多相型レコードに基づく Ruby オブジェクトの型推論に関する考察

松本 宗太郎　南出 靖彦
Soutaro MATSUMOTO　Yasuhiko MINAMIDE
筑波大学大学院 システム情報工学研究科
Graduate School of System and Information Engineering, University of Tsukuba
{soutaro,minamide}@score.cs.tsukuba.ac.jp

Ruby などのスクリプト言語が大規模なプログラム開発に用いられるようになりつつあり，型推論などの静的なプログラム検査が有用であると考えられる．我々は多相型レコードに基づき，Ruby オブジェクトの型推論アルゴリズムの設計と実装を行っている．我々の型システムでは，多相的なメソッドはオブジェクトに与えられた多相型で実現される．また，オブジェクトのインスタンス変数の型は，単相型変数によって多相性を制限される．一方で，Ruby クラスライブラリには再帰的なクラス定義が多相的に利用されるプログラムや，正規ではない型を持つ Array クラスといった，多相型レコードで表現できないオブジェクトが含まれている．これらのオブジェクトに型を付ける方法について考察する．

1 はじめに

これまでスクリプト言語は小さなプログラムの開発に利用されることがほとんどだったが，大規模なプログラムの開発での利用が増えつつある．しかし，スクリプト言語はもともと小規模な開発での利用を前提とした設計になっているため，静的な解析によって誤りを検出する機能がほとんど存在しない．またスクリプト言語の高い柔軟性は，精度の高い静的な解析を困難にしている．そのためスクリプト言語で開発されたプログラムは，実際に実行し挙動を確認するテストによって検査されるが，プログラムの大規模化にともなって十分なテストが困難となってきている．しかし，ある程度の精度を実現できれば，スクリプト言語に対しても静的な解析は有効であると考えられる．

本研究では，静的なプログラム検査の技術として型推論に着目した．型推論は，型の記述されていないプログラムに含まれる式の型を，文脈から構築する技術であり，同時に型の整合性を検査する．スクリプト言語としては，オブジェクト指向プログラミングを積極的にサポートする Ruby を対象とした．

Ruby オブジェクトは，メソッドをフィールドとみなすことによって，多相型レコードとして考えることができる．多相型レコードは，カインド (kind) を利用した多相レコード型によって型付けされる．本研究では Garrigue によるカインドを用いて，Ruby オブジェクトの型を表現する [1].

2 多相型レコードと Ruby オブジェクト

本研究では，Ruby オブジェクトを多相型レコードと考え，カインドを伴う型変数によってその型を表現している．カインドによる多相レコード型の表現と，多相レコード型による Ruby オブジェクトの型付けについて説明する．

2.1 多相型レコードとカインド

多相型レコードの型は，カインド付きの型変数で表現される．多相型レコードを含む ML プログラムと，大堀によるカインドを用いた型の例を示す [3].

$$\begin{aligned}
&\{\mathtt{name} ='' \mathtt{Soutaro}'', \mathtt{age} = 24\} \\
&\qquad : \{\mathtt{name} : \mathit{string}, \mathtt{age} : \mathit{int}\} \\
&\mathtt{fun\ x} \rightarrow \mathtt{x.age} + 1 \\
&\qquad : \alpha :: \{\mathtt{age} : \mathit{int}\} \rhd \alpha \rightarrow \mathit{int} \\
&\mathtt{let\ f} = \mathtt{fun\ x} \rightarrow \mathtt{x.age} + 1 \\
&\qquad \mathtt{f} : \forall \alpha :: \{\mathtt{age} : \mathit{int}\} \rhd \alpha \rightarrow \mathit{int}
\end{aligned}$$

式 $\{\mathtt{name} ='' \mathtt{Soutaro}'', \mathtt{age} = 24\}$ は，レコードを表現している．この式の型は $\{\mathtt{name} : \mathit{string}, \mathtt{age} : \mathit{int}\}$ となり，name というフィールドの型が string であることと，age というフィールドの型が int であることを示している．

