論　文

モデル生成法に基づくJavaScriptプログラム型検査の機械実行

大久保弘崇[†a)]　山本晋一郎[†b)]　坂部　俊樹[††c)]　稲垣　康善[†d)]

Machine Execution of Type Inference for JavaScript based on Model-Generation Theorem Prover

Hirotaka OHKUBO[†a)], Shinichiro YAMAMOTO[†b)], Toshiki SAKABE[††c)], and Yasuyoshi INAGAKI[†d)]

あらまし　プログラムの実行の安全性を保証するために，プログラムに対する型検査がしばしば行われる．本論文では，型のないオブジェクト指向言語である JavaScript で書かれたプログラムの型検査について述べる．本論文の型検査は，型規則を形式的な推論規則で記述し，これを自動証明系で直接実行するアプローチを特徴とする．この方法により，型検査系の実装時の誤りがなく正確な型検査が行え，また型検査系の構築・拡張が形式的な枠組みに基づいて行える．自動証明系には MGTP を用いた．MGTP はモデル生成法に基づく証明系で，推論規則の記述が直観的に行える．本論文では，推論規則によって定式化した型検査系を MGTP により自動証明可能な形式で表現する方法を示す．この際，推論規則には現れない暗黙の条件である型整合性の検査規則を明示する必要があることを明らかにし，またそれを MGTP 規則で表現する際の手法について述べる．

キーワード　JavaScript, 型検査, 定理自動証明, MGTP

1. まえがき

最近の Web アプリケーションの広がりはめざましい．Web アプリケーションは，(1)XML や HTML などのコンテンツ記述，(2) レンダリングのための CSS や XSLT, (3) Java Server Pages や Servlet などのサーバーサイド・プログラム, (4) JavaScript や Applet などのクライアントサイド・プログラムなどから構成される大規模でヘテロジニアスな分散ソフトウェアである．信頼性が要求されないエンドユーザ向けの情報提供から始まった Web を基幹業務に使用する動きが広がりつつあるため，その信頼性を確保することは社会的課題である．サーバサイド・プログラムの信頼性向上のための研究が多い [1]〜[3] が，クライアントサイド・プログラムを対象とした信頼性向上に関する研究も始まっている [4], [5]．これらの論文では，XHTML 文書の一部がインラインの JavaScript プログラムで生成される動的コンテンツに対して，DTD に基づく構文検証を静的に行うシステムを提案し，JavaScript プログラムのデータフローを解析することによって，動的に出力される XHTML テキストの構文の正しさの検証を可能にしている．

本論文は，広く用いられているクライアントサイドのスクリプト言語 JavaScript を対象とした型検査系を提案する．JavaScript は型を持たない言語であるが，その実行時エラーには型を考えることによって検出できるものがある．静的な型検査を JavaScript プログラムに適用することにより，クライアントサイド・プログラムの信頼性を向上させることが目的である．また，本研究の特徴は，型推論規則を素直な形で定理証明系上に表現することにより，形式的な型体系が実際的な問題に対して直接に適用されることである．ここでいう素直な形とは，型推論規則を Prolog に代表される反駁方式の推論規則に変換するのではなく，モデル生

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

成方式の定理証明を用いることをいう．モデル生成方式を用いる理由として，基となる型推論規則と素直な対応が取れて見通しが良いこと，証明結果を利用者に提示するなどの応用が容易であることが挙げられる．

本論文では，型検査系として文献 [9] で提案されている BabyJ^T をとりあげる．BabyJ^T は，JavaScript 言語のサブセットである BabyJ に対する型システムである．定理証明系には MGTP [11] を用いる．推論規則によって形式的に定義された BabyJ^T の型検査系を MGTP により自動証明可能な形式で表現する方法について述べる．この際，推論規則には現れない暗黙の条件である型整合性の検査規則を明示する必要があることを明らかにし，またそれを MGTP 規則で表現する際の手法について述べる．

2. BabyJ, BabyJ^T

本章では BabyJ 言語ならびにその型検査系 BabyJ^T の構造を概説する．

2.1 BabyJ

言語 BabyJ は JavaScript のサブセットであり，機能を本質的な部分に限定している．文献 [9] ではその制限した文法の上でプログラムに対する型検査系の形式化，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．型体系に影響する JavaScirpt からの本質的な制限は，関数の引数が唯一 `x` のみであること，同じく関数本体の局所変数が唯一 `y` のみであること，値として扱われるときの関数がオブジェクトでないこと，オブジェクトからメンバの削除ができないこと，文字列の eval ができないこと，の 5 点である．

2.2 BabyJ^T

BabyJ プログラムに対する型体系が BabyJ^T である．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，関数は必ず区分に従った呼び出し方をされるとする．

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．これはクラスに基づくオブジェクト指向言語のクラスに対応すると理解できる．メンバ関数は，主にコンストラクタ関数によるオブジェクトの生成時にそのメンバに代入されることによってメンバ関数として機能するようになる．すなわち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 の型はオブジェクト型と関数型であり，いずれにも多相性はない．この型体系は BabyJ の動作的意味論に対し健全であることが証明されている．

P ∈ *Program*	::=	D *	
B ∈ *Body*	::=	`var y; e`	
D ∈ *FuncDecl*	::=	`function f (x) { B }`	
e ∈ *Exp*	::=	`this`	receiver
	|	`f`	function identifier
	|	`x`	parameter
	|	`y`	local variable
	|	`new f(e)`	object creation
	|	`null`	null
	|	`e;e`	sequence
	|	`e.m(e)`	member call
	|	`e.m`	member select
	|	`f(e)`	global call
	|	`v=e`	assignment
v ∈ *Var*	::=	`x` | `y` | `e.m`	
f ∈ *FuncID*			
m ∈ *MemberID*			

図 1　BabyJ の文法
Fig. 1　Syntax of BabyJ

2.3 BabyJ^T の型検査

BabyJ^T における型検査とは，次の二つの写像 $\mathcal{P}, \mathcal{D}$ を求めることである．$\mathcal{P}$ はプログラムを構成する関数 f に対して this の型 t_o, 引数 x の型 t_a, 局所変数 y の型 t_l, 返り値の型 t_r の 4 項組を割り当てる写像 $\mathcal{P}(f) = (t_o, t_a, t_l, t_r)$ ，$\mathcal{D}$ はオブジェクト型 t のメンバ m に対してその型 s を割り当てる写像 $\mathcal{D}(t, m) = s$ である．これらは，以下の条件 C を満たすものでなくてはならない．

