Progress report: Ruby 3における静的型解析の実現に向けて

遠藤 侑介[1], 松本 宗太郎[2], 上野 雄大[3], 住井 英二郎[4], 松本 行弘[5]

[1] クックパッド株式会社
yusuke-endoh@cookpad.com

[2] Sider 株式会社
matsumoto@soutaro.com

[3] 東北大学電気通信研究所
katsu@riec.tohoku.ac.jp

[4] 東北大学大学院情報科学研究科
sumii@ecei.tohoku.ac.jp

[5] 一般財団法人 Ruby アソシエーション
matz@ruby.or.jp

概 要 Ruby は，動的型付けやメタプログラミングを特徴とするプログラミング言語である．一方，Ruby の使用範囲が広がるにつれて，誤りの自動検出など，プログラムの品質を高めるための支援も求められている．この状況に対し，Ruby の設計者である第 5 著者は，近い将来，何らかの静的解析を導入するという目標を掲げた．この方針を受けて筆者らは詳細な検討を進め，Ruby の特徴を損なうことなく Ruby プログラムを静的解析するための要件を整理するとともに，適用可能なアプローチについて議論し，抽象解釈に基づく型プロファイラや，漸進的型付けの考え方を取り入れた型検査器を並行して開発している．本論文では，Ruby の簡潔性を損なわないなど静的解析システムに求められる要件について述べたのち，筆者らが開発している 2 つのシステムの現状を報告し，Ruby の静的解析の今後の方向性について議論する．

1 はじめに

Ruby は動的型付けとメタプログラミングを特徴とするプログラミング言語である．これらの特徴は，言語やライブラリの動的な拡張を含む高い自由度をプログラマに与え，短く簡潔なプログラムの記述を可能とする．Ruby on Rails などに代表される Ruby 特有のプログラミングフレームワークは，Ruby のこれらの特徴の上に成立している．Ruby の簡潔性とその上に成立したソフトウェア資産の積み重ねにより，Ruby は産業的ソフトウェア生産の第一言語として選択されうる実用性を認められており，特にラピッドプロトタイピングに適していると考えられている．また，プロトタイピングの段階を脱した後も，そのまま Ruby がプロダクトラインのソフトウェア開発に用いられることも少なくない．

このような Ruby の普及と発展とは裏腹に，Ruby で書かれたソフトウェアの規模が大きくなるにつれて，Ruby の特徴たる動的機能およびメタプログラミングがソフトウェアの生産性を阻害する要因として大きなウェイトを占めるに至っている．例えば，数十万行を超える実用規模の Ruby プログラムにおいて，デッドコードと思われるコードを削除しようとしたとき，それが確実にデッドコードであることを確認することは人力では極めて困難である．また，メタプログラミングによって定義されたクラスやメソッドがあるとき，あるクラスに定義されたメソッドの一覧すら，プログ

ラムを実行せずに得ることは難しい．Ruby プログラムの品質を高めるため，Ruby の生産性を阻害することなくこれらの問題を解決するための支援が求められている．

この状況を踏まえ，Ruby の設計者である第 5 著者は，次期メジャーリリースである Ruby 3 に向けて Ruby 開発チームが達成すべき目標の一つとして，Ruby への静的型付けの導入の可能性について，いくつかの講演で言及した（例えば [9, 10] など）．これらの講演でいう「静的型付け」とは，Ruby プログラムに対する何らかの静的解析の枠組みを指す．一連の講演等で掲げられた方針の要点は以下のとおりである．

- 静的解析のために，Ruby の最大の特徴である動的機能，メタプログラミング，および簡潔性を失ってはならない．Ruby のこれまでの発展はこれらの特徴に裏付けられたものであり，これからの発展もこれらの特徴が基礎となるはずである．したがって，静的解析のために Ruby 全体の機能を制限することは受け入れられない．
- 静的解析のためだけに Ruby 言語を拡張してはならない．また，特別な埋め込み言語を同梱することも賛成できない．静的解析の有無を選択する余地はプログラマに残されるべきであり，また Ruby の今後の発展のためにも，Ruby は将来のプログラム解析技術の発展から独立でありたい．
- 型システムが健全であることよりも，上記の要求が優先される．Ruby の「静的型」はあくまで開発者の支援のために導入されるべきであり，プログラムのある種の性質を保証するものである必要はない．

この方針を受けて，筆者らは静的解析に対する機能要求や具体的な解析方式について，より詳細な検討を進めた．その結果，抽象解釈に基づく型解析，および漸進的型付け [18] の考え方を取り入れた静的型検査の 2 つの方針が，近い将来での実現に向けて適当であろうという感触を得た．より詳細な検討を進めるため，サブプロジェクトとして以下の 2 つの独立なシステムを並行して開発している．

- 抽象解釈に基づく型プロファイラ．このシステムは，オブジェクトが属するクラスの識別子を抽象値とする抽象評価器を備える．素の Ruby プログラムから読解に役立つ情報を引き出すのがこのシステムの目的である．
- プログラマが与えたシグネチャと実装の矛盾を検出する型検査器．このシステムは，Ruby ソースコードとは別のファイルに書かれたシグネチャを起点としてプログラムの型付けを試みる．Ruby で型を意識したプログラミングを行うことを提案および支援することが，このシステムの目的である．

本論文では，以下の構成で本プロジェクトの現状を報告する．まず 2 節では，Ruby 言語の性質やプログラミングパターンを概観することで静的解析器に求められる要件を整理する．次に 3 節および 4 節では，上述した 2 つのシステムの概要と開発状況を報告する．5 節では，本プロジェクト以外での Ruby の静的解析機能の実現に向けた動向を紹介する．最後に 6 節では本論文をまとめ，本プロジェクトの今後について述べる．

2　静的解析を考える上での Ruby の特徴

Ruby はクラスベースのオブジェクト指向言語である．整数などの基本的な値も含め，あらゆるデータ構造はオブジェクトであり，任意のオブジェクトはある一つのクラスの直接のインスタンスである．例えば，整数および浮動小数点数はそれぞれ `Integer` および `Float` クラスのインスタンスである．各クラスは高々1 つの他のクラスを継承する．クラスおよび継承の概念は，一般的なオブジェクト指向言語と同様である．

あらゆる構造をオブジェクトで表現する方針はクラスにも適用される．Ruby ではクラスは Class クラスのインスタンスである．クラス A を定義する構文 class A; ⋯; end は静的な宣言ではなく，新たなクラスオブジェクトをヒープに割り当て，A をそのクラスオブジェクトに束縛する実行文である．同様にメソッド定義構文 def m; ⋯; end も，文脈で指示されたクラスオブジェクトに対してメソッド m を破壊的に追加する実行文である．これらの定義構文に相当する機能は，後の例に示すように，メソッドとしても提供されている．クラスやメソッドの定義が実行時に行われるため，メソッド呼び出し時のメソッド検索も必然的に実行時に行われる．

あらゆる操作対象がオブジェクトであるのに対し，あらゆる操作はメソッドである．多彩なメソッドを直感的かつ簡潔に記述できるように，様々なメソッド呼び出し構文が用意されている．例えば，式 1 + 2 はレシーバオブジェクト 1 の + メソッドを引数 2 をともなって呼び出すことを表す．一般的なメソッド呼び出し構文においても，構文が曖昧でなければ引数列（空でも良い）を囲う括弧は省略でき，また，レシーバが self ならばレシーバの指定も省略できる．結果として，ただメソッド名のみを書いた式 foo は，そのスコープで同名のローカル変数が定義されていなければ，self をレシーバとする foo メソッドの呼び出しである．

