多相レコード型に基づくRubyプログラムの型推論

松本 宗太郎[†1] 南出 靖彦[†1]

本研究では，多相レコード型に基づいてRubyプログラムの型推論ツールを設計，実装した．型推論ツールは，組み込みライブラリの型を記述したシグネチャとRubyプログラムを入力とし，プログラムの型を推論し，誤りを検出する．しかし，Rubyの柔軟性を表現できる，実用的で健全な型体系を設計しようとすると，体系は非常に複雑になる．それを避けるため，多相レコード型によって拡張されたMLの型推論アルゴリズムを，直接，Rubyに適用した．型体系は，非常に制限されたRubyプログラムに対しては，健全になるように設計した．Rubyにおいては，組み込みクラスなどの既存のクラスを拡張することが許されており，実際に多くのプログラムで既存のクラスが拡張されている．このようなプログラムを処理するために，シグネチャとプログラムを分離して型推論するのではなく，プログラムの一部としてシグネチャが含まれるよう型体系を設計した．実際のRubyプログラムを型付けする場合には，多相レコード型の表現力やMLの型推論アルゴリズムにおける再帰的な定義の型推論に関する多相性の制限から，いくつかの問題が生じることが分かった．これらの制限は，特に組み込みライブラリの型付けにおいて問題になることから，クラス定義を複製し展開することによって型推論を行った．

Type Inference for Ruby Programs Based on Polymorphic Record Types

Soutaro Matsumoto[†1] and Yasuhiko Minamide[†1]

We design and implement a type inference tool for Ruby programs. The type system is based on polymorphic record types of Garrigue. The tool takes two inputs, a type signature of the built-in classes and a Ruby program, and then infers the type of the Ruby program and detect type errors. The type system is a direct adaptation of that of ML with polymorphic records, and designed to be sound only for a restricted tiny subset of Ruby. Ruby allows programmers to modify existing classes and many programs actually modify existing (built-in) classes. Thus we design our type system so that type signatures and method implementation coexist in a class definition. We encounter difficulties when typing common Ruby programs, since polymorphic methods are not expressible in our type system and the ML type inference does not infer polymorphic types in recursive definitions. We alleviate these difficulties by introducing transformations that duplicate class definitions.

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になってきている．一方で，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プログラム検査が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．

本研究では，プログラミング言語Ruby[4]によって記述されたプログラムの型推論を行うシステムを実装した．我々のシステムは，Rubyプログラムと組み込みクラスのシグネチャを入力とし，型推論を行う．Rubyでは，すべての値がオブジェクトである．また，組み込みクラスに対してメソッドを追加するなどの操

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

作が可能である．そのようなプログラムの型推論を可能にするため，シグネチャと Ruby プログラムを統合して型推論を行う．

型システムは，ML の型システムを基に設計した．オブジェクト型の表現には，多相レコード型を用いた．ML における多相レコード型は，Rémy による列変数を利用するものや，Ohori によるカインド（kind）を利用するものが，独立に開発されてきた[5),6)]．本システムでは，Ohori による多相レコード型の拡張として，Garrigue によって開発されたカインドによって多相レコード型を表現する[2)]．Garrigue によるカインドでは，再帰的なレコード型が容易に表現できる．

本研究では，実用的な Ruby プログラムの誤りを静的な解析によって検出するシステムの開発を目指している．実用的な Ruby プログラムに対して健全な型システムは非常に複雑になると考えられ，典型的なプログラムを取り扱えることが，理論的な正当性よりも優先した場合がある．型システムや型推論アルゴリズムについて，特に実装において，その健全性そのものは重要視していない．

2. プログラミング言語 Ruby

Ruby はオブジェクト指向プログラミング言語であり，すべての値はオブジェクトである．整数や文字列といった，プリミティブな値として扱われることが多い値もオブジェクトであり，`Integer` クラスや `String` クラスのインスタンスである．

メソッドは，def 構文によって定義する．メソッド定義の例を示す．

```
def succ(x)
  x+1
end
```

このプログラムではメソッド `succ` が定義される．このメソッドには，引数 `x` があり，メソッドの本体は `x+1` である．`x+1` という式は，`x` の+メソッドの呼び出しとして解釈される．そのため，実際の動作は引数 `x` の値に依存する．もしも，`x` として与えられたオブジェクトに+メソッドが定義されていない場合は，例外 `NoMethodError` が発生し，プログラムの実行中に型エラーが発生したことが通知される．これまでに代入されていないローカル変数に対する参照はエラーとなり，例外 `NameError` によって通知される．

クラスは，class 構文によって定義する．クラス定義には，メソッド定義が含まれる．

```
class A
  def initialize()
    @x = 0
  end

  def incr()
    @x = @x+1
  end
end

a = A.new
```

このプログラムは，クラス `A` を定義している．クラス `A` には 2 つのメソッド `initialize` と `incr` が定義されている．`initialize` メソッドは，インスタンス作成の際に呼び出されるメソッドである．クラス `A` の定義では，インスタンス変数`@x` を 0 で初期化している．Ruby では，`@`で始まる名前を持つ変数がインスタンス変数である．初期化されないインスタンス変数の値は，`nil` である．`incr` メソッドでは，インスタンス変数`@x` の値を，`@x+1` に更新している．また，このメソッドの返り値は，代入式がその右辺の値を返すことから，`@x+1` の値になる．インスタンスの作成は `A.new` として行う．これは，インスタンス作成の構文ではなく，定数 `A` の `new` メソッドの呼び出しである．クラス定義は，暗黙にクラスと同名の定数を定義し，インスタンス作成はその定数を通して行う．

クラス定義を行った際に，すでにクラスが存在していた場合は，既存のクラス定義に対する変更となる．先程のプログラムによってクラス `A` が定義されている状態で，次のようなクラス定義を行うとクラス `A` にメソッド `reset` が追加され，またメソッド `incr` の定義が置き換えられる．

```
class A
  def reset()
    @x = 0
  end

  def incr()
    @x = @x + 2
  end
end
```

このようなクラス定義の修正は，組み込みクラスに対しても可能であり，様々なプログラムで利用されている．

Ruby では，（イテレータ）ブロックと呼ばれる，制限された無名関数が利用できる．次のプログラムはブロックを利用するプログラムの例である．

```
class B
  def f()
    yield(1)
    yield(2)
    yield(3)
  end
end

b = B.new
b.f(){|x| puts x }
```

このプログラムではメソッド f がブロックを受け取るメソッドである．ブロックは {} で括られた部分であり，メソッド呼び出しの直後に表記することで，メソッドに与えることができる．ブロックを受け取ったメソッドは，yield 構文によってブロックを評価する．このプログラムは，f から，ブロックの引数にそれぞれ 1，2，3 が与えられてブロックが実行されるため，標準出力に 1，2，3 を順に出力する．ブロックは Ruby プログラムで頻繁に利用されている．配列に対応する組み込みクラスである Array には each というメソッドが定義されており，配列の要素を順にブロックに与えて評価を行う．

```
[1,2,3].each {|x| puts x }
```

このプログラムは，配列 [1,2,3] の each メソッドによって，配列の要素 1，2，3 を順に出力する．

3. Ruby オブジェクトと多相レコード型

3.1 カインドによる多相レコード型

多相レコード型は，型変数にカインドを与えることによって表現される．多相レコード型の表現力は，どのようなカインドを与えるかによって，変化する．ここでは ML プログラムを例に，Ohori によるカインドを用いた多相レコード型を示す．

```
{ name = "ruby"; age = 14 }
```

上のプログラムは，name というフィールドと age というフィールドを持つレコードを表している．それぞれのフィールドの値は"ruby"と 14 であり，それぞれの型は *string* と *int* と推論される．このレコードの型は，次のようになる．

$$\{\,\mathtt{name} : \mathit{string}, \mathtt{age} : \mathit{int}\,\}$$

中括弧 { } がレコードであることを示し，定義されているフィールドについて，フィールド名と型が記述される．

次のプログラムは，引数 x のフィールド name の値である文字列に対して，前後に文字列を連結した結果を返す関数 f を定義している．

```
let f x = "[" ^ x.name ^ "]"
```

