Rubyに対するGradual typingの導入に向けて

丹治　将貴[1,a)]　中野　圭介[1,b)]　岩崎　英哉[1,c)]

概要：プログラミング言語 Ruby は，広く使用されている動的型付け言語である．Ruby は実行時に型チェックを行うため，プログラムを実行し型エラーを含む部分に実行が及ばなければ型エラーは報告されない．このことは，プログラム中に型エラーによるバグが潜在的に残り，バグの発見が遅れたり，バグを見逃したりする要因となっている．このような動的型付けの欠点を補う方法として，Gradual typing が提案されている．Gradual typing とは，型注釈の有無により型付けの手法が異なるような型システムである．型注釈がある部分については静的に型付けをし，実行前に型チェックを行う．型注釈がない部分については動的型として型付けをし，実行時に型チェックを行う．本研究の目的は，Ruby の動的型付けによる柔軟性を残しつつ，プログラムの型エラーによるバグの発見を容易にすることである．その方針として，Ruby に Gradual typing を導入することで，静的型エラーの検出と，Ruby の柔軟性を両立させることを目指す．本発表では，Gradual typing に基づく動的型と型注釈の構文を加えた Ruby のサブセットを考え，そのサブセットについて型付け規則を与える．さらにその正当性を示し，実装に向けての基本的な考え方を示す．

キーワード：Ruby，Gradual typing，型システム

1. はじめに

プログラミング言語 Ruby は，広く使用されている動的型付け言語であり，Ruby on rails で有名である．Ruby の特徴のひとつは，強力なメタプログラミングが可能という点である．その例として，既存クラスへのメソッド追加･変更などを行えるオープンクラスや，未定義メソッド呼び出しの際の処理を定義できる method_missing 等がある．Ruby は動的型付け言語であるため，プログラム中の型エラーは実行時に報告される．そのため，プログラムを実行しなければ型エラーは報告されない．このことは，プログラム中に型エラーによるバグが潜在的に残り，バグの発見が遅れたり，バグを見逃す要因となる．

この問題点を解決するひとつの方法として，Gradual typing [1] が提案されている．Gradual typing とは，型注釈の有無により型付けの手法が異なるような型システムである．型注釈がある部分については静的に型付けを行い，実行前に型チェックを行う．例えば，変数 x は整数型であるという意味の型注釈があった場合は，その変数 x は整数型として静的に型付けされる．型注釈がない部分については動的型として型付けを行い，キャストを挿入し，実行時にキャストにより型チェックを行う．動的型は ? 型と表記する．例えば，変数 y

[1] 電気通信大学大学院 情報理工学研究科 情報・通信工学専攻 Department of Communication Engineering and Informat- ics, Graduate School of Informatics and Engineering,
[a)] tanji@ipl.cs.uec.ac.jp
[b)] ksk@cs.uec.ac.jp
[c)] iwasaki@cs.uec.ac.jp

に型注釈がない場合は，その変数 y は動的型として型付けされ，変数 y を含む式は動的に型検査される．Gradual typing は部分的に静的型チェックと動的型チェックを行うので，静的型付けと動的型付けの両方の利点を得ることができる．

本研究の目的は，Ruby の動的型付けによる柔軟性を残しつつ，プログラムの型エラーによるバグの発見を容易にすることである．その方針として，Ruby に Gradual typing を導入することを目指す．さらに Ruby のオープンクラスによるメソッド追加とメソッド更新に対応させることも目指す．Gradual typing を導入した Ruby を使用すれば，ユーザは静的に型エラーを検出したい部分に型注釈を入れ，Ruby の動的型付けによる柔軟な表現を残したい部分には型注釈を入れないことで，静的型エラーの検出と，Ruby の柔軟性を両立させることができる．

本稿の構成は次の通りである．2 節では，Ruby に Gradual typing を導入した際のプログラムの振る舞いについて述べ，3 節では，本研究で扱う Ruby サブセットや，それに対する型付け規則，評価規則を述べる．4 節では Ruby サブセットが型健全性を満たすことを証明し，5 節で関連研究について述べ，6 節で本稿をまとめる．

2. Ruby における Gradual typing

本節では，Ruby における Gradual typing の動作をプログラム例を用いて説明する．本研究で設計する Ruby サブセットでは，変数への代入式と，メソッド定義の部分に型注釈を挿入できる．したがって，変数への代入式と，メソッド定義の振る舞いと，その他に特徴的な，if 式とメソッド呼び出しの振る舞いについて説明する．

2.1 変数への代入式

図 1 にプログラム例を示す．

最初に変数への型注釈について説明する．本研究における Gradual typing の型注釈は，“:::” の後に型名 (クラス名) を続けて表現する．例えば，変数への代入式は，

```
1   x:::String = "abcde" # x は String 型
2   y:::Fixnum = 5       # y は Fixnum 型
3   z:::Float = "3.14"   # 静的型エラー
4   x:::Fixnum = 3       # x は Fixnum 型
5   y:::String = "fghij" # y は String 型
6   x = 100          # x は Fixnum 型のまま
7   x = "foo"            # 静的型エラー
8   w = 4      # w は?型 ?型 <= Fixnum 型
9   x = w                # 代入は成功
10  y = w                # 動的型エラー
11  z:::Numeric = 3
12          # Numeric 型<=Fixnum 型
13  v:::A = A.new
14          # A では Numeric クラスの
15          # メソッド全てを型の矛盾なく
16          # 定義済みかつNumeric クラスと
17          # 継承関係はないとする．
18  z = v  # Numeric 型 <= A 型
```

図 1 代入式における Gradual typing

変数 :::型名 = 式

と書く．型注釈をつけた場合，その変数は静的に型付けされる．図 1 の 1 行目から 3 行目がその例であり，x は String 型，y は Fixnum 型となる．z については，Float 型と型注釈しているのに対して，String 型の値を代入しているため，静的型エラーとなる．

すでに型がつけられている変数に対して，? 型以外の別の型注釈をつけて代入を行った場合，その変数の型を新しい型注釈の型に変更する．4 行目と 5 行目がその例であり，String 型だった変数 x が Fixnum 型，Fixnum 型だった変数 y が String 型となっている．

型注釈をつけずに代入式を記述すると，“:::?” が省略されているものとみなす．ここで，? 型は動的型を表す．すでに型がつけられている変数に型注釈を省略して代入すると，元の型をそのまま引き継ぐ，という意味になる．6 行目と 7 行目がその例となっており，変数 x は 4 行目で Fixnum 型として型付けされているため，Fixnum 型の値は代入できるが，String 型の値は代入できない．

初出の変数に型注釈を省略して代入すると，その変数の型は ? 型として型付けされる．8 行目に

おいて，変数 w は初出のため，? 型となる．ここで，? 型の変数に Fixnum 型の値 4 を代入しているため，右辺の値にキャストが挿入される．キャストとは，実行時に型を変換するための構文であり，

⟨ 型名 ⇐ 型名 ⟩ 式

と表現する．キャストは左辺の変数の型と右辺の式の型が異なる場合に式の前に挿入される．したがって，8 行目の式は w :::? = ⟨? ⇐ Fixnum⟩4 となり，このキャストは実行時に検査され，このキャストは成功する．続いて 9 行目では，Fixnum 型の変数 x に ? 型の変数 w を代入しているので，この場合においてもキャストが挿入され，x = ⟨Fixnum ⇐?⟩w となる．w の値は，⟨? ⇐ Fixnum⟩4 だったため，9 行目の代入式は，x = ⟨Fixnum ⇐?⟩⟨? ⇐ Fixnum⟩4 となる．これらのキャストは，最終的には Fixnum 型から Fixnum 型へのキャストとなり，キャストを行わないことと同等になるため，9 行目の代入式は動的型エラーを起こすことなく成功する．続いて 10 行目では，String 型の変数 y に w を代入しているので，9 行目と同様にキャストが挿入され，10 行目の代入式は y = ⟨String ⇐?⟩⟨? ⇐ Fixnum⟩4 となる．これらのキャストは，最終的には Fixnum 型から String 型へのキャストとなり，これは矛盾するため，動的型エラーを起こす．