条件 C: プログラムを構成する全ての関数 f について，$E \vdash e : t_r$ が図 2 に示す型規則により証明できる．ここで，e は f の関数本体，E は環境で，変数集合 $\{\mathtt{this}, \mathtt{x}, \mathtt{y}\}$ から型への写像であり，$\mathcal{P}(f) = (t_o, t_a, t_l, t_r)$ であるとき $E(\mathtt{this}) = t_o, E(\mathtt{x}) = t_a, E(\mathtt{y}) = t_l$ とする．図中の void 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．

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

(Null)
$$\frac{}{E \vdash \mathtt{null} : t}$$

(Seq)
$$\frac{E \vdash \mathtt{e}_1 : t' \quad E \vdash \mathtt{e}_2 : t}{E \vdash \mathtt{e}_1;\mathtt{e}_2 : t}$$

(New-cons)
$$\frac{\mathtt{f}\text{ is a constructor function} \quad \mathcal{P}(\mathtt{f}) = (\mathtt{f}, t_a, t_l, t_r) \quad E \vdash \mathtt{e} : t_a}{E \vdash \mathtt{new\ f(e)} : \mathtt{f}}$$

(Member-sel)
$$\frac{E \vdash \mathtt{e} : t' \quad \mathcal{D}(t', \mathtt{m}) = t}{E \vdash \mathtt{e.m} : t}$$

(Var-ass)
$$\frac{E \vdash \mathtt{e} : t \quad E(\mathtt{v}) = t}{E \vdash \mathtt{v} = \mathtt{e} : t}$$

(Member-ass)
$$\frac{E \vdash \mathtt{e1} : t' \quad E \vdash \mathtt{e2} : t \quad \mathcal{D}(t', \mathtt{m}) = t}{E \vdash \mathtt{e1.m} = \mathtt{e2} : t}$$

(Member-call)
$$\frac{E \vdash \mathtt{e1} : t_o \quad E \vdash \mathtt{e2} : t_a \quad \mathcal{D}(t_o, \mathtt{m}) = (t_o, t_a, t_r)}{E \vdash \mathtt{e1.m(e2)} : t_r}$$

(Global-call)
$$\frac{\mathtt{f}\text{ is a global function} \quad \mathcal{P}(\mathtt{f}) = (\mathit{void}, t_a, t_l, t_r) \quad E \vdash \mathtt{e} : t_a}{E \vdash \mathtt{f(e)} : t_r}$$

(Member-func)
$$\frac{\mathtt{f}\text{ is a member function} \quad \mathcal{P}(\mathtt{f}) = (t_o, t_a, t_l, t_r)}{E \vdash \mathtt{f} : (t_o, t_a, t_r)}$$

図 2　BabyJ^T の型規則
Fig. 2　Typing rules of BabyJ^T

3. モデル生成法による型推論

3.1 モデル生成法

前章で述べたように，BabyJ^T の型検査には，推論規則を用いる一つの証明により判定することではなく，条件 C を満たすような写像 $\mathcal{P}$ を何らかの方法で発見することが求められる．このような計算に適したシステムとして，モデル生成法に基づく定理証明系が挙げられる．モデル生成法は，与えられた推論規則を満たすモデルの候補を，それらの規則を利用して生成しつつ検証し，モデルを探索することで証明を行う証明法である．

BabyJ^T 型検査にモデル生成法を適用することを考える．条件 C とこれが参照する図 2 の型規則は推論規則である．これに加えて写像 $\mathcal{P}$ が満たすべき性質を推論規則により記述してモデル生成法による証明を行う．$\mathcal{P}$ のとりうる値をモデルの候補として順次生成し，それが条件 C を満たしているか否かを検証し，証明が成功したときはモデルとして正しい型割当を得ることができる．可能な全てのモデル候補がいずれも推論規則を充足するものでない場合，すなわち証明の失敗はプログラムに型の誤りがあることを表す．このように，モデル生成法は型推論系の自動実行に適合性が高い証明法である．

モデル生成法は一種のタブロー法であり，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うことができ，特に否定を含む推論を効率よく実行できることに大きな特徴がある．自動証明系の実装に関しても，タブローに基づく方法が近年再び注目されるようになってきており，その性能は向上してきている．

3.2 MGTP

MGTP [11] 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．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．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．

4. 素朴な変換

BabyJ^T の型規則を MGTP の表記に書き換え，その機械実行を試みる．

4.1 述語の定義

MGTP による表記に必要な述語および関数記号を定義する．BabyJ 構文の *Exp* を表現するための関数記号を表 1 のように定める．部分式をもつ式は，部分式を再帰的に変換して MGTP 項を得る．例えば，BabyJ の式

```
x=y; this.prop
```

を表現する MGTP 項は

```
seq(ass(x,y),sel(this,prop))
```

である．

写像 $\mathcal{P}, \mathcal{D}$ は，一つ一つの対応関係を意味する述語それぞれ pee, dee を用いて表すこととする．

```
   pee(f,this  ,to)  ∧  pee(f,arg   ,ta)
∧  pee(f,local ,tl)  ∧  pee(f,return,tr)
```

表 1　BabyJ 式と MGTP 関数記号の対応
Table 1　Correspondence of BabyJ expressions with MGTP constructor symbols

	BabyJ 式	MGTP 項
レシーバ	this	this
関数	f	f
引数参照	x	x
局所変数参照	y	y
オブジェクト生成	new f(e)	new(f,e)
空	null	null
逐次実行	e1;e2	seq(e1,e2)
メンバ関数呼び出し	e1.m(e2)	mcal(e1,m,e2)
属性読み出し	e.m	sel(e,m)
大域関数呼び出し	f(e)	gcal(f, e)
代入	x=e	ass(x, e)
	y=e	ass(y, e)
属性代入	e1.m=e2	mass(e1,m,e2)

が真であることが

$$\mathcal{P}(f) = (to, ta, tl, tr)$$

であることを表し，同様に

```
dee(t,m,s)
```

が真であることが

$$\mathcal{D}(t, m) = s$$

であることを表すとする．

BabyJ^T の型検査が成功したとき，プログラムへの型割当は証明木の中に示されている．得られた型割当を取り出して利用するためには，型割当は式の出現の位置（以降では単に出現とよぶ）と対応づけて得られていると扱いやすい．MGTP による型検査が成功したとき，推論規則を満たすモデルが得られる．このモデル中に，関数の型およびオブジェクトの型は上で定義した述語のリテラルとして含まれる．プログラム中の部分式の型を取り出すためには，式への型割当を与える述語の引数を，部分式そのものではなくプログラム中の出現の位置 (以降では単に出現とよぶ) であると都合がよい．このことを考慮して,『環境 E の下で式 ex が型 t をもつ』という意味の $E \vdash ex : t$ は『式 ex が関数 f の出現 o に存在する』ことを示す述語 `e(ex,f,o)` と『出現 o の式が型 t をもつ』ことを表す述語 `ta(o,t)` の組により表現することとする．

