Java のクラスローダ制約の定式化

萩谷昌己[†] 戸沢晶彦[†]
高橋孝一[††] 西崎真也[†††]

Saraswat が指摘したように，Java1 では同じ名前のクラスが異なるクラスローダでロードされたとき型安全性が一般には成り立たなかった．Java2 では複数のクラスローダの存在する状況における型安全性を保証するためにクラスローダ制約と呼ばれる機構が導入された．戸沢は，クラスローダ制約の存在下での JVM の型安全性を証明したが，その証明は極めて複雑であり理解しづらかった．本論文では，戸沢の議論を単純化することにより，クラスローダの役割を明確化する．理想的クラスローダ環境という概念を用いて，Java の宣言的なサブセットの定式化を与える．

Formalization of Classloader Constraints of Java

Masami Hagiya,[†] Akihiko Tozawa,[†] Koichi Takahashi[††]
and Shin-ya Nishizaki[†††]

As Saraswat pointed out, the type safety in Java1 does not always hold when classfiles that share the same name are loaded by different classloaders. In Java2, the mechanism of loader constraints was introduced to guarantee the type safety in case that multiple class loaders coexisted. Although Tozawa proved the type safety of the JVM with loader constraints, his proof was extremely complex and hard to understand. This paper clarifies the role of loader constraints by simplifying Tozawa's arguments. We formulate classloaders and loader constraints for a declarative subset of Java and prove its type safety.

1. はじめに

1.1 Java とクラスローダ

Java[3),6)] の特徴の一つにクラスローダを用いたローディングの機能がある．クラスローダは，クラスのバイトコードをロードするためのオブジェクトである．Java では，Java のプログラムによりクラスローダを再定義することができ，種々のローディング方法を実現することが可能になっている．実際，この機能を用いて Java では次のようなローディング機能を実現している．システムクラスがロードされる場合には JVM (Java Virtual Machine) により提供されているシステムクラスローダが使用され，ローカルに存在するクラスファイルがロードされる．一方，アプレットのクラスがロードされるときには，アプレットごとにアプレットクラスローダが生成され，それらのクラスローダはネットワークを経由してクラスファイルをロードする．単一の JVM 上では一般にアプレットは複数動作し，それらのアプレットは必ずしも信頼できるとは限らない．アプレットと同様，アプレットクラスローダ自体も Java により記述されている．JVM では，アプレットごとにクラスローダを設けて，名前空間を分離することにより，セキュリティの向上をはかっている[8)]．

ない．JVM では，アプレットごとにクラスローダを設けて，名前空間を分離することにより，セキュリティの向上をはかっている[8)]．アプレットと同様，アプレットクラスローダ自体も Java により記述される．

Java のクラスローダの機構は，Java のプログラムが名前空間など，言語の実行系自体の機能を変更・拡張することを可能にしており，非常に強力な記述性を有している．その強力さゆえに，セキュリティの面で問題[2)] を生み出してしまった．Saraswat[12)] は，Java の型システムがクラスローダの存在を前提としていないため，同じ名前のクラスが異なるクラスローダでロードされたとき，コンパイル時に型付けされたプロ

† 東京大学大学院理学系研究科
Graduate School of Science, University of Tokyo
†† 電子技術総合研究所
Electrotechnical Laboratory
††† 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

グラムであっても，実行時に型エラーが発生する場合があることを指摘した．

Saraswat 自身 [12] も，この問題を避けるためにいくつかの解決方法を提案しているが，その一つは，クラスの等価性を常にクラスとクラスローダの組に対して判断するというものである．同様の方法を Jensen ら [4] も提案している．また，Dean は，ファーストクラス環境 [1),10)] を用いてクラスローダの定式化を試みている．

1.2 クラスローダ制約

Saraswat の問題を避けるために最終的に Java2 で採用されたのは，クラスローダ制約（class loader constraint）の仕組みである [5),7)]．クラスローダ制約とは，1 つのクラス名 c に対して 2 つのクラスローダ L_1, L_2 がローディングをおこなったとき，それらの結果が同一にならなければならないという制約であり，次のように書く：

$$(c, L_1) \approx (c, L_2)$$

なお，Liang と Bracha[5] は上の制約を $c^{L_1} = c^{L_2}$ と書いている．また，戸沢 [13),14)] は $L_1 \overset{c}{\sim} L_2$ と書いている．

Java2 では，既にロードされたクラスのコンスタント・プールをもとにして，上記のようなクラスローダ制約を蓄積する．クラスローダ制約が蓄積された時点から後に，L_1 と L_2 の両方が c をロードしたとき，もしそれらの結果が食い違っていたならば，クラスローダ制約が満たされなくなるため，どちらかのローディングは無効となる（ローディングの際に例外が発生する）．