最後にクラス階層について説明する．本研究で設計する Ruby サブセットでは，クラス継承を扱っており，子クラスの型から親クラスの型へのキャストが発生することがあり，そのキャストは成功する．11 行目では，変数 z を Numeric 型としており，それに対して Fixnum 型の値 3 を代入している．Fixnum 型は，Numeric 型の子孫クラスであるため，11 行目の式はキャストが挿入され，z ::: Numeric = ⟨Numeric ⇐ Fixnum⟩3 となり，このキャストは成功する．また，Numeric クラスと継承関係にないクラスでも，Numeric クラスに属するメソッドをすべて定義していて，かつそのメソッドの型がすべて Numeric クラスのメソッドの型と矛盾がないクラスの場合は，このキャストが成功する．13 行目では，あるクラス A が存在し，そのクラスインスタンスを生成して，変数 v に代入している．クラス A では，Numeric クラスのメソッドが型の矛盾なく定義されていて，かつ Numeric クラスと継承関係がないとする．続いて 18 行目で，Numeric 型の変数 z に v を代入している．この時もキャストが挿入され，代入式は，z = ⟨Numeric ⇐ A⟩v となり，このキャストは成功する．

2.2 メソッド定義とメソッド呼び出し

図 2 にメソッド定義とメソッド呼び出しのプログラム例を示す．

1 行目から 6 行目で，既存のクラス Numeric に対して，オープンクラスによって新たなメソッド mysucc を追加している．ここでは，mysucc に対して，Fixnum 型を受け取り Fixnum 型を返すという型注釈を挿入している．Ruby では，最後に評価された式の結果が返り値となるので，このメソッド定義では 4 行目の n+1 が返り値となる．引数 n が Fixnum 型と型注釈されているため，4 行目は Fixnum 型の値となり，型注釈の返り値の型と一致する．したがって，このメソッド定義は静的型エラーを起こすことなく成功する．

次にメソッド呼び出しについての振る舞いを，7 行目以降のプログラムで説明する．変数 x は Fixnum 型，y は ? 型，z は Numeric 型である．z

```
1   class Numeric
2    def mysucc(n) ::: Fixnum -> Fixnum
3    # メソッドへの型注釈
4     n+1
5    end
6   end
7   x:::Fixnum = 3
8   y = 5
9   z:::Numeric = Numeric.new
10                  # Numeric 型
11  z.mysucc(x)     # Fixnum 型
12  z.mysucc(y)     # 引数にキャストが発生
13  z.mysucc("abc") # 静的型エラー
14  w = Numeric.new # ?型
15  w.mysucc(x)  # レシーバにキャストが発生
```

図 2 メソッド定義，メソッド呼び出しにおける Gradual typing

のメソッド mysucc を 11 行目から 13 行目で呼び出している．11 行目のメソッド呼び出しは，定義時の型注釈の引数の型と実引数 x の型が Fixnum 型と一致しているので，Fixnum 型の返り値が得られる．12 行目のメソッド呼び出しでは，実引数 y の型が ? 型のため，引数に，? 型から型注釈の引数の型である Fixnum 型へのキャストが挿入される．その結果，12 行目の式は，z.mysucc(⟨Fixnum ⇐?⟩y) となる．y は，⟨? ⇐ Fixnum⟩5 という値であるため，mysucc の実引数は ⟨Fixnum ⇐?⟩⟨? ⇐ Fixnum⟩5 となり，このキャストは成功する．

13 行目のメソッド呼び出しは，String 型の実引数を与えており，型注釈と矛盾するため，静的型エラーとなる．

14 行目では，w にクラス Numeric の生成式を代入しているが，型注釈がないので w は ? 型となる．15 行目で　w のメソッド mysucc を呼び出しているが，レシーバである w の型が ? 型のため，w に対してキャストが挿入される．したがって，15 行目の式は，(⟨[mysucc : Fixnum →?] ⇐?⟩w).mysucc(x) となる．ここで，[mysucc : Fixnum →?] は，「Fixnum 型の値を受け取り ? 型の値を返すメソッド mysucc を持つクラス」の型を意味する．15 行目の引数 x の型が Fixnum と判明しているため，キャストにおける mysucc の引数の型を Fixnum 型としている．実行時において，15 行目のメソッド呼び出し式全体にキャストが挿入され，⟨? ⇐ Fixnum⟩(w.mysucc(x)) となる．このキャストは成功し，メソッドの返り値を得られるが，その型は ? 型となる．

ここで，図 2 の後に図 3 のようなプログラムが存在したとする．

Numeric クラスの子クラスである Float クラスにおいて，同名のメソッド mysucc を定義している．親クラスに存在するメソッドを再定義する場合は，再定義前のメソッドの型と，再定義する際の型注釈の型の関係により，静的型エラーになりうる．詳しい定義については 3 節で述べる．図 3 の 1 つ目のメソッドの再定義では，同名のメソッド mysucc に対して，String → String と型注釈している．再定義前は引数と返り値の型が，ともに Fixnum 型であるので，それらを String 型とすることはできず，静的型エラーとなる．2 つ目の再定義では，引数と返り値の型が Fixnum 型となっている．この場合は，静的型エラーとはならず，再定義は成功する．

```
16  class Float
17   def mysucc(n) ::: String -> String
18    ...
19   end
20  end
21  # 静的型エラー発生
22  class Float
23   def mysucc(n) ::: Fixnum -> Fixnum
24    ...
25   end
26  end
27  # 静的型エラーなし
```

図 3　子クラスにおける親クラスメソッドの再定義

```
1   if e then              # e は適当な式
2    x:::Fixnum = 3        # x は Fixnum 型
3    y = "abc"             # y は?型
4   else
5    x:::Fixnum = 5        # x は Fixnum 型
6    y = 3                 # y は?型
7   end
8                  # if 式の型は Nil 型
9                  # x は Fixnum 型
10                 # y は?型
11  if e then              # e は適当な式
12   z:::String = "def"    # z は String 型
13   w = 4                 # w は?型
14  else
15   z:::Fixnum = 5        # z は Fixnum 型
16  end
17  # z，w の型の不一致のため，静的型エラー
```

図 4　if 式における Gradual typing

2.3　if 式

図 4 に if 式のプログラム例を示す．

最初に，if 式の結果の型について説明する．if 式の型は，then 節と else 節の内容に関わらず，必ず Nil 型とする．したがって，図 4 の 1 行目から 7 行目の if 式の型は Nil 型である．

次に，then 節と else 節のそれぞれの中で，変

数への代入があった場合について説明する．この場合，then 節を通過した後に変数に与えられている型と else 節を通過した後に変数に与えられている型が等しくなければならない．図 4 の 1 行目から 7 行目の if 式では，then 節と else 節でそれぞれ x は Fixnum 型，y は?型が等しく与えられているため，静的型エラーは発生せず，if 式には型がつけられる．しかし，図 4 の 11 行目から 16 行目の if 式では，then 節で z は String 型，w は?型が与えられているが，else 節で z は Fixnum 型が与えられ，w については代入式が存在しない．z に与えられた型が，then 節と else 節で一致せず，また，w についても，then 節のみで型が与えられているため，型が一致しないとみなす．したがって，この if 式は成功せず，静的型エラーとなる．