以上の準備により，図 2 の規則を MGTP 化することができる．例えば，環境から変数 x の型を求める推論規則 (Var) は，述語 `e` で対応づけるべき変数参照を発見し，その出現が評価される環境を `pee` で取得し，結果として出現 `O` に `pee` の与える変数の型を `ta` により割り当てる，すなわち

```
e(x,F,O),pee(F,arg,X)->ta(O,X).
```

のように書ける．また，(Member-sel) は部分式の出現への型割当を `ta` で取得し，その型と参照しているメンバ `M` に対して $\mathcal{D}$ が割り当てる型を取り出して `O` の型とするので,

```
e(sel(E,M),F,O),ta(o1(O),T1),
dee(T1,M,T) -> ta(O,T).
```

となる．全ての規則に対応する MGTP 規則を表 2 に示す．ここで，`o1()`, `o2()` は出現からその部分式の出現を作る構成子である．`ft()` は関数型の構成子である．

4.2　写像 $\mathcal{P}, \mathcal{D}$ の表現

上で構成した MGTP 記述を動作させるには，写像 $\mathcal{P}, \mathcal{D}$ を表す述語 `pee`,`dee` が必要である．特に $\mathcal{P}$ が中心的な役割を担っている．3. 1 節で述べたように，写像としての性質の自然な記述からこれを満たすモデルを自動生成させる．

準備のため補助的な述語を定義する．BabyJ^T の関数はコンストラクタ，大域関数，メンバ関数のいずれかの区分に属する．これは構文解析により判定できる．関数がコンストラクタ，大域関数，メンバ関数であることを表す述語 `cf`, `gf`, `mf` を導入する．関数 c,g,m がそれぞれコンストラクタ，大域関数，メンバ関数であるとき，`cf(c)`, `gf(g)`, `mf(m)` が真であるとする．また述語 `func(f)` を任意の関数 f に対して真となるような述語とする．

BabyJ の動作的意味論および BabyJ^T 型規則から，コンストラクタ関数の this は必ずそのコンストラクタが生成するオブジェクトだけが代入される．すなわちコンストラクタ関数 c に対して `pee(c,this,c)` は真となる．これを MGTP で表現すると

```
cf(F) -> pee(F,this,F).
```

となる．同様に，大域関数の this には何も代入されない．この this の型を意味する BabyJ^T の特殊なオブジェクト型が void である．MGTP では

```
gf(F) -> pee(F,this,void).
```

と表現できる．メンバ関数の this には任意のオブジェクト型が割り当てられる．

ある関係 r が写像である条件は，定義域の各要素に対してそれと関係 r が成立する値域の要素がたかだか一つであることである．逆にいえば，ある定義域の要素に値域の異なる 2 つの要素がいずれも関係しているならば，その関係は写像ではない．これを論理式で表せば以下のようになる:

表 2　BabyJ^T 規則の MGTP 記述
Table 2　BabyJ^T typing rules in MGTP

規則名	MGTP 規則

(Var)	`e(this,F,O),pee(F,this,S)->ta(O,S).`
	`e(x,F,O),pee(F,arg,X)->ta(O,X).`
	`e(y,F,O),pee(F,local,Y)->ta(O,Y).`

(Null)	`e(null,F,O),type(T)->ta(O,T).`
(Seq)	`e(seq(E1,E2),F,O),ta(o1(O),T1),ta(o2(O),T2)->ta(O,T2).`
(New-cons)	`e(new(CO,E),F,O),pee(CO,arg,X),ta(o1(O),X)->ta(O,CO).`
(Member-sel)	`e(sel(E,M),F,O),ta(o1(O),T),dee(T,M,S)->ta(O,S).`
(Var-ass)	`e(ass(x,E),F,O),pee(F,arg,T),ta(o1(O),T)->ta(O,T).` `e(ass(y,E),F,O),pee(F,local,T),ta(o1(O),T)->ta(O,T).`
(Member-ass)	`e(mass(E1,M,E2),F,O),ta(o1(O),T1),ta(o2(O),T2),dee(T1,M,T2)->ta(O,T2).`
(Member-call)	`e(mcal(E1,M,E2),F,O),ta(o1(O),T1),ta(o2(O),T2),dee(T1,M,ft(T1,T2,R))->ta(O,R).`
(Global-call)	`e(gcal(G,E),F,O),pee(G,arg,T),ta(o1(O),T),pee(G,return,R)->ta(O,R).`
(Member-func)	`e(F1,F,O),pee(F1,this,S),pee(F1,arg,X),pee(F1,return,R)->ta(O,ft(S,X,R)).`

表 3　述語 pee,dee の定義
Table 3　Definition of predicates pee and dee

pee の唯一性	`->talr(this), talr(arg), talr(local), talr(return).`
	`func(F),talr(A),type(X),type(Y),-eq(X,Y) -> peetf(F,A,X,Y) ; peetf(F,A,Y,X) ; peeff(F,A,X,Y).`
	`peetf(F,A,X,Y) -> pee(F,A,X), -pee(F,A,Y).`
	`peeff(F,A,X,Y) -> -pee(F,A,X), -pee(F,A,Y).`

dee の唯一性	`cf(F),member(M),type(X),type(Y),-eq(X,Y) -> deetf(F,M,X,Y) ; deetf(F,M,Y,X) ; deeff(F,M,X,Y).`
	`deetf(F,M,X,Y) -> dee(F,M,X), -dee(F,M,Y).`
	`deeff(F,M,X,Y) -> -dee(F,M,X), -dee(F,M,Y).`

$$
\begin{aligned}
s \neq t \Rightarrow \quad & r(a,s) \quad \wedge \quad \neg r(a,t) \\
\vee \quad & \neg r(a,s) \quad \wedge \quad r(a,t) \\
\vee \quad & \neg r(a,s) \quad \wedge \quad \neg r(a,t)
\end{aligned}
$$

pee の全ての要素はこの条件を満たしていなければならない．ただし，コンストラクタ関数と大域変数の this に関しては上で述べたように一意に定められるので確認は不要である．この条件を MGTP で表現すると表 3 の規則になる．

写像 $\mathcal{D}$ についても同様にその性質を記述する．表 3 の member はメンバ名を表す述語である．これはプログラムから構文解析により判定できる．