多くの JVM の実装では，インスタンスが生成されたりメソッドが呼び出されたりするなど，クラスの中身が本当に必要になってからはじめてクラスをロードする．このように，必要になってからクラスをロードすることを遅延ローディング (lazy loading) という．多くの Java の実装では遅延ローディングを行なうため，クラスローダ制約が蓄積された時点では，L_1 や L_2 は c をロードしていないかもしれない．L_1 もしくは L_2 のどちらかがローディングを行なっていなければ，クラスローダ制約はみたされているとみなされる．

ただし，二つ以上のクラスローダ制約が関連することがありうる．例えば，次のような二つのクラスローダ制約が蓄積されたとしよう．

- $(c, L_1) \approx (c, L_2)$
- $(c, L_2) \approx (c, L_3)$

そして，L_2 はまだ c をロードしていないが，L_1 と L_3 はともに c をロードしていたとする．この場合，L_1 と L_3 が c をロードした結果は同じにならなければならない．すなわち，クラスローダ制約は遷移性をみたさなければならないことがわかる．

そこで，具体的な例を考察する．以下のようなクラス C, D, E があるとする：

```
class C {
  static void foo()
  {
    D.bar(new A());
  }
}

class D {
  static void bar(A x)
  {
    E.boo(x);
  }
}

class E {
  static void boo(A x)
  {
    x.field = 0;
  }
}
```

そして，

- クラスローダ L_1 によってクラス C が読み込まれたとする．
- L_1 は，クラス D のローディングをクラスローダ L_2 に委譲したとする．
- さらに，L_2 は，クラス E のローディングをクラスローダ L_3 に委譲したとする．

このとき，L_2 はクラス A をロードする必要はない．なぜなら，D のメソッド bar は，受け取ったインスタンス x をそのまま E のメソッド boo に渡しているからである．しかし，この状況においては，

- $(\mathtt{A}, L_1) \approx (\mathtt{A}, L_2)$
- $(\mathtt{A}, L_2) \approx (\mathtt{A}, L_3)$

というクラスローダ制約が蓄積される．したがって，クラスローダ制約の推移性より，

- $(\mathtt{A}, L_1) \approx (\mathtt{A}, L_3)$

が導かれる．

C は A のインスタンスを生成している．E は，A のインスタンスのフィールド field を参照している．したがって，L_1 と L_3 は A をロードせざるをえない．

すると，上のクラスローダ制約によって，L_1 が A

をロードした結果と L_3 が A をロードした結果が同じかどうかがチェックされる．同じでない場合は例外が発生する．

インスタンス，クラス，クラスローダがみたすべき関係 $v::_L t$ は，

インスタンス v は，クラスを表わす式 t から
クラスローダ L によりロードされて得られ
るクラスに属する

と定義するのが自然だろう．しかし，これを「値 v が型 t に属する」という関係としてみると，上の例の場合，C で作られた A のインスタンスを v としたとき，確かに C の中では $v::_{L_1}$ A が成り立っているが，D の中では $v::_{L_2}$ A を導くことができない．なぜなら，L_2 は A を実際にはロードしていないからである．しかし，E の中では $v::_{L_3}$ A が成り立つ．このように，$v::_L t$ という関係は，クラスをロードしたかどうかに依存するために，あらゆるクラスローダに対して大域的に定義されるものではなく，したがって，この関係を用いて型安全性を与えることには問題がある．

この問題を回避するために，戸沢[13),14)]は，この関係を拡張したものとして次のようなものを与えた：

$v ::^{+}_{L} t$

この関係は，

$v ::_{L'} t$ かつ $(t, L) \approx (t, L')$ を満たす L'
が存在する．

と定義される．関係 $v ::_L t$ におけるクラスローダに関しての局所性が，関係 $v ::^{+}_{L} t$ では緩和されている．前述の例の場合，$v::_{L_2}$ A は成り立たなかったが，$v::^{+}_{L_2}$ A は成り立つ．

戸沢は，Java のバイトコードの実行がこの型付けの関係を保存することを示し，クラスローダ制約の型安全性を証明することに成功した．しかし，実行においてこの関係が常に保存されることを示すためには，どのようなクラスローダ制約が蓄積されたかを分析する必要があるために，証明は複雑になり，この定義がどのようにして有効に機能しているのかということを理解することは容易ではなかった．

1.3 クラスローダ制約の新しい定式化

クラスローダ制約が未来のローディングに対する制約であるとすると，$(c, L_1) \approx (c, L_2)$ という関係は，未来のローディングの可能性も含めたクラス間の等価性を表していると考えることができる．本論文では，この考えをもとにしてクラスローダ制約を反映した Java の宣言的なサブセットに対する型システムを提唱する．まず，天下り的に Λ という関数を導入する．Λ は，クラスローダとクラス名を引数にとる．$\Lambda(L, c)$ は，ローディングが完全に行なわれた理想的な状況において c を L でロードした結果を表している．

