情報処理学会論文誌　プログラミング　Vol. 3　No. 2　9–25 (Mar. 2010)

Ruby プログラムの制御フロー解析とその健全性の証明

松本　宗太郎[†1]　　南　出　靖　彦[†1]

Ruby プログラムの制御フロー解析を設計し，その健全性を操作的意味論に基づき証明する．Ruby などのスクリプト言語に関する静的プログラム解析は多く提案されているが，その健全性について言語の意味論に基づき証明がなされたものは少ない．本研究では，Ruby のコア言語について操作的意味論を定義し，その意味論に基づき制御フロー解析の健全性を証明する．Ruby のコア言語は，ruby 1.9 処理系の構文解析器によって得られる中間言語をモデルとし，メソッドの動的な定義・イテレータブロック・大域脱出などを含む．Ruby では，クラス・メソッドの定義は，宣言ではなく動的に評価される式であり，さらにメソッド定義の上書きが可能である．そのため，プログラムの実行時の制御フローによって，異なるクラス定義・メソッド定義が得られる可能性がある．本研究では，メソッド定義が評価されたかどうかについて，制御フローを区別する解析を行う．これにより，特に多く見られる，トップレベルでクラス・メソッドの定義を行うプログラムについては，メソッド呼び出しの正確な解析が可能となる．

Control Flow Analysis of Ruby Programs and Its Soundness

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

We design a semi-flow-sensitive control flow analysis for Ruby, and prove its soundness. Static analysis of scripting languages including Ruby has been actively studied recently, but few of the analysis proved their correctness with respect to language semantics. We formalize an operational semantics of the core of Ruby. It is designed based on the intermediate language obtained by the parser of ruby-1.9, and supports dynamic method definition, iterator blocks, and non-local exits. Our analysis is semi-flow-sensitive in the sense that it is not sensitive on the values of a variable at each program point, but on which method definitions have been evaluated. This sensitivity makes it possible to precisely analyze typical Ruby programs where the definition of a class is split into several top-level class definitions.

1. はじめに

プログラムの制御フローの解析は，様々なプログラム解析の基礎となる技術である[2),7)]．関数型言語やオブジェクト指向言語では，高階関数やオブジェクトの存在によりプログラムの制御フローは自明ではない．さらに，Ruby[10)] などのスクリプト言語では，制御フローに大きな影響を与える言語要素であるメソッドの定義が，通常の式として表現されるため，メソッド定義そのものが制御フローに依存する．そのため，プログラムの制御フローの解析はより複雑になっている．本論文では，メソッド定義に関して制御フローを区別する，Ruby プログラムの制御フロー解析を提案する．通常の制御フロー依存の解析では，変数の値の変化について制御フローを区別するが，そのような解析は解くべき制約の数が非常に大きくなることから実際に利用されることは稀である．我々の解析では，メソッド定義の変化のみに注目して制御フローを区別する．

Ruby の制御フロー解析において重要なのは，あるクラスのメソッド名にどのメソッド定義が対応するかという情報である．プログラム中に，同一メソッド名を持つメソッド定義が複数ある場合，制御フローを区別しない解析ではメソッド名に対してすべてのメソッド定義が対応するとして取り扱う．そのため，メソッド定義式について条件分岐などを含まない，本来は静的な解析によってメソッド定義を一意に定められるプログラムの場合に，解析の精度が低下する．このような条件分岐とは関係ないメソッド定義の上書きは，ライブラリの設計で多く見られることから，多くの Ruby プログラムにおいて解析の精度を向上させることができると考えられる．メソッド名に対応付けられたメソッド定義の情報をメソッド状況（configuration）として表現する．メソッド呼び出し式では，その式に対応付けられたメソッド状況を参照し，どのメソッド定義が有効なのかを考慮に入れながら解析を行う．メソッド状況は，メソッド定義式によって更新される．与えられたメソッド状況について，新しいメソッド定義で対応付けを与える．

さらに，Ruby のサブセットについて，制御フロー解析に基づく安全性解析が健全であることを証明する．安全性解析では存在しないメソッドの呼び出しがないことを検査する．まず，メソッド定義の動的な取扱いやイテレータブロックなどを含む Ruby のサブセットにつ

†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

いて操作的意味論を定義する．その操作的意味論に基づいて，安全性解析が健全であることを示す．Ruby プログラムに対する静的なプログラム解析はいくつか提案されているが，その解析の健全性について十分な議論がされたものはなかった．

1.1　Ruby におけるメソッド定義

Ruby におけるメソッド定義は，Java などと異なり，宣言ではなく式である．そのためプログラム中のメソッド呼び出し式について，プログラムの評価がそこにたどり着くまでの制御フローによって，呼び出されるメソッド定義が異なる可能性がある．

次のプログラムを実行すると，ローカル変数 x と y には，それぞれ整数 1 と文字列"a"が代入される．

```
class A                                    # 例 1 : pro.example1.rb
  def f() 1 end
end

a = A.new
x = a.f()

class A
  def f() "a" end
end

y = a.f
```

変数 x に対する代入式が評価された時点では 1 つ目の f の定義が評価されている．そのため，x に代入される値は整数 1 となる．しかし，y に対する代入式が評価された時点では，2 つ目の f の定義が評価されており，クラス A のメソッド f の定義は上書きされ，y に代入される値は文字列"a"となる．

さらに，メソッド定義式はネストすることが可能である．メソッド定義の本体にメソッド定義式を書くことで，外側のメソッド呼び出しにともなって新しいメソッドが定義される．

```
class A                                    # 例 2 : pro.example2.rb
  def f()
    def f() "a" end
    1
  end
end

a = A.new
x = a.f()
y = a.f()
```

変数 x に対する代入式が評価される時点では，クラス A に定義された外側のメソッド f が呼び出される．このメソッドは本体にメソッド f の定義式を含むため，外側のメソッドが呼び出された後メソッド定義式が評価された時点で，内側のメソッド f が上書きされる．変数 y への代入では，x への代入で呼び出されたメソッドとは異なるメソッドが呼び出される．x への代入で呼び出されるメソッドは外側の f であり，y への代入で呼び出されるメソッドは内側の f である．

また，メソッド定義式は，通常の式と同様に条件分岐などの構文の内側に記述することが可能である．

```
class A                                    # 例 3 : pro.example3.rb
  if g()
    def f() 1 end
  else
    def f() "a" end
  end
end

a = A.new
x = a.f
```

このようなプログラムでは，メソッド f の定義が if 式の内側に記述されていることから，if 式の条件であるメソッド呼び出し g() の返り値によって，どちらのメソッドが定義されるかが決定される．

これらの例から，Ruby プログラムの解析には，プログラムの制御フローに依存するような解析が有効であると考えられる．制御フローを区別しない解析を行う場合，メソッド f の呼び出しが，どちらのメソッド定義に対応するのかという情報がまったく失われてしまう．そのため，メソッド呼び出しにおいて，両方のメソッド定義が呼び出されると近似することになり，プログラムの制御フローを解析する際に大きく精度が失われることとなる．

本論文では，本節で例示した最初の 2 つのプログラムについて，メソッド呼び出しの正確な解析が可能な制御フロー解析アルゴリズムを提案する．この 2 つの例では，メソッド定義はプログラムの評価が進むに従って変化するが，条件分岐はないことから，静的にメソッド定義の正確な解析が可能である．最後の例に示した，メソッド定義が条件分岐に含まれるようなプログラムについては，実行時までどちらのメソッド定義が評価されるか決定されないことから，正確な解析は本質的に不可能であり，保守的な近似を行う．

1.2　制御フロー解析

前節で示したプログラムについて，制御フロー解析の例を示す．

Ruby ではクラス定義の内側にメソッド定義式を記述するが，ここではクラス定義を省略し，メソッド定義式自体にメソッドが定義されるクラスの情報を与える．クラスは，あらかじめ定義されているものとする．このとき，最初の例で示したプログラムは次の式によって表現される．

$$[\,\texttt{def A\#f} : \lambda x_1.\texttt{1}\,]^1$$
$$[\,a = [\,new(\texttt{A})\,]^3\,]^2$$
$$[\,x = [\,a.\texttt{f}(nil)\,]^5\,]^4$$
$$[\,\texttt{def A\#f} : \lambda x_2.\text{"}\texttt{a}\text{"}\,]^6$$
$$[\,y = [\,a.\texttt{f}(nil)\,]^8\,]^7$$

プログラムに含まれる式は一意なロケーションによって区別されるものとする．メソッド定義では，メソッドが定義されるクラス A を明示し，メソッド名とメソッド本体の式の対応を記述する．オブジェクトの作成は，Ruby では定数 A の new メソッドの呼び出しであるが，new 式から作成されるオブジェクトのクラスを判定するため，クラス名を与えた $new(\texttt{A})$ として表現する．

このプログラムについて，それぞれの式がプログラムの評価中に取る可能性のある値について，以下のような制約が得られる．式 e のとりうる値の集合を $[\![e]\!]$ と表記する．

$$\texttt{A} \in [\![[new(\texttt{A})]^3]\!]$$
$$[\![[new(\texttt{A})]^3]\!] \subseteq [\![a]\!]$$
$$[\![[\texttt{a}.\texttt{f}(nil)]^5]\!] \subseteq [\![x]\!]$$
$$[\![[\texttt{a}.\texttt{f}(nil)]^8]\!] \subseteq [\![y]\!]$$

最初の制約 $\texttt{A} \in [\![[\,new(\texttt{A})\,]^3]\!]$ は，new 式の値がクラス A のインスタンスであることから得られる．さらに，この式が代入式の右辺であることから，次の変数 a の値についての制約 $[\![[new(\texttt{A})]^3]\!] \subseteq [\![a]\!]$．が得られる．次の 2 つの制約は，変数 x と y に対する代入式から得られる．さらに，メソッド呼び出し式から，次のような条件付きの制約が得られる．

$$\texttt{A} \in [\![a]\!] \wedge D_5(\texttt{A},\texttt{f}) = \lambda x_1.\texttt{1} \Rightarrow \texttt{1} \in [\![[\texttt{a}.\texttt{f}(nil)]^5]\!]$$
$$\texttt{A} \in [\![a]\!] \wedge D_8(\texttt{A},\texttt{f}) = \lambda x_2.\text{"}\texttt{a}\text{"} \Rightarrow \text{"}\texttt{a}\text{"} \in [\![[\texttt{a}.\texttt{f}(nil)]^8]\!]$$

条件は，2 つの要素で構成されている．1 つ目の要素は，メソッド呼び出しのレシーバのクラスに関する条件である．2 つ目の要素は，実際に呼び出されるメソッド本体の式に関する条件である．ここで，D_5，D_8 はメソッドの環境で，クラス名とメソッド名の組からメソッド本体の式への写像であり，ロケーションごとに与えられる．1 つ目の制約では，ロケーション 5 においてクラス A のメソッド f が $\lambda x_1.\texttt{1}$ であることを条件としており，2 つ目の制約ではロケーション 8 においてクラス A のメソッド f が $\lambda x_2.\text{"}\texttt{a}\text{"}$ であることが条件となる．これらの条件が成り立つ場合に，メソッド呼び出しによって呼び出されるメソッド定義から，メソッド呼び出し式自体の値に関する制約が得られる．この場合は，メソッド本体の式から，式 (5) の値に 1 が含まれることや式 (8) の値に"a"が含まれることが，制約として得られる．

ロケーション 1 とロケーション 6 にあるメソッド定義式によってメソッド環境が更新される．前後の式に対するメソッド環境との関係を，次のような制約で表現する．

$$D_3 = D_1[\,(\texttt{A},\texttt{f}) \mapsto \lambda x_1.\texttt{1}\,]$$
$$D_8 = D_6[\,(\texttt{A},\texttt{f}) \mapsto \lambda x_2.\text{"}\texttt{a}\text{"}\,]$$