オブジェクトとメソッドによる統一的な抽象と，多くの省略を許すメソッド呼び出し構文が，見た目が統一された簡潔な記述を許す．高い記述性を追求するため，プログラムの堅牢性を捨てている側面もある．例えば，ローカル変数名の書き間違い（typo）でさえ発見は容易ではない．以下に例を示す（各行頭には行番号を付している）．

```
 1: class A
 2:   def foo
 3:     bar = 1        # ローカル変数 bar を定義
 4:     baz            # ここで bar を baz と書き間違えている
 5:   end
 6: end
 7: class B < A
 8:   def baz
 9:     2
10:   end
11: end
12: B.new.foo          # 結果は 2 である
13: A.new.foo          # 未定義メソッド例外（NameError）が発生する
```

4 行目の書き間違いは，メソッド foo 内にローカル変数 baz が定義されていないため，self をレシーバとするメソッド baz の呼び出しと構文解析される．12 行目でのクラス B のインスタンスに対する foo メソッドの呼び出しでは，B の定義よりレシーバは baz メソッドを持つため，4 行目の baz メソッドの呼び出しは成功する．一方，13 行目で A のインスタンスに対して foo を呼ぶ場合は，baz の検索に失敗し，実行時例外 NameError が発生し，プログラムの実行が中断される．もし 13 行目が存在しなければ，typo を含むプログラムでさえ正常に終了する．以上の状況から分かるように，たとえプログラムを実行したとしても typo が見つかるとは限らず，また typo を typo と断定することも容易でない．

記述の簡潔さが重視されることは，ライブラリやユーザープログラムの設計にも以下の 2 つの点で現れる．一つは，似たような形のコードを繰り返し書く手間を避けるためにメタプログラミングを多用することである．Ruby では，C 言語でマクロを使うのと同程度の気軽さでメタプログラミングが用いられる．例えば，以下は Ruby で書かれた CGI ライブラリ cgi/core.rb（Ruby 2.6.0 に標準添付）からの抜粋である（読みやすさのためにやや改変している）．

```
[ "CONTENT_LENGTH", "SERVER_PORT" ].each {|env|
  define_method(env.downcase) {
    (val = env_table[env]) && Integer(val)
  }
}
```

このコードは，環境変数 CONTENT_LENGTH と SERVER_PORT からそれぞれ整数を読み出す 2 つのメソッド content_length および server_port を定義する．文字列の配列に対するループの中でメソッドを定義する define_method メソッドを使うことで，環境変数名を小文字にしただけのメソッド名や，共通するメソッド本体を繰り返し書くことを避けている．同様のことが一般のアプリケーションコードでも平然と行われる．

もう 1 点は，メソッド検索が実行時に行われることを活用し，共通の親クラスを持たない複数のクラスが共通の性質を持つことを期待することである．例えば以下の 2 つのコードを考える．

```
def say_hello_to(x)
  x << "Hello!"
end
```

```
def lshift_with_one(x)
  x << 1 | 1
end
```

左のコードは，x がファイルハンドル（File クラスのインスタンス）ならばファイルへの書き出し，文字列（String）ならば末尾への追記，配列（Array）ならば要素の追加を行う．なぜなら，これらのクラスではメソッド << がそれぞれそのように定義されているからである．もしプログラマがこれらの定義を意識した多相的なコードとして左のコードを書いたならば，プログラマはメソッド << に「レシーバが指す場所に文字列を書き出す」という共通の機能を暗に想定し，x にはそのような << を持つ任意のオブジェクトが来ることを期待している．一方，右のコードでは，<< は左シフト演算であることが想定されており，同じ << を用いてはいるが左のコードとは想定される << の働きが異なる．Java などの静的型付きオブジェクト指向言語であれば，このような多相性は抽象クラスの継承やインターフェースの実装を通じて表現される．左右のコードにおける << への想定の違いも実装するインターフェースの違いとして現れるはずである．一方 Ruby では，共通の振る舞いに共通のメソッド名 << を割り当てるだけで同様の多相性が得られる．<< に関する想定の違いは文脈で暗に区別される．この性質はライブラリの様々な箇所で巧みに利用される．例えば，to_str メソッドを持つオブジェクトは文字列に暗黙に変換される，local_to_utc と utc_to_local メソッドを持つオブジェクトはタイムゾーンオブジェクトとみなされるなど，広く利用されている．

以上のように堅牢性より記述性を優先した，メタプログラミングや動的メソッド検索による高い記述力は Ruby の大きな特徴である．その一方で，前述した変数の typo の例にも見られる通り，この特徴はプログラムの可読性を下げ，ソフトウェアの品質の改善を妨げる要因となる．例えば，不要なメソッドを削除するなどの比較的軽微なリファクタリングですら，プログラムの意味を変えないことを確認するのは困難である．Ruby をソフトウェア開発に用いる現場では，このような悩みを解決するための努力が ad hoc に試みられてきた．例えば，あるコードがデッドコードであることを確認するため，大規模 Web サービスを実行する Ruby インタプリタを独自に改造して，そのコードの実行状況を記録し，ある一定期間実行されていないことを調べる，などの工夫がなされてきた [4].

本プロジェクトの目的は，Ruby の記述性を変えることなく，上述のような苦労を軽減するための静的解析機能を提供し，その機能を用いた新たなプログラミング体験を可能にすることである．したがって，静的解析しやすいように Ruby の言語仕様を改変することや，コード中に注釈を数多く書かせたり，特別な埋め込み言語を強要したりして Ruby の記述性を低下させることは避ける．また，静的解析の健全性や完全性よりも，Ruby の記述性や従来との互換性・実用性を優先し，前者については補助的な検討にとどめる．

以上を踏まえ，本プロジェクトでは，以下の 2 つの方式について検討を進めている．

- Ruby プログラムを抽象解釈する方式．Ruby プログラムの正確な意味は実行して初めて得られるので，プログラムの誤りを探す最も直接的な方法は，プログラムを実行してみることである．しかし，プログラムの実行には種々の設定が必要となるだけでなく，入力の可能性は無数（一般には無限）に存在し，実行トレースの量も膨大となる．そこで，抽象解釈の考え方を応用し，少ないコストで適切に抽象化された実行トレースを得ることができれば，プログラムの読解や誤りの発見においては，本当にプログラムを実行するよりも有益な情報が得られる可能性がある．
- Ruby プログラムに静的な型を与える方式．一般に推奨されるようなソフトウェアのモジュール化を行なっているならば，たとえクラスやメソッドの定義がメタプログラミングなどを通じて動的に行われるとしても，定義が完了したクラスはある種の静的なシグネチャを持つはずである．このシグネチャを記述する型言語を Ruby 自体とは独立に導入し，プログラマにシグネチャを書かせ，シグネチャと実装が矛盾しないことを型検査することができれば，従来の Ruby の記述性と，静的型も意識したプログラミングを両立できる可能性がある．

本プロジェクトでは，これらの 2 件のサブプロジェクトを並行して推進し，実装を進めつつ詳細な検討を行なっている．以下に続く 2 つの節では，各サブプロジェクトの取り組みと途中経過を報告する．

3　抽象解釈に基づく型プロファイラ

本節では，抽象解釈で Ruby プログラムを解析する「型プロファイラ」の開発について報告する．型プロファイラは，素の Ruby プログラムを入力として受け取り，エントリポイントから到達する可能性のある制御フローをトレースし，トレースの過程で発見したメソッド呼び出し，ブロック呼び出し，およびインスタンス変数の読み書きに関する情報を出力する．（ブロックとはコードをオブジェクト化する構文要素である．詳細は Ruby のマニュアル [1] を参照されたい．）型プロファイラの使用によって，プログラマは以下の恩恵を受けることが期待される．