3. 設計

本節では，本研究で扱う Ruby のサブセットとそれに対する型付け規則と評価規則を示す．

3.1 Ruby のサブセット

本研究で扱う Ruby のサブセットを図 5 に示す．このサブセットは，Ren らの Ruby-Like 言語 [4] を参考にして定めた．

ξ は値であり，nil またはクラスのインスタンス $[A]$ である．v も値であり，ξ と同じであるが，キャスト挿入の構文追加の時に拡張する．x は変数の識別子である．self はレシーバを表す．A はクラスであり，A.new でそのクラスのインスタンスを生成する．$e;e$ は連続した式を表している．その他には，if 式とメソッド呼び出し $e.m(e)$ が式の内容に含まれている．ここで m はメソッド名である．

さらに，Gradual typing を導入するために，型注釈の構文を式として定義している．今回の研究では，代入式における左辺の変数と，メソッドの引数と返り値に型注釈を与える．$x ::: \tau = e$ が変数への型注釈であり，τ は変数が取り得る型である．また，def $A.m = \lambda x.e ::: \tau_s$ がメソッドへの型注釈である．ここで，τ_s の右矢印の左が引数の型，右が返り値の型である．型 τ の定義には，Nil

$$
\begin{array}{rcl}
\text{expressions } e & ::= & v \mid x \mid \text{self} \mid \\
 & & e;e \mid A.\text{new} \mid \\
 & & \text{if } e \text{ then } e \text{ else } e \mid \\
 & & e.m(e) \mid x ::: \tau = e \mid \\
 & & \text{def } A.m = \lambda x.e ::: \tau_s \\
\text{ground values } \xi & ::= & \text{nil} \mid [A] \\
\text{values } v & ::= & \xi \\
\text{value types } \tau & ::= & \text{Nil} \mid A \mid ? \\
\text{method types } \tau_s & ::= & \tau \to \tau \\
\text{methods } c_m & ::= & A.m \\
\text{variables } vars & ::= & x \mid \text{self} \\
 & & \\
\text{method table } MT & ::= & (c_m \mapsto \tau_s)* \\
\text{type env } \Gamma & ::= & (vars \mapsto \tau)* \\
TT & ::= & (c_m \mapsto \tau_s)* \\
\text{dynamic table } DT & ::= & (c_m \mapsto \lambda x.e ::: \tau_s)* \\
\text{dynamic env } E & ::= & (vars \mapsto v)* \\
CT & ::= & (c_m \mapsto \lambda x.e ::: \tau_s)*
\end{array}
$$

extend to Cast Insertion

$$
\begin{array}{rcl}
\text{expressions } e & += & \langle \tau_d \Leftarrow \tau \rangle e \\
\text{values } v & += & \langle \tau_d \Leftarrow \tau \rangle \xi \\
\tau_d & ::= & \tau \mid [m : \tau_s]
\end{array}
$$

図 5 本研究で扱う Ruby サブセット

型とクラス型 A と，Gradual typing で扱う動的型を表す ? が含まれる．なお，代入式とメソッド定義を型注釈無しで書いた場合は，?型と型注釈した場合の構文糖衣となる．c_m はメソッドであり，クラス A のメソッド m という意味となる．$vars$ は self と変数の集合となる．

メソッドテーブル MT と型環境 Γ はそれぞれメソッドと型，変数と型の対応関係を表すものであり，型付け規則で用いる．今回のサブセットにおいては，クラスの親子関係に基づくメソッドの継承を扱っているが，MT には継承によるメソッドの型を含まない．したがって，$A.m \in dom(MT)$ である場合は，メソッド m はクラス A で必ず定義されている．また TT は，継承を含めたメソッドと型の対応を表す．TT は MT を参照する．$A.m \notin dom(MT)$ かつ $A.m \in dom(TT)$ であれば，クラス A におけるメソッド m は，先祖クラスから継承していることを表す．

DT，E は動的テーブル，環境であり，それぞれメソッドと定義本体 (型も含む)，変数と値の対応

$$
\begin{array}{rcl}
badcast\ e & ::= & \exists v\ \tau_0\ \tau_1\ \tau_1'\ \tau_d \\
 & & e = \langle \tau_d \Leftarrow \tau_1' \rangle \langle \tau_1 \Leftarrow \tau_0 \rangle v \wedge \\
 & & \tau_0 \not\lesssim \tau_d \\
BadCast\ e & ::= & \exists e' : e \text{ の部分式}, badcast\ e' \\
typeof & ::= & (v \mapsto \tau)*
\end{array}
$$

図 6 *badcast*，*BadCast*，*typeof* の定義

関係を表す．DT は MT と同様に，継承によるメソッド定義は含まないものとする．CT は，継承を含むメソッドとの対応関係を表す．CT は DT を参照する．これらは評価規則で用いる．

さらに，Gradual typing では，? 型を明らかにするためのキャストを挿入する必要があるため，キャストのための構文を追加している．キャストは $\langle \tau_d \Leftarrow \tau \rangle$ で表され，τ がキャスト前の型，τ_d がキャスト先の型である．また，ξ にキャストが挿入されたものが値 v に追加されている．ξ は，キャストが含まれない基本型を表すので，基本型にキャストがひとつだけ付けられたものは値であることを意味する．キャスト先の型には，τ に加えて $[m : \tau_s]$ が含まれている．これは，メソッド m を持つクラスを表現する型を意味しており，レシーバが ? の時のメソッド呼び出しで用いる．

また，キャストに関連して，*badcast* と *BadCast* と *typeof* を図 6 に定義する．*badcast* は，両辺の型の間に，Gradual typing で用いられる consistent-subtyping 関係 $\lesssim_{TT}$ を満たしていないキャストを値に適用した式を意味している．$\lesssim_{TT}$ については，3.2 節で詳しく説明する．

BadCast は，部分式に *badcast* が含まれた式を意味している．

typeof は値を受け取り，それに対応する型を返す．したがって，*typeof* で得られる型は，Nil または A である．

3.2 consistent-subtyping 関係

consistent-subtyping 関係 $\lesssim_{TT}$ には，Gradual typing で新たに定義されている consistent 関係 $\sim_{TT}$ と，subtyping 関係 $<:_{TT}$ が関わっている．右下に添え字 TT があるのは，これらの関係は，TT により変化するためである．$\sim_{TT}$ 関係とは，両方の型において，?型でない既知の部分において等価である関係のことである．クラス間の $\sim_{TT}$ 関係をチェックするときは，クラスに属するメソッドの引数における $\sim_{TT}$ 関係と返り値における $\sim_{TT}$ 関係をチェックする．図 7 に $\sim_{TT}$ 関係を示す．ここで M_{TT} は，TT に含まれるクラス A で利用できるメソッドの集合であり，

$$M_{TT}(A) = \{m \mid A.m \in dom(TT)\}$$

のように定義できる．

$$
\frac{}{\tau \sim_{TT} \tau} \quad \frac{}{\text{Nil} \sim_{TT} ?} \quad \frac{}{? \sim_{TT} \text{Nil}}
$$

$$
\frac{}{? \sim_{TT} A} \quad \frac{}{A \sim_{TT} ?}
$$

$$
\frac{\tau_1 \sim_{TT} \tau_1' \quad \tau_2 \sim_{TT} \tau_2'}{\tau_1 \to \tau_2 \sim_{TT} \tau_1' \to \tau_2'}
$$

$$
\frac{M_{TT}(A) = M_{TT}(A') \quad \forall m \in M_{TT}(A), TT(A.m) \sim_{TT} TT(A'.m)}{A \sim_{TT} A'}
$$