関数 f は，name という文字列型のフィールドを持つ，任意のレコードに適用することができる．この関数 f の型は，次のようになる．

$$\forall\alpha\,.\,\alpha :: \{\,\mathtt{name} : \mathit{string}\,\} \triangleright \alpha \to \mathit{string}$$

型に含まれる，$\alpha :: \{\,\mathtt{name} : \mathit{string}\,\}$ はカインド環境である．カインド環境は，型変数と，その型変数に与えられたカインドを関連付ける．ここでは型変数 α にカインド $\{\,\mathtt{name} : \mathit{string}\,\}$ が与えられていることを表現する．このカインドは，name という *string* 型のフィールドが要求されていることを表現している．

ここまでで示した Ohori によるカインドは次のようなプログラムに型を与えることができない．

```
List.map (fun x -> x.name)
  [{ name="ruby"; age=14 };
   { name="MacBookPro"; weight=2.5 }]
```

このプログラムの意図は，リストの各要素の name フィールドの値のみを抽出したリストを求めることである．この意図からは，リストの各要素に name フィールドが定義されておりフィールドの型が等しければ，プログラムは正常に実行できると考えられる．

Ruby プログラムにおいては，このリストのようにそれぞれ型が異なる要素を格納した，異種混合的なコレクションが多用される．そのようなコレクションに対していっさい型を与えることができない場合，実用的な型推論システムを開発することは困難である．そこで Garrigue によって提案されたカインドによって，型付けを行う[2]．このカインドでは，異なるレコード型に対して，共通部分のみからなる共通の型を与えることができる．この場合，異種混合的なコレクションに対してその要素の共通部分を，要素の型とすることができる．コレクションのすべての要素に対して，一定の処理を行うプログラムについては，型付けが可能である．

Garrigue の型体系においては，レコード型はカインドを与えられた型変数で表現される．先ほどの $\{\,\mathtt{name} : \mathit{string}, \mathtt{age} : \mathit{int}\,\}$ という型を与えられたレコードは，次のような型を与えられる．

$$\alpha :: (\emptyset, \{\,\mathtt{name}, \mathtt{age}\,\}, \{\,\mathtt{name} : \mathit{string}, \mathtt{age} : \mathit{int}\,\}) \triangleright \alpha$$

カインドは，三つ組 (L, U, R) となる．L は，要求されているフィールドのフィールド名の集合である．U は，定義されているフィールドのフィールド名の集合である．R は，フィールド名とその型の対応を表現する．この型から，レコードに name と age という 2 つ

のフィールドが定義されていることが分かる．要求されているメソッドはないため，空集合 $\emptyset$ となる．また，関数 f の型は次のようになる．

$$\forall\alpha\,.\,\alpha :: (\{\,\mathtt{name}\,\}, \mathcal{L}, \{\,\mathtt{name} : \mathit{string}\,\}) \triangleright \alpha \to \mathit{string}$$

このカインドから関数の引数の型には，name フィールドが要求されていることが分かる．定義されているフィールドの集合に表れている $\mathcal{L}$ は，フィールド名の全集合を表している．定義されているメソッドに関する情報がない場合，空集合ではなく全集合を用いる．異種混合的なリストについては，その要素に共通のフィールドのみが定義されたレコード型が要素の型となる．

```
[{ name="Ruby"; age=24 };
 { name="MacBookPro"; weight=2.5 }]
```

上のリストの型は，要素であるレコードに共通するフィールドである name のみが定義されたレコード型を要素の型とするリスト型になる．

$$\alpha :: (\emptyset, \{\,\mathtt{name}\,\}, \{\,\mathtt{name} : \mathit{string}, \mathtt{weight} : \mathit{float}, \mathtt{age} : \mathit{int}\,\}) \triangleright \alpha\ \mathit{list}$$

このリストの要素には name というフィールドのみが定義されている．weight および age は，定義されたフィールドではないため，フィールドの型は保存されているが，アクセスはできない．

3.2 Ruby オブジェクトの型

次の Ruby プログラムは，クラス A を定義している．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，引数を 1 つとり，引数に 1 を足した結果を返す．

```
class A
  def f(x)
    x+1
  end
end
```

ここで x+1 は，x の+メソッドの呼び出しである．このとき A のインスタンスの型として次のような多相型が考えられる．

$$\forall\alpha_1,\alpha_2,\alpha_3\,.\ \alpha_1 :: (\emptyset, \{\,\mathtt{f}\,\}, \{\,\mathtt{f} : \alpha_2 \to \alpha_3\,\}),\ \alpha_2 :: (\{\,\mathtt{+}\,\}, \mathcal{L}, \{\,\mathtt{+} : \mathit{int} \to \alpha_3\,\}) \triangleright \alpha_1$$

インスタンスの型は α_1 であり，α_1 に与えられたカインドは，$(\emptyset, \{\,\mathtt{f}\,\}, \{\,\mathtt{f} : \alpha_2 \to \alpha_3\,\})$ である．このカインドから，α_1 にメソッド f が定義されており，その型が $\alpha_2 \to \alpha_3$ であることが分かる．メソッド f の引数の型は α_2 であるが，f の定義において引数のメソッドもまた呼び出されていることから，α_2 に対してもカインドが与えられている．カインドは，$\alpha_2 :: (\{\,\mathtt{+}\,\}, \mathcal{L}, \{\,\mathtt{+} : \mathit{int} \to \alpha_3\,\})$ であり，引数のオブジェクトにメソッド+が要求されていることが分かる．また+メソッドの型は，$\mathit{int} \to \alpha_3$ となる．int は，クラス Integer のインスタンスの型の省略表記とする．本来は，Ruby においては整数もまたオブジェクトであり，Integer のインスタンスであるが，ここで Integer のインスタンスの型を表記すると煩雑になるため，省略表記を利用する．同様に，*string* などを String などの略記とする．

Ruby におけるクラス C のインスタンス作成は C.new として行う．次のプログラムは，クラス A のインスタンスを作成し，ローカル変数 x に代入する．

```
x = A.new()
```

インスタンス作成式 A.new() の型は，クラス A のインスタンスの型として推論された多相型をインスタンス化した型である．また，変数 x の型は，右辺の型を一般化して得られる，次の多相型である．

$$\forall\alpha_1,\alpha_2,\alpha_3\,.\ \alpha_1 :: (\emptyset, \{\,\mathtt{f}\,\}, \{\,\mathtt{f} : \alpha_2 \to \alpha_3\,\}),\ \alpha_2 :: (\{\,\mathtt{+}\,\}, \mathcal{L}, \{\,\mathtt{+} : \mathit{int} \to \alpha_3\,\}) \triangleright \alpha_1$$

ローカル変数への代入は，ML における let 式と同様に取り扱う．右辺の型を一般化した多相型が変数の型となる．また，代入式自体の型は，右辺の型と等しい．$x = e; e'$ という式は，$\mathtt{let}\ x = e\ \mathtt{in}\ e'$ と同様に取り扱う．また，$x = e$ という式は，$\mathtt{let}\ x = e\ \mathtt{in}\ x$ とする．

クラス A の定義からは，メソッド f の型として多相型を考えることができる．Ruby においては，すべての値がオブジェクトであることからメソッドの型は型変数を含むことになり，多相型へと一般化することが可能である．しかし，本研究では ML の型推論アルゴリズムを利用するため，メソッドの型としては多相型を認めない．多相的なメソッドの型は，メソッドの型自体は単相型とし，オブジェクトの型を多相型とすることで表現する．

再帰的なクラス定義においては，カインドを通して再帰型が表現される．次のプログラムのクラス B は，self を返す get_self メソッドが定義されている．

```
class B
  def get_self()
```

```
    self
  end
end
```

このとき，クラス B のインスタンスの型は次のようになる．

$$\forall\alpha\,.\,\alpha :: (\emptyset, \{\,\texttt{get\_self}\,\},\\ \{\ \texttt{get\_self}\ : \mathit{unit} \to \alpha\,\}) \triangleright \alpha$$