- 識別子未定義エラー（`NameError`）や型エラー（`TypeError`, `ArgumentError`）の検出．型プロファイラはすべての実行時エラーを見つけることはできず，逆に誤検出を行うこともあるが，人間が目視で実行時エラーを探すのに比べれば網羅的な検査が可能である．特に typo の検出に高い実用性を発揮することが期待される．
- プログラムを構成するクラスやメソッドのシグネチャの推定．型プロファイラの出力を読むことで，プログラマは自身が想定していないような可能性（例えば，あるメソッドの引数は `nil` でないと想定していたのに，`nil` が来る可能性）に気がつくことが期待される．さらに，この情報は 4 節で述べるような型検査器のためのシグネチャファイルの雛形を得ることにも応用できると考えられる．

3.1　型プロファイラが報告する言明

プログラム中に現れるすべてのメソッド定義およびブロックについて，それぞれ一意な識別子が与えられているとする．これらの識別子は実装上はメソッドやブロック本体のコードアドレスである．b および m をそれぞれブロックおよびメソッドの識別子の集合を動くメタ変数とする．Ruby のメソッドにはインスタンスメソッドとクラスメソッドの 2 種類がある．クラス K のインスタンスメソッド m の識別子を K`#`m と書く．クラス K のクラスメソッドは，クラスオブジェクト K を唯一のインスタンスとするクラス（K の特異クラス）のインスタンスメソッドのことである．K の特異クラスの識別子を class(K) と書く．class(K) 自身もメタ変数 K が動く集合の元であることに注意されたい．したがって，クラス K のクラスメソッド m の識別子は class(K)`#`m である．

型プロファイラが報告する言明に現れる型 τ は以下のいずれかである．

$$\tau \quad ::= \quad K \mid b \mid \mathit{Unknown}$$

K はクラス K の直接のインスタンス (K のインスタンスのうち，K の子クラスのインスタンスでないもの）の型である．b はブロック b をコードとするオブジェクトの型である．*Unknown* は，静的解析不能な組み込みメソッド（`eval` など）の呼び出しやエラーが発生したことを表す型である．型をこのように定義した意図は抽象解釈の方式と密接に関連する．各型のより詳細な説明は 3.2 節で抽象解釈の方式とともに述べる．

型プロファイラは，制御フローを抽象的にトレースする過程で，以下の事象を発見するたびに以下の言明を出力する．

- クラス K のインスタンスメソッド m が型 $\tau_1, \ldots, \tau_n$ の n 個の引数をともなって呼び出され，その結果 τ 型の値が返されるたびに，言明

  $$K\texttt{\#}m :: (\tau_1, \ldots, \tau_n) \to \tau$$

  を出力する．メソッド呼び出しがブロック b をともなう場合，引数列の最後に `&`b が付加される．メソッドだけでなくブロックの呼び出しについても，そのブロックが呼び出されて値を返すたびに同様の言明を出力する．
- クラス K の直接のインスタンスが持つインスタンス変数 `@`i（以下 K`#@`i と書く）に型 τ の値が書き込まれるたびに，言明

  $$K\texttt{\#@}i :: \tau$$

  を出力する．グローバル変数についても同様である．

K`#`m に関する言明は K の子クラスのインスタンスのメソッド m が呼び出された時も生成される可能性があるのに対し，K`#@`i に関する言明は K の直接のインスタンスのみを対象とすることに注意されたい．この違いは，Ruby ではメソッドはクラスに属しており，クラスの継承関係を通じて探索されるのに対し，インスタンス変数はクラスではなく各オブジェクトに属していることに由来する．

型プロファイラは，Ruby の組み込みメソッドそれぞれについて引数列と返り値の型を公理として内蔵し，ユーザー定義のメソッドとは区別して取り扱う．組み込みメソッドはオーバーロードされていることがあるため，1 つのメソッドに対して 1 つ以上の型が公理として与えられる．組み込みメソッドが呼び出されたとき，型プロファイラは言明を出力する代わりに，メソッドの引数列がその組み込みメソッドの型のいずれかと一致することを検査する．いずれの型とも一致しなかった場合，型エラーを報告する．

3.2 型プロファイラの抽象解釈方式

型プロファイラは与えられたプログラムを，τ を抽象値として抽象解釈する．言明を出力するときは抽象値がそのまま型として出力される．抽象値としての τ の意味は以下の通りである．

- K はクラス K の直接のインスタンスを表す．前述の通り K は，あるクラス K' の特異クラス $\mathrm{class}(K')$ である場合がある．$\mathrm{class}(K')$ の唯一のインスタンスはクラスオブジェクト K' である．すなわち，抽象値 $\mathrm{class}(K')$ はクラス K' そのものを表す．抽象値としての $\mathrm{class}(K')$ は，主にメソッド定義文を実行するときのメソッド定義先の指定に用いられる．
- b はブロック構文 b から生成されたブロックオブジェクトを表す．ブロックには自由変数が含まれない，すなわちクロージャは作られないと仮定する．自由変数を含むブロックの扱いは今後の課題として 3.3 節で述べる．
- *Unknown* は，静的な評価が不可能な式に解析器が到達したとき，解析を継続するため，仮に置く値である．型プロファイラは可能な限り多くの情報を Ruby プログラムから取り出すことを目的とする．そのためには，エラーを発見した後に続くコードも可能な限り解析を続けることが望ましい．型プロファイラは以下の場合に *Unknown* を導入し，解析を続ける．

– 未定義のメソッドを呼び出したとき，その返り値を *Unknown* とし，評価を続行する．
– 組み込みメソッドを呼び出したとき，引数列がそのメソッドに関するどの公理とも合致しなければ，その返り値を *Unknown* とし，評価を続行する．
– *Unknown* を返すと公理で指定されたメソッドを呼び出したとき，公理のとおり *Unknown* が返される．例えば，`eval` メソッドは，`String` を受け取り *Unknown* を返す．
– *Unknown* をレシーバとするメソッド呼び出しは直ちに *Unknown* を返す．

プログラムの実行の抽象的な 1 ステップは，命令を 1 つ実行するごとに抽象評価器の状態を次の状態に遷移することで行う．型プロファイラでは，抽象評価器の状態の大きさが有限となるように状態を抽象する．抽象解釈に関わる Ruby インタプリタの状態は，環境（ローカル変数，グローバル変数，演算スタック，現在のクラス），ヒープ（クラスオブジェクトを含む，各オブジェクトのインスタンス変数），およびコールスタックからなる．これら各構成要素に対して行った抽象は以下の通りである．