図 7 consistent $\sim_{TT}$ 関係

$$
\frac{}{\tau <:_{TT} \tau}
$$

$$
\frac{\tau_1' <:_{TT} \tau_1 \quad \tau_2 <:_{TT} \tau_2'}{\tau_1 \to \tau_2 <:_{TT} \tau_1' \to \tau_2'}
$$

$$
\frac{\forall m \in M_{TT}(A), m \in M_{TT}(A') \wedge TT(A.m) <:_{TT} TT(A'.m)}{A <:_{TT} A'}
$$

図 8 subtyping $<:_{TT}$ 関係

$$
\frac{\exists \tau', \tau_1 \sim_{TT} \tau' \wedge \tau' <:_{TT} \tau_2}{\tau_1 \lesssim \tau_2}
$$

図 9 consistent-subtyping $<:_{TT}$ 関係

次に，$<:_{TT}$ 関係を図 8 に示す．同じ型どうしにおいては $<:_{TT}$ 関係にある．また，クラス間の $<:_{TT}$ 関係 $A <:_{TT} A'$ は，クラス A' に属すメソッドすべてがクラスに属しており，それぞれのメソッドの型が $<:_{TT}$ 関係を満たす場合に，$A <:_{TT} A'$ であると定義している．

$\lesssim_{TT}$ 関係は，$\sim_{TT}$ 関係と $<:_{TT}$ 関係により，推移できる型間の関係である．$\lesssim_{TT}$ の定義を図 9 に示す．

3.3 型付け規則

3.1 節で定義した Ruby のサブセットにおける型

$\boxed{(MT, \Gamma, e) \Rightarrow (MT', \Gamma', \tau, e')}$

$$\frac{}{(MT, \Gamma, [A]) \Rightarrow (MT, \Gamma, A, [A])}\ (\text{G\_Object}) \qquad \frac{}{(MT, \Gamma, \text{nil}) \Rightarrow (MT, \Gamma, \text{Nil}, \text{nil})}\ (\text{G\_Nil})$$

$$\frac{\Gamma(\text{self}) = \tau}{(MT, \Gamma, \text{self}) \Rightarrow (MT, \Gamma, \tau, \text{self})}\ (\text{G\_Self}) \qquad \frac{\Gamma(x) = \tau}{(MT, \Gamma, x) \Rightarrow (MT, \Gamma, \tau, x)}\ (\text{G\_Variable})$$

$$\frac{}{(MT, \Gamma, A.\text{new}) \Rightarrow (MT, \Gamma, A, A.\text{new})}\ (\text{G\_New}) \qquad \frac{\begin{array}{c}(MT, \Gamma, e_1) \Rightarrow (MT_1, \Gamma_1, \tau_1, e_1') \\ (MT_1, \Gamma_1, e_2) \Rightarrow (MT_2, \Gamma_2, \tau_2, e_2')\end{array}}{(MT, \Gamma, e_1; e_2) \Rightarrow (MT_2, \Gamma_2, \tau_2, e_1'; e_2')}\ (\text{G\_Sequence})$$

$$\frac{\begin{array}{c}(MT, \Gamma, e_0) \Rightarrow (MT_0, \Gamma_0, \tau_0, e_0') \\ (MT_0, \Gamma_0, e_1) \Rightarrow (MT_1, \Gamma_1, \tau_1, e_1') \qquad (MT_0, \Gamma_0, e_2) \Rightarrow (MT_2, \Gamma_2, \tau_2, e_2') \\ MT' = MT_1 = MT_2 \qquad \Gamma' = \Gamma_1 = \Gamma_2\end{array}}{(MT, \Gamma, \text{if } e_0 \text{ then } e_1 \text{ else } e_2) \Rightarrow (MT', \Gamma', \text{Nil}, \text{if } e_0' \text{ then } e_1' \text{ else } e_2')}\ (\text{G\_If})$$

$$\frac{(MT, \Gamma, e) \Rightarrow (MT', \Gamma', \tau', e') \quad \Gamma'(x) = \tau \quad \tau' \lesssim_{TT} \tau}{(MT, \Gamma, x :::? = e) \Rightarrow (MT', \Gamma', \tau, x ::: \tau = \langle\!\langle \tau \Leftarrow \tau' \rangle\!\rangle e')}\ (\text{G\_Assign1})$$

$$\frac{\begin{array}{c}(MT, \Gamma, e) \Rightarrow (MT', \Gamma', \tau', e') \quad \tau' \lesssim_{TT} \tau \\ x \notin dom(\Gamma') \vee \tau \neq ? \quad \Gamma'' = \Gamma'[x \mapsto \tau]\end{array}}{(MT, \Gamma, x ::: \tau = e) \Rightarrow (MT', \Gamma'', \tau, x ::: \tau = \langle\!\langle \tau \Leftarrow \tau' \rangle\!\rangle e')}\ (\text{G\_Assign2})$$

$$\frac{(MT, \Gamma[\text{self} \mapsto A, x \mapsto \tau_1], e) \Rightarrow (MT', \Gamma', \tau', e') \quad \tau' \lesssim_{TT} \tau_2 \quad TT(A.m) = \tau_1 \to \tau_2}{(MT, \Gamma, \text{def } A.m = \lambda x.e :::? \to ?) \Rightarrow (MT, \Gamma, \text{Nil}, \text{def } A.m = \lambda x.\langle\!\langle \tau_2 \Leftarrow \tau' \rangle\!\rangle e' ::: \tau_1 \to \tau_2)}\ (\text{G\_Def\_Mth1})$$

$$\frac{\begin{array}{c}(MT', \Gamma[\text{self} \mapsto A, x \mapsto \tau_1], e) \Rightarrow (MT'', \Gamma', \tau', e') \quad \tau' \lesssim_{TT} \tau_2 \\ A.m \notin dom(TT) \quad MT' = MT[A.m \mapsto \tau_1 \to \tau_2] \\ \forall C \in Dec(A), C.m \notin dom(MT)\end{array}}{\begin{array}{c}(MT, \Gamma, \text{def } A.m = \lambda x.e ::: \tau_1 \to \tau_2) \\ \Rightarrow (MT', \Gamma, \text{Nil}, \text{def } A.m = \lambda x.\langle\!\langle \tau_2 \Leftarrow \tau' \rangle\!\rangle e' ::: \tau_1 \to \tau_2)\end{array}}\ (\text{G\_Def\_Mth2})$$

$$\frac{\begin{array}{c}(MT', \Gamma[\text{self} \mapsto A, x \mapsto \tau_1], e) \Rightarrow (MT'', \Gamma', \tau', e') \quad \tau' \lesssim_{TT} \tau_2 \\ TT(A.m) = \tau_1' \to \tau_2' \quad MT' = MT[A.m \mapsto \tau_1' \to \tau_2'] \\ \forall C \in Dec(A), C.m \notin dom(MT) \\ (\tau_1 \to \tau_2) \lesssim_{TT} (\tau_1' \to \tau_2') \quad e'' = e'[x \mapsto \langle \tau_1 \Leftarrow \tau_1' \rangle x]\end{array}}{\begin{array}{c}(MT, \Gamma, \text{def } A.m = \lambda x.e ::: \tau_1 \to \tau_2) \\ \Rightarrow (MT', \Gamma, \text{Nil}, \text{def } A.m = \lambda x.\langle\!\langle \tau_2' \Leftarrow \tau' \rangle\!\rangle e'' ::: \tau_1' \to \tau_2')\end{array}}\ (\text{G\_Def\_Mth3})$$

$$\frac{\begin{array}{c}(MT, \Gamma, e_0) \Rightarrow (MT', \Gamma', A, e_0') \\ (MT', \Gamma', e_1) \Rightarrow (MT'', \Gamma'', \tau, e_1') \\ TT(A.m) = \tau_1 \to \tau_2 \quad \tau \lesssim_{TT} \tau_1\end{array}}{\begin{array}{c}(MT, \Gamma, e_0.m(e_1)) \\ \Rightarrow (MT'', \Gamma'', \tau_2, e_0'.m(\langle\!\langle (\tau \leftharpoonup \tau_1) \Leftarrow \tau \rangle\!\rangle e_1'))\end{array}}\ (\text{G\_Ivk1}) \qquad \frac{\begin{array}{c}(MT, \Gamma, e_0) \Rightarrow (MT', \Gamma', ?, e_0') \\ (MT', \Gamma', e_1) \Rightarrow (MT'', \Gamma'', \tau_1, e_1')\end{array}}{\begin{array}{c}(MT, \Gamma, e_0.m(e_1)) \\ \Rightarrow (MT'', \Gamma'', ?, (\langle [m : \tau_1 \to ?] \Leftarrow ? \rangle e_0').m(e_1'))\end{array}}\ (\text{G\_Ivk2})$$