クラス B の定義中では，self の型は α であり，型推論の結果もそのようになる．ここで *unit* は，ML の型システムにおける *unit* と同様に，引数のないメソッドにおける引数の型を表現するためのダミーの型であるとする．

3.3 インスタンス変数を含むオブジェクト

次のプログラムはインスタンス変数を含むクラス C を定義する．2 つのメソッド get と set が定義されており，それぞれインスタンス変数@x の値の取得と設定を行う．

```
class C
  def get()
    @x
  end

  def set(x)
    @x = x
  end
end
```

このときクラス C のインスタンスの型は次のようになる．

$$\forall\alpha, \_\beta\,.\,\alpha :: (\emptyset, \{\,\texttt{get}, \texttt{set}\,\}\\ \{\,\texttt{get} : \mathit{unit} \to \_\beta,\\ \texttt{set} : \_\beta \to \_\beta\,\}) \triangleright \alpha$$

インスタンス変数@x の型は $\_\beta$ である．インスタンス変数の型には命令的型変数が与えられ，通常の型変数と区別される．インスタンス変数は，オブジェクトの状態を保存し，代入によって更新されるため，多相性を制限しなくてはならない．通常，ML では多相性の制限は値多相によって行われる[8]．値多相では，式が値式とそれ以外に分類され，let の右辺が値式である場合のみ多相型が推論される．Ruby ではプログラム中における値は new メソッドの呼び出しによって導入される．値多相を利用した場合，メソッド呼び出しが値式でないことから，多相型がいっさい推論されないことになる．そこで，値多相が導入される以前に Standard ML において利用されていた，命令的型変数を利用した型付けを用いる．これは，Tofte によって提案された手法で，型変数を適用的型変数と命令的型変数の 2 種類に分類し，適用的型変数は通常の型変数と同様に扱う一方で，命令的型変数について多相性を制限する[7]．参照の更新といった副作用を生じる操作について，命令的型変数を利用し型システムの健全性を保つ．

次のプログラムではクラス C がインスタンス変数を含むことから，変数 y の型として推論される多相型において一般化されない命令的型変数が含まれる．

```
y = C.new()
```

y の型は次のようになる．

$$\forall\alpha\,.\,\alpha :: (\emptyset, \{\,\texttt{get}, \texttt{set}\,\},\\ \{\,\texttt{get} : \mathit{unit} \to \_\beta_1,\\ \texttt{set} : \_\beta_1 \to \_\beta_1\,\}) \triangleright \alpha$$

通常の型変数である α は多相型の束縛変数に含まれるが，命令的型変数である $\_\beta_1$ は束縛されない．他の操作によって，インスタンス変数@x の型にカインドが与えられた場合，その制約は保存され，他の型に伝搬する．

```
x = A.new
y.set(x)
```

先ほどのプログラムに続いてこのプログラムを入力し，型推論を行った場合，y の型は次のようになる．

$$\forall\alpha\,.\\ \alpha :: (\emptyset, \{\,\texttt{get}, \texttt{set}\,\},\\ \{\,\texttt{get} : \mathit{unit} \to \_\beta_1,\\ \texttt{set} : \_\beta_1 \to \_\beta_1\,\}),\\ \_\beta_1 :: (\emptyset, \{\,\texttt{f}\,\}, \{\,\texttt{f} : \_\beta_2 \to \_\beta_3\,\}),\\ \_\beta_2 :: (\{\,\texttt{+}\,\}, \mathcal{L}, \{\,\texttt{+} : \mathit{int} \to \_\beta_3\,\}) \triangleright \alpha$$

ここでは，@x の型 $\_\beta_1$ はカインドによって制約を受ける．メソッドを経由して行われる@x への操作は，$\_\beta_1$ 型に反映される．

4. シグネチャ定義

組み込みライブラリなどは，Ruby プログラムによる実装を持たないため，そのメソッドの型を推論できない．そこで，シグネチャによってメソッドの型を宣言することにより，インスタンスの型を与える．メソッド型の宣言は def m :: C → C という形式で行う．メソッド名について，引数の型と返り値の型に対応するクラス名を記述して，宣言する．

以下に示すシグネチャは，クラス A の型を記述している．

```
class A
  def f :: unit -> A
end
```

このシグネチャから，クラス A のインスタンスの型として，次の多相型が得られる．

$$\forall\alpha\,.\,\alpha :: (\emptyset, \{\,\mathtt{f}\,\}, \{\,\mathtt{f} : \mathit{unit} \to \alpha\,\}) \triangleright \alpha$$

定義されたメソッドの集合は，f のみからなる集合である．メソッド f の定義より，引数の型が unit で返り値の型が A であることが分かる．シグネチャに返り値の型として記述されている A から，その型は A のインスタンスの型 α と等しい．

メソッドの引数の型を記述したシグネチャを考える．

```
class B
  def g :: A -> B
end
```

このクラス B のインスタンスの型は，次のようになる．

$$\forall\alpha,\beta\,.\,\alpha :: (\emptyset, \{\,\mathtt{g}\,\}, \{\,\mathtt{g} : \beta \to \alpha\,\}),\\ \beta :: (\{\,\mathtt{f}\,\}, \mathcal{L}, \{\,\mathtt{f} : \mathit{unit} \to \beta\,\}) \triangleright \alpha$$

クラス B のインスタンスの型は α である．メソッド g の引数の型は A であるが，g の引数の型 β に与えられたカインドは上に示した A のインスタンスの型とは異なる．ここでは，メソッド f は，要求されるメソッドの集合に含まれており，またその型も変化している．このように，シグネチャから型を構築する際には，2 種類の型が必要になる．

2 種類の型は，クラス名が出現する位置の正負に対応する．正の位置に対応する型はそのインスタンスの型と等しい．負の位置に対応する型は，要求されるメソッドの集合が，そのクラスに定義されているメソッドすべてとなる．正の位置に対応する型はインスタンスの型としてプログラムの型推論で利用されるが，負の位置に対応する型はシグネチャからの型の構築のみに利用されプログラムの型推論には利用されない．

4.1 パラメータ化されたクラス

配列の型は，その要素の型によってパラメータ化されている．このようなパラメータ化されたクラス定義は，型抽象と型適用をシグネチャに記述することによって表現する．

配列に対応する Array クラスのシグネチャは次のように定義する．

```
class Array<'_a>
  def push :: '_a -> Array<'_a>
  def pop :: unit -> '_a
  ...
end
```

Java 風の表記によって，クラス名に対するパラメータを宣言する．ここでは'_a という型変数をパラメータとして宣言している．Array クラスの push メソッドは，配列に要素を追加し，配列自身を返すメソッドである．引数の型は，格納するオブジェクトの型であり，返り値はその型を型適用した Array クラスである．pop メソッドは，配列から要素を取得する．返り値の型は，配列の要素の型である．この Array クラスのシグネチャは，他のクラスのシグネチャで利用される．

```
class C
  def f :: unit -> Array<Integer>
  def g :: unit -> Array<String>
end
```

このシグネチャでは，f メソッドの返り値が要素が Integer の配列であること，g メソッドの返り値が要素が String の配列であることが表現されている．プログラムから Array クラスを利用する場合には，型パラメータに関する特別な処理は行われない．

```
a = Array.new
a.push 3
```

型パラメータはこのプログラムには現れないが，a.push 3 というメソッド呼び出しから push メソッドの型が推論され，Array クラスのシグネチャに含まれる型変数'_a に対応する型には，Integer のインスタンスの型が代入される．

パラメータ化されたクラス定義は，シグネチャ自体を変換することで処理される．先ほどの，クラス A の定義における Array<Integer>から，Array の定義中の'_a をすべて Integer に置き換えた次に示すような定義が生成される．

```
class Array#Integer
  def push :: Integer -> Array#Integer
  def pop :: unit -> Integer
  ...
end
```

さらに，クラス C のシグネチャ定義に含まれる Array<Integer>が Array#Integer に置換される．

```
class C
  def f :: unit -> Array#Integer
  def g :: unit -> Array<String>
end
```