ここで $D_1[\,(\texttt{A},\texttt{f}) \mapsto \lambda x_1.\texttt{1}\,]$ として環境の更新を表現する．$D_3(\texttt{A},\texttt{f}) = \lambda x_1.\texttt{1}$ となる．また，プログラムの制御フローから，各ロケーションに対するメソッド環境の間の関係を，次のように記述できる．

$$D_2 = D_3 = D_4 = D_5 = D_6$$
$$D_7 = D_8$$

プログラムの評価を開始する時点でのメソッドの環境を $D_1 = \emptyset$ として与え，D と式の値に関する制約を解くことで，次のように制御フロー解析の結果が得られる．

$$D_5 = \{\ (\texttt{A},\texttt{f}) \mapsto \lambda x_1.\texttt{1}\ \}$$
$$D_8 = \{\ (\texttt{A},\texttt{f}) \mapsto \lambda x_2.\text{"}\texttt{a}\text{"}\ \}$$
$$[\![x]\!] = \{\ \texttt{1}\ \}$$
$$[\![y]\!] = \{\ \text{"}\texttt{a}\text{"}\ \}$$

実際のプログラムでは条件分岐などの制御構造から，プログラム中の各ポイントでありうるメソッド環境の集合を考える必要がある．メソッド環境の集合をメソッド状況 $\mathcal{D}$ と呼び，メソッド状況に含まれるメソッド環境によって，メソッド呼び出しを解析する．前節で示し

た，if 式にメソッド定義を含むプログラムは，次のように表現される．

$$[\,\texttt{if}\ [\,\texttt{self.g}(\mathit{nil})\,]^2\ \texttt{then}\ [\,\texttt{def A\#f} : \lambda x_1.\texttt{1}\,]^3\ \texttt{else}\ [\,\texttt{def A\#f} : \lambda x_2.\text{"}\texttt{a}\text{"}\,]^4\ \texttt{end}\,]^1$$
$$[\,a = [\,\mathit{new}(\texttt{A})\,]^6\,]^5;\ \ [\,x = [\,a.\texttt{f}(\mathit{nil})\,]^8\,]^7$$

このプログラムの制御フロー解析の結果は次のようになる．

$$\mathcal{D}_8 = \{\ \{\ (\texttt{A},\texttt{f}) \mapsto \lambda x_1.\texttt{1}\ \},\ \{\ (\texttt{A},\texttt{f}) \mapsto \lambda x_2.\text{"}\texttt{a}\text{"}\ \}\ \}$$
$$[\![x]\!] = \{\ \texttt{1}, \text{"}\texttt{a}\text{"}\ \}$$

メソッド状況 $\mathcal{D}_8$ は 2 つのメソッド環境を含む．if 式の，then 分岐からメソッド環境 $\{\ \ (\texttt{A},\texttt{f}) \mapsto \lambda x_1.\texttt{1}\ \ \}$ を含むメソッド状況が得られ，else 分岐からメソッド環境 $\{\ (\texttt{A},\texttt{f}) \mapsto \lambda x_2.\text{"}\texttt{a}\text{"}\ \}$ を含むメソッド状況が得られる．合流によって，それぞれの分岐から得られたメソッド状況の和集合が計算され，2 つのメソッド環境の両方を含むものとなる．ここから，メソッド呼び出し式でどちらのメソッド定義を呼び出すのかは決定できない．変数 `x` の値は `1` か`"a"`のいずれかとなる．

制御フロー解析は，プログラムから値に関する制約およびメソッド状況に関する制約を抽出し，それらの制約を解消するような，システムとして設計される．制御フロー解析の結果は，式のとりうる値の集合である．メソッド状況に関する制約は，式の制御構造に対応する関係によって表現されるほか，メソッド定義式の前後の式に与えられたメソッド環境の更新として表現される．

2. *SemiRuby*

本論文で取り扱う Ruby のサブセットを表現する抽象言語 *SemiRuby* を定義する．サブセットには，メソッド定義や Ruby の基本的な構文構造，インスタンス変数の操作，イテレータブロック，大域脱出が含まれる．Ruby のサブセットに対する形式的な意味論には文献 12) がある．Scheme の操作的意味論をもとに，クラスやオブジェクトに関する操作が付け加えられている[8]．本論文ではその操作的意味論をもとに，定数やクラス定義などを省略しメソッド定義の取扱いを変更した．

Ruby と *SemiRuby* の主な違いは次のとおりである．

クラス・メソッドの定義　*SemiRuby* には，クラス定義が含まれない．Ruby では class 式によってクラス定義を行うが，*SemiRuby* ではすべてのクラスがあらかじめ定義されているものとする．また，Ruby ではメソッド定義はメソッド定義の本体を含む def 式によって表現されるが，*SemiRuby* では def 式はクラス・メソッド名の組とメソッド定義識別子によって表現される．メソッド定義の本体の式は，識別子と式の組の形で，メソッド定義環境によって与えられる．

定数・インスタンス作成の取扱い　*SemiRuby* には定数が含まれない．Ruby ではインスタンスの作成は定数の new メソッドの呼び出しによって表現されるが，*SemiRuby* ではクラス名を引数にとる new 式 $\mathit{new}(C)$ によって表現する．

ローカル変数のスコープの取扱い　*SemiRuby* では，ローカル変数のスコープはスコープ式によって明示される．Ruby ではローカル変数のスコープはメソッド定義とイテレータブロックの 2 種類があるが，*SemiRuby* ではそれぞれに対応するスコープ式によって表現される．

イテレータブロック　*SemiRuby* では，イテレータブロックはラムダ式によって表現される．Ruby ではイテレータブロックはつねにメソッド呼び出し式に付随する形で与えられるが，*SemiRuby* ではラムダ式によって別の構文要素として表現される．イテレータブロックの起動は yield 式によって表現されるが，Ruby における yield 式は暗黙のうちにメソッド呼び出しに与えられたイテレータブロックを起動するのに対し，*SemiRuby* では起動するラムダ式を示す引数を yield 式に与える．

インスタンス変数の操作　*SemiRuby* では，インスタンス変数の操作に対し，どのオブジェクトのインスタンス変数を操作するのか対象を明示する．

大域脱出の表現　*SemiRuby* では，大域脱出はスロー式とキャッチ式の組によって表現される．Ruby では return や break といった大域脱出はそれぞれに対応する脱出先への脱出であるが，*SemiRuby* では対応するキャッチ式への脱出を表現するジャンプ式によって表現される．return 式，break 式は，対応する脱出先へキャッチ式を挿入し，キャッチ式へのジャンプ式によって表現される．

2.1 *SemiRuby* の構文規則

SemiRuby の構文規則を**図 1** に示す．プログラムは，メソッド環境 D，ヒープ H，プログラム式 $\bar{e}$ の組で表現される．プログラムは，スーパクラス環境 S とメソッド定義環境 M のもとで実行される．プログラム式は，次に評価される式を明示するフォーカス $\langle\ \rangle$ が含まれる．フロー解析においてプログラム中の各部分式を区別するためにロケーションを付加する．ロケーションは，未評価の式に付加するロケーション l と，評価開始後の式に付加するロケーション $\overline{l}$ を考える．式には未評価のロケーションが付き，評価文脈中では文脈に応じて評価開始後のロケーションが付く．未評価のロケーション l および評価開始後のロケー

$$
\begin{array}{rcll}
P & ::= & D, H \triangleright \overline{e} & \text{プログラム} \\
\overline{e} & ::= & E[\,\langle e \rangle\,] & \text{プログラム式} \\
e & ::= & x^l \mid s^l \mid p^\ell \mid p_{nil}^\ell & \text{式} \\
 & \mid & (x = e)^l & \\
 & \mid & e_1.\mathtt{f}(e_2)^l \mid (e_1; e_2)^l \mid nil^l & \\
 & \mid & \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{end}^l & \\
 & \mid & new(C)^l & \\
 & \mid & \mathtt{def}\, C\mathtt{\#f} : d^l & \\
 & \mid & load(se, \mathtt{@}x)^l \mid store(se, \mathtt{@}x, e)^l & \\
 & \mid & (\lambda x.e)^l \mid yield(e_1, e_2)^l & \\
 & \mid & (\nu x.e)^l \mid throw(t, e)^l \mid catch(t, e)^l & \\
se & ::= & s \mid p & \text{セルフまたはポインタ} \\
q & ::= & p_{nil} \mid p & \text{ポインタ} \\
E & ::= & E[\, catch(t, F)^{\overline{l}}\,] \mid F & \text{評価文脈} \\
F & ::= & [] & \\
 & \mid & F.\mathtt{f}(e)^{\overline{l_0}} \mid q^{\overline{l_1}}.\mathtt{f}(F)^{\overline{l_0}} \mid (F; e)^{\overline{l_0}} & \\
 & \mid & store(p, \mathtt{@}x, F)^{\overline{l_0}} \mid yield(F, e)^{\overline{l_0}} \mid yield(q^{\overline{l_1}}, F)^{\overline{l_0}} & \\
 & \mid & throw(t, F)^{\overline{l_0}} \mid (x = F)^{\overline{l_0}} & \\
 & \mid & \mathtt{if}\, F\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{end}^{\overline{l_0}} & \\
H & ::= & \{\, sf, \ldots \,\} & \text{ヒープ} \\
sf & ::= & x \mapsto q \mid p \mapsto v_h^l & \\
v_h & ::= & \langle C, IM \rangle \mid \lambda x.e & \\
IM & ::= & \{\, \mathtt{@}x \mapsto q, \ldots \,\} & \\
D & ::= & \{\, (C, \mathtt{f}) \mapsto d, \ldots \,\} & \text{メソッド環境} \\
M & ::= & \{\, d \mapsto \lambda s.\lambda x.e, \ldots \,\} & \text{メソッド定義環境} \\
S & ::= & \{\, C \mapsto C', \ldots \,\} & \text{スーパークラス環境}
\end{array}
$$

図 1　*SemiRuby* の構文規則
Fig. 1　Syntax rules of *SemiRuby*.

ション $\overline{l}$ を表すメタ変数として ℓ を利用する．議論に影響がない場合には，適宜ロケーションを省略する．プログラム式中のフォーカスやロケーションの区別は，簡約意味論に基づいて制御フロー解析の健全性を示すために導入した．

式 e には，代入式 $x = e$，メソッド呼び出し $e_1.\mathtt{f}(e_2)$，逐次実行 $e_1; e_2$，nil，条件分岐などの Ruby と同様の構文がある．また，ローカル変数 x，セルフ s，ポインタ p，nil ポインタ p_{nil} がある．セルフ s は通常の変数のように名前を持ち，それぞれのメソッド定義について区別される．これによって，制御フロー解析中で異なるメソッド中のセルフを区別する．ポインタには，通常のポインタ p のほかに，nil を表す特殊なポインタ p_{nil} がある．p_{nil} は他のどのポインタとも異なる．クラスのインスタンス作成は new 式 $new(C)$ によって表現される．クラスはクラス名で指定される．メソッド定義式 $\mathtt{def}\, C\mathtt{\#f} : d$ は，Ruby と異なりメソッドが定義されるクラス C を指定する．さらに，メソッド名とメソッド定義の識別子 d を与える．メソッド定義式の評価によって，クラス C のメソッド `f` がメソッド定義 d と関連付けられる．インスタンス変数の操作は，$load(se, \mathtt{@}x)$ と $store(se, \mathtt{@}x, e)$ によって表現される．インスタンス変数の操作の対象となるオブジェクトを se によって明示する．se はセルフかポインタである．イテレータブロックはラムダ式 $\lambda x.e$ によって表現される．イテレータブロックの適用 $yield(e_1, e_2)$ は，ラムダ式の適用を表現する．イテレータブロックを表すラムダ式を e_1 として与え，イテレータブロックへの引数を e_2 とする．スコープ式 $\nu x.e$ によってローカル変数のスコープが明示される．Ruby では，ローカル変数のスコープは 2 種類あり，メソッド定義の内側の場合とイテレータブロックの内側に限定される場合がある．この 2 つをスコープ式によって表現する．大域脱出はスロー式 $throw(t, e)$ とキャッチ式 $catch(t, e)$ の組で表現する．スロー式 $throw(t, e)$ は，対応するタグ t を持つ外側のキャッチ式 $catch(t, e)$ への脱出となる．break や return といった大域脱出は，スロー式とキャッチ式を利用して表現される．Ruby における break や return は脱出先が異なるが，この違いはキャッチ式を挿入する場所によって表現される．break に対応するキャッチ式は，メソッド呼び出しのすぐ外側に挿入される．return に対応するキャッチ式は，その式を含むメソッドの最も外側に挿入される．**図 2** に例を示した．図の左に示した Ruby プログラムでは，メソッド `f` の呼び出しに与えたイテレータブロック内に break が含まれており，`g` の呼び出しに与えたイテレータブロック内には return が含まれている．break は，これはタグ t_2 を持

```
def f(x)
  a = x.f {|y| break y }
  b = x.g {|y| return 1 }
end
```

$$
\begin{array}{l}
\lambda x. \\
\quad catch(t_1, \\
\quad\quad a = catch(t_2, \\
\quad\quad\quad\quad x.\mathtt{f}(\lambda y_1.throw(t_2, y_1))); \\
\quad\quad b = x.\mathtt{g}(\lambda y_2.throw(t_1, \mathtt{1})))
\end{array}
$$

図 2　スロー式・キャッチ式による大域脱出の表現
Fig. 2　Non-local exits representation with throw/catch.

