がある．これは §11.6.1 The addition operator の扱いを複雑にする．これは煩雑ではあるが困難ではない．

JavaScript の配列には異なる型の要素を格納できるため，静的な型検査は困難である．このため，配列に関する構文 §11.1.4 Array reference は除外した．

オブジェクトの生成はコンストラクタ関数を用いたものに限定している．§11.1.5 Object initialiser でのオブジェクトの生成は，特定のコンストラクタと結びつけることができないため，オブジェクト型をそのコンストラクタで分類する BabyJ^T 型体系では扱うことができない．この構文については，object initialiser の出現をコンストラクタ関数と同じレベルのオブジェクト型の要素とみなすという方針で型体系を拡張するれば，扱いが可能になると考えられる．

演算子については，§11.6.1 Addition operator のみ扱った．他の演算子の扱いは，これと同様にすればよい．

§11.4.1 delete operator は，オブジェクトから属性を削除できる．BabyJ^T 型体系は，オブジェクト型の属性が動的に変化する計算に対応していないので，delete 演算子に関する型検査はしていない．すなわち，オブジェクトの属性を delete 演算子で削除した後にこの属性にアクセスすると実行時エラーになるが，そのようなプログラムに対して提案した型検査系

[*4] すなわち，一つの型に関数型構成子 ft は 1 度しか出現せず，ft の引数は基本型とオブジェクト型のみである．

を delete 演算子を無視して適用すると型安全であると判定してしまう．

§11.2.1 Property accessor には 2 つの記法があるが，それらのうちアクセスする属性の名前が静的に決定できるドットを用いる記法にしか対応していない．もう一方の属性名が動的に決定される構文には，静的な型検査では対応できない．

制御構文については if 文についてのみ扱いを示した．他の制御構文についても同様に考えることができる．

JavaScript の例外処理は，throw 文で例外を発生させ catch 文で捕捉する．このときに値を受け渡すことができる．例外は動的な呼び出し関係で捕捉する場所が決まる．したがって，メソッド呼び出しにより関数呼び出し関係が動的に構成されることが一般的な JavaScript では，例外の発生と捕捉の位置を静的に対応づけることは困難である．そのため，例外を通じて渡される値の型を静的に関連づけることができないので，例外処理構文は対応から除外した．

以上のように，動的な型検査が必要なものについては対応から除外したが，静的な型検査で対応できる構文の扱い方については，方針を十分に明らかにできたと考えられる．

現実の JavaScript プログラムへの適用の際には，配列の対応がまず求められると予想される．配列の要素が全て BabyJ^T の意味で同じ型であることを仮定する型体系ならば，提案手法の延長で実現が可能であろう．要素の型が異なるが，それらが共通に持つ属性にのみアクセスするようなプログラムを型安全と判定するには，BabyJ^T の複数の型を多相型として扱える，より強力な型体系が必要になる．

8.2 BabyJ^T の制限について

著者らは文献 [2] において，BabyJ^T 型体系に基づく型検査手法に 4 つの問題があることを述べた．それらは (1) 関数の引数，局所変数がそれぞれ一つに限られていること，(2) 値としての関数はオブジェクトでないこと，(3) オブジェクトからメンバの削除ができないこと，(4) eval がないこと，である．

(1) については，推論規則を導出する際に CASE ツールプラットフォーム Sapid を利用し，導出規則の構成にパラメータを導入できたこと，そして変数の照応関係を id を用いることによって，任意個の引数，局所変数を許す型検査系に拡張でき解決できた．

(2) については，オブジェクト型と関数型の大別は BabyJ^T 型体系のアイデアの根幹であるので，考え方をそのままにした拡張によりこの問題を解決するのは難しい．BabyJ^T 型体系より強力な型体系に基づく型検査系を構築する必要がある．そのような型検査系も，4.3 節のように，構文に基づいた論理式の導出を形式的に定義することで，本提案手法の枠組みにしたがって形式的に実装を与えることができると考えられる．

(3) については前節で述べたように対応できていない．BabyJ^T を提案した Anderson らは，この問題を解決することを目的とした，より強力な型推論の枠組みを文献 [8] で提案している．ここでは，属性が確実に存在するかどうかを追跡する情報を型に付加することで，delete によって削除された属性へのアクセスを検出できる．この型体系も形式的に定義されているので，本論文の枠組みで直接実行することが考えられる．