すると，

$(c, L_1) \approx (c, L_2)$

という関係は，

$\Lambda(L_1, c) = \Lambda(L_2, c)$

と定義することができる．これは明らかに同値関係となる．この同値関係は，ローディングが完全に行なわれた理想的な状況において，c を L_1 でロードした結果と c を L_2 でロードした結果が同じになることを意味している．

関数 Λ と同値関係 $\approx$ を先に与えたので，ローディングとローダ制約は，これらと整合性を保っていなければならない．この整合性は，型検査における付帯条件の一種になる．その結果，型安全性の議論は，静的な型検査が成功したならば動的な型エラーが生じないという通常の形で定式化することができる．

1.4 本論文の構成

本論文の以下の部分では，Java の宣言的なサブセットに対してクラスローダとローダ制約を定式化し，クラスローダ制約の存在下での型付けを定義した後，その型安全性を証明する．次節でサブセットを定義する．3 節でクラスローダとローダ制約を導入する．そして，4 節で動的意味論，5 節で静的意味論を与える．それらをもとに，6 節で型安全性の命題を証明する．

2. 式とクラス定義

本節では，Java の非常に小さな，しかも，宣言的なサブセットを，Standard ML の型として定式化する．

2.1 変数名，クラス名，メソッド名

まず，変数名，クラス名，メソッド名を用意する．

```
type vname = string;    % 変数名
type cname = string;    % クラス名
type mname = string;    % メソッド名
```

vname は変数名の型，cname はクラス名の型，mname はメソッド名の型を意味する．すべて string として定義されているが，データの等価性が判定できるような型であれば何でもよい．

以下の議論では，次のような（メタ）変数を用いる．

$x, x', x_1, \cdots$: vname
$c, c', c_1, \cdots$: cname
$m, m', m_1, \cdots$: mname

2.2 式

次に式の型 exp を定義する．

```
datatype exp =          % 式
  this                  % 自分自身
```

```
  | var of vname          % 変数参照
  | capp of (cname*exp)   % インスタンス生成
  | mapp of (exp*cname*mname*exp*texp)
                          % メソッド呼び出し
  | iconst of int         % 整数定数
  | pair of exp*exp       % 組
  | proj1 of exp          % 組の第1要素
  | proj2 of exp          % 組の第2要素
  ;
```

this はオブジェクト自身を表す式である．var は変数参照を表す．capp はクラス・コンストラクタの適用を表し，引数としてはクラス名と（クラス・インスタンスを表現する）式をとる．mapp はメソッドの呼び出しを表し，引数として，オブジェクトを表す式，（そのオブジェクトの）クラス名，メソッド名，引数の式，引数の型をとる．texp は型を表す式の型である．後に 5.1 節で定義する．iconst は整数定数を表し，引数として整数をとる．

以下の議論では，次のような（メタ）変数を用いる．

$e, e', e_1, a, b, \cdots$: exp

2.3 クラス定義

次に，クラス定義の型 class を定義する．

```
type class =          % クラス定義
  texp *              % 表現型
  (mname*texp, vname*texp*exp*texp)
    table             % メソッド・テーブル
  ;
```

クラス定義は二つの構成要素から成り立っている．最初の要素は，クラスのインスタンスを表現するデータの型である．例えば，平面上の点を表すオブジェクトのクラスを定義しようとするとき，そのインスタンスは二つの整数の組によって表現することができる．従って，表現型は int*int となる．2 番目の要素はメソッドのテーブルである．このテーブルは，メソッド名にメソッドの定義を対応させる．個々のメソッドの定義は，引数の変数名，引数の型，メソッドの本体，本体の型（すなわち返り値の型）から成り立っている．ここでは，各メソッドのとる引数の個数は 1 つであるという制限をもうけることにより，体系の簡略化をはかっている．なお，一般に ('a, 'b) table は'a の要素に 'b の要素を対応させるテーブルの型を意味する．

メソッドの本体の中で式 this はオブジェクト自身を表現するデータを値とする．従って，this の型はクラス定義の表現型となる．

以下の議論では，次のような (メタ) 変数を用いる．

$C, C', C_1, \cdots$: class

クラス定義の各要素を取り出すために，以下のような関数を用意する．

```
reptype : class -> texp
```

reptype はクラス定義の表現型を返す．

```
lookup : class*mname*texp ->
           (vname*texp*exp*texp) option
```

lookup はクラス定義のメソッド・テーブルを検索する．'a option は次のように定義される型であり，lookup(C,m,t) は，クラス定義 C の中で名前が m，引数の型が t であるメソッドを検索し，そのようなメソッドがあれば，メソッドの定義に構成子 SOME の付いたものを返す．なければ構成子 NONE を返す．

```
datatype 'a option = SOME of 'a | NONE;
```