図 10 本研究における型付け規則

付け規則を図 10 に示す．

この型付け規則では，式から型へのジャッジメントとしており，その際にメソッドテーブルと型環境も変化し得るようになっている．また，右辺の e'

は，型付け後のキャストを挿入した式を表す．キャスト挿入に関わる部分以外では，e' は e と同等である．ここで，キャストに関する関数 $\langle\!\langle \tau_d \Leftarrow \tau \rangle\!\rangle e$ を以下のように定義している．

$$\langle\!\langle \tau_d \Leftarrow \tau \rangle\!\rangle e \equiv \begin{cases} e & (\tau = \tau_d \text{の場合}) \\ \langle \tau_d \Leftarrow \tau \rangle e & (\tau \neq \tau_d \text{の場合}) \end{cases}$$

これは，キャスト元の型とキャスト先の型が一致していれば，キャストを挿入しないことを意味する．

G_Object は，あるクラスのインスタンスは，そのクラス型であることを定義しており，G_New も，クラス生成式はそのクラス型であると定義している．nil についても同様であり，Nil 型に型付けされる (G_Nil)．self と変数は，型環境においてそれらに対応する型を与える (G_Self および G_Variable)．G_Sequence は連続した式 $e_1; e_2$ に対する型付けを定義している．e_1，e_2 を順番に型付けするたびにメソッドテーブルと型環境も変化し得る．最終的には，e_1 を型付けした後のメソッドテーブル MT_1 と型環境 Γ_1 において式 e_2 を型付けした結果の型が付けられる．

G_If は，if 式への型付けを定義している．型付けの順番は，最初に条件式 e_0 を型付けし，その後に e_1 と e_2 を型付けする．if 式の型は，e_1 と e_2 の型に関わらず，必ず Nil 型となる．型環境 Γ については，e_1 を型付けした型環境 Γ_1 と，e_2 を型付けした型環境 Γ_2 が一致していなければならない．MT についても，e_1 を型付けした環境 MT_1 と，e_2 を型付けした環境 MT_2 が一致していなければならない．したがって，if 式内で変数への代入とメソッド定義をする場合は，e_1 と e_2 の両方で同じ型を持つように定義しなければならない．

G_Assign1，G_Assign2 では，変数への代入式に対する型付けを定義している．基本的に，代入式自体の型は，代入後に与えられる変数の型と等しい．代入式の型付けは，右辺式の型となっており，右辺式の型付けにより，メソッドテーブルと型環境が変化し得る．さらに，変数の型が更新された場合は，型付け後の型環境 Γ' には，左辺の変数と右辺式の型の対応関係が加えられる．また，代入式においては，どの場合においても型付け後の代入式の右辺にキャストが挿入される可能性がある．

G_Assign1 は，定義済みの変数に対する代入式で，その変数に ? 型と型注釈した場合 (すなわち型注釈がない場合) の型付けを定義している．この場合は，変数の型は代入前の型のまま更新されない．この時，右辺式に対しては，右辺式の型 τ' から変数の元の型 τ へのキャストが，必要ならば挿入される．

G_Assign2 は，未定義変数への代入式に対する型付けを定義している．代入式の右辺式の型と，左辺において注釈した型が $\lesssim_{TT}$ 関係を満たしていれば，変数の型は型注釈した通りの型になる．また，ここでは，変数が未定義または型注釈の型 τ が ? 型でないことが仮定されている．変数が未定義の場合は，τ が ? 型であってもこの規則が適用される．また，変数が定義済みの場合は，τ が ? 型でないことが条件となる．キャスト挿入については，G_Assign1 と同様であり，右辺式の型 τ' から，型注釈の型 τ に (必要ならば) キャストする．

G_Def_Mth1，G_Def_Mth2，G_Def_Mth3 は，メソッド定義における型付けを定義している．定義式の結果自体は Nil 型としている．ここで使用している $Dec(A)$ は，クラス A の子孫クラスの集合を意味する．G_Def_Mth1 は，自身で定義しているか継承しているかに関わらず定義済みのメソッド m を，型注釈なしで再定義した場合である．この場合は，再定義前の型と同じ型とするため，MT は変化しない．また，e の型 τ' から，返り値の型 τ_2 へのキャストを e' へ挿入する．G_Def_Mth2 は，未定義のメソッド m を定義した場合である．ただし，クラス A の子孫クラスに m が定義されていないことを仮定している．この場合，型注釈通りの型を持つクラス A 上のメソッド m を，MT に追加する．キャストについては，G_Def_Mth1 と同様で，e の型 τ' から新しいメソッド定義の型注釈における返り値の型 τ_2 へのキャストを挿入する．G_Def_Mth3 は，(継承を含み) 定義済みのメソッド $A.m$ を型注釈を挿入して再定義した場合であり，G_Def_Mth2 と同様に，子孫クラスで m が定義されていないことも前

提としている．さらに，型注釈の型 $\tau_1 \to \tau_2$ と，再定義前の型 $\tau_1' \to \tau_2'$ の間に $\lesssim_{TT}$ 関係があることも前提としている．この仮定は，継承において矛盾を生じさせないための制約である．この型付けでは，再定義するメソッドの型は，親クラスでのメソッドの型 $\tau_1' \to \tau_2'$ として，MT に型情報を追加する．キャストについては，親クラスのメソッドの型へのキャストとなる．したがって，メソッド本体の式 e' の中の引数 x に τ_1' から τ_1 へのキャスト，e' に τ_2 から τ_2' へのキャストが挿入される．

G_Ivk1，G_Ivk2 はメソッド呼び出しにおける型付けを定義しており，レシーバ e_0，引数 e_1 の順に型付けする．

G_Ivk1 は，レシーバが ? 型ではない明らかなクラス型であるメソッド呼び出しの場合である．この型付けでは，実引数の型が呼び出すメソッドの仮引数の型と $\lesssim_{TT}$ 関係にあれば，メソッド呼び出し式はメソッドの返り値の型に型付けられる．その際に，実引数に，実引数の型 τ から，$\tau \leftharpoonup \tau_1$ へのキャストを挿入する．ここで用いられている $\tau \leftharpoonup \tau_1$ は，τ と $\sim_{TT}$ 関係にあり，かつ，τ_1 の部分型である型 (τ' とする) を，すなわち $\tau \sim_{TT} \tau' \wedge \tau' <:_{TT} \tau_2$ となるような τ' を表している．