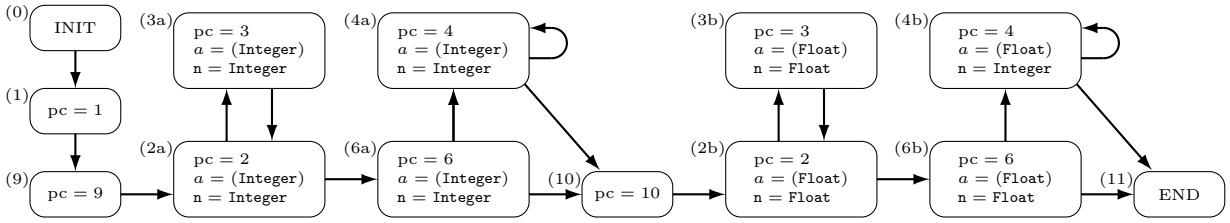
- クラスの数は有限とする．したがって抽象値の数も有限である．ローカル変数やグローバル変数の名前は有限とおりとする。したがって、抽象化された環境も有限とおりとなる．また，メソッドの数も有限とする（`def` 文以外のメソッド定義には対応していない）．
- 個々のオブジェクトが持つインスタンス変数の内容を省略する．代わりに，クラス K の直接のインスタンスのひとつに対するインスタンス変数 `@`i への書き込みは，クラス K の直接のインスタンス全てに対する `@`i の読み込みから観測されるとみなす．個々のオブジェクトの一意性も追跡しない（例えば参照を比較する組み込みメソッドは常に「`Bool`」を返すとする）ため，無限の大きさを持つヒープは不要となる．
- コールスタックを省略する．メソッドからのリターンは，引数（の抽象値）以外の呼び出し文脈を無視して（context insensitive に）行う．すなわち，ある引数（の抽象値）をともなって呼び出されたメソッドからのリターンは，同じメソッドを同じ引数で呼び出すすべての呼び出し命令の次の命令にリターンするとみなす．呼び出しの文脈として引数を考慮するのは，Ruby の動的な性質より，引数によってメソッドの抽象的な振る舞いが大きく変わる可能性があるためである．

プログラムの抽象解釈は，初期状態から到達するすべての状態をトレースすることで行う．到達する状態の集合は以下の抽象評価規則で帰納的に定義される．

1. 評価器の初期状態には無条件に到達する．
2. 条件分岐命令を実行する状態に到達するとき，いずれの分岐先を実行する状態にも到達する．
3. ある組み込みメソッド m について，m は $\tau_1, \ldots, \tau_n$ を引数として受け取ると τ を返すという公理が与えられているとする．メソッド m を引数 $\tau_1, \ldots, \tau_n$ をともなって呼び出す状態に到達するとき，返り値 τ を受け取ったとして次の命令を実行する状態にも到達する．
4. あるユーザー定義メソッド K`#`m をある引数列をともなって呼び出す状態を S，同じ引数をともなって呼び出された同じメソッドが返り値 τ をともなってリターンする状態を L とする．S および L の両方に到達するとき，返り値 τ を受け取ったとして S の次の命令を実行する状態にも到達する．ブロックについても同様である．
5. インスタンス変数 K`#@`i から値を読み込む状態を R，同じインスタンス変数に値 τ を書き込む状態を W とする．R および W の両方に到達するとき，K`#@`i から τ を読み込んだとして R の次の命令を実行する状態にも到達する．グローバル変数についても同様に扱う．
6. インスタンス変数 K`#@`i から値を読み込む状態に到達するとき，K`#@`i から `NilClass` を読み込んだとして次の命令を実行する状態にも到達する．この規則は，値が書き込まれていないインスタンス変数の読み込みは `nil` を返す，という Ruby の振る舞いに対応する．

```
 1:  def f(n)
 2:    if n > 0 then
 3:      n = f(n - 1)
 4:      return n + 1
 5:    else
 6:      return 1
 7:    end
 8:  end
 9:  f(N)   # N は外部から与えられる整数（Integer クラスのインスタンス）
10:  f(R)   # R は外部から与えられる浮動小数点数（Float クラスのインスタンス）
```

(a) プログラムの例



(b) 抽象解釈で到達する状態の集合の例

図 1. 抽象解釈の例: (a) プログラムの例 (b) 抽象解釈で到達する状態の集合の例

7. これら以外の状態に到達するとき，実行する命令に関する Ruby インタプリタの評価規則に準じて作られる次の状態にも到達する．

型プロファイラはこれらの条件を満たす最小の有限集合を不動点反復の一種により求める．抽象状態の数は有限であるから，この帰納的条件を満たす最小の有限集合は必ず存在する．したがって，どのような入力に対しても型プロファイラは必ず終了する．

例として，図 1(a) のプログラムの抽象解釈を考える．このプログラムを抽象解釈した結果得られる実行トレース全体を図 1(b) に示す．図では，状態としてプログラムカウンタ pc，引数列 a，および変数 `n` の内容を表示している．プログラムカウンタの値は行番号である．状態番号はプログラムカウンタの値に準じてつけている．矢印は実行トレースの帰納的構成の順序を表す．抽象解釈は初期状態 (0) から始まる．状態 (2a) に至るまでは Ruby インタプリタに準じた評価が行われる．(2a) は分岐命令のため，`then` 節を実行する状態 (3a) および `else` 節を実行する状態 (6a) の両方に到達する．(3a) で `f` の再帰呼び出しを行った後の状態は，コールスタックがないため (2a) に等しい．リターンする状態 (6a) に到達したとき，`f` を引数 `Integer` をともなって呼び出す状態 (9) および (3a) にすでに到達しているため，9 行目および 3 行目のリターン先である 10 行目および 4 行目を実行する状態 (10) および (4a) に到達する．(4a) からも同様に，(4a) 自身と (10) に到達する．(10) でのメソッド `f` の呼び出しは，(9) とは異なり `Float` を引数とするため，状態 (2a) とは異なる状態 (2b) に到達する．状態 (2b) からのトレースは上述した (2a) からのトレースと同様である．解析結果として，以下の 2 種類の言明が出力される．

```
Object#f  ::  (Integer) → Integer
Object#f  ::  (Float) → Integer
```

3.3 評価と今後の課題

本プロジェクトでは上述の型プロファイラの試験的な実装を進めている．実装言語は Ruby である．試験実装では，Ruby の構文木の代わりに Ruby インタプリタのバイトコード [23] を抽象評価する．この実装方式の利点は，整理された命令セットを持つバイトコードインタプリタをシミュレートするだけで実装が完了することである．一方，以下の 2 点に注意する必要があった．第 1 に，バイトコードコンパイラの最適化によって取り除かれたコードには到達できないこと，第 2 に，ソースコード上に現れないバイトコードに特有の値を具体的に管理する必要があることである．例えば，メソッド呼び出し命令 `send` はメソッド名をシンボル値のオペランドとして取るため，個々の具体的なシンボル値を抽象値に加えて対処した．

利便性の向上のために，言明の出力では以下の工夫を行っている．第 1 に，言明は全状態のトレースが終わってから整形し，重複を省いて出力する．第 2 に，ブロックの言明は，ブロックのコードアドレスを直接出力するのではなく，そのブロックをともなうメソッド呼び出しに関する言明に展開して出力する．最後に，型エラーの報告では，型エラーの原因を特定しやすいように，エラーの発生箇所（ソースファイル名と行番号）に加え，エラーを発生させる状態へのパスを擬似的なバックトレースとして表示する．

型プロファイラを有効に適用できるのは，以下の 2 条件を共に満たすときである．第 1 に，実行可能なプログラム全体が与えられていなければならない．第 2 に，プログラムの各メソッドに直接あるいは間接的に到達できるトップレベルコードが存在しなければならない．これらの条件は，例えばテストコードをトップレベルコードとして使うことである程度満足されるはずである．静的解析の適用が困難であった Ruby ではテストフレームワークが充実しており，テストコードを書くことが広く普及しているため，テストコードの存在は多くの場合期待できる．しかも，メソッドへは抽象評価で到達できればよく，通常の意味でのコードカバレッジが高い必要は必ずしもない．以上より，実践的な多くの場合において本手法は適用可能であると期待される．

予備実験として，型プロファイラ自身に対して型プロファイラを適用した．型プロファイラのソースコードは 1911 行，クラスは 25 個，メソッドは合計で 190 個，インスタンス変数は 56 個である．これに対して型プロファイラを適用したところ，解析で到達した状態数は 3911 個，解析にかかった時間は 0.59 秒であった．出力された言明の数は，メソッドについての言明が 68 個，インスタンス変数についての言明が 58 個であった．少なくとも 1 つの言明が得られたメソッドは 48 個であった．いずれかの言明に現れたクラスの数は 17 個であった．