つスロー式によって表現される．タグ t_2 を持つキャッチ式は，メソッド呼び出しのすぐ外側に挿入される．一方で，return に対応するスロー式はタグ t_1 を持つが，これはメソッドの最も外側に挿入されたキャッチ式に対応する．

評価文脈 E は，キャッチ式を含まない文脈 F を利用して定義される．

ヒープは，ローカル変数 x からポインタ q および，ポインタ p から値 v_h への写像である．ポインタ q は，通常のポインタ p か nil ポインタ p_{nil} である．v_h には，どの式から得られた値かを示すロケーション l が与えられ，v_h^l がヒープに格納される．値 v_h は，通常のオブジェクト $\langle C, IM\rangle$ かラムダ式 $\lambda x.e$ である．通常のオブジェクトは，オブジェクトのクラス名 C とインスタンス変数の環境 IM の組である．

あるクラスのメソッド名に関連付けられているのがどのメソッド定義であるかは，メソッド環境 D によって表現される．D において，クラス C のメソッド $\mathtt{f}$ がメソッド定義 d に関連付けられているとき，$D(C,\mathtt{f}) = d$ である．

クラスの継承関係はスーパクラス環境 S によって表現される．クラス C の親クラスは $S(C)$ として得られる．クラスの継承関係は，プログラムの実行を通じて変化しない[⋆1]．メソッド定義環境 M は，メソッド定義 d からメソッド本体を表現するラムダ式への写像である．メソッド定義本体は $\lambda s.\lambda x.e$ として表現される．1 つ目の引数 s はメソッド定義中におけるセルフであり，2 つ目の引数 x はメソッドへの引数である．

クラスからメソッドを探索する *lookup* 関数の定義を**図 3** に示す．$lookup_{S,M}(D,C,\mathtt{f})$ として，S と M のもとでの，D に定義されたクラス C のメソッド $\mathtt{f}$ の本体の式が得られる．$lookup^0_S(D,C,\mathtt{f})$ によって，対応するメソッド定義 d を求めたあと，M を参照しメソッド本体の式を得る．クラス C に対応するメソッドが定義されていない場合には，S よりクラスの継承関係を参照し，親クラスのメソッドを探索する．親クラスがない場合，つまり継承関係を一番上までたどったうえでメソッドが定義されていない場合には，メソッドが未定義であるとして $\perp$ を返す．

2.2　*SemiRuby* の操作的意味論

SemiRuby プログラムの操作的意味論を定義する．*SemiRuby* プログラムの操作的意味論は，プログラムの簡約関係で定義される．フォーカス $\langle\ \rangle$ の移動を明示的に示す必要があるため，通常よりも定義が複雑になっている．

⋆1 Ruby では，プログラム実行中にクラスの継承関係を変更することはできない．クラス定義式に別のスーパクラスを指定すると，実行時エラーとなる．

$$\frac{D(C,\mathtt{f}) = d}{lookup^0_S(D,C,\mathtt{f}) = d}$$

$$\frac{(C,\mathtt{f}) \notin \mathrm{dom}(D) \quad C' = S(C) \quad lookup^0_S(D,C',\mathtt{f}) = d}{lookup^0_S(D,C,\mathtt{f}) = d}$$

$$\frac{(C,\mathtt{f}) \notin \mathrm{dom}(D) \quad C \notin \mathrm{dom}(S)}{lookup^0_S(D,C,\mathtt{f}) = \perp}$$

$$\frac{lookup^0_S(D,C,\mathtt{f}) = d \quad M(d) = \lambda s.\lambda x.e}{lookup_{S,M}(D,C,\mathtt{f}) = \lambda s.\lambda x.e}$$

$$\frac{lookup^0_S(D,C,\mathtt{f}) = \perp}{lookup_{S,M}(D,C,\mathtt{f}) = \perp}$$

図 3　*lookup* 関数
Fig. 3　*lookup* function.

定義 1（*SemiRuby* プログラムの簡約）　*SemiRuby* プログラムの 1 ステップ簡約を次の形で定義する．

$$S, M \vdash D, H \triangleright \overline{e} \longrightarrow D', H' \triangleright \overline{e}'$$

S と M のもとで，プログラム $D, H \triangleright \overline{e}$ を 1 ステップ簡約すると，新しいプログラム $D', H' \triangleright \overline{e}'$ が得られる．簡約の規則を**図 4** に示す．フォーカスの移動に関する簡約規則を**図 5** に示す．□

scope の規則により，ローカル変数はスコープごとに p_{nil} で初期化される．ローカル変数の代入はヒープの更新により表現される．ローカル変数の参照には，var の規則が対応する．未定義の変数を参照した場合は，簡約がスタックする．pointer 規則によって，ポインタのロケーションが付け替えられる．この簡約によってポインタが評価済みに判別される．条件分岐における条件の式は，nil 以外の値の場合に真と判断され，nil の場合に偽となる．

メソッド呼び出し式は call 規則により，セルフの代入，引数名の付け替えを経て，メソッド定義本体に簡約される．セルフはポインタに置き換えられるが，引数は新しいローカル変数に置き換えられる．これは，ローカル変数は代入の可能性がある一方で，セルフに代入されることはないためである．未定義メソッドに対する呼び出しの場合（$lookup_{S,M}(D,C,\mathtt{f}) = \perp$）簡約がスタックする．yield 規則と lambda 規則によって，イテレータブロック付きのメソッド呼び出しと yield が表現される．ブロックに対応するラムダ式はヒープを経由し yield に

$S, M \vdash D, H \triangleright E[\langle (\nu x.e^{l}) \rangle] \longrightarrow D, H[y \mapsto p_{nil}] \triangleright E[\langle e[y/x]^{l} \rangle]$	$(\mathrm{fresh}\, y)$	**scope**
$S, M \vdash D, H \triangleright E[(x = \langle q^{\bar{l}} \rangle)^{\overline{l_0}}] \longrightarrow D, H[x \mapsto q] \triangleright E[\langle q^{\bar{l}} \rangle]$		**assign**
$S, M \vdash D, H \triangleright E[\langle x^{l} \rangle] \longrightarrow D, H \triangleright E[\langle q^{\bar{l}} \rangle]$	$(q = H(x))$	**var**
$S, M \vdash D, H \triangleright E[\langle q^{l} \rangle] \longrightarrow D, H \triangleright E[\langle q^{\bar{l}} \rangle]$		**pointer**
$S, M \vdash D, H \triangleright E[(\langle q^{\overline{l_1}} \rangle; e^{l_2})^{\overline{l_0}}] \longrightarrow D, H \triangleright E[\langle e^{l_2} \rangle]$		**seq**
$S, M \vdash D, H \triangleright E[\langle nil^{l} \rangle] \longrightarrow D, H \triangleright E[\langle p_{nil}^{\bar{l}} \rangle]$		**nil**
$S, M \vdash D, H \triangleright E[\mathtt{if}\, \langle p^{\bar{l}} \rangle\, \mathtt{then}\, e_1^{l_1}\, \mathtt{else}\, e_2^{l_2}\, \mathtt{end}^{\overline{l_0}}] \longrightarrow D, H \triangleright E[\langle e_1^{l_1} \rangle]$		**if(true)**
$S, M \vdash D, H \triangleright E[\mathtt{if}\, \langle p_{nil}^{\bar{l}} \rangle\, \mathtt{then}\, e_1^{l_1}\, \mathtt{else}\, e_2^{l_2}\, \mathtt{end}^{\overline{l_0}}] \longrightarrow D, H \triangleright E[\langle e_2^{l_2} \rangle]$		**if(false)**
$S, M \vdash D, H \triangleright E[p^{\overline{l_1}}.\mathtt{f}(\langle q^{\overline{l_2}} \rangle)^{\overline{l_0}}] \longrightarrow D, H[y \mapsto q] \triangleright E[\langle e_3[p/s, y/x]^{l_3} \rangle]$	$(H(p) = \langle C, \ldots \rangle^{l}, \mathit{lookup}_{S,M}(D, C, \mathtt{f}) = \lambda s.\lambda x.e_3^{l_3}, \mathrm{fresh}\, y)$	**call**
$S, M \vdash D, H \triangleright E[\mathit{yield}(p^{\overline{l_1}}, \langle q^{\overline{l_2}} \rangle)^{\overline{l_0}}] \longrightarrow D, H[y \mapsto q] \triangleright E[\langle e[y/x]^{l_3} \rangle]$	$(H(p) = (\lambda x.e^{l_3})^{l}, \mathrm{fresh}\, y)$	**yield**
$S, M \vdash D, H \triangleright E[\langle (\lambda x.e)^{l} \rangle] \longrightarrow D, H[p \mapsto (\lambda x.e)^{l}] \triangleright E[\langle p^{\bar{l}} \rangle]$	$(\mathrm{fresh}\, p)$	**lambda**
$S, M \vdash D, H \triangleright E[\langle (\mathtt{def}\, C\mathtt{\#f} : d)^{l} \rangle] \longrightarrow D', H \triangleright E[\langle p_{nil}^{\bar{l}} \rangle]$	$(D' = D[(C, \mathtt{f}) \mapsto d])$	**def**
$S, M \vdash D, H \triangleright E[\langle new(C)^{l} \rangle] \longrightarrow D, H[p \mapsto \langle C, \emptyset \rangle^{l}] \triangleright E[\langle p^{\bar{l}} \rangle]$	$(\mathrm{fresh}\, p)$	**new**
$S, M \vdash D, H \triangleright E[\langle load(p, @x)^{l} \rangle] \longrightarrow D, H \triangleright E[\langle q^{\bar{l}} \rangle]$	$(H(p) = \langle C, IM \rangle, IM(@x) = q)$	**load**
$S, M \vdash D, H \triangleright E[\langle load(p, @x)^{l} \rangle] \longrightarrow D, H \triangleright E[\langle p_{nil}^{\bar{l}} \rangle]$	$(H(p) = \langle C, IM \rangle, @x \notin \mathrm{dom}(IM))$	**load'**
$S, M \vdash D, H \triangleright E[store(p, @x, \langle q^{\bar{l}} \rangle)^{l_0}] \longrightarrow D, H' \triangleright E[\langle q^{\bar{l}} \rangle]$	$(H(p) = \langle C, IM \rangle, H' = H[p \mapsto \langle C, IM[@x \mapsto q] \rangle^{l}])$	**store**
$S, M \vdash D, H \triangleright E[\langle catch(t, e^{l})^{l_0} \rangle] \longrightarrow D, H \triangleright E[catch(t', \langle e[t'/t]^{l} \rangle)^{\overline{l_0}}]$	$(\mathrm{fresh}\, t')$	**catch**
$S, M \vdash D, H \triangleright E[catch(t, F[throw(t', \langle q^{\bar{l}} \rangle)^{\overline{l_1}}])^{\overline{l_0}}] \longrightarrow D, H \triangleright E[\langle q^{\bar{l}} \rangle]$	$(t = t')$	**throw**
$S, M \vdash D, H \triangleright E[catch(t, F[throw(t', \langle q^{\bar{l}} \rangle)^{\overline{l_1}}])^{\overline{l_0}}] \longrightarrow D, H \triangleright E[throw(t', \langle q \rangle^{\bar{l}})^{\overline{l_1}}]$	$(t \neq t')$	**throw'**
$S, M \vdash D, H \triangleright E[catch(t, \langle q^{\bar{l}} \rangle)^{\overline{l_0}}] \longrightarrow D, H \triangleright E[\langle q \rangle^{\bar{l}}]$		**throw"**

図 4　*SemiRuby* の簡約規則
Fig. 4　Reduction rules of *SemiRuby*.