G_Ivk2 は，レシーバが ? 型の場合である．この型付けでは，引数の型に関わらず必ず ? 型とする．キャストについては，レシーバが ? であり，メソッド m を呼び出しているため，? からメソッド m を持っているクラスの型 $[m : \tau_1 \to ?]$ へのキャストをレシーバに挿入している．m の型は，τ_1 型となった式 e_1 を受けとり，返り値の型が ? となっているため，$\tau_1 \to ?$ としている．

図 10 の型付け規則を式 e に対して適用すると，必要に応じてキャストが挿入された式 e' が得られる．4 節での型健全性の証明においては，e' に対する型付けが必要となる．e' と e の違いはキャストの有無だけなので，キャスト以外の部分については，図 10 における型付け規則 (ただしキャストを挿入した式の部分は無視する) を e' にそのまま適用することができる．さらに，キャストのある式については，図 11 に示す型付け規則 G_Cast を新たに導入する．ここで，G_Cast は図 10 に合わせるため，$\Rightarrow$ の右側は 4 つ組を書いているが，実際には 4 つ目の要素は使わない．G_Cast は，キャストが挿入された値の型は，キャスト先の型としているだけである．

$$\frac{v = \langle \tau_d \Leftarrow \tau \rangle \xi}{(MT, \Gamma, v) \Rightarrow (MT, \Gamma, \tau_d, v)} \quad \text{(G\_Cast)}$$

図 11 キャスト挿入された値への型付け規則

3.4 評価規則

3.1 節で定義した Ruby のサブセットに対して，3.3 節で述べた型付け規則を適用した結果の式の評価規則を，図 12 に示す．

この評価規則では，式の 1 ステップ評価をジャッジメントとしており，その際に，DT と E が変化し得る．

E_Self は self の評価であり，self は，それに対応する値に評価される．E_Variable は変数についての評価で，self と同様である．

E_New は，A.new の評価である．インスタンスの生成のため，これは $[A]$ に評価される．

E_Assign1，E_Assign2 は，変数への代入式 $x ::: \tau = e$ を評価している．E_Assign1 は，右辺式が値ではなく，1 ステップ評価できる場合である．E_Assign2 は，右辺式が値になった場合であり，代入式の値を右辺式の値とする．この時，環境 E に左辺の変数と右辺の値の対応関係を追加する．

E_Sequence1，E_Sequence2 は，連続した式 $e_1; e_2$ についての評価であり，1 つ目の式 e_1 について場合分けしている．E_Sequence1 は e_1 を 1 ステップ評価できる場合の評価である．E_Sequence2 は e_1 が値になった場合であり，引き続き e_2 を評価する．

E_If，E_If_True，E_If_False は，if 式についての評価であり，条件式 e_0 について場合分けしている．ここで SV は，values から (あれば) キャストを除去するものであり，次のように定義される．

$\boxed{(DT, E, e) \to (DT', E', e')}$

$$\frac{E(\text{self}) = v}{(DT, E, \text{self}) \to (DT, E, v)}\ (\text{E\_Self}) \qquad \frac{E(x) = v}{(DT, E, x) \to (DT, E, v)}\ (\text{E\_Variable})$$

$$\frac{(DT, E, e) \to (DT', E', e')}{(DT, E, x ::: \tau = e) \to (DT', E', x ::: \tau = e')}\ (\text{E\_Assign1}) \qquad \frac{E' = E[x \mapsto v]}{(DT, E, x ::: \tau = v) \to (DT, E', v)}\ (\text{E\_Assign2})$$

$$\frac{}{(DT, E, A.\text{new}) \to (DT, E, [A])}\ (\text{E\_New}) \qquad \frac{(DT, E, e_1) \to (DT', E', e')}{(DT, E, (e_1; e_2)) \to (DT', E', (e'; e_2))}\ (\text{E\_Sequence1})$$

$$\frac{}{(DT, E, (v; e)) \to (DT, E, e)}\ (\text{E\_Sequence2}) \qquad \frac{(DT, E, e_0) \to (DT', E', e')}{(DT, E, \text{if } e_0 \text{ then } e_1 \text{ else } e_2) \to (DT', E', \text{if } e' \text{ then } e_1 \text{ else } e_2)}\ (\text{E\_If})$$

$$\frac{SV(v) = [A]}{(DT, E, \text{if } v \text{ then } e_1 \text{ else } e_2) \to (DT, E, e_1; \text{nil})}\ (\text{E\_If\_True}) \qquad \frac{SV(v) = \text{nil}}{(DT, E, \text{if } v \text{ then } e_1 \text{ else } e_2) \to (DT, E, e_2; \text{nil})}\ (\text{E\_If\_False})$$

$$\frac{DT' = DT[A.m \mapsto \lambda x.e ::: \tau_1 \to \tau_2]}{(DT, E, \text{def } A.m = \lambda x.e ::: \tau_1 \to \tau_2) \to (DT', E, \text{nil})}\ (\text{E\_Def\_Mth}) \qquad \frac{(DT, E, e_0) \to (DT', E', e_0')}{(DT, E, e_0.m(e_1)) \to (DT', E', e_0'.m(e_1))}\ (\text{E\_Ivk1})$$

$$\frac{(DT, E, e_1) \to (DT', E', e_1')}{(DT, E, v.m(e_1)) \to (DT', E', v.m(e_1'))}\ (\text{E\_Ivk2}) \qquad \frac{\begin{array}{c} SV(v_0) = [A] \\ CT(A.m) = \lambda x.e ::: \tau_1 \to \tau_2 \\ E' = E[\text{self} \mapsto [A], x \mapsto v_1] \end{array}}{(DT, E, v_0.m(v_1)) \to (DT, E', e)}\ (\text{E\_Ivk3})$$

$$\frac{CT(A.m) = \lambda x.e ::: \tau_1 \to \tau_2}{\begin{array}{l}(DT, E, (\langle (A \leftarrow [m : \tau' \to ?]) \Leftarrow A \rangle [A]).m(v_1)) \\ \to (DT, E, \langle\!\langle ? \Leftarrow \tau_2 \rangle\!\rangle ([A].m(\langle\!\langle \tau_1 \Leftarrow \tau' \rangle\!\rangle v_1)))\end{array}}\ (\text{E\_Ivk4}) \qquad \frac{(DT, E, e) \to (DT', E', e')}{(DT, E, \langle \tau_d \Leftarrow \tau \rangle e) \to (DT', E', \langle \tau_d \Leftarrow \tau \rangle e')}\ (\text{E\_Cast})$$

$$\frac{\tau_0 \neq \tau_2}{\langle \tau_2 \Leftarrow \tau_1' \rangle \langle \tau_1 \Leftarrow \tau_0 \rangle v \to \langle\!\langle (\tau_0 \leftharpoonup \tau_2) \Leftarrow \tau_0 \rangle\!\rangle v}\ (\text{E\_Cast\_Merge}) \qquad \frac{\tau_0 = \tau_2}{\langle \tau_2 \Leftarrow \tau_1' \rangle \langle \tau_1 \Leftarrow \tau_0 \rangle v \to v}\ (\text{E\_Cast\_Remove})$$

図 12 本研究における評価規則

$$\begin{cases} SV(\xi) = \xi \\ SV(\langle\!\langle \tau_d \Leftarrow \tau \rangle\!\rangle \xi) = \xi \end{cases} \tag{1}$$

E_If は，e_0 を 1 ステップ評価できる場合の評価である．E_If_True は，e_0 が型注釈の有無に関わらず値 $[A]$ になった場合であり，この場合は，条件成立という扱いとなり，$e_1; \text{nil}$ を評価する．E_If_False は， e_0 が型注釈の有無に関わらず値 nil となった場合であり，この場合は，条件が不成立の扱いとなり，$e_2; \text{nil}$ を評価する．G_If において，if 式の型は必ず nil 型となるとしているため，E_If_True と E_If_False において，評価後の式の後に nil を追加している．