4.3 条　件　C

以上の推論規則より，MGTP は $\mathcal{P}$ の候補を生成でき，その中から表 2 の BabyJ^T 型規則を満たすものを選び出せる．この型規則によって関数本体に割り当てられる型と，$\mathcal{P}$ の与える関数本体の型が等しいという条件 C を満たすことを確認できたものが，プログラムへの正しい型割当を含む推論規則のモデルである．これは MGTP では以下のように表現できる．

```
pee(F,return,T),ta(F,S) -> eq(T,S).
```

ここで，関数本体の出現は関数名で表すこととする．

4.4 プログラムの情報

以上で BabyJ^T 型検査の固定部は MGTP に書き換えられた．これを用いて特定の BabyJ プログラムの型検査を行うには，そのプログラムの情報を MGTP 化して与える．関数本体は表 1 によって MGTP 項に変換し，4.3 節で述べたように関数名を出現として“`->e(ex,f,f).`”の形の正節に変換する．このリテラルから全ての出現を生成するには図 3 の規則を用いる．この規則は部分式の出現を項の形に従って展開している．

```
e(seq(E1,E2),F,O) -> e(E1,F,o1(O)),e(E2,F,o2(O)).
e(new(CO,E),F,O) -> e(E,F,o1(O)).
e(sel(E,M),F,O) -> e(E,F,o1(O)).
e(ass(x,E),F,O) -> e(E,F,o1(O)).
e(ass(y,E),F,O) -> e(E,F,o1(O)).
e(mass(E1,M,E2),F,O) -> e(E1,F,o1(O)),e(E2,F,o2(O)).
e(mcal(E1,M,E2),F,O) -> e(E1,F,o1(O)),e(E2,F,o2(O)).
e(gcal(G,E),F,O) -> e(E,F,o1(O)).
```

図 3　全ての部分式の出現を得る推論規則
Fig. 3　Inference rules to extract all occurrence of subexpressions

関数の使用区分は，2.2 節で述べたように構文から自明に判定できる．この区分は 4.2 節で述べた述語で記述する．

4.5 型空間の宣言

モデル生成法は，値域限定性という制約がある．これは，述語の引数の変域が有限集合でなければならな

いという条件である．BabyJ^T 型検査では，型の空間がこの条件の影響を受ける．BabyJ^T の型はオブジェクト型と関数型である．オブジェクト型の要素は，プログラム中に存在するコンストラクタ関数なので，これは有限個である．一方，関数型は任意の型 2 つとオブジェクト型から帰納的に構成される，すなわち原理的に無限の要素が存在する．この無限に要素のある関数型を制限なしにモデル生成法で扱うことはできない．関数を引数にとる関数や結果に関数を返す関数といった高階関数がこれらの型をとる．JavaScript 言語の機能にはこれらの高階関数も仕様に含まれている．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充分対応できると予想される．

この予想に基づき，関数型の入れ子の深さの制限段数をパラメータとして，入力された BabyJ プログラムに対してモデル候補として用いる関数型の要素をあらかじめ生成し，これらだけからなる型の空間を用いて近似することとする．制限段数をパラメータ化することで，任意の有限の段数で関数型の要素をモデル候補の探索空間とすることができる．ただし，段数を深くすることは関数型の要素数を指数的に増加させ，結果として証明の計算量を増加させることになる．

全てのオブジェクト型 `c` に対して，それが型であることを `type(c)` と，そしてそれが関数型でないことを `-ftype(c)` と宣言する．全ての関数型 `ft(・,・,・)` に対して，型であることを `type(ft(・,・,・))` と，またそれが関数型であることを `ftype(ft(・,・,・))` と宣言する．全ての型の間に，等価性を判定する述語 `eq(t1,t2)` を定義する．

4.6 推論の流れ

以上で，BabyJ プログラムを入力として，そのBabyJ^T 型検査を MGTP により機械実行しプログラムの型安全性を検査する条件が記述できた．4.1 節で述べた BabyJ^T 型体系の実体である推論規則，4.2 節で定義した関数への型割当 $\mathcal{P}$ の条件とともに，型検査を行う対象 BabyJ プログラムの情報に関して，4.4 節の定義に従って関数本体の内容を，4.5 節の定義によって走査する型の空間を，それぞれ推論規則の形に変換し，これらをまとめたものに対して MGTP の証明の機械実行をする．プログラムの型が正しい場合は正しい型割当を含むモデルが得られる．

モデル生成法は，モデル候補となるリテラルの集合を，推論規則を用いて順次拡大することで推論を行う．最初は前件を持たない正節のリテラルをモデル候補に入れる．以降，モデル候補内に存在するリテラルとマッチする前件をもつ推論規則を探し，変数代入を行って得られる後件のリテラルをモデル候補に追加していく．モデル候補の拡張が完了したときが証明の完成である．モデル候補の拡張の過程で正リテラル `p(a)` と負リテラル`-p(a)` の両方がモデル候補に追加されたとき，これは矛盾を意味するのでそのモデル候補は棄却される．後件が or(;) で結合されている場合，その全ての可能性を別々に検証する．

本章で構築した MGTP 規則における証明の全体の流れは次のようである．はじめに図 3 の式の出現を表す述語 `e` に関する規則によりプログラム中の全ての式の出現がモデル候補に生成される．次いで表 3 の述語 `pee` に関する規則が適用され，多くの `pee` の正負リテラルがモデル候補に入れられる．この規則の or 結合に従って，MGTP はさまざまな $\mathcal{P}$ の可能性を数え上げて以下の規則を検証する動作を行う．述語 `e`,`pee` を前件に持つ表 2 の BabyJ^T 型規則が使われ，式の出現に対して述語 `ta` のリテラルが順次生成される．これは，現在検証中の $\mathcal{P}$ の下での，式の出現への型割当である．式への型割当は，他の部分式の型に依存しない抽象構文木の末端から根の方へ順に求められる．構文木の根である関数本体の式への型割当が得られた後，それが条件 C を満たすものか否かが 4.3 節の推論規則により確認される．$\mathcal{P}$ の示す関数の返り値の型 t と関数本体の型 s が異なるならば，それらを引数にもつリテラル `eq(t,s)` がこの規則でモデル候補に追加される．4.5 節で述べたように，異なる 2 つの型 a,b の間には `-eq(a,b)` をあらかじめ宣言しておくので，条件 C を満たさない型割当は矛盾をおこして棄却される．条件 C を満たすモデル候補が，推論規則のモデルとして提示される．