式 $\mathtt{fun\ x} \rightarrow \mathtt{x.age} + 1$ は，引数 x の age フィールドの値に 1 を加算した値を返す関数である．この式の型に含まれる型変数 α が x の型を示している．$\{\mathtt{age} : \mathit{int}\}$ が型変数 α に与えられたカインドである．このカインドは，α に代入できるレコード型に，少

なくとも age という *int* 型のフィールドが定義されていなければならないことを表現している．

式 let f = fun x → x.age + 1 は，f を上の関数に束縛している．let によって束縛することで，関数の型は多相型となる．関数 f の引数は，少なくとも age という int 型のフィールドが定義されていればどのようなレコードでも良い．

2.2 Ruby オブジェクトの型

前節では，大堀によるカインドを用いて多相レコード型の例を示した．本研究では，Garrigue によるフィールドのマスクが可能なカインドを利用して，Ruby オブジェクトの型付けを行っている．

ここではカインドは (L, U, R) という三つ組となる．L は定義されているメソッドの集合であり，U は少なくとも定義されていなくてはならないメソッドの集合であり，R はメソッドとメソッドの型の関連付けである．

型 τ_f のメソッド f が定義されているオブジェクトの型は，次のように表現される．

$$\alpha :: (\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \tau_f\}) \rhd \alpha$$

ここで α がオブジェクトの型を示す型変数であり，$(\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \tau_f\})$ が α に与えられたカインドである．定義されているメソッドは $\{\mathtt{f}\}$ である．また，関連付け $\{\mathtt{f} : \tau_f\}$ から，f の型が τ_f であることがわかる．

型 τ_g のメソッド g が少なくとも定義されていなければならないオブジェクトの型は，次のように表現される．

$$\beta :: (\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \tau_g\}) \rhd \beta$$

集合 $\{\mathtt{g}\}$ が，少なくともメソッド g が定義されていなければならないことを示している．

実際に Ruby プログラムとその型の例を示す．次のプログラムはクラス A を定義する．クラス A にはメソッド f が定義されている．メソッド f は引数 x を持ち，x のメソッド g を呼び出す．

```
class A
  def f(x); x.g(); end
end
```

クラス A のインスタンスにはメソッド f が定義されている．またメソッド f の引数には，少なくともメソッド g が定義されていなければならない．このクラス A のインスタンスの型は次のようになる．

$$\begin{aligned}&\mathtt{A} :: (\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \alpha \to \beta\}),\\ &\alpha :: (\{\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \mathtt{unit} \to \beta\}) \rhd \mathtt{A}\end{aligned}$$

ここで A がオブジェクトの型を示す型変数である．A には f メソッドが定義されておりその型は $\alpha \to \beta$ であることが，カインドからわかる．このとき α もまたオブジェクトであり，α のカインドは $(\{\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \mathtt{unit} \to \beta\})$ である．このカインドはメソッド g が定義されていなければならないことを表現している．

Ruby ではクラス定義と同時に，クラスと同名の定数が定義される．クラスのインスタンス作成は，この定数の new メソッドを呼び出すことで行われる．以下のプログラムは，上で定義したクラス A のインスタンスを作成し，変数 a に代入している．

```
a = A.new()
```

定数 A の型は次のようになる．

$$\begin{aligned}&\mathtt{A} :: (\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \alpha \to \beta\}),\\ &\alpha :: (\{\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \mathtt{unit} \to \beta\}),\\ &\gamma :: (\{\mathtt{new}\}, \emptyset, \{\mathtt{new} : \mathtt{unit} \to \mathtt{A}\}) \rhd \gamma\end{aligned}$$

型の不整合は，(L, U, R) というカインドに対して，$U \not\subseteq L$ となったときに検出される．全てのカインドにおいて，定義されていなければならないメソッドの集合 U が，定義されているメソッドの集合 L に含まれていることが，型の整合性が保たれるための条件である．