E_Def_Mth は，メソッド定義式の評価である．型付け規則において，メソッド定義式の型は Nil としているため，評価についても，nil に評価している．この時，動的テーブル DT に定義したメソッドの場所 (クラス名)，メソッド名と，メソッド本体の対応関係を追加する．

E_Ivk1，E_Ivk2，E_Ivk3 は，メソッド呼び出し $e_0.m(e_1)$ の評価である．E_Ivk1 はレシーバ e_0 を 1 ステップ評価する．E_Ivk2 は，e_0 が型注釈の有無に関わらずクラス A のインスタンス $[A]$ と判明した後に，実引数 e_1 を 1 ステップ評価する．なお，e_0 を $[A]$ としていることにより，レシーバが nil のメソッド呼び出しは実行時エラーとなる．E_Ivk3 は，e_0 と e_1 が，それぞれ値 v_0，v_1 $(SV(v_0) = [A])$ となった場合の評価であり，この場合は，メソッド本体の評価に移行する．そのため，仮定において，CT にクラス名，メソッド名と，メソッド本体の対応関係が存在していることが条件となっている．また，クラス A のメソッド m の本体においては，評価後の E の self は v_0，メソッドの引数 x は実引数 v_1 に関係づけられる．

E_Ivk4 は，G_Ivk2 によりレシーバが？型と型付けされた時のメソッド呼び出しである．レシーバが $[A]$ と判明し，CT においてメソッド $A.m$ が定義されていれば，不明であった m が判明したことになるので，実引数 (τ' 型) に対して，$A.m$ の引数の型である τ_1 へのキャストを挿入する．さらに，G_Ivk2 において，このメソッド呼び出し式は？型になるとしているため，メソッド呼び出し式に対して，$A.m$ の返り値の型 τ_2 から？型へのキャストを挿入する．

E_Cast は，キャストを含んだ式についての評価である．キャストを含んだ式では，キャストは評価せず，式のみを 1 ステップ評価する．したがって，式が値に評価され，$\langle \tau_d \Leftarrow \tau \rangle \xi$ となった場合，値として扱う．

E_Cast_Merge，E_Cast_Remove は，1 つの値に対して 2 つのキャストが連続して出現したときに，キャストを削減するための評価である．E_Cast_Merge は，2 つのキャストを 1 つに統合する．統合したキャストは，$\langle\!\langle (\tau_0 \leftharpoonup \tau_2) \Leftarrow \tau_0 \rangle\!\rangle$ となる．E_Cast_Remove は，2 つのキャストにおいて，元の型 τ_0 と，最終的な目的の型 τ_2 が一致しているときには，実際にはキャストを行う必要はないので，2 つのキャストを消去する．

4. 型健全性の証明

本研究で定めた Ruby サブセットにおいて，次の 2 つの定理を証明することで，型健全性を証明する．型健全性とは，正しく型付けされた項において，以下の Progress と Preservation の両方を満たすことである．

- Progress
  正しく型付けされ，キャストが挿入された項 e' は，値となるか，$Badcast\ e'$ となるか，1 ステップ評価を進めることができる．
  if $(MT, \Gamma, e) \Rightarrow (MT', \Gamma', \tau, e')$
  then $(e' = v) \vee (Badcast\ e') \vee$
  $(\forall DT\ E, (dom(MT') = dom(DT)) \wedge$
  $(dom(\Gamma') = dom(E))\ \ \exists e''\ DT'\ E',$
  $(DT, E, e') \rightarrow (DT', E', e''))$
- Preservation
  正しく型付けされた項が評価できるならば，評価後の項も正しく型付けされている．
  if $((MT_0, \Gamma_0, e_0) \Rightarrow (MT_0', \Gamma_0', \tau, e_0')) \wedge$
  $((DT, E, e_0') \rightarrow (DT', E', e_1'))$
  then $\exists e_1''\ MT_1\ \Gamma_1\ MT_1'\ \Gamma_1',$
  $(MT_1, \Gamma_1, e_1') \Rightarrow (MT_1', \Gamma_1', \tau, e_1'')$

本稿では，$(MT, \Gamma, e) \Rightarrow (MT', \Gamma', \tau, e')$ に関する帰納法に基づく Progress の証明のみを記載する．

G_Object と G_Nil と G_Cast は値についての型付けのため，Progress を満たす．

それ以外の型付け規則により型付けされた項について考える．

G_Self の場合

$e = e' =$ self

G_Self により self の型は $\tau = \Gamma$(self) と型付けされる．self $\in dom(\Gamma)$ であるならば，必ず self $\in dom(E)$ となる．したがって，評価規則 E_Self が適用できて，E において束縛されている値に評価されるため，Progress を満たす．

G_Variable の場合

$e = e' = x$

G_Self と同様に Progress を満たす．

G_New の場合

$e = e' = A.\mathrm{new}$

A.new は，G_New により，A と型付けされる．A.new は，E_New により，$[A]$ と評価されるため，Progress を満たす．

G_Sequence の場合

$e = e_1; e_2 \quad e' = e_1'; e_2'$

帰納法の仮定により，e_1' は値であるか，$BadCast\ e_1'$ か，ある e_1'' が存在し，e_1' は e_1'' に 1 ステップ評価が進む．e_2' についても同様である．e_1' が値である場合は，E_Sequence2 により，e' は e_2' に評価される．$BadCast\ e_1'$ であれば $BadCast\ e'$ である．e_1' が e_1'' に 1 ステップ評価が進むならば，E_Sequence1 により，$e_1''; e_2'$ に評価される．

G_If の場合

$e = \mathrm{if}\ e_0\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2$

$e' = \mathrm{if}\ e_0'\ \mathrm{then}\ e_1'\ \mathrm{else}\ e_2'$

帰納法の仮定により，e_0' は値であるか，$BadCast\ e_0'$ か，ある e_0'' が存在し，e_0' は e_0'' に 1 ステップ評価が進む．e_1'，e_2' についても同様である．e_0' が値であるならば，e_0' は，$[A]$，nil，$\langle \tau_d \Leftarrow \tau \rangle [A]$，$\langle \tau_d \Leftarrow \tau \rangle \mathrm{nil}$ のいずれかになり得る．$SV(v) = [A]$ ならば，E_If_True により，e' は e_1' に評価が進む．$SV(v) = \mathrm{nil}$ ならば，E_If_False により，e' は e_2' に評価が進む．$BadCast\ e_0'$ ならば，$BadCast\ e'$ である．e_0' が e_0'' に 1 ステップ評価が進むならば，E_If により，e' は $\mathrm{if}\ e_0''\ \mathrm{then}\ e_1'\ \mathrm{else}\ e_2'$ に評価される．

G_Assign1 の場合

$e = x :::? = e_0 \quad e' = x ::: \tau = \langle\!\langle \tau \Leftarrow \tau' \rangle\!\rangle e_0'$

帰納法の仮定により，e_0' は値であるか，$BadCast\ e_0'$ か，または e_0'' が存在し，e_0' は e_0'' に評価が進む．

最初に，$\tau \neq \tau'$ の場合について考える．この時，$e' = x ::: \tau = \langle \tau \Leftarrow \tau' \rangle\ e_0'$ となる．e_0' が値 ξ であれば，E_Assign2 により評価が 1 ステップ進む．e_0' が $\langle \tau'' \Leftarrow \tau''' \rangle \xi$ であれば，代入右辺はキャストが 2 つ重なった形になり，$BadCast$ を起こすか，E_Cast_Merge あるいは E_Cast_Remove で評価が進む．e_0' が $BadCast$ であれば，$BadCast\ e'$ となる．e_0' が e_0'' に 1 ステップ評価が進むならば，E_Assign1 により評価が進む．