この結果から分かる通り，予備実験では，インスタンス変数については実際の数よりも言明の数が多く，その一方でメソッドについては実際の数よりも言明の数が少なかった．インスタンス変数については，同じインスタンス変数が別のクラス（典型的には子クラス）を経由して利用されていることが主な原因である．メソッドについては，その主要な要因は以下の 2 点と推察される．第 1 に，未実装の組み込みメソッドが多かったことである．このため，未実装の組み込みメソッドの呼び出しによって返り値が *Unknown* となり，その返り値をレシーバとする後続のメソッド呼び出しが省略された．また，`Array#each` などブロックを受け取る組み込みメソッドが未実装のため，ブロック内のコードが抽象解釈されなかった．第 2 に，未使用のメソッドが存在することである．典型的な例は，デバッグプリントのための `inspect` メソッドである．このメソッドはデバッグプリントを行ったときのみ呼び出される．従って，プログラム中にデバッグ出力のコードが残されていない限り，このメソッドに到達する実行パスは存在せず，型プロファイラではこのメソッドに到達できない．第 1 の原因は，今後実装の完成度が高まるごとに解決するはずである．第 2 の問題の深刻さの程度は，型プロファイラの適用範囲を広げるごとに明らかになると期待する．

型プロファイラの出力によるコード読解実験では以下の事例に遭遇した．型プロファイラからの出力には,「あるメソッドが `NilClass` を受け取る」という言明が含まれていたが，これはプログラマの想定と異なるものであった．この情報を元にソースコードを調査した結果，別の箇所で記述した

nil が，当初想定していなかったパスでそのメソッドに間接的に渡されていることがわかった．この事例から分かる通り，型プロファイラの実装は不完全ながらも，当初の目論見どおりソースコードの品質を高める機能をすでに果たしており，この方式が有望であるとの印象を得た．

本方式が Ruby の全機能をどの程度合理的な範囲で網羅できるか，および本方式が実用上どの程度の規模までスケールするかは，今後の開発と実験で順次明らかになると期待する．現時点で未対応の機能を含む今後の課題と展望は以下の通りである．

- 値を抽象するレベルの調整．特に，Array クラスや Hash クラスなど，頻繁かつ多義的に使われる組み込みのコンテナ型のサポートは実用上必須である．これらのサポートには，generic（パラメタ多相的）なクラスを抽象値に加えるなどの拡張が必要と考えられる．しかし，Ruby では Array や Hash をタプルやレコードのように使うことがあるため，要素型をパラメタ化するだけでは不十分である．また，再帰的にネストした Array や Hash を作るコードでは，それらの抽象値が有限にならないおそれがある．組型，レコード型，再帰型などの導入も含めた検討が必要である．
- 自由変数を含むブロックへの対応．ブロックは一般に自由変数を含み，クロージャを作る．Ruby のクロージャは環境を通じて自己参照をする可能性があるため，暗黙に再帰的なデータ構造が作られ，上述した再帰したコンテナ型と同様の問題が生じる．また，ブロック内での自由変数の書き換えも考慮しなければならない．試験実装では，クロージャが捕捉した環境に含まれる変数の抽象値は更新されないという前提をおいて，限定的にクロージャに対応している．
- 未初期化のインスタンス変数への対処．値が書き込まれていないインスタンス変数を読み込む可能性があるため，3.2 節に述べた抽象評価規則 6 は全てのインスタンス変数の読み込みに対して NilClass を生成する．しかし，この生成の影響でコード読解に有益でない言明が多く出力されてしまう．この問題への ad hoc な対処として，インスタンス変数は必ず初期化される（読み込みは必ず書き込みより後に起こる）と仮定し，試験実装では規則 6 を外すこととした．より正確な解析のためには，インスタンス変数が初期化されることを追跡するなどの対応が必要と思われる．
- 例外のサポート．例外発生の可能性の検出は型プロファイラによる解析が効果的な分野の一つと考えられる．例外に関する抽象解釈方式や，適切な言明の粒度は今後の課題である．
- 動的なクラス生成，モジュールの mix-in，一般の特異クラスなど，クラスの動的な構成への対応．Ruby ではクラスを動的に作るため，無限にクラスを生成し続けるプログラムを書くことができる．また，クラスに限らずすべてのオブジェクトは特異クラスを持つことができる．これらへの対応は未整理である．
- 出力される言明の簡単化．試験実装では言明の重複を取り除いて出力したが，予備実験で出力された言明は依然として冗長に見える部分があった．予備実験で遭遇した典型例を以下に示す．

  ```
  GlobalEnv#add_method :: (Type::Class, Symbol, CustomMethodDef) -> GlobalEnv
  GlobalEnv#add_method :: (Type::Class, Symbol, TypedMethodDef) -> GlobalEnv
  ```

  これらの言明は第 3 引数のみ異なる．しかし，CustomMethodDef と TypedMethodDef は共に MethodDef の子クラスである．プログラマがこの継承関係を知っているならば，第 3 引数を MethodDef とする 1 つの言明にこれら 2 つの言明を集約したほうが，プログラマにとって理解しやすい可能性が高い．クラスの継承関係を利用して複数の言明を包含する汎用的な言明を作るなど，出力される言明の簡単化を検討したい．ただし，簡単化しすぎないように注意する必要がある．

- 分岐先の限定．抽象値から分岐先が一意に決まる場合がいくつかある．例えば，`if` 文の条件式が `NilClass` に評価されたならば `then` 文は評価されない．また，ダウンキャスト時の動的型チェックコードの分岐も，抽象値に応じて分岐先の可能性を狭めることができると期待できる．
- メタプログラミングへの対応．試験実装における想定は，メタプログラミングを行うメソッド（`attr_reader` など）の抽象解釈を与えられるように，組み込みメソッドの意味をユーザーがプラグインすることである．この是非も含めて詳細は今後の課題である．

4 静的型検査ツール Steep

Steep は，プログラマによって与えられたシグネチャに対する Ruby プログラムの整合性を検査するツールである．漸進的型付け [18] とローカル型推論 [15] の考え方を取り入れた型推論を行い，シグネチャと実装に矛盾がないことを検査する．Steep の一般的な目標は，Ruby の意味論に対しておおよそ健全と期待される型システムを与え，型検査をしながら Ruby プログラムを書くことを推進する開発環境を提供することである．Steep の基本的な設計方針は以下の通りである．

- 定義完了後のクラスやモジュールのシグネチャをプログラマ自身に記述させる．このシグネチャを中心として，ライブラリの実装と使用の両側の整合性を検査する．これは，たとえクラスの定義中にメタプログラミングが使用されたとしても，定義が完了した後ならばクラスは静的なシグネチャを持つはずである，という観察に基づく．ただし，メタプログラミングされたクラスやメソッドがシグネチャと矛盾しないことはプログラマの責任に帰する．
- メソッドの仮引数や返り値の型およびクラスやメソッドの多相性はプログラムから推論しない．多相性はプログラマがシグネチャに明示した場合に限り導入される．
- 式の型を一意に推論できない場合，その式に動的型を与える．プログラマは必要に応じて Steep が解釈する特別なコメントをプログラムに挿入することで型推論を補助する．動的型がプログラマによる指定なしに導入された場合は警告を表示する．