以下の部分においても，部分関数を明示的に表現するために，返り値の型に option 型を用いるであろう．

3. クラスローダとクラスローダ制約

先にも述べたように，本論文では，クラスローダとクラスローダ制約の定式化を行なう．まず，クラスローダとクラスローダ環境を定義する．

3.1 クラスローダ

本論文では，クラスローダは番号で参照することとする．

```
type loader = int;      % クラスローダ
```

以下で定義するように，クラスローダ環境がクラスローダ（の番号）とクラス名に対してクラス定義を対応させる．

以下の議論では，次のような（メタ）変数を用いる．

$L, L', L_1, \cdots$: loader

3.2 クラスローダ環境

クラスローダ環境は，次のような型をもつテーブルである．

```
type lenv =
  (loader*cname, class*loader) table;
```

すなわち，クラスローダ（の番号）とクラス名の組に対して，クラス定義とクラスローダの組を対応させる．後者のクラスローダは，クラス定義の中のクラスをロードする際に用いられる．

クラスローダ環境のもとでクラスをロードするために，次のような関数を用意する．

```
load : lenv*loader*cname ->
         (class*loader) option
```

クラスローダ環境 E，クラスローダ L，クラス名 c に対して，load(E,L,c) は，クラス定義とクラスローダの組に SOME が付いたものか，NONE を返す．前者

はクラスのロードに成功した場合に対応し，後者は失敗した場合に対応する．

以下では，クラスローダ環境 E を一つ固定して議論を進める．

3.3 理想的クラスローダ環境とクラスローダ制約

Λ は，クラスローダとクラス名に対して，クラス定義とクラスローダの組を返す関数とする．

Λ : `loader*cname -> class*loader`

この関数は部分関数ではない．すなわち，任意のローダとクラス名に対して，クラス定義とそれが参照するクラスをロードするためのローダとを必ず返す．いいかえると，関数 Λ は，ローディングが完全に行なわれた理想的な状況におけるクラスローダ環境と考えることができる．

クラスローダ制約 $\approx$ は，Λ から定義される同値関係である．すなわち，

$(c, L_1) \approx (c, L_2)$

という関係は，

$\Lambda(L_1, c) = \Lambda(L_2, c)$

と定義される．$(c, L_1) \approx (c, L_2)$ は，クラス名 c をクラスローダ L_1 でロードした結果と，c をクラスローダ L_2 でロードした結果が同じになるという制約を表している．

3.4 クラスローダ環境の整合性

クラスローダ環境 E が Λ に対して整合的(compatible)であるとは，以下の条件が満たされることである．

`load(`E`,`L`,`c`)` = `SOME(`C'`,`L'`)` ならば，
$\Lambda(L, c) = (C', L')$.

本論文の以下の部分では，E は Λ に対して整合的であると仮定する．

なお，本論文では Λ を天下り的に与えたが，Java2の実行中に蓄積されたクラスローダ制約とクラスローダ環境 E から Λ を逆に構成することができる．1.2節で説明したとおり，戸沢の研究[13),14)]では，クラスローダに対して個別にさだめられる局所的な関係 $v ::_L \mathtt{A}$ を，より大域的な関係 $v ::^{\dagger}_L t$ に拡張するという手法が重要であった．これは，局所的に定義されているクラスローダ制約とクラスローダ環境から大域的に定義されている Λ を構成するということに相当している．

4. 動的意味論

本節では，以上で定義した Java のサブセットに対して動的意味論を与える．

4.1 値

まず，式を評価した結果として得られる値を定義する．

```
datatype value =          % 値
  error                   % 実行時エラー
| cvalue of (class*loader*value)
                          % インスタンス
| ivalue of int           % 整数
| pvalue of value*value
                          % 組
;
```

構成子 `error` は実行時エラーを意味する．クラスのインスタンスは，`cvalue`(C,L,v) という形をしている．C がクラス定義，L は C の中に現れるクラスをロードするためのクラスローダである．v は C の表現型を持つデータである．`ivalue` は整数値を表す．また，`pvalue` は値の組である．

以下の議論では，次のような（メタ）変数を用いる．

$v, v', v_1, w, \cdots$: `value`

4.2 動的意味論の定義

動的意味論は，$C, L, v, r \vdash e \Downarrow v'$ という述語によってあたえられる．この述語は図 1 にあげられた規則により帰納的に定義される．ここで，C はクラス定義，L はクラスローダ，v はオブジェクト自身 `this` の値，r はメソッドの仮引数に実引数を対応させる環境である．$e \Downarrow v'$ は，式 e を評価した結果が値 v' であることを意味する．$[w/x]$ によって，引数の名前 x に値 w を対応させる環境を表す．なお，メソッドの引数が1つに制限されているので，環境は必ず $[w/x]$ という形になる．

一般に，各規則にもうけられている付帯条件が満たされなければ，

$$\frac{}{C, L, v, r \vdash e \Downarrow \mathtt{error}}$$

であることとする．例えば，

$$\frac{C, L, v, r \vdash e \Downarrow v' \quad v' \neq \mathtt{cvalue}(\ldots)}{C, L, v, r \vdash \mathtt{mapp}(e, c, m, a, t') \Downarrow \mathtt{error}}$$