メソッド g の定義に含まれる Array<String>も同様に処理される．

型適用は，シグネチャの変換によって処理され，型

システムでは考慮されない．これは，Ruby の組み込みライブラリに含まれる，多相再帰的に定義されたクラスの型推論を行うためである．本型システムは ML の型システムに基づいており，再帰的に定義された値を，定義中で多相的に利用することはできない．型システムによって型適用を取り扱った場合，この制限から，厳しすぎる制約が生成されることがある．

具体的な例では，Ruby の組み込みクラスである String，Integer，Array は相互再帰的に定義されている．それぞれのシグネチャの抜粋を示す．

```
class String
  def size :: unit -> Integer
  def split :: Regexp
                -> Array<String>
  ...
end

class Integer
  def to_s :: unit -> String
  def divmod :: Integer
                 -> Array<Integer>
  ...
end

class Array<'_a>
  def to_s :: unit -> String
  def size :: unit -> Integer
  ...
end
```

ここで，String クラスの定義中に Array<String>が出現し，Integer クラスの定義中に Array<Integer>が出現することから，Array クラスは多相再帰的に定義されている．この定義を型推論すると，String および Integer クラスの定義を型推論する過程において Array クラスのインスタンスの型は単相型となり，Array<String>と Array<Integer>は同一の型となる．このとき，2 つのクラス String と Integer のインスタンスの型は，同一の型として推論され，String と Integer に共通のメソッドのみが定義された型となる．そのため，String にのみ定義されたメソッドを利用するプログラムを型推論した場合に型エラーを誤検出することとなり，実用的な型推論システムとはならない．また，配列の要素の型についても，String と Integer 両方のクラスに定義されたすべてのメソッドが要求される．シグネチャ定義の操作によって型適用を処理することによって，Array<String>と Array<Integer>は無関係な 2 つの型として取り扱われるため，このような問題を回避できる．

4.2 正則な型で表現できないメソッドの近似

Ruby の配列には，map メソッドが定義されている．このメソッドは，ブロックを受け取り，そのブロックを評価した結果を要素とした配列を返り値とする．

```
class Array<'_a>
  def map :: &{ '_a -> '_b }
              -> Array<'_b>
  ...
end
```

ここでブロックの型は，&{ } の内側に引数と返り値の型を記述することで表現した．この map メソッドの型は本システムにおいて表現できない．型を擬似的に表現するならば，次のようになる．

$$\alpha :: (L, U, \{\texttt{map} : \forall \_\beta'.(\_\beta \to \_\beta') \to \alpha[\_\beta'/\_\beta], \ldots\}) \triangleright \alpha$$

ここで $\_\beta$ がもともとの配列の要素の型であり，$\_\beta'$ は map メソッドを呼び出した結果の配列の要素の型である．map の返り値の型は，配列の型 α の要素の型 $\_\beta$ を新しい要素の型 $\_\beta'$ で置き換えた型となる．しかし，この型には多相再帰的で，なおかつ多相メソッドが出現している．我々の型システムでは多相メソッドが許されていないため，このような型を表現しようとすると，無限にパラメータの展開を行わなければならない．

一方で，このような型を許さない場合には，map メソッドの返り値の型が引数と同じになるように近似することにより，次のような型を推論できる．

$$\alpha :: (L, U, \{\texttt{map} : (\_\beta \to \_\beta) \to \alpha, \ldots\}) \triangleright \alpha$$

この型は我々の型システムで表現が可能であるが，Ruby における map メソッドの意味とは大きく異なる制限されたものである．map メソッドを用いて配列を別の型のオブジェクトの配列に変換するような操作は頻繁に行われるため，この制限は許容しがたいものと考えられる．

次のプログラムは配列 [1,2,3] をそれぞれ to_s メソッドによって文字列に変換したあと，末尾に文字列の 0 を付加し，さらに to_i メソッドによって整数に変換している．

```
[1,2,3].map{|x|
  x.to_s
}.map{|x|
  (x+"0").to_i
}
```

ここで [1,2,3] の要素の型 Integer と，最初の map を適用した結果の型 String が等しいとすると，map の結果の配列の型が Integer と String の共通部分となる．その場合，Integer クラスのオブジェクトに定義された+メソッドは引数として数値をとらなくてはならないことから，2 番目の map の適用の時点で型エラーが誤検出される．

そこで，map に関する再帰を展開することによって，この制限を緩和する．上記のシグネチャ定義における再帰を展開し，次のようなシグネチャを得る．

```
class Array#0<'_a>
  def map :: &{ '_a -> '_a }
              -> Array#0<'_a>
  ...
end

class Array#1<'_a, '_b>
  def map :: &{ '_a -> '_b }
              -> Array#0<'_b>
  ...
end
```

このとき，配列として Array の代わりに Array#1 を用いれば，1 回目の map 呼び出しについては，結果の配列の要素の型を別の型に変更することが可能である．再帰的なメソッド定義などにおいて，この展開では多相性が不足する場合があるが，ここで示したような例においては必要な多相性を確保できる．

4.3 型推論との統合

ここまでで説明したシグネチャには，シグネチャによるメソッド定義のみが含まれていた．実際には，シグネチャによって定義されたクラスと同一のクラスについて，Ruby プログラムによるメソッド定義が含まれる可能性がある．Ruby プログラムとの統合は，同じクラスに含まれる Ruby プログラムによるメソッド定義とシグネチャによるメソッド定義を，両方含むクラスを生成することにより行われる．

このとき，メソッドの引数などの負の位置に対応する型は，シグネチャによって定義されているメソッドのみを対象として構築する．一方で，メソッドの返り値などの正の位置に対応する型は，シグネチャだけでなくプログラムによって定義されるメソッドも含む型として構築する．

返り値の型が Ruby プログラムで定義されたメソッドも含むことにより，次のようなプログラムを扱うことができる．

```
class String
  def f()
    "string"
  end
end

1.to_s.f()
```

ここで，Integer クラスの to_s メソッドは，$unit \to$ String という型でシグネチャによって定義されているとする．String クラスの f メソッドは，組み込みライブラリには定義されておらず，このプログラムにおいて新たに定義されたメソッドである．ここで，返り値の型 String がシグネチャのみで表現されるとすると，この新しいメソッド f の定義は to_s メソッドの返り値に含まれないことになる．その場合は，この Ruby プログラムを型推論した際に，f メソッドの呼び出しが型の不整合として検出されることになる．このような誤検出を防ぐため，正の位置に対応する型には，プログラムによって定義されたメソッドを含まなくてはならない．

本システムの設計では，シグネチャによって型が与えられたメソッドには，Ruby プログラムによる実装が与えられないことを仮定している．その場合，それらのメソッドが，Ruby プログラムによって追加されたメソッドを呼び出すことはない．そのため，負の位置の型に対する制約として，Ruby プログラムによって定義されたメソッドの要求を含む必要はない．

5. 型システム

Ruby のサブセットをモデル化した言語について，型システムを定義する．型システムは Garrigue による型システムに基づいている．プログラミング言語として Ruby を採用したことから構文が変更され，また型変数が適用的型変数と命令的型変数に分類されている．

本型システムにおける制約ドメインには，Garrigue による「フィールドをマスク可能なレコード（Records with maskable fields）」について与えられた制約ドメインを用いた．制約 C は，組 (L,U) で表現される．L は要求されるメソッドの名前の集合であり，U は定義されたメソッドの名前の集合あるいはメソッド名の全集合 $\mathcal{L}$ である．含意関係（entailment relation）は，次のように定義される．

$$(L,U) \models (L',U') \Leftrightarrow L \supseteq L' \wedge U \subseteq U'$$

カインドの妥当性は，L と U の包含関係によって検査される．$L \not\subseteq U$ の場合妥当でないカインドとなり，型推論において単一化が失敗する．

$$
\begin{array}{rcll}
p & ::= & \{\,\overline{c}\,\}, p & \text{プログラム} \\
 & | & e & \\
c & ::= & \texttt{class}\ \mathrm{C}\ \overline{\texttt{@x} : u}, \overline{m}\ \texttt{end} & \text{クラス定義} \\
m & ::= & \texttt{def}\ \mathrm{m}(x)\ e\ \texttt{end} & \text{メソッド定義} \\
 & | & \texttt{def}\ \mathrm{m} :: \mathrm{K} \triangleright \alpha \to \mathrm{C} & \text{メソッド型定義} \\
e & ::= & x & \text{ローカル変数} \\
 & | & @x & \text{インスタンス変数} \\
 & | & \mathrm{C}.\texttt{new} & \text{インスタンス作成} \\
 & | & e.\mathrm{m}(e) & \text{メソッド呼び出し} \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{let 式} \\
 & | & @x = e & \text{インスタンス変数代入}
\end{array}
$$