2.3 多相型の導入

プログラムの意味を考えれば，先ほどのクラス A の f メソッドに多相型を与えることができる．メソッド f の引数とできるオブジェクトは，少なくともメソッド g が定義されていなければならないという制約を満たせば，どのような型でもよい．そこで，メソッドに多相型を与えた型を考える．

$$\begin{aligned}\mathtt{A} :: (\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \forall \alpha\, \beta.\ &\\ \alpha :: (\{\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \mathtt{unit} \to \beta\})&\\ \rhd \alpha \to \beta\}) \rhd \mathtt{A}&\end{aligned}$$

しかし，メソッドの型が多相になることを許すと，限量子が型の内側に表れる．このような限量子がネストする型を許す型推論は判定不能問題である．そこで，限量子をオブジェクトの型に対して付けた型を考えた．

$$\begin{aligned}&\forall \mathtt{A}\, \alpha\, \beta.\\ &\quad \mathtt{A} :: (\{\mathtt{f}\}, \emptyset, \{\mathtt{f} : \alpha \to \beta\}),\\ &\quad \alpha :: (\mathcal{L}, \{\mathtt{g}\}, \{\mathtt{g} : \mathtt{unit} \to \beta\}) \rhd \mathtt{A}\end{aligned}$$

メソッドの型自体は単相とし，オブジェクトの型を多相型とした．このとき，ML と同様に多相型が導入されるとすると，変数を参照する度に束縛型変数は新たな型へとインスタンス化されることになる．メソッドの型も変数参照の度に新しくインスタンス化されるため，メソッドの型自体を多相型にした場合と変わらない多相性が得られる．

2.4 単相型変数による多相性の制限

代入などの副作用を含む式に多相型を推論すると，型推論の健全性が失われることが知られている．副作用を許すプログラミング言語である ML では，値多相という形で多相型を推論してよい式と単相型に制限しなくてはいけない式を区別している．値多相は，構文的に式を値とそれ以外に区別し，値にのみ多相型を推論する方法である．関数や変数といった式が値とされ，関数適用は値とされない．

Ruby は副作用を許すため，代入などで多相性を制限する必要がある．しかし，値多相を利用すると，多相性が厳しく制限されすぎてしまう．メソッド呼び出しは関数適用と考えられるが，Ruby ではオブジェクトのインスタンス作成も `new` メソッドの呼び出しで行われるため，ほとんど全ての式の型が単相型に制限されてしまう．そこで，自由であったとしても多相型に束縛されない単相型変数 (monomorphic type variable) を導入し，明示的に多相性を制限する [4].

ループなどの構文を別に取り扱うことにすれば，ローカル変数への代入は新しい変数の束縛とみなせる．しかし，インスタンス変数への代入はオブジェクトの状態を更新することが目的であり，本質的に副作用を伴う．そこで，インスタンス変数の型を単相型変数とする．これでオブジェクトのインスタンス変数の型は，各オブジェクトに共通に定められる．

実際のプログラムで型付けの例を示す．

```
class B
  def set(x); @x = x; end
  def get(); @x; end
end
b = B.new()
```

クラス `B` は，インスタンス変数`@x` と，二つのメソッド `get`，`set` を持つ．メソッド `set` の引数として与えられたオブジェクトをインスタンス変数`@x` に代入し，メソッド `get` で`@x` の値を読み出す．変数 `b` の型は次のようになる．

$$\forall \mathtt{B}.\mathtt{B} :: (\{\mathtt{set}, \mathtt{get}\}, \emptyset, \{\mathtt{set} : \_\alpha \to \_\alpha, \mathtt{get} : \mathtt{unit} \to \_\alpha\}) \rhd \mathtt{B}$$

ここで $\_\alpha$ がインスタンス変数の型である．また，$\_\alpha$ は単相型変数であり，多相性を制限される．

しかし，このとき`@x` の型がクラス `B` の全てのインスタンスで共通になってしまう．これを防ぐため，クラス定義で暗黙に定義されるクラス名と同名の定数の型を一般化するときには，単相型変数も束縛する．この例では，定数 `B` の型は次の β になる．

$$\begin{aligned}&\forall \_\alpha\, \beta\, \mathtt{B}.\\ &\quad \mathtt{B} :: (\{\mathtt{set}, \mathtt{get}\}, \emptyset,\\ &\qquad \{\mathtt{set} : \_\alpha \to \_\alpha, \mathtt{get} : \mathtt{unit} \to \_\alpha\}),\\ &\quad \beta :: (\{\mathtt{new}\}, \emptyset, \{\mathtt{new} : \mathtt{unit} \to \mathtt{B}\}) \rhd \beta\end{aligned}$$