モデルが出力された場合，入力した BabyJ プログラムは型が正しいといえる．出力されるモデルには，プログラムへの型割当が含まれている．関数に対する型割当 $\mathcal{P}$ と，オブジェクトの属性への型割当 $\mathcal{D}$ についてはその意味は 4.1 節に定義した。式の出現に対する型割当は述語 `ta` によって表される．

5. 型不整合検査規則

5.1 不完全なモデル

前章で構成した MGTP 規則は，BabyJ^T 型体系に関して正しい BabyJ プログラムに対して，正しい型

割当をもつモデルを提示する．代入も参照もされない変数への型割当は制約がないため，どのような型を割り当てても矛盾しない．この場合全ての可能性が正しい型割当となり，それらに相当する複数のモデルが提示される．

一方、型の誤りを含む BabyJ プログラムに対しては，本来この MGTP 規則は全てのモデル候補を棄却して矛盾を示し，プログラム中に型の誤りがあることを示さなければならない．しかし前章で構成した推論規則は，不完全な型割当を含む誤ったモデルを提示してしまう．この原因を以下に述べる．

(Var-ass) 規則を例に考える．これは，代入する値の型と代入される変数の型が等しいならば，代入式全体の型が定義されることを意味している．証明の途中のある時点で，以下のリテラルがモデル候補に含まれていたとする：

```
e(ass(y,e1),f1,o)
pee(f1,local,t1)
ta(o1(o),t1)
```

すなわち，出現 o に変数 y への代入式があり，代入される値の式は型 t1, 出現 o は関数 f1 の中にあり，関数 f1 の局所変数の型は t1 である．これらのリテラルに対して (Var-ass) 規則の前件がマッチし，変数代入を行った結果の後件のリテラル

```
ta(o,t1)
```

がモデル候補に追加される．すなわち代入式全体の型が推論される．型の誤りが含まれる場合を考える．上の時点で，3 行目のリテラルの代わりに

```
ta(o1(o),t2),  -eq(t1,t2)
```

の 2 つのリテラルがモデル候補に含まれていたとする．すなわち代入する値の型と代入される変数の型が異なる場合である．(Var-ass) 規則の前件はこれらのリテラルとはマッチしない．ここでモデル候補の拡張は停止する．矛盾が発生せずモデル候補の拡張が停止したときがモデル生成法の証明の成功なので，MGTP はこのモデル候補をモデルとして返す．このモデルは，型の誤りがある箇所以前の型割当だけを含む不完全なモデルである．

このような不完全なモデル候補を排除するためには，(Var-ass) の前件が満たされない場合に，これを積極的に型の誤りとして検出し，矛盾を発生させなければならない．これらの条件は図 2 の BabyJ^T 型規則には明記されない暗黙の条件であり，これを MGTP 規則で明示的に記述しなければならない．型の不整合を検出するこの条件を，ここでは型不整合検査規則と呼ぶ．

モデル生成法による証明は，型割当 $\mathcal{P}$ のさまざまな候補に対して型規則の適用を試みる．わずかな正解を除いたそれらのほとんどは誤った型割当である．プログラムの型が正しくても，型割当が誤っていれば型規則の適用は失敗する．そのため，前章で構成した MGTP 規則は型の正しいプログラムに対しても多くの誤ったモデルを提示する．よって，型不整合検査規則を追加してこれらのモデルを排除しなければ正しく型検査が実行できたとはいえない．

型不整合検査規則が不完全なモデル候補を棄却することで，ある BabyJ プログラムに対して一つもモデルが提示されず矛盾が示されたとき，このプログラムに型の誤りがあると判定できるようになる．型の正しいプログラムに対しても，正しい型割当を持つ全てのモデルだけが型検査の結果として返される．

5.2 型不整合検査規則

表 2 の BabyJ^T 型規則に暗黙に存在する不整合に関する条件は，写像の定義域に関するものと等式に関するものの二つに分類できる．

写像 $\mathcal{P}$ の定義域に関して，関数の使われ方はプログラム中でそれぞれ 1 つに定まることを仮定したので，暗黙の条件は存在しない．

写像 $\mathcal{D}$ の定義域に関して，その第 1 引数はオブジェクト型でなければならない．そのために，第 1 引数の型 `T` を `dee` に引き渡すより前に`-ftype(T)` であることを確認する．(Member-sel) 規則はこの確認が必要な規則の一つであり，修正の結果は表 4 のようになる．`T` が関数型である場合は `ftype(T)` なので矛盾となり，モデル候補が棄却される．

等式に関する暗黙の条件は，前節 5.1 に示した例がその一つである．(Var-ass) 規則の前件が `t1` と `t2` が異なるために満たされない場合を，型の誤りとして検出するために，表 4 の (Var-ass) 規則 2 行目のように，後件で 2 つの型が等しいことを確認する．これは元の図 2 の規則の等式が成立しない場合を型の不整合として検出することを意味する．同様の修正が (Member-ass),(Member-call) の規則の等式に対して，また関数の引数渡しの際の型の判定を行う (Global-call),(Member-call),(New-cons) 規則の t_a に対して必要である．

以上の型検査規則を追加した推論規則を表 4 に示す．特に，(Member-call) 規則については，検査する項目が多段になり，MGTP 規則が 3 つに増えている．

表 4　型不整合検査規則を追加した BabyJ^T 型規則の MGTP 記述
Table 4　BabyJ^T typing rules in MGTP with type inconsistency test

規則名	MGTP 規則
(New-cons)	`e(new(C0,E),F,O),pee(C0,arg,T),ta(o1(O),U)->eq(T,U),ta(O,C0).`
(Member-sel)	`e(sel(E,M),F,O),ta(o1(O),T)->-ftype(T).` `e(sel(E,M),F,O),ta(o1(O),T),dee(T,M,S)->ta(O,S).`
(Var-ass)	`e(ass(x,E),F,O),pee(F,arg,T),ta(o1(O),U)->eq(T,U),ta(O,T).` `e(ass(y,E),F,O),pee(F,local,T),ta(o1(O),U)->eq(T,U),ta(O,T).`
(Member-ass)	`e(mass(E1,M,E2),F,O),ta(o1(O),T1)->-ftype(T1).` `e(mass(E1,M,E2),F,O),ta(o1(O),T1),ta(o2(O),U),dee(T1,M,T2)->eq(T2,U),ta(O,T2).`

(Member-call)	`e(mcal(E1,M,E2),F,O),ta(o1(O),T1)->-ftype(T1).`
	`e(mcal(E1,M,E2),F,O),ta(o1(O),T1),dee(T1,M,S)->ftype(S).`
	`e(mcal(E1,M,E2),F,O),ta(o1(O),T1),ta(o2(O),T2),dee(T1,M,ft(U1,U2,R))->eq(T1,U1),eq(T2,U2),ta(O,R).`