図 2 構文規則
Fig. 2 Syntax rules.

$$
\begin{array}{rcll}
\alpha & ::= & t & \text{適用的型変数} \\
 & | & u & \text{命令的型変数} \\
R & ::= & \{\,\overline{r(\mathrm{m}, \alpha \to \alpha)}\,\} & \\
\mathrm{K} & ::= & \alpha :: (C, R), \mathrm{K} & \text{カインド環境} \\
\sigma & ::= & \forall \overline{\alpha}\,.\,\mathrm{K} \triangleright \alpha & \text{多相型} \\
 & | & \alpha & \text{単相型} \\
\Gamma & ::= & x : \sigma, \Gamma & \text{型環境} \\
 & | & @x : u, \Gamma & \\
\Delta & ::= & \mathrm{C} : \sigma, \Delta & \text{クラス環境}
\end{array}
$$

図 1 型の定義
Fig. 1 Types.

型の定義を**図 1** に示す．単相型は型変数 α のみからなる．オブジェクト以外の値が存在しないことから，基底型や関数型は含まれない．型変数は，適用的型変数 t と命令的型変数 u の 2 種類に区別される．*AppTyVar* をすべての適用的型変数の集合とし，*ImpTyVar* をすべての命令的型変数の集合とする．カインドは組 (C, R) である．C は制約ドメインの要素であり，メソッドに関する制約を表現する．R は関係述語（relating predicate）の集合である．R に述語 $r(\mathrm{m}, \alpha \to \beta)$ が含まれていたとき，メソッド m の引数の型が α であり返り値の型が β である．K はカインド環境であり，$\alpha :: (C, R)$ が含まれていたとき，型変数 α に与えられたカインドは (C, R) である．あるカインド環境においては，1 つの型変数についてたかだか 1 つのカインドが与えられる．多相型 σ は，標準的な定義を，束縛変数に対するカインド環境によって拡張したものである．型環境 Γ は，変数とその型の対応を表現する．インスタンス変数の型は，つねに命令的型変数 u である．クラス環境 Δ は，クラス名とそのインスタンスの型の対応を表現する．

自由型変数の定義を以下に示す．

$$
\begin{aligned}
\mathrm{FV}_{\mathrm{K}}(\forall \overline{\alpha}.\mathrm{K}' \triangleright \alpha) &= \mathrm{FV}_{\mathrm{K},\mathrm{K}'}(\alpha) \backslash \{\,\overline{\alpha}\,\} \\
\mathrm{FV}_{\mathrm{K},\alpha::(U,L,R)}(\alpha) &= \{\,\alpha\,\} \cup \mathrm{FV}_{\mathrm{K}}(R) \\
\mathrm{FV}_{\mathrm{K}}(\alpha) &= \{\,\alpha\,\}
\end{aligned}
$$

この定義は，Garrigue による定義と等しい．標準的な多相型の自由型変数の定義と同様であるが，カインド環境 K によって拡張されている．

カインド環境に対して，許容可能な代入（admissible substitution）を定義する．代入 θ がカインド環境 K と K′ に対して許容可能であるとき $\mathrm{K} \vdash \theta : \mathrm{K}'$ とする．$\mathrm{K} \vdash \theta : \mathrm{K}'$ となるのは，すべての $\alpha :: (C, R) \in \mathrm{K}$ について，$\theta(\alpha) :: (C', R') \in \mathrm{K}'$ であり，次の条件を満たすときである．

(1) $\alpha \in \mathit{ImpTyVar} \Rightarrow \theta(\alpha) \in \mathit{ImpTyVar}$
(2) $C' \models C$
(3) $\theta(R) \subseteq R'$

この定義は，Garrigue による定義に命令的型変数に関する条件 (1) を追加したものである．命令的型変数から適用的型変数への代入が発生しないように，条件 1 によって制限している．直感的には $\mathrm{K} \vdash \theta : \mathrm{K}'$ であった場合，K においてカインドで制約が与えられた型 α について，代入の結果 $\theta(\alpha)$ には，K′ においてより厳しい制約が与えられる．

構文規則の定義を**図 2** に示す．プログラムはクラス定義群 $\{\,\overline{c}\,\}$ の列と式からなる．クラス定義群は，相互再帰的なクラス定義によって構成される．またクラス定義群の列は依存関係に従って整列されているものとする．クラス定義には，インスタンス変数の宣言とメソッド定義が含まれる．プログラムによるメソッド定義は，メソッド名と引数と本体の式からなる．メソッド型定義では，多相レコード型とクラス名によってメ

ソッドの型を表現する．引数の型は多相レコード型であるが，返り値の型はクラス名となる．式には，ローカル変数，インスタンス変数，インスタンス作成，メソッド呼び出し，let 式，インスタンス変数への代入がある．変数はローカル変数とインスタンス変数に分類されており，インスタンス変数の名前は @ で開始する．ローカル変数への代入はなく，let 式によって表現されるとする．Ruby とこのモデル言語の主な違いとして，ローカル変数への代入が存在しない点とインスタンス作成が構文として独立している点がある．Ruby におけるローカル変数の代入は let 式へと変換され，インスタンスの作成はメソッド呼び出しではなく構文として扱われる．

関数 $l(m)$ および $l(c)$ は，メソッド名およびクラス名をメソッド定義あるいはクラス定義から取得する関数である．定義は以下のようになる．

$$\begin{aligned} l(\texttt{def}\ \mathrm{m}(x)\ e\ \texttt{end}) &= \mathrm{m} \\ l(\texttt{def}\ \mathrm{m} :: \mathrm{C}_1 \to \mathrm{C}_2) &= \mathrm{m} \\ l(\texttt{class}\ \mathrm{C}\ \overline{m}\ \texttt{end}) &= \mathrm{C} \end{aligned}$$

単相型 α と多相型 σ の関係 $\alpha \sqsubseteq_{\mathrm{K}} \sigma$ を定義する．

$$\begin{aligned} &\alpha \sqsubseteq_{\mathrm{K}} \beta = \exists\theta\,.\,(\mathrm{K} \vdash \theta : \mathrm{K} \wedge \alpha = \theta(\beta)) \\ &\alpha \sqsubseteq_{\mathrm{K}} \forall B.\mathrm{K}' \triangleright \beta = \\ &\quad \exists\theta\,.\,\mathrm{K}, \mathrm{K}' \vdash \theta : \mathrm{K} \wedge \alpha = \theta(\beta) \wedge \mathrm{Dom}(\theta) \subseteq B \end{aligned}$$

$\alpha \sqsubseteq_{\mathrm{K}} \sigma$ が成立するとき，α は σ の例である．標準的な例の定義に，カインド環境に関する条件が加わった定義である．

図 2 に示した言語における型付け規則を，**図 3** に示す．式の型判定は $\mathrm{K};\Delta;\Gamma \vdash e : \alpha$ という形になる．この型判定が導出された場合，カインド環境 K およびクラス環境 Δ，型環境 Γ のもとで式 e が型 α を持つ．メソッド定義の型判定は $\mathrm{K};\Delta \vdash m : \alpha \to \beta$ となる．この型判定が導出された場合，カインド環境 K およびクラス環境 Δ，型環境 Γ のもとで，メソッド $l(m)$ は型 $\alpha \to \beta$ を持つ．メソッド定義の型判定における型環境は，インスタンス変数の型のみを保持し，ローカル変数の型を保持しない．クラス定義の型判定は $\mathrm{K};\Delta \vdash c : \alpha$ となる．この型判定が導出された場合，カインド環境 K とクラス環境 Δ のもとで，クラス $l(c)$ のインスタンスの型は α に含まれるすべての自由型変数を束縛変数とする多相型となる．プログラム全体の型判定は $\Delta \vdash p : \alpha$ となる．この型判定が導出されたとき，クラス環境 Δ のもとで，プログラム p は型 α を持つ．