これで，オブジェクトの作成のために `new` メソッドを呼び出す度に，$\_\alpha$ が別々の型へと代入される．

3 型システム

Ruby オブジェクトを表現するための型の定義を示す．

τ	::=	α	型変数
	$\mid$	$\mathtt{unit}$	ユニット型
α	::=	β	多相型変数
	$\mid$	$\_\beta$	単相型変数
K	::=	$\alpha :: (L, U, R), \ldots$	カインド環境
R	::=	$\{m : \overline{\tau} \to \tau; \ldots\}$	
σ	::=	τ	単相型
	$\mid$	$\forall \alpha \ldots \alpha.\mathrm{K} \rhd \tau$	多相型

型 τ は型変数かユニット型である．Ruby では整数や文字列といった値も全てオブジェクトであるため，基底型は存在しない．オブジェクト型はカインド付き型変数で表現される．型変数は，通常の型変数と単相型変数に区別される．多相型は束縛型変数のカインド環境を含む．

Ruby の構文のうち，特に型について考慮する必要のあるサブセットを示す．

P	::=	$\mathcal{C} \ldots \mathcal{C}\, e$	プログラム
$\mathcal{C}$	::=	$\mathtt{class}\, C\, \overline{D}\, \mathtt{end}$	クラス定義
D	::=	$\mathtt{def}\, m(\overline{x})\, e\, \mathtt{end}$	メソッド定義
e	::=	x	ローカル変数
	$\mid$	$@x$	インスタンス変数
	$\mid$	C	定数
	$\mid$	$x = e; e$	ローカル変数代入
	$\mid$	$@x = e$	インスタンス変数代入
	$\mid$	$e.m(\overline{e})$	メソッド呼び出し

$$\frac{C : \forall B.\mathrm{K}_1 \rhd \alpha \in \Gamma \quad B = FV_{\mathrm{K}_1,\mathrm{K}}(\beta) \backslash FV_{\mathrm{K}}(\Gamma) \qquad \dfrac{\alpha :: (\{\mathtt{new}, \ldots\}, U, \{\mathtt{new} : \mathtt{unit} \to \beta, \ldots\}) \in \mathrm{K}_1}{\mathrm{K}, \mathrm{K}_1; \Gamma|_{\overline{C}}, C : \alpha, \mathtt{self} : \beta \vdash m \Rightarrow \mathrm{OK}}}{\mathrm{K}; \Gamma \vdash \mathtt{class}\ C\ m\ \mathtt{end} \Rightarrow \mathrm{OK}}\ \text{クラス定義}$$

$$\frac{\mathrm{K}; \Gamma, x : \tau_x \vdash e : \tau \quad \mathrm{K}; \Gamma \vdash \mathtt{self} : \alpha \quad \alpha :: (\{m, \ldots\}, U, \{m : \tau_x \to \tau, \ldots\}) \in \mathrm{K}}{\mathrm{K}; \Gamma \vdash \mathtt{def}\ m(x)\ e\ \mathtt{end} \Rightarrow \mathrm{OK}}\ \text{メソッド定義}$$

$$\frac{\mathrm{K}; \Gamma \vdash e_1 : \alpha \quad \mathrm{K}; \Gamma \vdash e_2 : \tau_2 \quad \alpha :: (L, \{\mathtt{m}, \ldots\}, \{\mathtt{m} : \tau_2 \to \tau, \ldots\}) \in \mathrm{K}}{\mathrm{K}; \Gamma \vdash e_1.\mathtt{m}(e_2) : \tau}\ \text{メソッド呼び出し}$$