(Global-call)	`e(gcal(G,E),F,O),pee(G,arg,T),ta(o1(O),U),pee(G,return,R)->eq(T,U),ta(O,R).`

5.3 実　　装

以上の修正を含めることで，図 2 の型規則の暗黙の条件を含めた型推論規則の MGTP 記述が得られた．この推論規則に基づく型検査では，BabyJ プログラムに対してその正しい型割当のみがモデルとして示されることでプログラムの型安全性の判定が行える．モデルが存在しないプログラムは型の誤りを含む．

実際の型検査の実行の様子を付録に示す．3 つの関数からなる JavaScript プログラムに対して，関数型の深さ 1 段の範囲で推論を実行させた．MGTP 処理系には JavaMGTP [12] を用い，著者らの環境 (SunBlade 2000) で約 100msec でモデルが得られた．

6. 検　　討

6.1 BabyJ^T の制限

BabyJ は JavaScript のサブセットであり，2.1 節で述べた 5 つの制限が BabyJ^T 型体系を現実の JavaScript プログラムへ適用する際に問題となる．これらの制限について考察する．

a）引数，局所変数の数

引数および局所変数の数については、型のベクトルを導入することで対応できる．変数の名前とベクトルの要素との対応づけなどで冗長な処理が必要になる．JavaScript に対応した CASE ツールプラットフォーム Sapid [14] を利用してこれらの処理を自動的に行う方法について [8] で論じている．

b）値としての関数はオブジェクトでないこと

BabyJ の実行では，値は関数かオブジェクトのいずれかである．BabyJ^T はこの性質を用いて関数型とオブジェクト型を厳密に区別し，簡潔な型体系としている．構文でみると，厳密に関数が要求される箇所はメンバ関数呼び出し式のみである．(Member-call) 規則の $\mathcal{D}(t_o, m)$ は関数型でなければならない．この条件は JavaScript でも必要である．一方，BabyJ でオブジェクトが厳密に要求される箇所は，推論規則の条件部に $\mathcal{D}$ が出現する全ての構文である．JavaScript では関数もオブジェクトとして扱われるので，これらの箇所に関数が出現してもよいが，そのような計算を行うプログラムに BabyJ^T 型検査を適用すると全て型エラーとなる．

BabyJ^T の設計上の制限であるため，現状ではこの問題を回避することはできない．BabyJ^T では型の 3 項組で関数の型を区別するが，関数値のメンバを操作するプログラムに妥当な型付けを行うには，3 項組に加えてメンバの型により関数の型を区別するような型体系を構築する必要がある．

c）オブジェクトからメンバの削除ができないこと

この問題は，特別な null 値の扱いに関する問題の一部である．次の例を考える．

```
1: y = new Object(x); // y は属性 m を持たない
2: y.m(x);            // エラー
3: y.m = memberfunc1; // 属性 m を追加
4: y.m(x);            // 成功
5: delete(y.m);       // y の属性 m を削除
6: y.m(x);            // エラー
```

3 回の `y.m(x)` の呼び出しがあるが，4 行目のものを除き実行時エラーとなる．5 行目の `delete` を無視して 4 行目までのプログラムを考えてみると，2 行目で実行時エラーが発生するにもかかわらず，このプログラムは BabyJ^T 型体系に関して正しい．文献 [9] で証明されている BabyJ^T の健全性は，『プログラムの型検査が成功したならば，プログラムは正常に終了し返

り値の型は推論された型になる，あるいはプログラムは null 値に対する操作により実行時エラーを発生させる』と定義されている．上の例で示した実行時エラーは，この健全性の定義で無視する例外に相当する．すなわち，この実行時エラーでのプログラムの異常終了は，BabyJ^T によるプログラムの安全性の保証の関心の対象の外にある．`delete` 命令は，このような場合をより顕著に発生させるものである．同様な制約が他の型推論系でもみられる [16], [18], [19] ように，この問題は，静的な型検査では扱いが困難である．

クライアントサイド・スクリプト言語として JavaScript をとらえると，不要になった値を確実にメモリから消去することがセキュリティの観点から求められると考えられる．単一の型検査系だけで JavaScript プログラムの問題を全て検出するのではなく，この要求には，`delete` 命令による値や属性の削除が要求通りに行われているかを検査するシステムを別に用意して対処する必要があると考える．

d） eval がないこと

文字列をコードとして評価する `eval` 関数は，値の中身により結果の型が変化する．このような動的演算は静的な型検査では扱えない．クライアントサイド・スクリプト言語としての JavaScript を考えると，一般的な形で `eval` が使われる機会は多くないと考えられる．Web の form の内容のように，外部から得た文字列を直接 `eval` することは危険である．JavaScript コードの難読化のためにあらかじめスクランブルしたコードをデコードして，その後に `eval` して実際のプログラムを得るといった場面で使われるが，これは特殊な用途であろう．

同様の問題が，文字列の数値への自動変換のような場面でも起きるが，これは型の間での変換が限定されており，推論規則の拡張で対応できると思われる．

6.2 型体系の拡張

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く排除されている．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．

この問題に対処しようとして，C. Anderson らは対応構文と型体系を強化するために BabyJ^T を発展させた型体系 JS_0 を文献 [10] で提案している．そこでは，オブジェクト型の表現に不動点による循環レコード型を導入し，その上の部分型を定義することによって型の多相性を表現しているが，直観的な把握に難しさがある．

これに対して，より単純に型の多相性を扱える型体系として，著者ら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 [16]．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, [16] 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ことができると考えられる．

もう一つの課題として，前節 6.1 項目 c) で述べたメンバの削除の問題について付言する．文献 [17] では，手続き型プログラムにおけるデータフローに類した概念としてオブジェクト指向プログラムに対するオブジェクトフローグラフを定義している．オブジェクトフローグラフでは，オブジェクトはコンストラクタだけでなくそのコンストラクタの呼び出しの位置によっても区別される．フローという細かい粒度でオブジェクトの型を定義することで，メンバの削除を追跡できる型体系も期待できる．

これらの型体系の拡張については本研究の今後の課題とする．

6.3 関 連 研 究

オブジェクト指向言語の型体系に関する研究は，クラスベースの言語に対するものとオブジェクトベースの言語に対するものに大きく分類される．

クラスベースの言語に対しては，オブジェクトの型を定義するためにクラスが使えるため研究が進んでおり，手続き型の言語に対する型体系も多く提案されている．文献 [23] で提案されたレコード型に基づく方法や，クラスそのものを型の構成子として使う方法 [16], [18], [19] がある．JavaScript は近年注目されているいわゆるライトウェイト言語に属するが，その一つである Python に対する型体系も提案されている [22].