となる．また，部分式の評価が `error` になれば，全体も `error` とする．例えば，

$$\frac{C, L, v, r \vdash e \Downarrow \mathtt{error}}{C, L, v, r \vdash \mathtt{capp}(c, e) \Downarrow \mathtt{error}}$$

となる．

$$\frac{C,L,v,r \vdash e_1 \Downarrow v_1 \quad C,L,v,r \vdash e_2 \Downarrow v_2 \quad v_1 \neq \texttt{error} \quad v_2 \neq \texttt{error}}{C,L,v,r \vdash \texttt{pair}(e_1,e_2) \Downarrow \texttt{pvalue}(v_1,v_2)}$$

$$\frac{C,L,v,r \vdash e \Downarrow \texttt{pvalue}(v_1,v_2)}{C,L,v,r \vdash \texttt{proj1}(e) \Downarrow v_1} \qquad \frac{C,L,v,r \vdash e \Downarrow \texttt{pvalue}(v_1,v_2)}{C,L,v,r \vdash \texttt{proj2}(e) \Downarrow v_2}$$

$$\frac{}{C,L,v,r \vdash \texttt{iconst}(i) \Downarrow \texttt{ivalue}(i)}$$

$$\frac{}{C,L,v,r \vdash \texttt{this} \Downarrow v}$$

$$\frac{}{C,L,v,[w/x] \vdash \texttt{var}(x) \Downarrow w}$$

$$\frac{\begin{array}{l}\texttt{load}(E,L,c) = \texttt{SOME}(C',L') \\ C,L,v,r \vdash e \Downarrow v' \\ v' \neq \texttt{error}\end{array}}{C,L,v,r \vdash \texttt{capp}(c,e) \Downarrow \texttt{cvalue}(C',L',v')}$$

$$\frac{\begin{array}{l}C,L,v,r \vdash e \Downarrow \texttt{cvalue}(C',L',v') \\ \texttt{load}(E,L,c) = \texttt{SOME}(C',L') \\ C,L,v,r \vdash a \Downarrow w \\ w \neq \texttt{error} \\ \texttt{lookup}(C',m,t') = \texttt{SOME}(x,t',b,t'') \\ C',L',v',[w/x] \vdash b \Downarrow v''\end{array}}{C,L,v,r \vdash \texttt{mapp}(e,c,m,a,t') \Downarrow v''}$$

図 1 動的意味論の帰納的定義
Fig. 1 Inductive definition of the dynamic semantics.

5. 静的意味論

本節では，以上で定義したサブセットに対する型付けを定義する．

5.1 型　　式

まず，型を表す式を定義する．

```
datatype texp =                 % 型式
  ctype of cname                % クラス型
| itype                         % 整数型
| ptype of texp*texp            % 直積型
;
```

ctype はクラス型を表す．クラス名がそのまま型を表す式となっている．itype は整数型を表し，ptype は直積型を表す．texp の要素を型式と呼ぶ．

以下の議論では，次のような（メタ）変数を用いる．

$t, t', t_1, u, \cdots,$: texp

クラスローダ制約 $\approx$ は，図 2 のように型式とクラスローダの組の間の同値関係に自然に拡張される．

5.2 静的意味論の定義

静的意味論は，$C,L,t,s \vdash e : t'$ という述語により与えられる．この述語は図 3 にあげられた規則により帰納的に定義される．ここで，C はクラス定義，L はクラスローダ，t はオブジェクト自身 this の型式，s はメソッドの仮引数に型式を対応させる環境である．$e : t'$ は，式 e の型式が t' であることを意味する．$[u/x]$ によって，引数の名前 x に型式 u を対応させる環境を表す．

5.3 クラスローダ環境の型付け

クラスローダ環境 E が以下の条件を満たすとき，E は型付けされているといい，$\vdash E$ と書く．

> $\texttt{load}(E,L,c) = \texttt{SOME}(C',L')$ ならば，$\texttt{lookup}(C',m,t) = \texttt{SOME}(x,t',b,t'')$ を満たす任意の m と t に対して，$t = t'$ かつ，
> $C', L', \texttt{reptype}(C'), [t'/x] \vdash b : t''$
> が成り立つ．

5.4 値の型付け

値に型を与えるために，型を表す値を以下のように定義する．

```
datatype tvalue =               % 型値
  ctvalue of class*loader
| itvalue
| ptvalue of tvalue*tvalue
;
```

tvalue を型とするデータを型値と呼ぶ．ctvalue はクラス型の型値を表し，クラスをロードした結果得られるクラス定義とクラスローダを引数としている．itvalue は整数型の型値を表し，pvalue は直積型の型値を表す．