$$\frac{\mathrm{K}, \mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{Dom}(\theta) \subseteq B}{\mathrm{K}; \Gamma, x : \forall B.\mathrm{K}_0 \rhd \tau \vdash x : \theta(\tau)}\ \text{ローカル変数} \qquad \frac{\mathrm{K}, \mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{Dom}(\theta) \subseteq B}{\mathrm{K}; \Gamma, C : \forall B.\mathrm{K}_0 \rhd \tau \vdash C : \theta(\tau)}\ \text{定数}$$

$$\frac{\mathrm{K}; \Gamma \vdash e_1 : \tau_1 \quad B = \mathrm{FV}_{\mathrm{K}}(\tau_1)|_{poly} \backslash \mathrm{FV}_{\mathrm{K}}(\Gamma) \quad \mathrm{K}|_{\overline{B}}; \Gamma, x : \forall B.\mathrm{K}|_B \vdash e_2 : \tau}{\mathrm{K}; \Gamma \vdash x = e_1;\ e_2 : \tau}\ \text{ローカル変数代入}$$

$$\frac{@x : \_\beta \in \Gamma}{\mathrm{K}; \Gamma \vdash @x : \_\beta}\ \text{インスタンス変数} \qquad \frac{\mathrm{K}; \Gamma \vdash e : \tau \quad \mathrm{K}; \Gamma \vdash @x : \tau}{\mathrm{K}; \Gamma \vdash @x = e : \tau}\ \text{インスタンス変数代入}$$

図 1: 型付け規則

プログラムは 0 個以上のクラス定義と式で構成される．クラス定義は 0 個以上のメソッド定義を含む．メソッドの可視性の指定は考慮しない．ここでは条件分岐やループなどの構文は省略する．クラス定義はクラスと同名の定数を定義し，オブジェクトはその定数の `new` メソッド呼び出しで生成される．クラス定義は相互再帰的でもかまわない．

型付け規則を図 1に示す．式の型判定は型環境 Γ とカインド環境 K の元で，$\mathrm{K}, \Gamma \vdash e : \tau$ という形になる．ローカル変数および定数の型は，型環境に含まれる多相型をインスタンス化して得られる．メソッド呼び出しでは，呼び出されるメソッドが式の型に定義されていなくてはならない．ローカル変数の代入では，ML の `let` 式と同様に，代入されたローカル変数の型は多相型になる．インスタンス変数の型は常に単相型変数となる．メソッド定義の型判定は，型環境 Γ とカインド環境 K の元で，$\mathrm{K}, \Gamma \vdash D \Rightarrow \mathrm{OK}$ という形になる．メソッド定義が型判定を得たときには，`self` にそのメソッドが定義されているというカインドが与えられている．クラス定義は暗黙の定数を定義し，定数にインスタンス作成のための `new` メソッドが定義される．単相型変数は，クラスと同時に定義される定数の型の一般化では束縛される．

4 型システムの問題点

3 節の型システムによって型付けを行うと，Ruby プログラムの意味と一致しない型が推論される場合がある．問題が発生するプログラム例と，現在検討している解決策を述べる．

多相再帰的なクラス定義

ML の型システムでは，再帰的に定義される値を，定義中で多相的に利用することができない．この制限から，クラス定義中で `self` を多相的に利用することができない．メソッドの多相性はオブジェクトの多相性として表現されることから，これは `self` のメソッドを多相的に利用できないことを意味する．`self` に多相型を与えるには，多相再帰 (polymorphic recursion) を許す型システムが必要である．

Ruby では，クラス名の定数の `new` メソッドの呼び出しによって，インスタンス作成が行われる．この定数もクラス定義によって定義される．そのため，相互再帰的なクラス定義中では，他のクラスのインスタンスまで常に単相型となる．この型システムの挙動が問題となるプログラムの例を示す．次のプログラムは組み込みの `String` クラスを拡張し，与えられた文字で自分自身を分割する `my_split` メソッドを定義する．

```
class String
  def my_split(sep)
    ret = Array.new(); a = '';
    self.each {|c|
      if c == sep
        ret.push(c); a = '';
      else
        a = a + c
      end
    }
  end
end
```