変数の参照に関する規則は，通常の ML と同様である．環境から変数に対応する多相型を取得し，インスタンス化した型を式の型とする．メソッド呼び出しでは，メソッドの型に関する制約をカインドによって表現する．インスタンス変数への代入では，型環境に保存されているインスタンス変数の型と，右辺の型が等しい．

多相型は，プログラムの型付けに関する 2 番目の規則 $\Delta \vdash \{\overline{c}\}, p : \alpha$ と let 式の型付けに関する規則 $\mathrm{K};\Delta;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ によって導入される．プログラムの型付けに関する規則では，$\{\overline{c}\}$ に含まれるクラス定義のインスタンスの型として多相型が推論され，得られたクラス環境のもとで p の型が推論される．クラス群 $\{\overline{c}\}$ に含まれる各クラスの定義が相互再帰的であることから，クラス群の内部においては，それぞれのクラスのインスタンスの型は単相型である．let 式の型付けでは，右辺の式の型に含まれる適用的型変数を一般化した多相型が得られる．

Tofte の型システムにおいては，let 式の型付け規則は 2 種類あり，右辺が展開的（expansive）な式[⋆1]であるかどうかによって適用される規則が異なる．我々の型システムでは，let 式の型付け規則が Tofte の型システムにおける展開的な式の型付け規則に対応し，クラス定義の型付けが非展開的な式の型付けに対応する．Tofte の型システムにおいては変数は非展開的な式に分類されるが，我々の型システムにおいては，変数も展開的な式と同様に型付けを行う．型環境 Γ に登録されている変数の型はつねに $\forall t_1, \ldots, t_n.\mathrm{K} \triangleright \alpha$ という形をしており，束縛変数に命令的型変数を含まない．そのため，let の右辺となった変数の型に含まれる命令的型変数の集合は，つねに型環境の自由変数の集合の部分集合となり，let について非展開的な式に関する規則を導入したとしても得られる多相型は変化しない．

本章で形式化した範囲の型システムは，Garrigue の型システムと命令的多相を統合した型システムに変換できる．その型システムの健全性を仮定すれば，ここで形式化した型システムに関する健全性が成り立つ．ただし，初期化されていないインスタンス変数を参照するプログラムも型を持つ．Java の型システムと同様に，健全性は「NoMethodError が発生するのは，未初期化のインスタンス変数に対してメソッド呼び出しを行った場合のみである」という意味になる．

この型システムに，Garrigue の型推論アルゴリズムを適用し，型推論を行った．

⋆1 値でない式[7]．

プログラム

$$\frac{\emptyset;\Delta;\emptyset \vdash e : \alpha}{\Delta \vdash e : \alpha} \qquad \frac{\begin{array}{l}\mathrm{K};\Delta, l(c_1):\beta_1,\ldots,l(c_n):\beta_n \vdash c_i : \beta_i \\ B = \mathrm{FV_K}(\beta_i) \quad \sigma_i = \forall B.\mathrm{K} \triangleright \beta_i \\ \Delta, l(c_1):\sigma_1,\ldots,l(c_n):\sigma_n \vdash p : \alpha\end{array}}{\Delta \vdash \{\, c_1,\ldots,c_n \,\}, p : \alpha}$$

クラス定義

$$\frac{\begin{array}{l}\Delta(\mathrm{C}) = \theta(\alpha) \quad \mathrm{K};\Delta;@x_1:\theta(u_1),\ldots,@x_m:\theta(u_m) \vdash m_i : \theta(\beta_i) \to \theta(\gamma_i) \\ \alpha :: ((\emptyset, \{l(m_1),\ldots,l(m_n)\}), \{r(l(m_1), \beta_1 \to \gamma_1),\ldots,r(l(m_n), \beta_n \to \gamma_n)\}) \vdash \theta : \mathrm{K}\end{array}}{\mathrm{K};\Delta \vdash \texttt{class}\ \mathrm{C}\ @x_1 : u_1,\ldots,@x_m : u_m, m_1,\ldots,m_n\ \texttt{end} : \theta(\alpha)}$$

メソッド定義

$$\frac{\mathrm{K};\Delta;\Gamma, x:\alpha \vdash e : \beta}{\mathrm{K};\Delta;\Gamma \vdash \texttt{def}\ \mathrm{m}(x)\ e\ \texttt{end} : \alpha \to \beta} \qquad \frac{\mathrm{K}' \vdash \theta : \mathrm{K} \quad \Delta(\mathrm{C}) = \sigma \quad \beta \sqsubseteq_{\mathrm{K}'} \sigma}{\mathrm{K};\Delta;\Gamma \vdash \texttt{def}\ \mathrm{m} :: \mathrm{K}' \triangleright \alpha \to \mathrm{C} : \theta(\alpha) \to \theta(\beta)}$$

式

$$\frac{\alpha \sqsubseteq_{\mathrm{K}} \Gamma(x)}{\mathrm{K};\Delta;\Gamma \vdash x : \alpha}\ \textbf{ローカル変数} \qquad \frac{\Gamma(@x) = u}{\mathrm{K};\Delta;\Gamma \vdash @x : u}\ \textbf{インスタンス変数}$$

$$\frac{\alpha \sqsubseteq_{\mathrm{K}} \Delta(\mathrm{C})}{\mathrm{K};\Delta;\Gamma \vdash \mathrm{C}.\texttt{new} : \alpha}\ \textbf{インスタンス作成} \qquad \frac{\begin{array}{l}\alpha_0 :: ((\{\,\mathrm{m}\,\}, \mathcal{L}), \{\, r(\mathrm{m}, \beta \to \alpha)\,\}) \vdash \theta : \mathrm{K} \\ \mathrm{K};\Delta;\Gamma \vdash e_1 : \theta(\alpha_0) \quad \mathrm{K};\Delta;\Gamma \vdash e_2 : \theta(\beta)\end{array}}{\mathrm{K};\Delta;\Gamma \vdash e_1.\mathrm{m}(e_2) : \theta(\alpha)}\ \textbf{メソッド呼び出し}$$

$$\frac{\mathrm{K};\Delta;\Gamma \vdash e_1 : \beta \quad B = \mathrm{FV_K}(\beta) \backslash (\mathrm{FV_K}(\Gamma) \cup \mathit{ImpTyVar}) \quad \mathrm{K}|_{\overline{B}};\Delta;\Gamma, x : \forall B.\mathrm{K}|_B \triangleright \beta \vdash e_2 : \alpha}{\mathrm{K}|_{\overline{B}};\Delta;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \alpha}\ \textbf{let 式}$$

$$\frac{\Gamma(@x) = u \quad \mathrm{K};\Delta;\Gamma \vdash e : u}{\mathrm{K};\Delta;\Gamma \vdash @x = e : u}\ \textbf{インスタンス変数代入}$$

図 3 型付け規則
Fig. 3 Typing rules.