以下の議論では，次のような（メタ）変数を用いる．

$\tau, \tau', \tau_1, \cdots$: tvalue

型値は型式の評価結果である．ここでいう「型式の評価」とは，クラスの型式に現れるクラス名を与えら

$$\frac{(c,L_1) \approx (c,L_2)}{(\mathtt{ctype}(c),L_1) \approx (\mathtt{ctype}(c),L_2)} \qquad \frac{}{(\mathtt{itype},L_1) \approx (\mathtt{itype},L_2)}$$

$$\frac{(t_1,L_1) \approx (t_2,L_2) \quad (t_1',L_1) \approx (t_2',L_2)}{(\mathtt{ptype}(t_1,t_1'),L_1) \approx (\mathtt{ptype}(t_2,t_2'),L_2)}$$

図 2 クラスローダ制約 $\approx$ の同値関係への拡張
Fig. 2 The extension of the loader constraint $\approx$ to the equivalence relation.

$$\frac{}{C,L,t,s \vdash \mathtt{this} : t} \qquad \frac{}{C,L,t,[u/x] \vdash \mathtt{var}(x) : u} \qquad \frac{}{C,L,t,s \vdash \mathtt{iconst}(i) : \mathtt{itype}}$$

$$\frac{\begin{array}{l}\mathtt{load}(E,L,c) = \mathtt{SOME}(C',L') \\ C,L,t,s \vdash e : t' \\ (t',L) \approx (\mathtt{reptype}(C'),L')\end{array}}{C,L,t,s \vdash \mathtt{capp}(c,e) : \mathtt{ctype}(c)} \qquad \frac{\begin{array}{l}C,L,t,s \vdash e : \mathtt{ctype}(c) \quad C,L,t,s \vdash a : t' \\ \mathtt{load}(E,L,c) = \mathtt{SOME}(C',L') \\ \mathtt{lookup}(C',m,t') = \mathtt{SOME}(x,t',b,t'') \\ (t',L) \approx (t',L') \quad (t'',L) \approx (t'',L')\end{array}}{C,L,t,s \vdash \mathtt{mapp}(e,c,m,a,t') : t''}$$

$$\frac{C,L,t,s \vdash e_1 : t_1 \quad C,L,t,s \vdash e_2 : t_2}{C,L,t,s \vdash \mathtt{pair}(e_1,e_2) : \mathtt{ptype}(t_1,t_2)}$$

$$\frac{C,L,t,s \vdash e : \mathtt{ptype}(t_1,t_2)}{C,L,t,s \vdash \mathtt{proj1}(e) : t_1} \qquad \frac{C,L,t,s \vdash e : \mathtt{ptype}(t_1,t_2)}{C,L,t,s \vdash \mathtt{proj2}(e) : t_2}$$

図 3 静的意味論の帰納的定義
Fig. 3 Inductive definition of the static semantics.

れたクラスローダからロードすることにより，型式の意味である型値を求めることである．以下で定義される Λ' は，型式の評価関数に相当する．

関数 Λ から関数 Λ' を次のように定義する．Λ' は，型式に現れるクラス名をクラス定義とクラスローダで置き換えることにより，型式を型値に変換する．

Λ' : `loader*texp -> tvalue`

Λ' は以下のように定義される．

$$\begin{array}{l}\Lambda'(L,\mathtt{ctype}(c)) = \mathtt{ctvalue}(C',L') \\ \quad \mathtt{where}\ \Lambda(L,c) = (C',L') \\ \Lambda'(L,\mathtt{itype}) = \mathtt{itvalue} \\ \Lambda'(L,\mathtt{ptype}(t_1,t_2)) = \\ \quad \mathtt{ptvalue}(\Lambda'(L,t_1),\Lambda'(L,t_2))\end{array}$$

Λ' と拡張された $\approx$ の定義より，$(t,L_1) \approx (t,L_2)$ ならば，$\Lambda'(L_1,t) = \Lambda'(L_2,t)$ が成り立つ．

$v :: \tau$ は，値 v の型値が τ であることを意味し，図 4 にあげられた規則によって帰納的に定義される．

$$\frac{v :: \Lambda'(L,\mathtt{reptype}(C))}{\mathtt{cvalue}(C,L,v) :: \mathtt{ctvalue}(C,L)}$$

$$\frac{}{\mathtt{ivalue}(i) :: \mathtt{itvalue}}$$

$$\frac{v_1 :: \tau_1 \quad v_2 :: \tau_2}{\mathtt{pvalue}(v_1,v_2) :: \mathtt{ptvalue}(\tau_1,\tau_2)}$$

図 4 関係 :: の帰納的定義
Fig. 4 Inductive definition of the relation ::.