この `my_split` メソッドは分割の結果として配列を

返す．ここでは，String クラスのオブジェクトを要素とする，Array クラスのオブジェクトである．また Array クラスには，自分自身を文字列に変換する to_s メソッドが定義されており，ここで String クラスと Array クラスは相互再帰的な関係になる．

プログラムの意味としては，配列の要素の型は配列オブジェクト毎に定められている．しかし，定数 Array の型が単相型であることから，Array クラスの型や要素の型が単相型になる．そのため，上のプログラムを単純に解析すると，Array クラスの要素の型は String に固定されてしまう．多相再帰を許す型システムであれば，定数 Array の型が多相型になることから，この問題は生じず，Array クラスの要素の型はインスタンス毎に定まる．

多相再帰を許す型推論は判定不能問題であることが知られている．Mycroft によって提案されたアルゴリズムは，停止性が保証されていない [2]．しかし，停止性に関する条件を別途設けたうえでこのアルゴリズムを利用し，多相再帰を許す型推論を実装することを検討している．このアルゴリズムによって，my_split メソッドのような，多相再帰的なクラス定義を含むプログラムの型が推論できるようになると考えられる．

正規でない型が必要なクラス定義

前節で示した my_split メソッドは多相再帰を許す型推論アルゴリズムによって，取り扱うことができる．しかし，Array クラスに定義された map メソッドは，多相再帰を許したとしても型推論できない．このメソッドは，クロージャに相当する手続きオブジェクトを受け取り，配列のそれぞれの要素に手続きを適用した結果から構成される配列を返す．

この Array オブジェクトの型を擬似的に記述すると，次のようになる．

$$
\begin{aligned}
&\forall \alpha_0 \alpha_1 \alpha_2 \ldots \_\beta_1 \_\beta_2 \ldots . \\
&\quad \alpha_0 :: (L, U, \{\mathtt{map} : (\_\beta_0 \to \_\beta_1) \to \alpha_1, \ldots\}), \\
&\quad \alpha_1 :: (L, U, \{\mathtt{map} : (\_\beta_1 \to \_\beta_2) \to \alpha_2, \ldots\}), \\
&\qquad \vdots \\
&\quad \rhd \alpha_0
\end{aligned}
$$

Array オブジェクトの型は上の型に含まれる α_0 である．メソッド map の返り値の型 α_i は Array オブジェクトとなり，それぞれ map メソッドを含む．ここで map メソッドは意味的には多相型となるが，この型システムでは多相メソッドを許さない．そこで，メソッド自体の型は単相型としたうえで，オブジェクトの多相型として表現されるが，map メソッドの返り値が map メソッドを含むことから上の型は無限に続く．従って，束縛変数も無限個になるが，そのような束縛は不可能である．このことから多相レコード型によってオブジェクト型を表現する限り，Array オブジェクトの型は表現できない．

この問題は，map メソッドを含まない Array0 クラスを定義し，Array クラスの map メソッドの返り値の型は Array0 とすることで，不完全ながら解決できる．この場合，map の返り値として得られたオブジェクトに，さらに map を呼び出すことができないという問題が発生する．

5 まとめ

多相型レコードに基づいて，Ruby オブジェクトの型推論について考察した．オブジェクトの多相性によるメソッドの多相性の表現や単相型変数を利用したインスタンス変数の型付けによって，多相型を推論する型推論アルゴリズムの設計を行った．そのプロトタイプ実装を行い，簡単な Ruby プログラムの型検査の実験を行っている．

また，Ruby のライブラリに含まれる多相再帰的なクラス定義を解析する際に生ずる問題について考察した．多相再帰を許す型推論アルゴリズムと型を複製による再帰的な定義の解消により，問題は解決すると考えられ，現在実装を進めている．

型システムが複雑になることをできるだけ避けながら，推論の精度を向上させたい．実装を進め，実用上十分な精度を実現できたか検証を行う．

参考文献

[1] Jacques Garrigue. Simple type inference for structural polymorphism. In *9th Workshop on Foundations of Object-Oriented Languages*, 2002.

[2] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.

[3] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Prog. Lang. Syst.*, 17(6):844–895, 1995.

[4] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.