一方，オブジェクトベースの言語に対する型体系は，関数型の言語を対象として形式化するものが多い [20], [21]．オブジェクトベースの手続き型言語に対する型検査系には [24] がある．

文献 [19] では，型検査系の実装に関する評価を行っている．ここでは，制約の解消に基づく型検査系を C++言語で実装し，性能評価を行っている．

これらの研究に対して，本論文で提案した手法は，オブジェクトベースの手続き型言語に対するものであるが，型推論規則を形式的に定義し，モデル生成法に基づく定理証明系により直接検証するため，計算量の点で不利であるが，型体系が素直な形で表現されており，変更や拡張のコストが小さいという利点をもつと考えられる．

7. ま と め

本論文では，JavaScript のサブセット言語 BabyJ に対する型推論系である BabyJ^T 型検査系を対象とし，これを MGTP 定理証明系で推論可能な形式に変換する方法を示した．このアプローチにより，形式的に定義された型推論系を直接にプログラムに適用することができるようになった．モデル生成法に基づく証明系を用いることで，問題の記述が自然な形で行えることを示し，その上で，推論規則に含まれる暗黙の条件を型不整合検査規則として明示化して，不完全なモデルの報告を排除する必要があることを示した．

BabyJ は JavaScript のサブセットである．本手法の実用化には JavaScript の主要な構文への対応が不可欠である．また，入力プログラムの情報を MGTP で扱える形に変換する前処理系が必ず必要である．JavaScript は単体で用いられる言語ではなく HTML, XUL [15] 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 プログラムの実行時環境の一部になっている．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，本論文で JavaScript プログラムの情報を型検査系に渡すための公理を生成した方法と同様にして，外側の文書を読み込んで必要な公理を生成し，文書全体の型整合性を検査する方法が必要である．また，型検査の結果として得られる型割当はプログラムの検証や理解に有用な情報であり，これらを CASE ツールの枠組みで再利用できることが期待される．例えば，ソースプログラムブラウザと連携して型割当を同時に閲覧可能にすることが考えられる．

本論文では自動証明系として MGTP を採用しているが，MGTP の実装の一つに Java で実装された JavaMGTP [12] がある．JavaMGTP は任意の Java アプリケーションに組み込んで動作させることができるため，本型検査系を Sapid [13], [14] のような CASE ツールプラットフォームと連携させて動作させることが可能である．この拡張性は，本研究の実用化に関する重要な課題である前処理系の問題への対応と型割当情報の応用の両面に有効である．これらの課題については [7], [8] で議論を始めているが，引き続き検討を進め報告したい．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (A)(2)(16200001) の補助による．有益な御意見を頂いた査読者の方々に深く感謝致します．

文 献

[1] S. YUEN, K. KATO, D. KATO, S. YAMAMOTO, and S. AGUSA, “A Testing framework for web applications based on the MVC model with behavioral descriptions,” International Conference on Information Technology & Applications 2004, CD-ROM, Harbin, China, Jan. 2004.

[2] 根路銘 崇, 小野 康一, 田井 秀樹, 安部 麻里, “Web アプリケーションモデルに基づく JSP の動的検証,” ソフトウェアテストシンポジウム 2004, pp.61–67, Jan. 2004.

[3] F. Ricca, and P. Tonella, “Building a tool for the analysis and testing of web applications: problems and solutions,” Proc. of TACAS’2001, Tools and Algorithms for the Construction and Analysis of Systems, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS’2001, LNCS 2031, pp.373–388, Genova, Italy, Apr. 2001.

[4] 鷲尾 和則, 松下 誠, 井上 克郎, “JavaScript を含んだ HTML 文書に対するデータフロー解析を用いた構文検証手法の提案,” 信学技報, SS2002-22, Vol.102, No.370, pp.13–18, Oct. 2002.

[5] 松下 誠, 鷲尾 和則, 井上 克郎, “インラインスクリプトを含んだ XHTML 文書に対するデータフロー解析を用いた構文検証手法,” 情処学論, Vol.45, No.8, pp.2034-2042, Aug. 2004.

[6] 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,” SS2004-3, 信学技報 Vol.104, No.47, pp.13–18, Apr. 2004.

[7] 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,” SS2004-13, 信学技報 Vol.104, No.242, pp.41–46, Aug. 2004.

[8] 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 プログラムの型検査系”, SS2005-11, 信学技報 Vol.105, No.25, pp.25–30, Apr. 2005.

[9] C. Anderson, and S. Drossopoulou, “BabyJ: from object based to class based programming via types”, WOOD2003 Workshop on Object Oriented Developments, Electr. Notes. Theor. Compt. Sci., Vol. 82, No. 8, Warsaw, Poland, April 2003.

[10] C. Anderson, and P. Giannini, “Type checking for JavaScript”, WOOD’04 Workshop on Object-Oriented Developments, London, UK, August 2004.

[11] 長谷川 隆三, 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”, 情処学論, vol.37, No.1, pp.1–12, Jan. 1996.
[12] 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,” 情処学論, vol.41, No.6, pp.1791–1798, June 2000.
[13] 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,” 情処学論, Vol.39, No.6, pp.1990–1998, June 1998.
[14] Sapid : Sophisticated APIs for CASE tool Development, http://www.sapid.org/
[15] XUL : XML User Interface Language, http://www.mozilla.org/projects/xul/
[16] 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.255–268, Dec. 2000.
[17] P. Tonella, and A. Potrich, “Reverse Engineering of Object Oriented Code,” Springer, New York, 2005.
[18] J. Palsberg, and M.I. Schwartzbach, “Object-Oriented Type Inference,” Proc. OOPSLA ’91, ACM SIGPLAN NOTICES, Vol 26, No. 11, pp.146–161, 1991.
[19] N. Oxhøj, J. Palsberg, and M.I. Schwartzbach, “Making Type Inference Practical,” Proc. ECOOP ’92, LNCS 615, pp.329–349, 1992.
[20] M. Abadi, and L. Cardelli, “A Theory of Objects,” Springer-Verlag, 1996.
[21] K. Fisher, F. Honsell, and J.C. Mitchell, “A Lambda Calculus of Objects and Method Specialization,” Nordic Journal of Computing, 1(1):3.37, 1994. preliminary version appeared in Proc. of IEEE Symp. LICS ’93.
[22] M. Salib, “Static type inference (for python) with starkiller,” http://www.python.org/pycon/dc2004/papers/1/paper.pdf, 2004.
[23] L. Cardelli, and P. Wegner, “On understanding types, data abstraction, and polymorphism,” ACM Computing Surveys, vol. 17, No. 4, pp.471–522, 1985.
[24] O. Agesen, J. Palsberg, and M.I. Schwartzbach, “Type inference of SELF: Analysis of objects with dynamic and multiple inheritance,” Lecture Notes in Computer Science, 707, pp.247–262, 1993.