5.1 型推論の流れ

型推論は次の手順で行う．

(1) クラス定義ごとに，メソッド定義とメソッド型定義を統合する．
(2) パラメータ化されたクラス定義を展開する（無限に続く展開は有限回で近似）．
(3) クラス定義の依存関係を解析し，相互再帰的なクラス群へと分割する．
(4) 依存関係に従って，クラス群ごとに 5，6，7 を繰り返す．
(5) 各クラス定義の負の位置に対応する型を生成する．
(6) シグネチャ由来のメソッド定義 $\texttt{def}\ \mathrm{m} :: \mathrm{C} \to \mathrm{C}'$ から，引数の型を多相レコード型へと変換し $\texttt{def}\ \mathrm{m} :: \mathrm{K} \triangleright \alpha \to \mathrm{C}'$ を生成する．
(7) クラス定義それぞれのメソッド定義から，メソッドの型を推論し，インスタンスの型とする．

(3) では，すべてのクラスの依存関係を表すグラフを構築し，そのグラフを強連結成分へと分解することにより，相互再帰的に定義されたクラス群へ分解する．クラス群間の依存関係をトポロジカルソートすることにより，依存関係に従ってクラス群を処理することができる．

6. 実　　装

型推論システムは，Objective Caml[3] によって実

装した．Ruby の構文規則は複雑であることから，構文解析器の実装は行わず，Ruby インタプリタが構文解析した結果を出力する Ruby ライブラリ NodeDump を修正したものを利用した．NodeDump の出力にはインデントによって構文木の構造を表現する部分があり解析が煩雑であったため，構文木の構造を適当な文字列を出力することによって明示するよう修正した．

実装は Ruby のサブセットをサポートしている．実装がサポートするサブセットに含まれる主な機能をあげる．

- シグネチャあるいは Ruby プログラムによるクラス・モジュールの定義
- クラスの継承
- シグネチャによるモジュールの include
- 定数によるクラス・モジュールの表現
- ネストしたクラス・モジュールの定義
- ローカル変数，インスタンス変数への代入
- `if`，`case` などの条件分岐構文
- `while`，`for` などのループ構文
- 例外処理
- ブロックの定義・メソッド呼び出しでの利用

次に示すような機能への対応は実装されていない．

- 任意個の引数をとるメソッド定義
- クラス・モジュール定義中に記述された文の実行
- 多重代入
- 定数の宣言

クラスの継承は，親クラスに含まれるメソッド定義を，子クラスにコピーすることによって処理する．ネストされたクラス定義については，クラス定義をネストの外側へと移動する．クラスやモジュールは Ruby の実装に忠実に定数として定義される．クラスのインスタンスの型は，クラスに対応する定数の `new` メソッドの型から得られる．ローカル変数への代入は，形式化では let 式に完全に読み替えられると説明したが，実装では代入として型環境に対する変更が行われるものとして取り扱う．実装においては型判定は，式を評価した後の型環境 Γ' によって拡張されていると考え，$\mathrm{K}, \Gamma \vdash e : \alpha, \Gamma'$ という形になる．クラス環境は存在せず，インスタンスの型は型環境から得られる定数の型を通じて表現される．条件分岐構文は，それぞれの分岐を型推論した後，それぞれの分岐における式の評価後の環境の共通部分を分岐後の環境として利用する．ループ構文は，1 回ループが実行された場合と 1 回もループが実行されなかった場合に分岐するものとして，条件分岐と同様に取り扱う．ブロックは，高階関数と同様に型推論を行う．

`nil` は，任意の型を持つものとして型付けされる．したがって，nil 式のメソッド呼び出しや未初期化のインスタンス変数へのメソッド呼び出しは，型エラーとして検出されない．静的にクラス定義を統合して型推論を行うため，クラス定義を動的に変更し，変更の順序に依存するようなプログラムに対しては，健全性が成り立たない．また，ループやブロックの型推論に関しても，健全性が成立しない．

6.1 評　　価

実装を用いて，Ruby バージョン 1.8.5 に付属のサンプルプログラムの型推論を行ったところ，39 個のサンプルプログラムのうち，21 個のプログラムの型推論が可能であった．組み込みライブラリについては，`Object` や `Array`，`Regexp` などの主な組み込みクラスおよび組み込みモジュール 22 個の定義を含む 280 行のシグネチャファイルを用いた．

型推論が可能であったプログラムのうち，list.rb というプログラムについて説明する．list.rb は，連結リストの実装を含む 80 行の Ruby プログラムであり，リストを表現する `MyList` クラスと，リストの要素を表現する `MyElem` クラスが定義されている．型推論によって得られた `MyList` クラスと `MyElem` クラスのインスタンスの型のカインドを，**図 4** に示す．この図では，型変数 19015 が `MyElem` クラスのインスタンスの型である．`MyElem` には `succ` や `data` というメソッドが定義されている．これらの型が型変数 19015 のカインドとして表現されている．また `MyList` には `add_to_list` などのメソッドが定義されており，型に反映されている．list.rb の型推論に要した時間は，Intel Core Duo 2.16 GHz の CPU を搭載した Apple MacBook Pro において，1.26 秒であった．

list.rb においては，型推論を行うために，組み込みメソッドである `print` メソッドの修正が必要であった．`print` メソッドは，任意個の引数を受け付けるが，我々の型システムではこのような型を表現できない．そこで，引数の数に応じて `print1`，`print2` などと便宜的にメソッドを定義し，それらの呼び出しに修正することで，型推論を行った．

次に，list.rb において型の不整合となるようにプログラムを変更し，型推論によって検出できることを確認した．`MyElem` クラスの定義を変更し，リストの要素に対してメソッド `f` を呼び出すようにした．

```
class MyElem
  ...
  def data
    # @data
```

```
19015 :: { rqs: {}
           avs: { succ; eval; to_i; eql?; hash; ==; =~; print;
                  initialize; gets; succ=; data; rand; inspect; print2;
                  print3; puts; to_s; require }
           rels: { print: '18993 => ('18991) -> '18992;
                   succ=: '19666 => ('_42753) -> '_42753;
                   eval: '19004 => ('18785) -> '19003;
                   data: '19013 => () -> '_19451;
                   eql?: '18985 => ('18984) -> '19201;
                   gets: '19007 => () -> '18784;
                   initialize: '19012 => ('_19451) -> '_42753;
                   to_s: '18981 => () -> '18784;
                   to_i: '18982 => () -> '19510;
                   =~: '18989 => ('18988) -> '19510;
                   puts: '19010 => ('19008) -> '19009;
                   rand: '19011 => ('19511) -> '19510;
                   hash: '18983 => () -> '19510;
                   print3: '19002 => ('18998*'18999*'19000) -> '19001;
                   succ: '19014 => () -> '_42753;
                   print2: '18997 => ('18994*'18995) -> '18996;
                   require: '19006 => ('18785) -> '19005;
                   ==: '18987 => ('18986) -> '19201;
                   inspect: '18990 => () -> '18784 }
           desc: Kinstance [MyElem] }

52301 :: { rqs: {}
           avs: { eval; to_i; eql?; hash; ==;
                  add_to_list; =~; print; gets;
                  require; rand; each; inspect; print2;
                  print3; puts; to_s }
           rels: { print: '56017 => ('56015) -> '56016;
                   add_to_list: '56036 => ('_47278) -> '_47231;
                   eval: '56028 => ('55809) -> '56027;
                   gets: '56031 => () -> '55808;
                   eql?: '56009 => ('56008) -> '52486;
                   to_s: '49948 => () -> '55318;
                   =~: '56013 => ('56012) -> '52795;
                   to_i: '56006 => () -> '52795;
                   puts: '56034 => ('56032) -> '56033;
                   rand: '56035 => ('52796) -> '52795;
                   print3: '56026 => ('56022*'56023*'56024) -> '56025;
                   hash: '56007 => () -> '52795;
                   each: '49945 => &{ ('_52951) -> '49946 } => () -> '49944;
                   require: '56030 => ('55809) -> '56029;
                   print2: '56021 => ('56018*'56019) -> '56020;
                   inspect: '56014 => () -> '55808;
                   ==: '56011 => ('56010) -> '52486 }
           desc: Kinstance [MyList] }
```

図 4 型推論の結果
Fig. 4 Result of type inference.

```
      @data.f
    end
    ...
  end
```

この変更によって，f メソッドを実装していない値を，リストの要素として登録した場合にエラーとなる．このプログラムを型推論すると，**図 5** に示した実行結果のように，メソッド f が定義されていないことから型の不整合が検出されることが確認できた．実行時間はほとんど変化せず，1.23 秒であった．

図 4 に掲載した型環境の出力は，出力の一部であり，実際には組み込みライブラリなどの型がすべて表示される．型推論の全過程で利用された型変数の数は数十万を超え，クラス定義に対応する定数の型における束縛型変数の数は list.rb の結果に含まれる例で 57,479 個におよぶ．このように，このシステムにおいては，

```
../example/ruby-1.8.5/list.rb:69: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:70: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:71: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:72: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:74: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:75: undefined method {‘f’} (NoMethodError)
../example/ruby-1.8.5/list.rb:76: undefined method {‘f’} (NoMethodError)
```

図 5 型エラーの検出
Fig. 5 Detection of type errors.