6. 型安全性

型安全性とは，式の評価において式のもつ型と値のもつ型が同一であるという性質であり，静的意味論と動的意味論とが合致していることをあらわしている．上で定義した Java の宣言的なサブセットにおける型安全性の命題は以下のとおりである．

次のことを仮定する．

- $\vdash E$
- $v :: \Lambda'(L,t)$
- $w :: \Lambda'(L,u)$
- $C,L,t,[u/x] \vdash e : t'$
- $C,L,v,[w/x] \vdash e \Downarrow v'$

このとき，$v' :: \Lambda'(L,t')$ が成り立つ．

この命題の証明は $C,L,v,[w/x] \vdash e \Downarrow v'$ の導出に関する帰納法による．

最後の規則が以下のような場合を考える．

$$\frac{\begin{array}{l}C,L,v,r \vdash e \Downarrow \texttt{cvalue}(C',L',v')\\ \texttt{load}(E,L,c) = \texttt{SOME}(C',L')\\ C,L,v,r \vdash a \Downarrow w\\ w \neq \texttt{error}\\ \texttt{lookup}(C',m,t') = \texttt{SOME}(x,t',b,t'')\\ C',L',v',[w/x] \vdash b \Downarrow v''\end{array}}{C,L,v,r \vdash \texttt{mapp}(e,c,m,a,t') \Downarrow v''}$$

また，$C,L,t,s \vdash \texttt{mapp}(e,c,m,a,t') : t''$ とすると，これを導出する規則は以下のものに限られる．

$$\frac{\begin{array}{l}C,L,t,s \vdash e : \texttt{ctype}(c)\\ C,L,t,s \vdash a : t'\\ \texttt{load}(E,L,c) = \texttt{SOME}(C',L')\\ \texttt{lookup}(C',m,t') = \texttt{SOME}(x,t',b,t'')\\ (t',L) \approx (t',L')\\ (t'',L) \approx (t'',L')\end{array}}{C,L,t,s \vdash \texttt{mapp}(e,c,m,a,t') : t''}$$

$C,L,v,r \vdash e \Downarrow \texttt{cvalue}(C',L',v')$ に対して帰納法の仮定を適用すると，

$\texttt{cvalue}(C',L',v') :: \Lambda'(L,\texttt{ctype}(c))$

である．$\texttt{load}(E,L,c) = \texttt{SOME}(C',L')$ であり，整合性より，$\Lambda(L,c) = (C',L')$ である．これと，Λ' の定義により，$\Lambda'(L,\texttt{ctype}(c)) = \texttt{ctvalue}(C',L')$ となる．すると，関係 :: の定義より，$v' :: \Lambda'(L',\texttt{reptype}(C'))$ が成り立つ．

また，$C,L,v,r \vdash a \Downarrow w$ に対して帰納法の仮定を適用すると，$w :: \Lambda'(L,t')$ が導かれる．$(t',L) \approx (t',L')$ より $\Lambda(L,t') = \Lambda(L',t')$ が成り立つので，$w :: \Lambda'(L',t')$ が導かれる．

$\vdash E$ より，$t = t'$ かつ

$C',L',\texttt{reptype}(C'),[t'/x] \vdash b : t''$

が成り立っている．従って，$C',L',v',[w/x] \vdash b \Downarrow v''$ に対して帰納法の仮定を適用することができ，$v'' :: \Lambda'(L',t'')$ が成り立つ．

最後に，$(t'',L) \approx (t'',L')$ より $\Lambda(L,t'') = \Lambda(L',t'')$ が成り立つので，$v'' :: \Lambda'(L,t'')$ が導かれる．

7. おわりに

本論文では，戸沢[13),14)]のおこなったクラスローダ制約の存在下でのJVMの型安全性を証明を再検討し，彼の議論を単純化することにより，クラスローダの役割を明確化をおこなった．理想的クラスローダ環境という概念を用いて，Javaの宣言的なサブセットの定式化を与え，型安全性を示した．

戸沢[13),14)]は，クラスローダ制約の定式化を行なうことにより，JDK1.2の問題点，特にバイトコード検証系のバグを発見した．これは，彼の定式化した理想的なモデルとJDK1.2の実装が微妙に食い違っているためであり，型安全性の厳密な検証を行なってはじめて発見することが可能であったと考えられる．形式的な方法論がプログラミング言語の設計や実装に実際に役に立った事例として興味深い．

しかし，彼の発見したバグは継承やインタフェースに関するものであり，本論文のサブセットでは扱うことができない．ただし，クラスローダ制約の役割は本質的には同じであり，本論文で与えたJavaのサブセットに継承やインタフェースを導入することにより解析が可能だと考えられる．