渡される．オブジェクトに対して yield した場合，簡約がスタックする．

メソッド定義式の簡約には，def 規則が対応する．メソッド環境 D におけるメソッドの対応付けを更新する．メソッド定義の本体はあらかじめ M に含まれている．すでに D にメソッドが定義されていた場合は，新しいメソッドに上書きされる．

load 規則がインスタンス変数の参照，store 規則がインスタンス変数の更新に対応している．load' 規則は，未定義のインスタンス変数の参照に対応する．未定義のインスタンス変数が参照された場合は p_{nil} へと簡約される．

スロー式は，対応するタグを持つキャッチ式までの大域脱出を表現している．大域脱出では，通常の例外処理とは異なりタグが静的スコープ持つ．catch 規則では，$catch(t, e)$ についてタグ名を変更する．タグ名の変更によって，静的に脱出先が定められる．throw 規則は

$$S, M \vdash D, H \triangleright E[\langle (x = e)^l \rangle] \longrightarrow D, H \triangleright E[(x = \langle e \rangle)^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[\langle (e_1; e_2)^l \rangle] \longrightarrow D, H \triangleright E[(\langle e_1 \rangle; e_2)^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[\langle \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{end}^l \rangle] \longrightarrow D, H \triangleright E[\texttt{if}\, \langle e_0 \rangle\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{end}^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[\langle e_1.\texttt{f}(e_2)^l \rangle] \longrightarrow D, H \triangleright E[\langle e_1 \rangle.\texttt{f}(e_2)^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[\langle q^{\bar{l}} \rangle.\texttt{f}(e_2)] \longrightarrow D, H \triangleright E[q^{\bar{l}}.\texttt{f}(\langle e_2 \rangle)]$$
$$S, M \vdash D, H \triangleright E[\langle yield(e_1, e_2)^l \rangle] \longrightarrow D, H \triangleright E[yield(\langle e_1 \rangle, e_2)^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[yield(\langle q^{\bar{l}} \rangle, e_2)] \longrightarrow D, H \triangleright E[yield(q^{\bar{l}}, \langle e_2 \rangle)]$$
$$S, M \vdash D, H \triangleright E[\langle store(p, @x, e)^l \rangle] \longrightarrow D, H \triangleright E[store(p, @x, \langle e \rangle)^{\bar{l}}]$$
$$S, M \vdash D, H \triangleright E[\langle throw(t, e)^l \rangle] \longrightarrow D, H \triangleright E[throw(t, \langle e \rangle)^{\bar{l}}]$$

図 5　*SemiRuby* の簡約規則（フォーカスの遷移）
Fig. 5　Reduction rules of *SemiRuby* (focusing).

対応するタグを持つキャッチ式まで簡約される．

フォーカスの移動は，式に与えられたロケーションの付け替えをともなう．代入式の簡約を考えると，代入式自体に与えられたフォーカスが，代入の右辺に移動する．さらに，代入式が未評価であることを示していたロケーション l が，未評価ではないことを示すロケーション $\bar{l}$ へと付け替えられる．メソッド呼び出し式および yield 式は，複数の部分式を含むため，それらの部分式の評価順序に対応して，フォーカスが移動する．

また，特にポインタ q について $q^{\bar{l}}$ である場合，評価済みポインタと呼ぶ．

3. 制御フロー解析

SemiRuby プログラムの制御フロー解析を定義する．メソッド定義の情報を表現するためにメソッド状況を考え，プログラムの制御フローに応じてメソッド状況を更新する．これにより，メソッド定義に関して制御フロー依存の解析を定義する．*SemiRuby* の制御フロー解析は，各式のとりうる値の集合である付値 $\mathcal{A}$ を解析結果とする．

定義 2　付値 $\mathcal{A}$ は，次の型を持つ関数 f_1，f_2，f_3，f_4 の 4 つ組である．

$f_1 : L \cup Var \cup Self \cup Tag \cup Ptr \to 2^L$　　式の値の集合
$f_2 : L \times InstanceVar \to 2^L$　　インスタンス変数の値の集合
$f_3 : L \to 2^D$　　式の評価前のメソッド状況
$f_4 : L \cup Tag \to 2^D$　　式を評価後に得られるメソッド状況

ただし，L はロケーションの集合，Var は変数名の集合，$Self$ はセルフ名の集合，Tag はタグ名の集合，Ptr はポインタの集合，$InstanceVar$ はインスタンス変数名の集合とする．$\mathcal{A} = (f_1, f_2, f_3, f_4)$ のとき，$[\![]\!]_{\mathcal{A}}$，$\mathcal{F}[\![]\!]_{\mathcal{A}}$，$\mathcal{I}[\![]\!]_{\mathcal{A}}$，$\mathcal{O}[\![]\!]_{\mathcal{A}}$ と書き，それぞれ f_1，f_2，f_3，f_4 を参照する．また制御フロー解析中で，付値の各要素に関する制約を $[\![]\!]$，$\mathcal{F}[\![]\!]$，$\mathcal{I}[\![]\!]$，$\mathcal{O}[\![]\!]$ によって表現する．□

ロケーション l を持つ式のとりうる値の集合は $[\![l]\!]$ によって表現される．ローカル変数，セルフ変数，ポインタおよびタグについても $[\![]\!]$ によって，とりうる値が表現される．

$\mathcal{I}[\![]\!]$ および $\mathcal{O}[\![]\!]$ によって，メソッド状況を表現する．メソッド状況はメソッド環境の集合である．$\mathcal{I}[\![l]\!]$ として，式 l の評価の直前までの制御フローによって得られるメソッド状況を表現する．$\mathcal{O}[\![l]\!]$ によって，式 l を評価した後に得られるメソッド状況を表現する．メソッド定義式が実行されない場合，メソッド状況は変化しないため $\mathcal{I}[\![]\!]$ と $\mathcal{O}[\![]\!]$ は等しい．メソッド定義式では，$\mathcal{O}[\![]\!]$ は $\mathcal{I}[\![]\!]$ に含まれるメソッド環境を更新したものとなる．条件分岐では，それぞれのブランチから得られる $\mathcal{O}[\![]\!]$ が，合流地点でのメソッド状況に含まれる．

$[\![]\!]$，$\mathcal{F}[\![]\!]$，$\mathcal{I}[\![]\!]$，$\mathcal{O}[\![]\!]$ では，それぞれロケーションを引数とするが，未評価のロケーションと評価開始後のロケーションを区別しない．つまり $[\![\bar{l}]\!] = [\![l]\!]$ などとする．

3.1 式の制約

制御フロー解析では，メソッド呼び出し式や yield 式の解析で，オブジェクトのクラスやラムダ式の情報が必要である．この情報を表現するため，ロケーション情報を導入する．

定義 3（ロケーション情報）　オブジェクトが作成されるロケーションから，オブジェクトのクラスを与えるロケーション情報関数 $\mathcal{L}$ は，次のような型を持ち，プログラムごとに与えられる．

$\mathcal{L} : L \to Class \cup Lambda$

ここで $Class$ はクラス名の集合であり，$Lambda$ はラムダ式の集合である．ロケーションで示される式が，クラス C に対する new 式であるとき $\mathcal{L}(l) = C$ となる．ロケーションがラムダ式であるとき $\mathcal{L}(l) = \lambda x.e$ である．□

定義 4（式の制約）　S，M，$\mathcal{L}$ のもとで式 e^ℓ に与えられる制約を $\mathcal{C}_{e^\ell}^{S,M,\mathcal{L}}$ で表す[⋆1]．$\mathcal{C}_{e^\ell}^{S,M,\mathcal{L}}$ は，e^ℓ のすべての部分式に対して値に関する制約とメソッド状況に関する制約を求めた，その和集合である．それぞれの制約の生成規則は，**図 6** と**図 7** に示す．□

⋆1 自明な場合には，制約に対する S，M，$\mathcal{L}$ を省略して $\mathcal{C}_e$ などと表記する．

式	制約（値）
$(\texttt{def}\, C\texttt{\#f} : d)^{\ell_0}$	
$(e_1^{\ell_1}.\texttt{f}(e_2^{\ell_2}))^{\ell_0}$	$\forall l \in [\![\ell_1]\!].\forall D \in \mathcal{O}[\![\ell_2]\!].$ $\mathcal{L}(l) = C \wedge$ $lookup_{S,M}(D, C, \texttt{f}) = \lambda s.\lambda x.e_3^{\ell_3} \Rightarrow$ $[\![\ell_1]\!] \subseteq [\![s]\!] \wedge [\![\ell_2]\!] \subseteq [\![x]\!] \wedge [\![\ell_3]\!] \subseteq [\![\ell_0]\!]$
$(new(C))^{\ell}$	$\ell \in [\![\ell]\!]$
x^{ℓ}	$[\![x]\!] \subseteq [\![\ell]\!]$
s^{ℓ}	$[\![s]\!] \subseteq [\![\ell]\!]$
p^{ℓ}	$[\![p]\!] \subseteq [\![\ell]\!]$
$nil^{\ell}, p_{nil}^{\ell}$	
$(x = e^{\ell_1})^{\ell_0}$	$[\![\ell_1]\!] \subseteq [\![x]\!], [\![\ell_1]\!] \subseteq [\![\ell_0]\!]$
$(e_1^{\ell_1}; e_2^{\ell_2})^{\ell_0}$	$[\![\ell_2]\!] \subseteq [\![\ell_0]\!]$
$(\nu x.e^{\ell_1})^{\ell_0}$	$[\![\ell_1]\!] \subseteq [\![\ell_0]\!]$
$(throw(t, e_1^{\ell_1}))^{\ell_0}$	$[\![\ell_1]\!] \subseteq [\![t]\!]$
$(catch(t, e_1^{\ell_1}))^{\ell_0}$	$[\![\ell_1]\!] \subseteq [\![\ell_0]\!], [\![t]\!] \subseteq [\![\ell_0]\!]$
$load(se, @x)^{\ell_0}$	$\forall l \in [\![se]\!].\mathcal{F}[\![l, @x]\!] \subseteq [\![\ell_0]\!]$
$store(se, @x, e_1^{\ell_1})^{\ell_0}$	$[\![\ell_1]\!] \subseteq [\![\ell_0]\!],$ $\forall l \in [\![se]\!].[\![\ell_1]\!] \subseteq \mathcal{F}[\![l, @x]\!]$
$(\lambda x.e)^{\ell}$	$\ell \in [\![\ell]\!]$
$yield(e_1^{\ell_1}, e_2^{\ell_2})^{\ell_0}$	$\forall l \in [\![\ell_1]\!].\mathcal{L}(l) = \lambda x.e^{l'} \Rightarrow [\![\ell_2]\!] \subseteq [\![x]\!] \wedge [\![l']\!] \subseteq [\![\ell_0]\!]$
$\texttt{if}\, e_1^{\ell_1}\, \texttt{then}\, e_2^{\ell_2}\, \texttt{else}\, e_3^{\ell_3}\, \texttt{end}^{\ell_0}$	$[\![\ell_2]\!] \subseteq [\![\ell_0]\!], [\![\ell_3]\!] \subseteq [\![\ell_0]\!]$

図 6　値に関する制約
Fig. 6　Constraints on values.