次に，$\tau = \tau'$ の場合については，$e' = x ::: \tau = e_0'$ となる．e_0' が値の時はキャストの有無によらず，E_Assign2 により評価が進む．$BadCast\ e_0'$ ならば $BadCast\ e'$ となる．e_0' が e_0'' に評価が進むときは，E_Assign1 により評価が進む．

G_Assign2 の場合

$e = x ::: \tau = e_0 \quad e' = x ::: \tau = \langle\!\langle \tau \Leftarrow \tau' \rangle\!\rangle e_0$

G_Assign1 の場合と同様である．

G_Def_Mth1 の場合

$e = \mathrm{def}\ A.m :::? \to ? = \lambda x.e_0$

$e' = \mathrm{def}\ A.m ::: \tau_1 \to \tau_2 = \lambda x.\langle\!\langle \tau_2 \Leftarrow \tau' \rangle\!\rangle e_0'$

帰納法の仮定により，e_0' は値であるか，$BadCast\ e_0'$ か，または e_0'' が存在し，e_0' は e_0'' に 1 ステップ評価が進む．$BadCast\ e_0'$ 以外の場合は，e' は，E_Def_Mth により，nil に評価される．$BadCast\ e_0'$ の場合は，$BadCast\ e'$ となる．

G_Def_Mth2 の場合

$e = \mathrm{def}\ A.m ::: \tau_1 \to \tau_2 = \lambda x.e_0$

$e' = \mathrm{def}\ A.m ::: \tau_1 \to \tau_2 = \lambda x.\langle\!\langle \tau_2 \Leftarrow \tau' \rangle\!\rangle e_0'$

G_Def_Mth1 の場合と同様である．

G_Ivk1 の場合

$e = e_0.m(e_1)$

$e' = e_0'.m(\langle\!\langle \tau' \Leftarrow \tau \rangle\!\rangle e_1')$ ただし $\tau' = \tau \leftharpoonup \tau_1$

帰納法の仮定により，e_0' は値であるか，$BadCast\ e_0'$ か，または e_0'' が存在し，e_0' は e_0'' に 1 ステップ評価が進む．e_1' についても同様である．また，G_Ivk1 の仮定により，e_0' の型は A で，$MT(A.m) = \tau_1 \to \tau_2$ である．最初に，$\tau \neq \tau'$ の場合について考える．この時，$e' = e_0'.m(\langle \tau' \Leftarrow \tau \rangle e_1')$ となる．e_0' が値 v である場合は，次のようになる．e_1' が値 ξ である場合，E_Ivk3 により評価が進む．e_1' が $\langle \tau \Leftarrow \tau_3 \rangle \xi$ の形であれば，G_Assign1 の時と同様にして，E_Cast_Merge あるいは E_Cast_Remove で評価が進むか $BadCast$ になるかのいずれかである．e_1' が e_1'' に 1 ステップ評価が進むならば，E_Ivk2 により評価が進む．e_0' が $BadCast\ e_0'$ ならば，$BadCast\ e'$ となる．e_0' が e_0'' に 1 ステップ評価が進むならば，E_Ivk1 により評価が進む．

最後に，$\tau = \tau_1$ の場合，$e' = e'_0.m(e'_1)$ となるため，キャストが関わる部分を除いて，$\tau \neq \tau_1$ の場合と同様である．

G_Ivk2 の場合

$e = e_0.m(e_1)$

$e' = (\langle [m : \tau_1 \to ?] \Leftarrow ? \rangle e'_0).m(e'_1)$

帰納法の仮定により，e'_0 は値であるか，$BadCast\ e'_0$ か，または e''_0 が存在し，e'_0 は e''_0 に 1 ステップ評価が進む．e'_1 についても同様である．G_Ivk1 と異なる部分は，レシーバ e'_0 に必ずキャストが挿入され，キャスト先の型が $[m : \tau_1 \to ?]$ となる点である．G_Ivk2 の仮定により，e'_0 は ? 型である．そのため，e'_0 は値 ξ とはならない．e'_0 が $\langle ? \Leftarrow \tau_3 \rangle \xi$ の形であれば，G_Assign1 の時と同様にして，E_Cast_Merge あるいは E_Cast_Remove で評価が進むか $BadCast$ になるかのいずれかである．このとき，ξ が $[A]$ であり，E_Cast_Merge によりキャストがマージされ，$\langle (A \leftharpoonup [m : \tau_1 \to ?]) \Leftarrow A \rangle [A]$ の形になれば，E_Ivk4 によりさらに評価が進む．e'_0 が $BadCast\ e'_0$ の場合は，$BadCast\ e'$ となる．e'_0 が e''_0 に 1 ステップ評価が進むならば，E_Ivk1 により評価が進む．

5. 関連研究

Ruby Type Checker [2] は，型注釈に基づく型チェックを行う仕組みを Ruby のライブラリとして実装したものである．型注釈を挿入した部分の型チェックを，Ruby 本来の型チェックとは異なるタイミングで行う．Ruby のライブラリとしての実装のため，型チェックは実行時に行われる．これに対して本研究で導入する Gradual typing は，部分的な静的型チェック・動的型チェックを行う．

Siek ら [3] はオブジェクトを扱えるように Gradual typing を拡張している．Ruby はオブジェクト指向言語なので，本研究は，[3] のオブジェクトに対する型付け規則を参考にした．本研究では，オブジェクト指向言語である Ruby に Gradual typing を導入することに加えて，代入式での変数への型注釈やメソッド定義式によるメソッドの追加が行えるように対応させているため，Siek らによるものとは異なる．

Vitousek ら [5] は，Python に対して，Gradual typing を導入し，実装している．動的変換セマンティックスを 2 つ設計しており，従来のものを含めて 3 つのセマンティックスを実装しており，それぞれを評価し比較している．ただし，この研究においては，メソッド追加に対応していない．

6. まとめ

本稿では，Ruby に対して Gradual typing を導入するための基本的な考え方を，Ruby サブセット，型付け規則，評価規則の順に示した．さらに型健全性の証明を一部行った．本研究で扱う Ruby サブセットは，メソッドの更新，追加が行えるため，Ruby のオープンクラスによるメソッド追加，メソッド更新に対応することができた．

今後の課題は，まず Preservation 定理の証明を行い，型健全性の完全な証明を行うことである．

次に，Ruby の method_missing に対応させることである．本研究におけるメソッド呼び出しにおける型付け規則や評価規則を拡張することで，method_missing に対応できると考えられる．

最後に，本稿で示した考え方を元に，実装とその評価を行うことである．それにより，Ruby に対して Gradual typing を導入することの有用性を立証できる．

参考文献

[1] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. *Proceedings of Scheme and Functional Programming 2006* (*SFP 2006*), *pp 81–92, 2006*

[2] Brianna M. Ren, John Toman, T. Stephen Strickland and Jeffrey S. Foster. The Ruby Type Checker. *Proceedings of the 28th Annual ACM Symposium on Applied Computing 2013* (*SFP 2013*), *pp 1565–1572, 2013*

[3] Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. *Proceedings of the 21st European Conference on Object-Oriented Programming 2007* (*ECOOP 2007*), *pp 2–27, 2007*

[4] Brianna M. Ren, and Jeffrey S. Foster. Just-in-Time Static Type Checking for Dynamic Languages. *Proceeding of the 37th ACM SIGPLAN Conference on Programming Language Design*

and Implementation (*PLDI 2016*), *pp 462–476, 2016*

[5] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. *Proceedings of the 10th ACM Symposium on Dynamic Languages* (*DLS 2014*), *pp 45–56, 2014*