Steep が提案する型検査に基づくプログラミングは，一見，これまでの Ruby プログラミングと矛盾するスタイルであるように見える．特に，シグネチャの用意やコメントの挿入は，これまでの Ruby プログラミングには無かった要素である．Steep の開発では，2 節で述べたメタプログラミングや動的メソッド検索と高い親和性を持つように注意深く設計を行うことで，Ruby の記述力と静的型検査を両立した新しいスタイルを実現することを目指している．とはいえ，素の Ruby プログラムに比べると，シグネチャの分だけ記述量が増大することは事実である．しかしながら，Steep のシグネチャは，ライブラリの API に関してプログラマが書くドキュメントの機械可読な一形態と見なすこともできる．プログラマがシグネチャを書くことは，型検査を可能にすることだけに留まらず，API を一覧できる良質なドキュメントをユーザーに提供することにも繋がる．したがって，ドキュメンテーションも含めたソフトウェア開発の全行程を考えるならば，Steep を使うことによる記述量の増加は Ruby の簡潔性に影響を与えないと期待している．

4.1 シグネチャと型

Steep では，Ruby のソースファイル（`.rb` ファイル）とは別に，シグネチャファイル（`.rbi` ファイル）にシグネチャを書く．概念上は，プロジェクト全体（`.rb` ファイルの集合）に対して一つのプロジェクトシグネチャが対応づけられる．プロジェクトシグネチャは複数のクラスシグネチャおよび補助定義からなる．プロジェクトシグネチャは複数のシグネチャファイルに分割して書いてもよい．

クラスシグネチャには，そのシグネチャを持つクラスのメソッドやインスタンス変数の型を書く．Ruby のインスタンス変数はアクセス制限がないため，クラスの公開された API の一部をなすと見なす．クラスシグネチャは以下の例のように書く．

```
1: class Stack<'a>
2:   @elements: Array<'a>
3:   def push: ('a) -> Stack<'a>
4:           | <'x> ('x) { ('x) -> 'a } -> Stack<'a>
5:   def pop: () -> 'a
6:   def each: { ('a) -> any } -> Stack<'a>
7:   include Enumerable<'a, any>
8: end
```

これは `Stack` クラスシグネチャの定義である．クラスシグネチャはソースコード上の同名のクラスに対応づけられる．クラスシグネチャは 0 個以上の全称的な束縛型変数（上記例では 1 行目の`<'a>`）をその名前の後に持つことができる．クラスシグネチャには，インスタンス変数の型（2 行目），メソッドの型（3〜6 行目），および他のクラスやモジュールのシグネチャとの関係（7 行目）を書く．各メソッドには複数の型を与えることができる．先に書かれた型が優先的に，そのメソッドを呼び出す式の型付けで使用される．ブロックを受け取るメソッドの型にはブロックの型を`{}`で囲んで書き加える．Ruby のブロックは引数とは異なる構文要素であるため，ブロックの型は引数の型とは異なる記法を用いる．メソッドの型はパラメトリックな多相型であってもよい．ただし，型変数の（全称的な）束縛はメソッドの各型の先頭でのみ許される．例えば，3〜4 行目の `push` メソッドは 2 つの型を持ち，そのうち 2 つ目の型は`'x` を束縛型変数とする多相型である．7 行目の `include` 構文は，`Stack` がモジュール `Enumerable` を mix-in していることを意味する．Ruby の mix-in およびモジュールについての詳細は本論文では省略する．以下，クラスシグネチャの名前を表すメタ変数を k とする．

型変数に具体的な型を代入したクラスシグネチャの集合は部分型関係 $\leq$ をなす．この部分型関係は，クラスの継承関係ではなく，クラスが継承などを通じて獲得するメソッド集合全体の包含関係を用いて定義する．例えば，クラスシグネチャ`Foo` がメソッド `m` だけからなり，クラスシグネチャ `Bar` は同名で同じ型のメソッドを持つ時，`Foo` と `Bar` の継承関係に関わらず，`Bar` $\leq$ `Foo` である．この方針は，2 節で述べた動的メソッド検索を活用した多相性に由来する．

2 節で述べたように，Ruby ではクラスの一部のメソッドにのみ注目することがある．Steep では，この状況に対応して，メソッドの部分集合を表す「インターフェース」の概念を導入する．インターフェースは以下の例のような形でシグネチャファイルに記述する．

```
interface _Poppable<'a>
  def pop: () -> 'a
end
```

インターフェースはクラスの性質の一部を切り取った抽象的な概念であり，インターフェースに対応する実体は Ruby プログラムには現れない．クラスシグネチャに関する部分型関係 $\leq$ は，インターフェースも含めて準同型に拡張される．例えば上述の例において，任意の型 τ について `Stack`$\langle\tau\rangle \leq$ `_Poppable`$\langle\tau\rangle$ である．以下，インターフェースの名前を表すメタ変数を I とする．

Steep における式の型は以下の通りである．シグネチャファイルでは，τ はメソッドの仮引数，返り値，およびインスタンス変数の型に現れる．

$$\tau \quad ::= \quad \alpha \mid k\langle\tau,\ldots,\tau\rangle \mid I\langle\tau,\ldots,\tau\rangle \mid \mathrm{class}(k) \mid \mathrm{any} \mid \tau \vee \tau \mid \tau \wedge \tau$$

α は型変数である．式の型として現れる型変数は，シグネチャで束縛位置が明示されているため，すべて区別される．$k\langle\tau_1,\ldots,\tau_n\rangle$ は，クラスシグネチャk を型引数 $\tau_1,\ldots,\tau_n$ に適用した型である．型引数が無い場合は括弧を省略する．$I\langle\tau_1,\ldots,\tau_n\rangle$ はインターフェースに関する同様の記述である．class(k) はクラスシグネチャk を実装するクラス自身を指す型である．any は動的な型付けを表す型であり，プログラマが明示的に指定した場合の他，文脈から式の型を一意に決定できなかった場合に型検査器によって導入される．$\tau\vee\tau$ と $\tau\wedge\tau$ は，それぞれ union 型と intersection 型を表す．

部分型関係 $\leq$ は any を含む型を除く型の集合に対して準同型に拡張される．any を含む型全体に対して定義される関係 <: は，オブジェクトに対する漸進的型付け [20] の考え方に倣い，any が他のあらゆる型を超越する推移的でない関係として，$\leq$ を拡張することで導入される．Steep は，この <: に対するプログラムの整合性を検査し，<: を満足しない変数の書き換えやメソッド呼び出しを型エラーとして報告する．型検査を通ったプログラムは実行時にメソッド未定義エラー（`NoMethodError`）例外を発生しないと期待される．ただし，any に関する一切の操作は検査の対象外とする．例えば，any 型のレシーバに対するメソッド呼び出しの型は直ちに any 型とし，エラーも報告しない．

4.2 型推論アルゴリズム

Ruby プログラムの型検査は，Ruby プログラム中の `class` 構文および `def` 構文に対してシグネチャを対応付けた上で，各 `def` 構文の本体の型を推論することで行う．メタプログラミングなどを通じてこれらの構文以外の方法で定義されるメソッドへの対応については 4.3 節で後述する．

型推論は，ローカル型推論 [15] の考え方に従い，自分自身を含むすべてのメソッドの型を前提として，メソッド本体の先頭から順に，上向きおよび下向きの両方向で行う．式の型はそれぞれ構文的に隣接する式の型のみを用いて推論され，一度推論された式の型は他の式の型推論の結果によって変更されない．すべてのメソッドおよび仮引数の型は既知であるから，多相的なメソッドに対する暗黙の型適用を除いて，式およびローカル変数の型はほぼ自明である．したがって，主な推論の対象は多相メソッドに対する型引数である．型引数は，各メソッド呼び出し式ごとに，引数列および返り値の型についての <: に関する制約を解くことで求める．メソッドの型が複数与えられている場合には，シグネチャに記載された順でそれぞれ制約の生成と解消を行い，最初に解が得られたものがメソッド呼び出し式の型として採用される．解が存在しない場合は型エラーを報告する．部分型関係 $\leq$ の最大元 $\top$ または最小元 $\bot$ が解の場合は，それらの代わりに any を用いる．特に，制約集合が空の場合は型引数として any が用いられる．制約の作り方および解き方についての詳細は，本稿執筆時点では整理が十分でないため，今後の論文に譲る．