3.1.1　値に関する制約

S，M，$\mathcal{L}$ のもとでの，式の値に関する制約の生成規則を図 6 に示す．

メソッド定義式 $\texttt{def}\, C\texttt{\#f} : d$　値に関する制約は生成されない．

メソッド呼び出し式 $e_1.\texttt{f}(e_2)$　レシーバ e_1 がクラス C のインスタンスであり，クラス C のメソッド f の本体が $\lambda s.\lambda x.e_3$ である場合，ラムダ式の引数の値とメソッド呼び出し式自体の値に関する制約が得られる．引数 s，x の値は，メソッド呼び出しのレシーバおよび引数の値を含む集合となる．また，メソッド呼び出し式自体の値は，メソッド本体の式 e_3 の値を含む．呼び出されるメソッド定義は，メソッド呼び出し式に与えられたメソッド環境から決定される．複数のメソッド環境が与えられる場合，それぞれ別の

式	制約（メソッド状況）
$(\texttt{def}\, C\texttt{\#f} : d)^{\ell_0}$	$\{\, D[(C, \texttt{f}) \mapsto d] \mid D \in \mathcal{I}[\![\ell_0]\!] \,\} \subseteq \mathcal{O}[\![\ell_0]\!]$
$(e_1^{\ell_1}.\texttt{f}(e_2^{\ell_2}))^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{I}[\![\ell_2]\!]$ $\forall l \in [\![\ell_1]\!].\forall D \in \mathcal{O}[\![\ell_2]\!].$ $\mathcal{L}(l) = C \wedge$ $lookup_{S,M}(D, C, \texttt{f}) = \lambda s.\lambda x.e_3^{\ell_3} \Rightarrow$ $\mathcal{O}[\![\ell_2]\!] \subseteq \mathcal{I}[\![\ell_3]\!] \wedge \mathcal{O}[\![\ell_3]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$(new(C))^{\ell}$	$\mathcal{I}[\![\ell]\!] \subseteq \mathcal{O}[\![\ell]\!]$
$x^{\ell}, s^{\ell}, q^{\ell}$	$\mathcal{I}[\![\ell]\!] \subseteq \mathcal{O}[\![\ell]\!]$
$nil^{\ell}, p_{nil}^{\ell}$	$\mathcal{I}[\![\ell]\!] \subseteq \mathcal{O}[\![\ell]\!]$
$(x = e^{\ell_1})^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$(e_1^{\ell_1}; e_2^{\ell_2})^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{I}[\![\ell_2]\!], \mathcal{O}[\![\ell_2]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$(\nu x.e^{\ell_1})^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$(throw(t, e^{\ell_1}))^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{O}[\![t]\!]$
$(catch(t, e^{\ell_1}))^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{O}[\![\ell_0]\!], \mathcal{O}[\![t]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$load(se, @x)^{\ell}$	$\mathcal{I}[\![\ell]\!] \subseteq \mathcal{O}[\![\ell]\!]$
$store(se, @x, e^{\ell_1})^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$(\lambda x.e)^{\ell}$	$\mathcal{I}[\![\ell]\!] \subseteq \mathcal{O}[\![\ell]\!]$
$yield(e_1^{\ell_1}, e_2^{\ell_2})^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{I}[\![\ell_2]\!]$ $\forall l \in [\![\ell_1]\!].$ $\mathcal{L}(l) = \lambda x.e^{\ell'} \Rightarrow \mathcal{O}[\![\ell_2]\!] \subseteq \mathcal{I}[\![\ell']\!] \wedge \mathcal{O}[\![\ell']\!] \subseteq \mathcal{O}[\![\ell_0]\!]$
$\texttt{if}\, e_1^{\ell_1}\, \texttt{then}\, e_2^{\ell_2}\, \texttt{else}\, e_3^{\ell_3}\, \texttt{end}^{\ell_0}$	$\mathcal{I}[\![\ell_0]\!] \subseteq \mathcal{I}[\![\ell_1]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{I}[\![\ell_2]\!], \mathcal{O}[\![\ell_1]\!] \subseteq \mathcal{I}[\![\ell_3]\!]$ $\mathcal{O}[\![\ell_2]\!] \subseteq \mathcal{O}[\![\ell_0]\!], \mathcal{O}[\![\ell_3]\!] \subseteq \mathcal{O}[\![\ell_0]\!]$

ただし，未評価でない式 $e^{\bar{l}}$ について，$\mathcal{I}[\![\bar{l}]\!]$ が左辺に現れる場合はその包含関係は制約に含めない．

図 7　メソッド状況に関する制約
Fig. 7　Constraints on method configurations.

メソッド定義が呼び出されるものとして解析される．

new 式 $new(C)$　new 式自体のロケーションが，$[\![]\!]$ に含まれる．new 式によって新しく作成されるオブジェクトのクラスは，$\mathcal{L}$ から得られる．

変数，ポインタ x，s，p　それぞれ変数またはポインタの値が式の値となる．

nil 式　nil に関する制約は生成されない．Ruby では nil は `NilClass` クラスのオブジェクトとして扱われ，メソッド呼び出しなどの操作が可能であるが，ここでは nil に対する操作は不正とし，解析の対象に含めない．

スコープ式 $\nu x.e$　変数のスコープに関する制約はないが，式 e とスコープ式の関係に関する制約が生成される．

インスタンス変数読み書き $load(se, @x)$，$store(se, @x, e)$　se のとりうる値のロケーション l について，$\mathcal{F}[\![l, @x]\!]$ によってインスタンス変数に関する制約が表現される．

スロー式，キャッチ式 $throw(t, e)$，$catch(t, e)$　スロー式自体は値を返さないため，スロー式の値に対する制約は生成されない．しかし，対応するキャッチ式の値に関して，制約が生成される．これは，タグ t の値に関する制約を通して表現される．return，break などの大域脱出によって実現される値の伝搬は，この制約によって表現される．

ラムダ式，yield 式 $\lambda x.e, yield(e_1, e_2)$　yield 式について，e_1 の値にラムダ式が含まれている場合に，引数 x の値と返り値に関する制約が生成される．

3.1.2　メソッド状況に関する制約

S，M，$\mathcal{L}$ のもとでの，プログラムのメソッド状況に関する制約の生成規則を図 7 に示す．

メソッド定義式 `def` C`#f` : d　メソッド状況 $\mathcal{I}[\![]\!]$ に含まれるメソッド環境 D について，メソッド名 `f` とメソッド定義 d の関係を更新し，$\mathcal{O}[\![]\!]$ を求める．

メソッド呼び出し式 e_1`.f(`e_2`)`　レシーバ，引数の式の評価順に基づいて，メソッド状況の関係を設定し，さらにメソッド本体の式の $\mathcal{I}[\![]\!]$ との関係を設定する．この処理によって，メソッド内におけるメソッド定義の取扱いを実現する．

スロー式，キャッチ式 $throw(t, e)$，$catch(t, e)$　大域脱出の制御フローを表現するため，タグ t の $\mathcal{O}[\![t]\!]$ について制約を生成する．

ラムダ式，yield 式 $\lambda x.e$，$yield(x, e)$　yield 式において，実行されるラムダ式の本体の式との関係を設定する．ラムダ式にメソッド定義式が含まれている場合を取り扱うことができる．

if 式 `if` e_1 `then` e_2 `else` e_3 `end`　条件分岐のブランチ e_2，e_3 の評価後のメソッド状況は，両方とも if 式の評価後のメソッド状況に含まれる．

他の式については，部分式の評価順序に従って $\mathcal{I}[\![]\!]$ と $\mathcal{O}[\![]\!]$ の関係を設定する．

3.2　プログラムの制約

式の制約を用いて，プログラム全体に関する制約を定義する．

定義 5（文脈に対する制約）　S，M，$\mathcal{L}$ のもとで，文脈 E，F にロケーション ℓ を持つ式が埋められるときの制約 $\mathcal{C}^{S,M,\mathcal{L}}_{E,\ell}$，$\mathcal{C}^{S,M,\mathcal{L}}_{F,\ell}$ を次のように定義する．

$$\mathcal{C}^{S,M,\mathcal{L}}_{E,\ell} = \mathcal{C}^{S,M,\mathcal{L}}_{E[\,\_^{\ell}\,]}$$
$$\mathcal{C}^{S,M,\mathcal{L}}_{F,\ell} = \mathcal{C}^{S,M,\mathcal{L}}_{F[\,\_^{\ell}\,]}$$

ここで，$\_^{\ell}$ はダミーの式で，$\_^{\ell}$ に対する制約 $\mathcal{C}^{S,M,\mathcal{L}}_{\_^{\ell}}$ は $\emptyset$ とする．　□

$E \equiv ([]; e_2^{l_2})^{\overline{l_0}}$ に対して文脈に対する制約の例を示す．

$$\begin{aligned}
\mathcal{C}_{E,\ell} &= \mathcal{C}_{E[\,_^{\ell}\,]} \\
&= \mathcal{C}_{(_^{\ell}; e_2^{l_2})^{\overline{l_0}}} \\
&= \mathcal{C}_{_^{\ell}}, \mathcal{C}_{e_2^{l_2}}, \mathcal{I}[\![\overline{l_0}]\!] \subseteq \mathcal{I}[\![\ell]\!], \mathcal{O}[\![\ell]\!] \subseteq \mathcal{I}[\![l_2]\!], \mathcal{O}[\![l_2]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!], [\![l_2]\!] \subseteq [\![\overline{l_0}]\!] \\
&= \mathcal{C}_{e_2^{l_2}}, \mathcal{I}[\![\overline{l_0}]\!] \subseteq \mathcal{I}[\![\ell]\!], \mathcal{O}[\![\ell]\!] \subseteq \mathcal{I}[\![l_2]\!], \mathcal{O}[\![l_2]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!], [\![l_2]\!] \subseteq [\![\overline{l_0}]\!]
\end{aligned}$$

定義 6（プログラム式の制約）　S，M，$\mathcal{L}$ のもとでのプログラム式 $\overline{e} = E[\langle e'^{l} \rangle]$ に対する制約 $\mathcal{C}^{S,M,\mathcal{L}}_{\overline{e}}$ を，次のように定義する．

$$\mathcal{C}^{S,M,\mathcal{L}}_{\overline{e}} = \mathcal{C}^{S,M,\mathcal{L}}_{E,l} \cup \mathcal{C}^{S,M,\mathcal{L}}_{e'^{l}}$$

□

定義 7（メソッド環境の制約）　S，M，$\mathcal{L}$ のもとでのプログラム $D, H \triangleright \overline{e}$ において，D と $\overline{e}$ の関係を表現する制約 $\mathcal{C}^{S,M,\mathcal{L}}_{D,\overline{e}}$ を次のように定義する．

$$\mathcal{C}^{S,M,\mathcal{L}}_{D,\overline{e}} = \begin{cases} D \in \mathcal{I}[\![l]\!] & \overline{e} \equiv E[\langle e'^{l} \rangle] \text{ のとき} \\ D \in \mathcal{O}[\![\overline{l}]\!] & \overline{e} \equiv E[\langle q^{\overline{l}} \rangle] \text{ のとき} \end{cases}$$

□

プログラム式のフォーカスが未評価の式 e'^{l} に与えられている場合，評価前のメソッド状況 $\mathcal{I}[\![l]\!]$ にメソッド環境 D が含まれる．フォーカスが評価済みのポインタ $q^{\overline{l}}$ に与えられている場合，評価後のメソッド状況 $\mathcal{O}[\![l]\!]$ にメソッド環境 D が含まれる．

定義 8（プログラムの制約）　S，M，$\mathcal{L}$ のもとでのプログラム $D, H \triangleright \overline{e}$ に対する制約 $\mathcal{C}^{S,M,\mathcal{L}}$ を，次のように定義する．

$$\mathcal{C}^{S,M,\mathcal{L}} = \mathcal{C}_M^{S,\mathcal{L}} \cup \mathcal{C}_H^{S,M,\mathcal{L}} \cup \mathcal{C}_{\overline{e}}^{S,M,\mathcal{L}} \cup \mathcal{C}_{D,\overline{e}}^{S,M,\mathcal{L}}$$

ただし，$\mathcal{C}_M^{S,\mathcal{L}}$，$\mathcal{C}_H^{S,M,\mathcal{L}}$ は，それぞれメソッド定義環境，ヒープに対する制約で，以下のように定義する．

$$\begin{aligned}
\mathcal{C}_M^{S,\mathcal{L}} &= \textstyle\bigcup_{d \mapsto \lambda s.\lambda x.e^\ell \in M} \mathcal{C}_{e^\ell}^{S,M,\mathcal{L}} \\
\mathcal{C}_H^{S,M,\mathcal{L}} &= \{\, [\![p]\!] \subseteq [\![x]\!] \mid x \mapsto p \in H \,\} \cup \\
&\quad \{\, [\![p']\!] \subseteq \mathcal{F}[\![l, @x]\!] \mid p \mapsto \langle C, IM\rangle^l \in H, @x \mapsto p' \in IM \,\} \cup \\
&\quad \{\, l \in [\![p]\!] \mid p \mapsto \langle C, IM\rangle^l \in H \,\} \cup \\
&\quad \{\, l \in [\![p]\!] \mid p \mapsto (\lambda x.e)^l \in H \,\}
\end{aligned}$$

□

メソッド定義環境に対する制約では，メソッド定義に含まれるラムダ式の本体の式 e^ℓ について，制約を求め，その和集合を求める．ヒープに対する制約では，ヒープに含まれるポインタ，変数，インスタンス変数の値について，制約を求める．