推論される型が ML などに比べて大きくなる．

今回実験を行った中で，型推論を行うことが不可能であったプログラムを分類すると，次のようになる（括弧内の数字は，重複を許して数えた場合の件数）．

- Range クラスを利用するプログラム（6 件）
- 任意個の引数を与えるプログラム（5 件）
- 定数の宣言を行うプログラム（3 件）
- 多重代入を含むプログラム（2 件）
- クラス定義中に文を含むプログラム（1 件）
- その他（3 件）

Range クラスは，2 つの値の間の範囲を表現するクラスである．Range クラスのインスタンスの型は，範囲の両端を表すオブジェクトの型によってパラメータ化されていると考えられる．このとき，両端のオブジェクトには succ というメソッドが必要であるが，このような型抽象の引数に対する制約は現在の実装ではシグネチャに記述できない．そのため，Range クラスは実装において未対応となっており，Range クラスを利用するプログラムが解析できなかった．そのほかには，test.rb などのプログラムが含まれる．test.rb は，Ruby インタプリタのテストを行うためのプログラムであり，実装において対応していない構文が含まれている．

7. 今後の課題

現在の型システムには，次のような型を表現できないという問題がある．

(1) 引数の数が定まらないメソッド呼び出し
(2) 厳密に特定のクラスを表現する型

Wright によるソフトタイピングの研究[9] では，関数の引数の型をリストのような再帰型として推論することにより，任意個の引数を受け取る関数の型を表現している．Ruby における多重代入などの構文については，式に対してその型が特定のクラスであることが要求される．現在の型システムでは型がどのクラスのインスタンスの型であるかという情報を保持していないが，クラス定義についてそれぞれ一意なメソッドを定義することにより，特定のクラスである型を表現できると考えられる．

また実装に関する課題として，次のようなものがある．

(1) 定数への代入を行うプログラム
(2) クラス定義中に文を含むプログラム
(3) 型抽象の引数に制約が必要な型

(1)，(2) については適切なプログラム変換を行うことにより，現在の型システムで取り扱うことが可能であると考えられる．(3) について，シグネチャ記述言語において，型抽象の引数となった型に対して特定のメソッドを要求するなどの制約を表現できない．これについては，シグネチャ記述言語の表現力を向上させる必要がある．

配列などのコレクションに様々な型のオブジェクトを格納した場合，それらのオブジェクトの型は単一の型として推論される．これは，配列の走査を行うプログラムの型付けとしては十分であるが，ML などのタプルのように配列を利用するようなプログラムについては，それぞれの要素の厳密な型が必要となることから不十分である．このような場合，C や Java などの型付きの言語に見られるキャストのように，注釈によってプログラマが型情報を記述することが考えられる．

```
class Object
  def as(klass)
    self
  end
end

[1, "a", true].first.as(Integer) + 2
```

上に示した例では，as メソッドによってキャストを表現している．as メソッドの呼び出しについて型付け規則を新たに導入することにより，キャストを型付けすることができる．as メソッドはプログラムの実行結果に影響を与えない．

また，型推論によって検出された型エラーは，図 5 に示したように，断片的な情報としてユーザに通知される．これは，図 4 に示したような型の内部表現をそのまま表示するだけでは，型の大きさが膨大であるこ

とから，その情報を読み取ることは困難であると判断したためである．不整合を検出された型の情報を分かりやすくユーザに提示できるよう，検出したエラーの出力を改善する必要がある．

8. 関連研究

Cartwright らによるソフトタイピングの研究がある[1)]．ソフトタイピングは静的な型システムによる型推論のみではなく，実行時の型検査をも行うことで，より広い範囲のプログラムにおいて型安全性を保証し，実行時の型検査による実行速度の低下を抑えている．Cartwright らによるソフト型システムは，その基盤となる理論において Rémy による多相レコード型を用いている．Rémy による多相レコード型の研究は，本研究が基づいている Garrigue による多相レコード型の研究と同様の目的において開発されたものであり，その点で本研究とよく似た型システムとなっている．Cartwright らによるソフトタイピングにおいては，多相ヴァリアント型によって型が表現されている．本研究では，逆に多相レコード型を用いて，型が表現される．この違いは，対象とする言語が関数型言語である Scheme であるかオブジェクト指向言語である Ruby であるかという違いに由来すると考えられる．

Cartwright らによって開発されたソフトタイピングは，Wright によって実用的な Scheme プログラムへの適用が試みられた[9)]．Wright の手法では値多相によって多相性が制限されている．我々のシステムでは，命令的型変数を導入し，インスタンス変数の型付けについて多相性が制限される．

9. おわりに

Ruby プログラムの型推論ツールを設計，実装した．型推論ツールは，Ruby プログラムと組み込みライブラリのシグネチャを入力とし，Ruby プログラムにおける型の不整合を検出する．このシステムでは，Ruby における特徴的な機能のうち，組み込みクラスの拡張やブロックなどをサポートする．Ruby のサブセットについて型システムの形式化を行い，健全性を確認した．また，本システムを用いて小規模な Ruby プログラムを型推論した結果，型の不整合を検出できた．

我々の型システムは，ML の型システムと多相レコード型に基づいており，Ruby プログラムを型推論する際には，多相再帰的なクラス定義や正則な型を持たないメソッド定義などについて問題があった．多相再帰的なクラス定義はクラス定義自体を展開することで型推論を行い，正則な型とならないメソッドについては同様に型を有限回展開することにより近似を行った．

実験では，小規模な Ruby プログラムについて，型推論が利用できることが確認できた．タプルとして利用される配列や動的なクラス定義の変更などの精密な型付けについては，型体系の大幅な変更が必要になることから，対応することは考えていない．このような機能を利用していない場合は，大規模なプログラムについても有効な検査が提供できるのではないかと考えている．今後，Ruby のより広い範囲をサポートするようシステムを改善し，現実的なプログラムについて型推論の精度を確認したい．

参考文献

1) Cartwright, R. and Fagan, M.: Soft Typing, *Programming Language Design and Implementation*, pp.278–292 (1991).
2) Garrigue, J.: Simple Type Inference for Structural Polymorphism, *9th Workshop on Foundations of Object-Oriented Languages* (2002).
3) Leroy, X.: The Objective Caml system release 3.10: Documentation and user's manual (2007). http://caml.inria.fr/
4) Matsumoto, Y.: Ruby Programming Language (2007). http://www.ruby-lang.org/
5) Ohori, A.: A polymorphic record calculus and its compilation, *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.6, pp.844–895 (1995).
6) Rémy, D.: Type Inference for Records in a Natural Extension of ML, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, MIT Press (1993).
7) Tofte, M.: Type Inference for Polymorphic References, *Information and Computation*, Vol.89, No.1, pp.1–34 (1990).
8) Wright, A. K.: Simple Imperative Polymorphism., *Lisp and Symbolic Computation*, Vol.8, No.4, pp.343–355 (1995).
9) Wright, A.K. and Cartwright, R.: A Practical Soft Type System for Scheme, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.1, pp.87–152 (1997).

(平成 19 年 9 月 12 日受付)
(平成 19 年 12 月 6 日採録)

松本宗太郎

2007年筑波大学大学院システム情報工学研究科博士前期課程修了．同年同大学院同研究科博士後期課程入学．スクリプト言語によって記述されたプログラムの検証に興味を持つ．

南出　靖彦（正会員）

1993年京都大学大学院理学研究科数理解析専攻修士課程修了．同年同大学数理解析研究所助手．1999年筑波大学電子・情報工学系講師．2004年筑波大学大学院システム情報工学研究科講師．2007年同准教授．博士（理学）．プログラミング言語およびソフトウェア検証に興味を持つ．