型適用の推論のおおよその動きを例で示す．メソッド `discard` の型を

```
<'x> (_Poppable<'x>, Integer) -> any
```

とし，変数 `stack` の型を `Stack<String>`とするとき，式 `discard(stack, 3)` の型を推論する．まず，`discard` の多相型を fresh な型変数 α でインスタンス化する．次に，メソッドの型と実引数の型から α が満たすべき制約

$$\texttt{Stack}\langle\texttt{String}\rangle <: \texttt{\_Poppable}\langle\alpha\rangle$$

を得る．最後に，この制約を以下のようにして解く．$\texttt{\_Poppable}\langle\alpha\rangle$ は `pop : () -> `α のみからなるので，この制約を満たすには，`pop` メソッドの型に関して

$$\texttt{() -> String} <: \texttt{() -> }\alpha$$

を満たせばよい．関数型に関する標準的な部分型規則より，

$$\texttt{String} <: \alpha$$

である．この関係を満たす（any を除き）最小の型は `String` であるから，$\alpha = $ `String` を解とする．

4.3 Ruby プログラムに書き加える型注釈

Steep では，ローカル型推論の方針などの理由で，プログラマの意図に反して any が導入される場合がある．また，Ruby では動的な型検査がメソッド（`Object#is_a?`など）で行われるが，Steep ではこれらのメタプログラミング要素の結果が型検査に反映されない．このような場合にも，より精密な型検査を行うため，Steep ではローカル変数の型を型注釈として Ruby コード内に宣言する記法を導入した．型注釈は，Ruby プログラムの意味を変えることがないよう，Ruby のコメントとして記述する．コメントであるから，Ruby プログラムの実行に影響を与えることはなく，また将来的に不要になったとしても除去ないし無視できる．Ruby の簡潔さを妨げない点については，実際の使用感も含めた慎重な評価が必要と思われる．

あるコード位置でローカル変数 x が型 τ を持つことを，その位置に以下のコメントを書くことで宣言する．

```
# @type var x : τ
```

先頭の `#` は Ruby では行コメントの開始を表す記号である．この注釈には以下の 2 つの役割がある．

- 新たに定義されるローカル変数の型の指定．型推論器が推論したローカル変数の型よりも具体的な型を指定することで，より精密な型検査を実施することができる．例えば以下のコードを考える．

  ```
  1: numbers = []       # numbers の型は Array⟨Integer⟩ のつもり
  2: numbers[0] = 1
  3: numbers[1] = "2"   # ここで型エラーを報告してほしい
  ```

  ローカル型推論により，1 行目に定義された `numbers` の型は，2 行目以降の文脈を参照せずに決定される．型推論器は，1 行目だけでは配列の要素の型を一意に決められないため，`numbers` に `Array`⟨any⟩ 型を与える．そのため，2 行目，3 行目では要素の型は検査されず，3 行目は型エラーを起こさない．1 行目の前に

  ```
  # @type var numbers : Array<Integer>
  ```

  と書くことで，この問題を回避できる．

- 定義済みのローカル変数の型のキャスト．Ruby では，以下の例のように，引数が属するクラスに応じて場合分けをするコードがよく現れる．

  ```
  1: def ==(other)                # 型は any → bool
  2:   if other.is_a?(Person)     # Person クラスのインスタンスであるか確認
  3:     other.name == name
  4:   else false end
  5: end
  ```

  `Person` クラスには `name` メソッドが定義されているとする．このメソッドは，引数 `x` が `Person` クラスのインスタンスであった場合，`x` のメソッド `name` を呼び出す．この状況を静的型付けの観点から分析すると，3 行目に限定して `x` の型が `Person` にキャストされ，2 行目の `is_a?` による条件判定は安全にキャストするための動的型検査とみなすことができる．（上述のようなパターンに限れば注釈なしで対応することも可能だが，一般的に）このような状況を Steep に伝えるためには，2 行目と 3 行目の間に変数 `other` の型に関する以下の注釈を加える．

  ```
  # @type var other : Person
  ```

  この位置での`@type var` の指定は，`then` 節に限定して `other` の型を `Person` にキャストすることを表す．キャストの正しさは注釈を書いたプログラマの責任に帰する．

また，メソッドに関する以下の注釈を実験的に導入している．

`# @dynamic` m	このクラスのメソッド m の定義の有無に関する検査を省略する．
`# @type method` m `:` ty	このクラスのメソッド m の型は ty である．

`@dynamic` はメタプログラミングによって定義されるメソッドの取り扱いのために導入された．Steep は，Ruby に組み込みのメタプログラミングだけでなく，新たなメタプログラミングライブラリの開発も妨げないよう，特定のメタプログラミング機能に関する知識を持たない．その代わりに，メタプログラミングを行うコードの静的意味をプログラマが注釈として指定する方針を取る．`# @dynamic` m と書くことで，Steep は検査を省略し，メソッド m の定義が存在しシグネチャに書かれた m の型を持つことの検査はプログラマの責任に委ねられる．例えば，Ruby の組み込みメソッド `attr_reader :foo` は，シンボル`:foo` を引数に取り，インスタンス変数`@foo` を読むメソッド `foo` を定義するメソッドである．このような場合，`attr_reader :foo` の前に`# @dynamic foo` と指示することで，Steep に `foo` メソッドが定義されていることの検査を省略させることができる．

`@type method` m 注釈は，複数の型を持つメソッドの型検査のために導入された．前述のとおり，ローカル型推論は仮引数の型を既知としてメソッド本体の型検査を進める方式である．メソッドの型がただ 1 つの場合，仮引数の型は与えられているので自明である．一方，一つのメソッド本体が複数の型を持つ場合，そのメソッド本体の型検査のために仮引数の型を一つに定めることは難しい．例えば

```
foo : (Integer) -> Integer | (String) -> String
```

に対して以下の実装を与え，型検査をすることを考える．

```
def foo(x)
  if x.is_a?(Integer) then 42
  else "str" end
end
```

このメソッド `foo` の動的意味を考えるならば，`Integer` に対して `Integer` を返し，`String` に対して `String` を返す．しかし，Steep の型推論方式では，この実装は `(Integer) -> Integer` および `(String) -> String` のどちらの型でも型検査が通らない．このとき，このメソッド定義の直前に

```
# @type method foo: (any) -> (Integer ∨ String)
```

と書くことで，`foo` の実装をこの型に対して型検査することを Steep に指示することができる．この型で型検査したこととシグネチャに書かれた型との整合性はプログラマの責任に委ねられる．オーバーロードされたメソッドに対する，より適切な対応（例えば [3] などを参照）は今後の課題である．

メソッドに関するこれらの注釈をコード中に書く必要は必ずしもなく，メソッド型の一部としてシグネチャファイル内に記述するような設計もありうる．しかし，前述のとおり，シグネチャファイルは公開 API を記述するドキュメントとしての役割も持つ．実装の内部に関する局所的な注釈をシグネチャファイルに書くことを避けるため，現時点ではコード内の注釈として記述する方式を選択した．