3.3　制約の解消

この制御フロー解析アルゴリズムで得られる制約は，標準的な制御フロー解析における制約と同じ形で表現でき，制約の解消について既知のアルゴリズムが利用できる．*SemiRuby* プログラムに含まれるクラス，メソッド名，メソッド定義の数はそれぞれ有限であり，メソッド環境もまた，それらの組合せであることから，有限となる．したがって，図 6，図 7 に示した総称量化子を用いた制約を，量化子を用いない形に展開することができる．

既存の制御フロー解析では，有限集合上の以下の形式の制約を用いる[5),6)]．

$$X \subseteq Y,\ x \in X,\ x \in X \Rightarrow Y \subseteq Z$$

ここで X，Y，Z は有限集合上の変数で，x は有限集合の要素である．

本研究で得られる制約を，この形式で表現できる．メソッド呼び出し式に対する制約を例に説明する．

$$\begin{aligned}
&\forall l \in [\![\ell_1]\!].\forall D \in \mathcal{O}[\![\ell_2]\!]. \\
&\quad \mathcal{L}(l) = C \wedge \mathit{lookup}_{S,M}(D, C, \mathtt{f}) = \lambda s.\lambda x.e_3^{\ell_3} \Rightarrow \\
&\qquad [\![\ell_1]\!] \subseteq [\![s]\!] \wedge [\![\ell_2]\!] \subseteq [\![x]\!] \wedge [\![\ell_3]\!] \subseteq [\![\ell_0]\!]
\end{aligned}$$

この制約は，ロケーションおよびメソッド環境全体の集合が有限であることから，以下のように $\forall$ を展開できる．セルフ変数 s に対する制約のみを示す．

Datalog プログラムの例

```
invoke(call_loc, vblk, blk) :-
  call(call_loc, recv_loc, name, n, vblk),
  instance(cls, recv_loc), config(vblk, config),
  method(config, cls, name, def), def(_, _, _, def, blk, n).

config(blk_in, config) :-
  invoke(_, v, blk), config(v, config),
  block_expr(blk, loc), vertex(loc, blk_in, _).

points_to(obj_loc, param_loc) :-
  invoke(call_loc, _, blk),
  param(blk, i, param_loc), arg(call_loc, i, arg_loc),
  points_to(obj_loc, arg_loc).
```

Datalog 関係の意味

`points_to`　ある式が，どの式で作成されたオブジェクトを指すか．$[\![]\!]$ に対応．

`instance`　式の値が，どのクラスのインスタンスとなり得るか．

`config`　ある式が，どのメソッド状況と関連付けられているか．$\mathcal{I}[\![]\!], \mathcal{O}[\![]\!]$ に対応．

`method`　メソッド名とメソッド定義の関連付け．メソッド環境 D に対応．

`call`　ロケーションで与えられた式が，メソッド呼び出し式であることを表現．

`def`　ロケーションで与えられた式が，メソッド定義式であることを表現．

`arg`, `param`　メソッド呼び出し・メソッド定義の引数を表現．

`vertex`　式とメソッド状況の対応付けに利用．

`block_expr`　メソッド定義の本体の式．

図 8　Datalog による制御フロー解析
Fig. 8　Flow analysis in datalog.

$$\begin{aligned}
&\{\, l \in [\![l_1]\!] \Rightarrow [\![l_1]\!] \subseteq X_{l,D}, \\
&\quad D \in \mathcal{O}[\![l_2]\!] \Rightarrow X_{l,D} \subseteq [\![s]\!] \mid \mathcal{L}(l) = C, \mathit{lookup}_{C,M}(D, C, \mathtt{f}) = \lambda s.\lambda x.e_3^{\ell_3} \,\}
\end{aligned}$$

ここでは，複数の条件を持つ制約を表現するために，新しい変数 $X_{l,D}$ を導入している．

3.4　実　　装

ここまでに述べた，メソッド定義について制御フロー依存な制御フロー解析アルゴリズムを実装した．実装には，Datalog 処理系である BDDBDDB を用いた[9)]．

```
$ ./flow.byte ../pro.example1.rb
Instance of variables::
  ../pro.example1.rb:16 (x) : Fixnum
  ../pro.example1.rb:16 (a) : A
  ../pro.example1.rb:16 (y) : String

$ ./flow.byte ../pro.example2.rb
Instance of variables::
  ../pro.example2.rb:10 (y) : String
  ../pro.example2.rb:10 (x) : Fixnum
  ../pro.example2.rb:10 (a) : A

$ ./flow.byte ../pro.example3.rb
Instance of variables::
  ../pro.example3.rb:10 (x) : String
  ../pro.example3.rb:10 (x) : Fixnum
  ../pro.example3.rb:10 (a) : A
```

図 9　制御フロー解析の実行例
Fig. 9　Example of flow analysis.

実装には Objective Caml を用いた．Ruby の構文規則は非常に複雑であることから構文解析は実装せず，ruby 処理系に含まれる構文解析器を利用した．ruby-1.9.1 のオブジェクトファイルをリンクして構文解析器を実行し，構文木の中間表現を Objective Caml の値に変換した．

制約の解消には BDDBDDB を利用した．BDDBDDB は Datalog 処理系であり，制約解消のアルゴリズムを Datalog で記述することによって，自然な形で制御フロー解析を表現できる．**図 8** に，メソッド呼び出し式に関する制約の生成を記述した Datalog プログラムと，関連する関係の説明を抜粋する．図 8 には，3 つの規則が含まれている．各規則は，`:-`記号によって区切られ，右辺の関係のリストは，左辺の関係が生成されるための条件となっている．

1 つ目の規則は，メソッド呼び出しが実際に呼び出すメソッド定義の本体を，発見するための規則である．`call` 関係によってメソッド呼び出し式を認識し，`instance` 関係によってレシーバ式のクラスを判定する．さらに，`config` 関係から引数評価後のメソッド状況を得た後，呼び出されるメソッドの情報をまとめた `invoke` 関係を生成する．

2 つ目の規則によって，メソッド呼び出しにともなう，制御フローに関する制約を生成する．`invoke` 関係および `config` 関係によって，メソッド呼び出し時のメソッド状況を取得し，`config` 関係によってメソッド本体の式の評価を開始する際のメソッド状況を設定する．

3 つ目の規則によって，メソッド呼び出しにともなう，メソッド仮引数の値に関する制約を生成する．`param` 関係，`arg` 関係から，メソッド呼び出し時の実引数，仮引数を取得し，実引数と仮引数の間の値の関係を設定する．

このように，Datalog を用いることで，制約の取扱いをほぼアルゴリズムの説明と同様に表現でき，メンテナンス性の高い形で制御フロー解析を実装できた．

制御フロー解析の実行例を**図 9** に示す．1.1 節で示した，3 つの Ruby プログラムに対する実行結果である．プログラムに含まれるローカル変数に代入されるオブジェクトのクラスを出力するものである．例 1 と例 2 に対しては，変数 `x`，`y` の値が，それぞれ `Fixnum`，`String` と正確に求められている．一方で，例 3 については，メソッド定義が if 式のブランチに含まれていることから，変数 `x` のクラスが `Fixnum` および `String` のいずれかであるとしか求められていない．

4. 健　全　性

ここまで説明した制御フロー解析の健全性を示す．制御フロー解析に基づく，安全性解析（Safety Analysis）を定義し，その安全性解析の健全性を示す[6]．

4.1　安全性解析

安全性解析では，プログラム中に含まれるメソッド呼び出し式が未定義のメソッドを呼び出さないこと，および yield がラムダ式以外に対して実行されないことを検証する．本来の Ruby では yield 式はイテレータブロックを指定する引数を持たないことから，この yield 式に対する安全性制約は不要である．ここでは *SemiRuby* の形式化から，健全性の証明のために yield に関する制約が必要となった．

定義 9（安全性解析制約）　プログラムに含まれるメソッド呼び出し式および yield 式について，次のような制約を与える．

式	制約（安全性解析）
$(e_1^{l_1}.\mathtt{f}(e_2^{l_2}))^{l_0}$	$\forall l \in [\![l_1]\!].\forall D \in \mathcal{O}[\![l_2]\!].\exists C.\mathcal{L}(l) = C \wedge \mathit{lookup}_{S,M}(D, C, \mathtt{f}) \neq \bot$
$\mathit{yield}(e_1^{l_1}, e_2^{l_2})^{l_0}$	$[\![l_1]\!] \subseteq \{\ l \mid \mathcal{L}(l) = \lambda y.e\ \}$

□

メソッド呼び出し式に対する制約では，レシーバがオブジェクトであり，そのクラスにメソッドが定義されていることを検査する．メソッド呼び出し式に複数のメソッド環境が与え

られている場合には，そのすべてのメソッド環境について，メソッド呼び出しが未定義メソッドを呼び出さないことを検査する．yield 式に対する制約では，引数 e_1 の値がつねにラムダ式であることを検査する．

以上の制約が解を持つ場合に，上記の意味での安全性が検証できる．ただし，nil に対するメソッド呼び出しと nil に対する yield 式の評価が発生する可能性がある．

4.2　Well-formed なプログラム

SemiRuby プログラムについて，自由なローカル変数やポインタが存在しないといった，基本的な性質を well-formedness として導入する．Well-formed なプログラムを実行する過程で，well-formed でないプログラムが得られることはない．

定義 10（Well-formed なプログラム） 以下の条件がすべて満たされるときに，S，M，$\mathcal{L}$ のもとでプログラム $D, H \triangleright \bar{e}$ が well-formed であるという．

(1)　すべてのローカル変数，セルフ変数が，λ，ν またはヒープ H によって束縛されている
(2)　すべてのメソッド定義 `def` C`#f` : d について，$d \in \mathrm{dom}(M)$
(3)　プログラムにすべての含まれるポインタ p について，$p \in \mathrm{dom}(H)$
(4)　プログラムにすべての含まれるポインタ p について，$H(p) = \langle C, IM \rangle^l$ ならば $\mathcal{L}(l) = C$
(5)　プログラムに含まれるすべてのポインタ p について，$H(p) = (\lambda x.e_1^{l'})^l$ ならば，ある式 e_2 に対して $\mathcal{L}(l) = \lambda x.e_2^{l'}$
(6)　プログラムに含まれるすべての new 式 $new(C)^l$ について，$\mathcal{L}(l) = C$
(7)　プログラムに含まれるすべてのラムダ式 $(\lambda x.e_1^{l'})^l$ について，ある式 e_2 が存在して，$\mathcal{L}(l) = \lambda x.e_2^{l'}$

□

条件 (5) と (7) では，ラムダ式の引数と本体の式 e_1 のロケーションに注目する．ラムダ式の本体の式 e_1 は，ローカル変数やセルフ変数への代入によって，一致しない場合がある．

補題 1（well-formedness の保存） S，M，$\mathcal{L}$ のもとで，プログラム $D, H \triangleright \bar{e}$ が well-formed であり，$S, M \vdash D, H \triangleright \bar{e} \longrightarrow D', H' \triangleright \bar{e}'$ ならば，$D', H' \triangleright \bar{e}'$ も well-formed である．□

4.3　安全性解析の健全性

安全性解析の健全性は，簡約意味論に基づく型システムの健全性の証明と同じ手法で示す．すなわち，以下を示す．

- 保存：安全性解析制約が解を持つ式を簡約すると安全性解析制約が解を持つ式が得られること．
- Progress：安全性解析制約が解を持つ式の簡約がスタックしないこと．

プログラムの簡約によって，新たな変数，ポインタ，タグが導入されるため制約の解となる付値は簡約とともに拡張していく必要がある．

定義 11（付値の拡張） 付値 $\mathcal{A}$ の拡張 $\mathcal{A}[w \mapsto L]$ を次のように定義する．

$$\mathcal{A}[w \mapsto L] = (\llbracket \rrbracket', \mathcal{F}\llbracket \rrbracket_{\mathcal{A}}, \mathcal{I}\llbracket \rrbracket_{\mathcal{A}}, \mathcal{O}\llbracket \rrbracket_{\mathcal{A}})$$
$$\text{ただし } \llbracket w' \rrbracket' = \begin{cases} L & (w' = w \text{ のとき}) \\ \llbracket w' \rrbracket_{\mathcal{A}} & (\text{それ以外の場合}) \end{cases}$$