付　　録

実 行 例

Java により実装された MGTP 処理系である JavaMGTP [12] を用いて実行した例により，型推論の処理の流れを示す.

次のプログラムを型検査する．ただし，このプログラムは整数型の値を扱えるように拡張している．この拡張に対応するために，int 型の推論規則 `e(intval,F,O)->ta(O,int)`. を追加する.

```
// constructor
function c1(x) { this.i=x; this.m=f1; y=3 }
// global
function g1(x) { x=2; y=new c1(4); y.m(5) }
// member
function f1(x) { y=1; this.i }
```

整数を表す定数項 `intval` を表 1 に追加し，このプログラムを表す公理は

```
->cf(c1).
->e(seq(mass(this,i,x),seq(mass(this,m,m1),
ass(y,intval))),c1,c1).
->gf(g1).
->e(seq(ass(x,intval),seq(ass(y,new(c1,int
val)),mcal(y,m,intval))),g1,g1).
->mf(m1).
->e(seq(ass(y,intval),sel(this,i)),m1,m1).
```

となる.

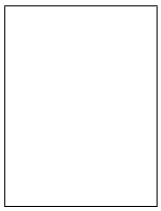
整数型 `int`，オブジェクト型 `c1`，これらから生成される深さ 1 の関数型 (`ft(int,int,int)`〜`ft(c1,c1,c1)` の 8 個) について，述語 `type`, `ftype`, `cf`, `eq` に関する公理を記述する.

プログラムから生成された公理と，本稿で定義した推論規則を JavaMGTP 処理系に与えて推論を実行する．モデルを発見するのにかかる時間は著者らの環境 (SunBlade 2000) で 97msec であった.

JavaMGTP 処理系の起動時のオプションによりモデルをテキスト出力することができる．型割当に関するリテラルのみ抜粋したものを以下に示す．写像 $\mathcal{P}, \mathcal{D}$ および部分式の型割当のいずれも正しいものになっていることが確認できる.

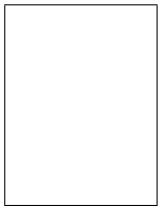
```
dee(c1,i,int)
dee(c1,m,ft(c1,int,int))

pee(c1,this,c1)
pee(c1,arg,int)
pee(c1,local,int)
pee(c1,return,int)
pee(g1,this,void)
pee(g1,arg,int)
pee(g1,local,c1)
pee(g1,return,int)
pee(m1,this,c1)
pee(m1,arg,int)
pee(m1,local,int)
pee(m1,return,int)

ta(c1,int)
ta(g1,int)
ta(m1,int)
ta(o1(c1),int)
ta(o1(g1),int)
ta(o1(m1),int)
ta(o1(o1(c1)),c1)
ta(o1(o1(g1)),int)
ta(o1(o1(m1)),int)
ta(o1(o1(o1(o2(g1)))),int)
ta(o1(o1(o2(c1))),c1)
ta(o1(o1(o2(g1))),c1)
ta(o1(o2(c1)),ft(c1,int,int))
ta(o1(o2(g1)),c1)
ta(o1(o2(m1)),c1)
ta(o1(o2(o2(c1))),int)
ta(o1(o2(o2(g1))),c1)
ta(o2(c1),int)
ta(o2(g1),int)
ta(o2(m1),int)
ta(o2(o1(c1)),int)
ta(o2(o1(o2(c1))),ft(c1,int,int))
ta(o2(o2(c1)),int)
ta(o2(o2(g1)),int)
ta(o2(o2(o2(g1))),int)
```

（平成 xx 年 xx 月 xx 日受付）



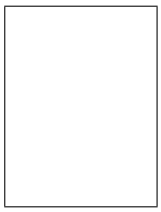
大久保弘崇

1992 名大・工・情報卒．1997 同大大学院博士課程了．1997 愛知県立大学助手，現在に至る．工学修士．オブジェクト指向計算モデル，代数的仕様記述法，ソフトウェア工学などの研究に従事．情報処理学会，日本ソフトウェア科学会各会員．



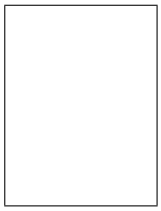
山本晋一郎　（正員）

1986 名大・工・電気卒．同大大学院に進学．同大工学部に勤務．同助手，講師を経て，1998 より，愛知県立大学情報科学部助教授．博士（工学）．専門はソフトウェア工学，言語処理系，ユーザインタフェースに関する研究に従事．情報処理学会，日本ソフトウェア科学会各会員．



坂部　俊樹　（正員）

1972 名大・工・電気卒．1977 同大大学院博士課程了．名古屋大学助手，三重大学助教授，名古屋大学助教授を経て，1993 より名古屋大学教授．ソフトウェアの基礎理論全般に興味を持つ．抽象データ型の理論，プログラミング言語の形式的意味論，自動プログラミング，書換え型計算モデル，オブジェクト指向計算モデルなどの研究に従事．工博．情報処理学会，人工知能学会，日本ソフトウェア科学会，EATCS 各会員．



稲垣　康善　（正員：フェロー）

1962 名大・工・電子卒．1967 同大大学院博士課程了．同大助教授，三重大学教授を経て，1981 名大・工・教授，2003 名大名誉教授．同年より愛知県立大学情報科学部教授．工博．コンピュテーションとコミュニケーションの理論，オートマトン言語理論，ソフトウェア基礎論，自然言語処理に関する研究に従事．本会情報・システムソサイエティ会長，副会長等を歴任，功績賞受賞．情報処理学会（フェロー），言語処理学会，人工知能学会，日本ソフトウェア科学会，IEEE, ACM, EATCS 各会員．

Abstract Type checking is a method to improve dependability of programs. This paper proposes a type checking method directly employing a theorem prover to examine type consistency, specified by formally defined rules. We adopted MGTP for the theorem prover, which is based on model-generation approach. This paper shows how to describe the type system defined as a set of inference rules in MGTP syntax. We clarify that our type inference process needs to write out the implicitly implied conditions of the type system, and we show how to describe such conditions explicitly as MGTP rules.

Key words JavaScript, type checking, theorem prover, MGTP