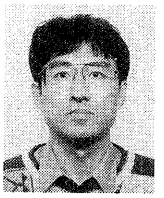
これまでにもJavaやJVMに対する定式化の研究がいくつかあったが，ローダ制約をともなったクラスローディングの機構を定式化したものはこれまでなく，それらが定式化の対象としたものは，基本的にはクラスローディングの機構とは独立性が高いと考えられる．もちろん，それら既存の定式化の研究の手法を導入することにより，本論文で与えたJavaのサブセットより拡張されたものに対して定式化を試みるのは興味深い．Nipkow[9)]は，Javaのソースプログラムレベルの意味論を与え，その型安全性を証明した．そこでは，継承やインターフェースなどの概念が形式化されている．彼らの研究では，クラスローダは定式化の対象ではなかった．彼らの研究でおこなわれている定式化により，本研究で与えたJavaのサブセットを拡張することは，今後の課題の一つである．Javaのバイトコードを定式化したものとしては，Qian[11)]やJensen[4)]による研究があった．特に，Jensen[4)]の研究では，クラスローダ制約は扱っていないものの，クラスローダの機能については定式化がなされていた．彼らの定式化の手法を用いることにより，本研究であつかったクラスローダの定式化をバイトコードレベルに拡張することが期待される．

参考文献

1) Dean, D.: *Formal Aspects of Mobile Code Security*, PhD Thesis, Princeton University (1999).
2) Dean, D., Felten, E. W., Wallach, D. S. and Balfanz, D.: Java Security: Web Browers and Beyond, *Internet Beseiged: Countering Cyberspace Scofflaws* (Denning, D. E. and Denning, P. J.(eds.)), ACM Press, pp. 241–269 (1997).
3) Gosling, J., Joy, B. and Steele, G.: *The*

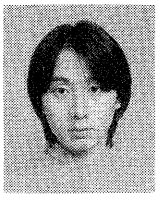
Java™ *Language Specification*, Addison-Wesley (1996).
4) Jensen, T., Metayer, D. L. and Thorn, T.: Security and Dynamic Class Loading in Java: A Formalisation, *Proceedings of IEEE International Conference on Computer Languages*, pp. 4–15 (1998).
5) Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *OOPSLA '98*, pp. 36–43 (1998).
6) Lindholm, T. and Yellin, F.: *The Java*™ *Virtual Machine Specification*, Addison-Wesley (1997).
7) Lindholm, T. and Yellin, F.: *The Java*™ *Virtual Machine Specification, Second Edition*, Addison-Wesley (1999).
8) McGraw, G. and Felten, E. W.: *Securing JAVA: Getting Down to Business with Mobile Code*, John Wiley & Sons (1999).
9) Nipkow, T. and von Oheimb, D.: Java$_{light}$ is Type-Safe — Definitely, *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 161–170 (1998).
10) Nishizaki, S.: Simply Typed Lambda Calculus with First-class Environments, *Publication of RIMS Kyoto Univ.*, Vol. 30, No. 6, pp. 1055–1121 (1995).
11) Qian, Z.: A Formal Specification of Java Virtual Machine Instructions, unpublished (1997). `http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html`.
12) Saraswat, V.: Java is not type-safe, unpublished (1997). Available from `http://www.research.att.com/~vj/bug.html`.
13) Tozawa, A. and Hagiya, M.: Careful Analysis of Type Spoofing, *JIT'99 Java-Informations-Tage 1999* (Clemens, H. and Cap, H.(eds.)), Springer, pp. 290–296 (1999).
14) Tozawa, A. and Hagiya, M.: New Formalization of the JVM, in preparation (1999). Draft available from `http://nicosia.is.s.u-tokyo.ac.jp/members/miles/paper/cl-99.ps`.

(平成 11 年 10 月 15 日受付)
(平成 12 年 2 月 22 日採録)



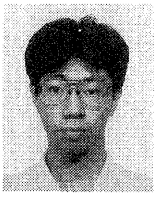
萩谷 昌己（正会員）

1957 年生．1982 年東京大学大学院理学系研究科情報科学専攻修士課程修了．京都大学数理解析研究所を経て現在東京大学大学院理学系研究科情報科学専攻教授．基本的に，演繹的推論を計算機上に実装することに興味をもっている．また，最近では，生命情報関連の研究（特に，分子計算）も行なっている．理学博士．



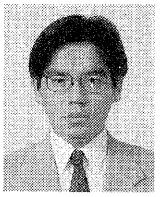
戸沢 晶彦

1973 年生．1998 年東京大学理学部情報科学科卒．1998 年より同大学大学院理学系研究科情報科学専攻修士課程在学中．Java の形式化に興味をもつ．



高橋 孝一

1963 年生．1986 年名古屋大学理学部数学科卒．1988 年同大学大学院理学研究科数学専攻修士課程修了．同年電子総合研究所入所．計算機による検証などに興味をもつ．日本ソフトウェア科学会会員．



西崎 真也（正会員）

1967 年生．1989 年京都大学理学部卒．1991 年同大学大学院理学研究科数理解析専攻修士課程修了．1994 年同大学大学院理学研究科数理解析専攻博士課程修了．1994 年岡山大学工学部助手，1995 年岡山大学大学院自然科学研究科助手，1996 年千葉大学理学部助教授，1998 年より東京工業大学大学院情報理工学研究科助教授．博士（理学）．プログラミング言語意味論，証明検証系上での形式化などに興味をもつ．日本ソフトウェア科学会，人工知能学会，ACM，EATCS 各会員．