ここで，w，w' は，変数，タグ，またはポインタ，L はロケーションの集合である．□

制約 $\mathcal{C}$ の任意の解が制約 $\mathcal{C}'$ の解であるとき，$\mathcal{C} \models \mathcal{C}'$ と書く．代入に関しては，次の補題が式の構造に関する帰納法で証明される．

補題 2（代入）

(1)　$\mathcal{C}_e, \llbracket p \rrbracket \subseteq \llbracket s \rrbracket \models \mathcal{C}_{e[p/s]}$
(2)　$\mathcal{S}$ が $\mathcal{C}_e$ の解ならば $\mathcal{S}[y \mapsto \llbracket x \rrbracket_{\mathcal{S}}]$ は $\mathcal{C}_{e[y/x]}$ の解である．
(3)　$\mathcal{S}$ が $\mathcal{C}_e$ の解ならば $\mathcal{S}[t' \mapsto \llbracket t \rrbracket_{\mathcal{S}}]$ は $\mathcal{C}_{e[t'/t]}$ の解である．

ただし，y，t' は fresh とする．□

補題 3（保存） プログラム $D, H \triangleright \bar{e}$ は，S，M，$\mathcal{L}$ のもとで well-formed であるとする．S，M，$\mathcal{L}$ のもとで，プログラム $D, H \triangleright \bar{e}$ に対する制約が解を持ち，$S, M \vdash D, H \triangleright \bar{e} \longrightarrow D', H' \triangleright \bar{e}'$ ならば，$D', H' \triangleright \bar{e}'$ に対する制約も解を持つ．□

保存補題の証明には，次の補題が必要となる．評価文脈に対する制約は，文脈の穴に埋められる式のロケーションに依存している．また，プログラムの簡約では評価文脈の中の式が置き換わるため，簡約の前後で評価文脈に対する制約は異なる．次の補題は，一定の条件の下で，簡約前の評価文脈に対する制約の解が，簡約後も解となることを保証する．

補題 4　以下が成り立つ．

- $\mathcal{C}_{E,l}, \llbracket l' \rrbracket \subseteq \llbracket l \rrbracket, \mathcal{O}\llbracket l' \rrbracket \subseteq \mathcal{O}\llbracket l \rrbracket \models \mathcal{C}_{E,l'}$
- $\mathcal{C}_{F,l}, \llbracket l' \rrbracket \subseteq \llbracket l \rrbracket, \mathcal{O}\llbracket l' \rrbracket \subseteq \mathcal{O}\llbracket l \rrbracket \models \mathcal{C}_{F,l'}$

□

この補題は，評価文脈の構造に関する帰納法で容易に証明できる．

保存補題の証明

$S, M \vdash D, H \triangleright \overline{e} \longrightarrow D', H' \triangleright \overline{e'}$ についての場合分けで証明する．主要な場合についてのみ証明を示す．

- $S, M \vdash D, H \triangleright E[\,\langle q_1^{\overline{l_1}} \rangle.\mathtt{f}(e_2^{l_2})^{\overline{l_0}}\,] \longrightarrow D, H \triangleright E[\,q_1^{\overline{l_1}}.\mathtt{f}(\langle e_2^{l_2} \rangle)^{\overline{l_0}}\,]$.
  $D, H \triangleright E[\,\langle q_1^{\overline{l_1}} \rangle.\mathtt{f}(e_2^{l_2})^{\overline{l_0}}\,]$，$D, H \triangleright E[\,q_1^{\overline{l_1}}.\mathtt{f}(\langle e_2^{l_2} \rangle)^{\overline{l_0}}\,]$ の制約を $\mathcal{C}$，$\mathcal{C}'$ とすると以下のように分解できる．

  $$\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}_0 \cup \mathcal{C}_{D,\overline{e}}$$
  $$\mathcal{C}' = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}'_0 \cup \mathcal{C}_{D,\overline{e'}}$$

  ただし，$\mathcal{C}_0$，$\mathcal{C}'_0$ は，$\langle q_1^{\overline{l_1}} \rangle.\mathtt{f}(e_2^{l_2})^{\overline{l_0}}$，$q_1^{\overline{l_1}}.\mathtt{f}(\langle e_2^{l_2} \rangle)^{\overline{l_0}}$ に対する制約とする．
  - ロケーションを変更しただけであり，$\mathcal{C}_0 \models \mathcal{C}'_0$.
  - $\mathcal{C}_0 \models \mathcal{O}[\![l_1]\!] \subseteq \mathcal{I}[\![l_2]\!]$ であり，$\mathcal{C}_{D,\overline{e}} = D \in \mathcal{O}[\![l_1]\!]$ かつ $\mathcal{C}_{D,\overline{e'}} = D \in \mathcal{I}[\![l_2]\!]$ なので，$\mathcal{C}_0, \mathcal{C}_{D,\overline{e}} \models \mathcal{C}_{D,\overline{e'}}$.
- $S, M \vdash D, H \triangleright E[\,p_1^{\overline{l_1}}.\mathtt{f}(\langle q_2^{\overline{l_2}} \rangle)^{\overline{l_0}}\,] \longrightarrow D, H[\,y \mapsto q_2\,] \triangleright E[\,\langle e_3^{l_3}[\,p_1/s, y/x\,] \rangle\,]$．ただし，$H(p) = \langle C, \ldots \rangle^l$，$lookup_{S,M}(D, C, \mathtt{f}) = \lambda s.\lambda x.e_3^{l_3}$.
  $D, H \triangleright E[\,p_1^{\overline{l_1}}.\mathtt{f}(q_2^{\overline{l_2}})\,]$ に対する制約を $\mathcal{C}$ とすると，次のように分解できる．$\mathcal{C}_0$ は，式 $p_1^{\overline{l_1}}.\mathtt{f}(q_2^{\overline{l_2}})^{\overline{l_0}}$ に対する制約である．

  $$\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}_0 \cup \mathcal{C}_{D,\overline{e}}$$

  $\mathcal{C}_0$ は $p_1^{\overline{l_1}}$，$q_2^{\overline{l_2}}$ に対する制約を含むので，次の制約が成り立つ．

  $$\mathcal{C}_0 \models [\![p_1]\!] \subseteq [\![\overline{l_1}]\!], [\![q_2]\!] \subseteq [\![\overline{l_2}]\!] \quad (1)$$

  メソッド状況に関する制約の定義より，$D \in \mathcal{O}[\![\overline{l_2}]\!]$．$H(p_1) = \langle C, \ldots \rangle^l$ と well-formedness より，$\mathcal{L}(l) = C$ である．D，l をメソッド呼び出しの制約（値），制約（メソッド状況）に適用することで，以下を得る．

  $$\mathcal{C}_0 \models [\![\overline{l_1}]\!] \subseteq [\![s]\!], [\![\overline{l_2}]\!] \subseteq [\![x]\!], [\![l_3]\!] \subseteq [\![\overline{l_0}]\!] \quad (2)$$
  $$\mathcal{C}_0 \models \mathcal{O}[\![\overline{l_2}]\!] \subseteq \mathcal{I}[\![l_3]\!], \mathcal{O}[\![l_3]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!] \quad (3)$$

  式 (1)，(2)，(3) を整理すると以下が得られる．

  $$\mathcal{C} \models [\![l_3]\!] \subseteq [\![\overline{l_0}]\!] \quad (4)$$
  $$\mathcal{C} \models \mathcal{O}[\![l_3]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!] \quad (5)$$

  $D, H[\,y \mapsto q_2\,] \triangleright E[\,\langle e_3^{l_3}[\,p_1/s, y/x\,] \rangle\,]$ に対する制約 $\mathcal{C}'$ は次のように分解できる．

  $$\mathcal{C}' = \mathcal{C}_M \cup \mathcal{C}_H \cup \{\, [\![q_2]\!] \subseteq [\![y]\!] \,\} \cup \mathcal{C}_{E,l_3} \cup \mathcal{C}'_3 \cup \mathcal{C}_{D,\overline{e'}}$$

  ただし，$\mathcal{C}'_3$ は $e_3^{l_3}[\,p_1/s, y/x\,]$ に対する制約とする．
  - 式 (4)，(5) と補題 4 より $\mathcal{C} \models \mathcal{C}_{E,l_3}$ が成り立つ．
  - $\mathcal{C}_{D,\overline{e'}} = D \in \mathcal{I}[\![l_3]\!]$ であり，$\mathcal{C} \models D \in \mathcal{O}[\![\overline{l_2}]\!]$，$\mathcal{C} \models \mathcal{O}[\![\overline{l_2}]\!] \subseteq \mathcal{I}[\![l_3]\!]$ なので，$\mathcal{C} \models \mathcal{C}_{D,\overline{e'}}$.
  - e_3 に対する制約が $\mathcal{C}_M$ に含まれることと補題 2 より，$\mathcal{C}$ の解を $\mathcal{S}$ とすると，$\mathcal{S}' = \mathcal{S}[\,y \mapsto [\![x]\!]_{\mathcal{S}}\,]$ は $\mathcal{C}'_3$ の解となる．
  - $\mathcal{C} \models [\![q_2]\!] \subseteq [\![x]\!]$ が成り立つので，$\mathcal{S}'$ は $[\![q_2]\!] \subseteq [\![y]\!]$ を満たす．

  さらに $\mathcal{S}'$ は $\mathcal{C}$ の解でもあるので，$\mathcal{C}'$ の解となる．
- $S, M \vdash D, H \triangleright E[\,\langle \mathtt{def}\, C\mathtt{\#f} : d^{l_0} \rangle\,] \longrightarrow D', H \triangleright E[\,\langle p_{nil}^{\overline{l_0}} \rangle\,]$．ただし，$D' = D[\,(C, \mathtt{f}) \mapsto d\,]$.
  $D, H \triangleright E[\,\langle \mathtt{def}\, C\mathtt{\#f} : d^{l_0} \rangle\,]$，$D', H \triangleright E[\,\langle p_{nil}^{\overline{l_0}} \rangle\,]$ に対する制約 $\mathcal{C}$，$\mathcal{C}'$ は次のように分解される．

  $$\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,l_0} \cup \mathcal{C}_d \cup \mathcal{C}_{D,\overline{e}}$$
  $$\mathcal{C}_d = \{\, \{\, D[\,(C, \mathtt{f}) \mapsto d\,] \mid D \in \mathcal{I}[\![l_0]\!] \,\} \subseteq \mathcal{O}[\![l_0]\!] \,\}$$
  $$\mathcal{C}' = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}_{D',\overline{e'}}$$

  ここで，$\mathcal{C}_d$ はメソッド定義式から生成される制約である．制約（メソッド状況）の定義より，$\mathcal{C} \models D \in \mathcal{I}[\![l_0]\!]$．上の制約から，$\mathcal{C} \models D' \in \mathcal{O}[\![l_0]\!]$ であり，$\mathcal{C}_{D,\overline{e'}} = D' \in \mathcal{O}[\![l_0]\!]$ なので成り立つ．
- $S, M \vdash D, H \triangleright E[\,catch(t, F[\,throw(t, \langle p^{\overline{l}} \rangle)^{\overline{l_2}}\,]^{\overline{l_1}})^{\overline{l_0}}\,] \longrightarrow D, H \triangleright E[\,\langle p^{\overline{l}} \rangle\,]$.
  $E[\,catch(t, F[\,throw(t, \langle p^{\overline{l}} \rangle)^{\overline{l_2}}\,]^{\overline{l_1}})^{\overline{l_0}}\,]$ に対する制約 $\mathcal{C}$ は，以下のように分解できる．

  $$\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}_{F,\overline{l_1}} \cup \mathcal{C}_g \cup \mathcal{C}_t \cup \mathcal{C}_{D,e}$$
  $$\mathcal{C}_g = [\![l_1]\!] \subseteq [\![\overline{l_0}]\!], [\![t]\!] \subseteq [\![\overline{l_0}]\!], \mathcal{O}[\![\overline{l_1}]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!], \mathcal{O}[\![t]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!]$$
  $$\mathcal{C}_t = [\![p]\!] \subseteq [\![\overline{l}]\!], [\![\overline{l}]\!] \subseteq [\![t]\!], \mathcal{O}[\![\overline{l}]\!] \subseteq \mathcal{O}[\![t]\!]$$

  これより，次の関係が成り立つ．

  $$\mathcal{C} \models [\![\overline{l}]\!] \subseteq [\![\overline{l_0}]\!] \quad (6)$$
  $$\mathcal{C} \models \mathcal{O}[\![\overline{l}]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!] \quad (7)$$