(4) について，動的に型が変化する eval 式については，静的な型検査に基礎をおく本論文の枠組みで扱うことはできない．

8.3 型割当の応用

本提案手法による型検査は，型が正しいプログラムに対して詳細なレベルでの型割当を同時に与える．この型割当をさらに高度な CASE に応用することが考えられる．型検査系を JSX-model に基づいて構築したことにより，プログラムに対する型割当は JSX-model の id に対応づけられる．例えば，図 2 の下から 2 行めでは，sapid id が 16 である式の出現に対してオブジェクト型 c1 が割り当てられることが示されている．Sapid に基づくプログラムブラウザの表示に，この詳細な型割当の情報を重ねて提示すれば，プログラム理解の支援になる．その他の応用についても，JSX-model を利用する CASE ツールからは，id を介して型割当を容易に参照できる．

8.4 推論時間

本提案手法では，特に型検査規則で二つの型が等しいかどうかを確認するため，型の間の等価性を述語 eq で定義した．この型の間の等価性は，単に型を表す項の同一性である．項の同一性を判断する組み込み述語を追加して JavaMGTP を改良することで，

証明に必要な時間を短縮できると考えられる．

文献 [2] に示した手法では，ある構文に対する型推論規則は一つしか存在せず，その構文の異なる出現は全て一つの推論規則で扱われる．本提案手法では，ある構文の出現それぞれに対して型推論規則を導出するため，論理式の数が多くなった．これが推論時間の増加を招いていると考えられる．導出する論理式の形を工夫して論理式の総数を減らすことで，証明にかかる時間を短縮できる可能性がある．

8.5 型の多相性

BabyJ^T 型体系は多相性を持たないため，型多相性を用いるようなプログラムに対しては，その実行が安全であっても本論文の型検査系は型エラーとみなす．今後，より強力な型体系に基づく型検査系を，本論文で提案した枠組に基づいて構築することが必要である．

9 むすび

本論文では，JavaScript プログラムに対する静的な型検査系を構築した．オブジェクトと関数からなる単純な型体系 BabyJ^T を，JavaScript の主要な構文を扱えるように拡張した．型検査系の形式的な定義に基づき，型検査のための論理式は，構文に対応して抽出される．抽出された論理式は，定理自動証明系により充足可能性を判断される．提案手法を形式的に実装し，実際に JavaScript プログラムの型検査が行えることを示した．すなわち，JavaScript 構文から論理式を導出する段階には，Sapid の XML 形式と XSLT 技術を用いた．定理自動証明系には MGTP を用いた．論理式の抽出が正常に行え，また正しく型検査が機械実行できることを示した．

JavaScript には，型の他にも検査することが望まれる性質がある．これらを検査するシステムも，構文に基づいて形式的な推論規則で検査系を記述できる問題については，本論文の枠組の上に実現できると考える．また，提案した型検査系が正しいプログラムに与える型割当は，より高度な CASE ツールの入力として利用できると考える．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (A)(16200001) の補助による．

参考文献

[1] Anderson C. and Drossopoulou S. BabyJ: from object based to class based programming via types. In *WOOD2003 Workshop on Object Oriented Developments*, April 2003. Warsaw, Poland.

[2] 大久保 弘崇, 山本 晋一郎, 坂部 俊樹, and 稲垣 康善. モデル生成法に基づく JavaScript プログラム型検査の機械実行. **電子情報通信学会誌**, J89-D(No.4):693–704, 2006.

[3] 長谷川 隆三 and 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**, 41(6):1791–1798, June 2000.

[4] Standard ECMA-262. *ECMAScript Language Specification*, 1999.

[5] 長谷川 隆三 and 藤田 博. MGTP:並列論理型言語 KL1 によるモデル生成型定理証明系. **情報処理学会論文誌**, 37(1):1–12, January 1996.

[6] 福安 直樹, 山本 晋一郎, and 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**, 39(6):1990–1998, June 1998.

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

[8] Anderson C., Giannini P., and Drossopoulou S. Towards Type Inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, LNCS 3586, pages 428–453, 2005.