  $D, H \triangleright E[\,\langle p^l \rangle\,]$ に対する制約 $\mathcal{C}'$ は，以下のように分解できる．

$$\mathcal{C}' = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\bar{l}} \cup \{\, [\![p]\!] \subseteq [\![\bar{l}]\!] \,\} \cup \mathcal{C}_{D,\overline{e'}}$$

式 (6)，(7) と補題 4 より $\mathcal{C}_{E,\overline{l_0}} \models \mathcal{C}_{E,\bar{l}}$ が成り立つ．また，$\mathcal{C}_{D,\bar{e}} = \mathcal{C}_{D,\overline{e'}}$ となるので，この場合も成り立つ．

- $S, M \vdash D, H \triangleright E[\, store(p_1, \mathtt{@}x, \langle\, q_2^{\overline{l_2}} \,\rangle)^{\overline{l_0}} \,] \longrightarrow D, H' \triangleright E[\, \langle\, q_2^{\overline{l_2}} \,\rangle \,]$．ただし，

$$H(p_1) = \langle C, IM \rangle^l$$
$$H' = H[\, p_1 \mapsto \langle C, IM[\, \mathtt{@}x \mapsto q_2 \,] \rangle^l \,]$$

$D, H \triangleright E[\, store(p_1, \mathtt{@}x, \langle\, q_2^{\overline{l_2}} \,\rangle)^{\overline{l_0}} \,]$ に対する制約 $\mathcal{C}$ は次のように分解できる．

$$\mathcal{C} = \mathcal{C}_M \cup \mathcal{C}_H \cup \mathcal{C}_{E,\overline{l_0}} \cup \mathcal{C}_s \cup \mathcal{C}_{D,\bar{e}}$$

store に関する制約（メソッド状況）から

$$\mathcal{C}_s \models \mathcal{O}[\![\overline{l_2}]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!]$$

また，$\mathcal{C}_H \models l \in [\![p_1]\!]$ であるので，l を store に関する制約（値）に適用することで

$$\mathcal{C}_H, \mathcal{C}_s \models [\![q_2]\!] \subseteq [\![\overline{l_2}]\!], [\![\overline{l_2}]\!] \subseteq \mathcal{F}[\![l, \mathtt{@}x]\!], \mathcal{F}[\![l, \mathtt{@}x]\!] \subseteq [\![\overline{l_0}]\!]$$

一方，$D, H' \triangleright E[\, \langle\, q_2^{\overline{l_2}} \,\rangle \,]$ に対する制約 $\mathcal{C}'$ は以下のように分解できる．

$$\mathcal{C}' = \mathcal{C}_M \cup \mathcal{C}_{H'} \cup \mathcal{C}_{E,\overline{l_2}} \cup \mathcal{C}_{D,\overline{e'}}$$

$\mathcal{C}_H, [\![q_2]\!] \subseteq \mathcal{F}[\![l, \mathtt{@}x]\!] \models \mathcal{C}_{H'}$ が成り立つので，$\mathcal{C}_H, \mathcal{C}_s \models \mathcal{C}_{H'}$．

また，$\mathcal{C}_H, \mathcal{C}_s \models [\![\overline{l_2}]\!] \subseteq [\![\overline{l_0}]\!]$，$\mathcal{O}[\![\overline{l_2}]\!] \subseteq \mathcal{O}[\![\overline{l_0}]\!]$ と補題（4）より，

$$\mathcal{C}_H, \mathcal{C}_s, \mathcal{C}_{E,\overline{l_0}} \models \mathcal{C}_{E,\overline{l_2}}$$

また，$\mathcal{C}_{D,\bar{e}} = \mathcal{C}_{D,\overline{e'}}$ となるので成り立つ．

□

Progress 補題の準備として，次の補題を示す．e の構造に関する帰納法で証明することができる．

補題 5　プログラム $D, H \triangleright E[\langle e \rangle]$ は，S，M，$\mathcal{L}$ のもとで well-formed とする．このとき，以下のいずれかが成り立つ．

(1)　e が評価済みポインタである．

(2)　ある D'，H'，$\bar{e}'$ に対して，$S, M \vdash D, H \triangleright E[\langle e \rangle] \longrightarrow D', H' \triangleright E[\bar{e}']$．

□

保存補題とともに，安全なプログラムの評価が実行時エラーにならないことを保証する Progress 補題は下の形になる．場合 2 の $\langle\, q^{\bar{l}} \,\rangle$ が実行が正常に終了しポインタを返す場合に，$F[\, throw(t, \langle\, q^{\bar{l}} \,\rangle) \,]$ はスコープをぬけた後に大域脱出（ブロックからの return や break）をした場合に，それぞれ対応する．本論文では，p_{nil} に関する値のフローを無視することにしたため，p_{nil} に対するメソッド呼び出しや yield が発生する場合があり，場合 3 の可能性がある．

補題 6（Progress）　$D, H \triangleright \bar{e}$ を S，M，$\mathcal{L}$ のもとで well-formed なプログラムとする．このとき，S，M，$\mathcal{L}$ のもとで，プログラム $D, H \triangleright \bar{e}$ に対する安全性解析制約が解を持つならば，以下のいずれかが成り立つ．

(1)　ある D'，H'，$\bar{e}'$ に対して，$S, M \vdash D, H \triangleright \bar{e} \longrightarrow D', H' \triangleright \bar{e}'$．

(2)　$\bar{e}$ が $\langle\, q^{\bar{l}} \,\rangle$ または $F[\, throw(t, \langle\, q^{\bar{l}} \,\rangle) \,]$ の形をしている．

(3)　$\bar{e}$ が $E[\, p_{nil}.\mathtt{f}(\langle\, q^{l} \,\rangle) \,]$ か $E[\, yield(p_{nil}, \langle\, q^{\bar{l}} \,\rangle) \,]$ の形をしている．

□

証明 補題 5 より，$\bar{e} \equiv E[\langle\, q^{\bar{l}} \,\rangle]$ のときを示せばよい．E の構造に関する場合分けで示す．主要な場合についてのみ証明を示す．$\mathcal{C}$ を $D, H \triangleright \bar{e}$ に対する制約とする．

- $E \equiv []$ のとき．2 が成り立つ．
- $E \equiv E'[\, [].\mathtt{f}(e_2) \,]$ のとき．$S, M \vdash D, H \triangleright E'[\, \langle\, q^{\bar{l}} \,\rangle.\mathtt{f}(e_2) \,] \longrightarrow D, H \triangleright E'[\, q^{\bar{l}}.\mathtt{f}(\langle\, e_2 \,\rangle) \,]$．
- $E \equiv E'[\, q_1^{\overline{l_1}}.\mathtt{f}([]) \,]$ のとき．
  - $q_1 \equiv p_{nil}$ のとき．3 が成り立つ．
  - $q_1 \equiv p_1$ のとき．Well-formedness と安全性制約から，ある C と l に対して，$H(p_1) = \langle C, \ldots \rangle^l$ かつ $\mathcal{L}(l) = C$．また，メソッド状況に関する制約の定義から $\mathcal{C} \models D \in \mathcal{O}[\![l_2]\!]$．$l$ と C をこのメソッド呼び出しに関する安全性制約に適用すると，$lookup_{S,M}(D, C, \mathtt{f}) \neq \bot$ が得られる．すなわち，ある s，x，e_3 に対して，$lookup_{S,M}(D, C, \mathtt{f}) = \lambda s.\lambda x.e_3$ となり，call の簡約を適用できる．
- $E \equiv E'[\, throw(t, []) \,]$ のとき．
  - $E' \equiv F$ のとき．2 が成り立つ．
  - $E' \equiv E''[\, catch(t', F) \,]$ のとき．簡約規則 throw または throw' を適用できる．

□

5. 関 連 研 究

安全性解析と同様の検査を行う Ruby の型推論が提案されている[1)]．フロー解析と同様に，式の値に関する制約を抽出，解消することにより，式の型を推論する．また，型推論の

結果から，未定義メソッドの呼び出しや引数型の不一致の検査を行う．多相型の取扱いには制限があり，プログラムに型注釈を付加することによって多相型を宣言する．型推論は制御フロー非依存であるため，メソッド定義の解析の精度が劣る．また，型システム・型推論アルゴリズムの健全性について議論されていない．

多相レコード型と ML の型推論アルゴリズムに基づく Ruby の型推論が提案されている[11]．この論文では，多相レコード型に基づく型システムを提案し，型推論の実装を示している．ML の型システムに基づく型システムであることから，多相再帰的な型を持つ組み込みクラスの定義などの解析に制限があり，また制御フローについても区別されていない．また，型システムの健全性に関する議論も不十分である．

JavaScript に対する型検査が提案されている[3]．抽象解釈に基づく型解析を行うもので，フルセットの JavaScript 言語を対象とする．制御フロー解析による手法よりも高い精度が実現できると考えられるが，解析の健全性が示されていない．JavaScript については，フルセットに対する操作的意味論が発表されている[4]．この論文では well-formedness の保存について証明が与えられており，プログラム解析の健全性まで含めた証明に対する障害は低くなりつつある．

6. まとめと今後の課題

Ruby プログラムの制御フロー解析を提案した．提案した制御フロー解析は，メソッド定義について制御フロー依存であり，メソッド定義式の評価に依存するプログラムの振舞いを正確に解析できる．メソッド定義式が評価済みかどうかのみに注目して制御フローを識別することにより，Ruby プログラムの解析で特に重要な，動的なメソッド定義の解析精度の向上を実現できた．また，提案した制御フロー解析アルゴリズムに基づく，安全性解析の健全性について証明を与えた．

今後の課題として，実験による評価や文脈依存な解析の導入があげられる．本論文で述べた解析は，メソッド呼び出しの文脈を区別しない．その結果，メソッド呼び出しにおいて，複数のメソッド状況が区別されずに取り扱われることになり，解析の精度が向上しない．また，文献 3) に示されているような，インスタンス変数についてのより正確な解析も課題である．

制御フロー解析は，安全性解析だけでなく，プログラムの最適化などにも広く利用できる結果を与えるものである．今後，Ruby プログラムの最適化なども含め解析結果の応用を進めてゆきたい．

謝辞　本研究の一部は，科学研究補助金（基盤（B）20300001，基盤（C）21500028）の補助を得て行った．

参 考 文 献

1) Furr, M., An, J.D., Foster, J.S. and Hicks, M.W.: Static type inference for Ruby, *Symposium on Applied Computing*, pp.1859–1866, ACM (2009).
2) Heintze, N.: Set-Based Analysis of ML Programs, *LISP and Functional Programming*, pp.306–317 (1994).
3) Jensen, S.H., Møller, A. and Thiemann, P.: Type Analysis for JavaScript, *16th International Static Analysis Symposium*, Lecture Notes in Computer Science, Vol.5673, pp.238–255, Springer (2009).
4) Maffeis, S., Mitchell, J.C. and Taly, A.: An Operational Semantics for JavaScript, *6th Asian Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, Vol.5356, pp.307–325, Springer (2008).
5) Nielson, F., Nielson, H.R. and Hankin, C.: *Principles of Program Analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999).
6) Palsberg, J. and Schwartzbach, M.I.: Safety Analysis versus Type Inference, *Inf. Comput.*, Vol.118, No.1, pp.128–141 (1995).
7) Shivers, O.: Control-Flow Analysis in Scheme, *Programming Language Design and Implementation*, pp.164–174 (1988).
8) Sperber, M., Dybvig, R.K., Flatt, M. and van Straaten, A.: Revised^6 Report on the Algorithmic Language Scheme (2007). http://www.r6rs.org
9) Whaley, J. and Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, *Programming Language Design and Implementation*, pp.131–144, ACM (2004).
10) まつもとゆきひろ：オブジェクト指向スクリプト言語 Ruby (2009). http://www.ruby-lang.org
11) 松本宗太郎，南出靖彦：多相レコード型に基づく Ruby プログラムの型推論，情報処理学会論文誌　プログラミング，Vol.49, No.3, pp.39–54 (2008).
12) 松本宗太郎，南出靖彦：Ruby のコア言語の操作的意味論，日本ソフトウェア科学会第 26 回大会予稿集 (2009).

(平成 21 年 9 月 29 日受付)
(平成 22 年 1 月 5 日採録)

松本宗太郎（学生会員）

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．