4.4 現時点での評価と今後の課題

Steep は Ruby で実装されており，オープンソースソフトウェアとして公開されている [21]．実装では，上述した基本的な設計に加え，型による分岐，`nil` や `bool` などの基底型，関数オブジェクト型，タプル，レコード，シングルトン型など，実用上必須の拡張を備える．

ツールとしてのSteepは，型推論器本体に加え，Rubyソースコードからシグネチャファイルの雛形を生成する`scaffold`コマンドを提供する．`scaffold`コマンドはRubyソースコードから`class`文と`def`文を抜き出すことしか行わない．より正確なシグネチャの作成支援には，3節で報告した型プロファイラが応用できると期待する．

第2著者は，Steepを自社のソフトウェア開発プロジェクトで利用されているライブラリに適用することで，実用のRubyプログラムに対する適用可能性や使いやすさに関する予備的な評価を行った．型検査には，1853行のライブラリに対して，677行のシグネチャおよび38件の型注釈が必要であった．シグネチャには40個のクラス，2個のモジュール，1個のインタフェース，および232件のメソッドが含まれていた．シグネチャがライブラリのAPIに関するドキュメントとして機能することも確認した．Steepを適用したことによって得られた大きな利点の一つは，互換性を失う形でのアップデートに以前より積極的に取り組むことができるようになったことである．以上の結果より，SteepはRubyプログラムの品質を高めるのに有用であるとの印象を得て，実際にSteepのより広範な利用を社内で推進しているところである．

一方，ソースコード全体の3分の1に匹敵する量のシグネチャが必要であったことについては，Rubyの簡潔性の観点から慎重な評価が必要と思われる．このシグネチャの多さの原因として以下の2点が考えられる．

- メタプログラミングによるコードの短縮．評価に用いたプログラムでは，全メソッドの約3分の1（84個）が`attr_reader`などのメタプログラミングによって1メソッドにつき1行以下で定義されていた．Steepでは，メタプログラミングによって定義されるメソッドについてもシグネチャでの型定義が必要であり，Rubyプログラムの行数と比較したときのシグネチャの行数を大きくしている．
- 局所的なメソッドに対するシグネチャの記述．例えば，`private`と宣言されたメソッドの多くは各クラスに局所的にのみ使われており，従って一般には，シグネチャに現れないはずである．一方，Steepでは，ローカル型推論の方針により，各クラスに局所的なメソッドについてもシグネチャの記述が必要である．インスタンス変数に関しても同様である．

Steepで採用した型注釈をコメントとして記入する方針は，Rubyの構文を拡張せず，またRubyプログラムの意味を変更しない点で受け入れられるものであると考えている．一方で明らかに「Rubyプログラムへの特別な言語の埋め込み」でもあり，1節に示された型検査に求められる要求と完全には一致していない．実用上プログラム中の型注釈が必要となる場合が減らせるよう，型推論アルゴリズムやツール全体の設計について，検討が必要である．

型システムの性質や型推論アルゴリズムの詳細を整理することは今後の課題である．筆者らは，実用的なRubyプログラムにおいて現れるほとんどの場合で，型検査が通ったプログラムは`NoMethodError`例外を発生しないことを保証することを念頭にSteepを設計した．しかしながら，型推論アルゴリズムなどに未整理の点が多く残るため，筆者ら自身もSteepの性質を完全に把握しているわけでない．2節で述べた通り，本プロジェクトは健全性を示すことを目的としていないが，Steepの振る舞いを理論的側面から整理することは実用上も価値があると考えられる．Steepの型検査器の性質や実用上の問題点は，今後の理論的な整理と実践的なSteepの適用を通じて，明らかにしていく予定である．その過程において，漸進的型付けにおける型推論に関する研究（例えば[19, 7]など）も参考になると思われる．

5　関連プロジェクト

本節では，Rubyプログラムの静的解析に向けた，筆者ら以外による取り組みをいくつか紹介する．

mruby-meta-circular [12] は，Ruby の別実装である mruby 向けの静的解析器である．mruby-meta-circular はプログラムを抽象的に実行し，その過程で遭遇したメソッド呼び出しを記録し，シグネチャのような形式に集約して表示する．本プロジェクトの型プロファイラは，mruby-meta-circular のアプローチに着想を得て開発が始められたものである．しかしながら，様々なヒューリスティクスを利用した経験的な解析手法を用いている [24] こと以上の mruby-meta-circular の技術的詳細は不明である．

動的言語への静的型付けに関して Ruby に焦点を当てた研究が Foster を共著者に含む一連の論文で報告されており，DRuby [6]，$\mathcal{P}$Ruby [5]，Rubydust [2]，RTC [17]，RDL [22], Hummingbird [16]，および RTR [8] などのツールが提案されている．近年の研究成果はソフトウェアとしての RDL に集約されている．RDL では，ドメイン固有言語（DSL）でプログラム内に埋め込まれたメソッドのシグネチャを用いて，実行時にメソッド定義本体の型検査を行う．型検査は検査対象のメソッドを呼び出した時点でのクラスおよびシグネチャの内容に基づいて行われるため，複雑なメタプログラミングにも自然に対応する．本プロジェクトの狙いは Ruby プログラムを実行せずに型検査することであり，実行時に型検査を行う RDL とは方向性が異なる．

Sorbet は Stripe 社によって開発されている型検査ツールである．同社における製品開発に利用されていると報告されている [14, 13]．Sorbet は，型注釈を書くための埋め込み DSL で拡張した Ruby プログラムに対し静的型検査器を提供する．メタプログラミングについては，いくつかの組み込みメソッドに関する知識を型検査ツールに組み込むことでサポートしている．本論文執筆時点では 2 件の口頭発表のみが公表された資料であり，それ以上の詳細は公開されていない．

最後に，Ruby ではないが動的言語 JavaScript に静的型付けを加えた言語 TypeScript [11] と，本プロジェクトの Steep との関連について述べる．TypeScript と Steep は，動的言語に構造的な部分型を導入し，プログラマが書いたシグネチャに対する実装の矛盾を検査する点で，対象言語は異なるものの方向性は共通している．Steep が TypeScript と異なる点は，TypeScript は JavaScript に対する前方互換性がない拡張言語であるのに対し，Steep プログラムは Ruby プログラムとしてそのまま動くことである．この方針の違いは，システム全体の設計にも影響を与えている．例えば，独自構文を導入しなければ書くことが難しい型適用に関する注釈を Steep は提供しない．

6　まとめ

本論文では，Ruby の次期メジャーリリースである Ruby 3 に向けて静的解析機能を設計・開発するプロジェクトの経過報告を行った．メタプログラミングや動的メソッド検索による Ruby の記述力を妨げることなく，プログラムの品質向上を支援する静的解析機能の実現を目指し，抽象解釈に基づく型プロファイラと，漸進的型付けの考え方を取り入れた型検査器 Steep の開発に取り組んでいる．これらのシステムは未完成ではあるものの，人工的ではない例に対する適用をすでに試みており，予備的ではあるが有望な結果が得られた．今後もこれらのシステムの完成を目指し開発を継続する予定である．

本プロジェクトの終着点は静的解析機能のリリースであるが，最終的なリリース形態は未定である．本プロジェクトで開発している 2 つのシステムはあくまで例にすぎず，二者のうちのどちらが主流となるか，相補的な二者が一つのシステムに統合されるか，あるいはそのどちらでもない方式が採用されるかは，今後のプロジェクトの進展により次第に定まると思われる．本論文で経過報告した 2 つのシステム以外の提案も，1 節および 2 節で述べた方針に大きく反しない限り，歓迎・検討したいと考えている．

謝辞

本プロジェクトの立ち上げおよび運営にご尽力いただき，Ruby インタプリタの詳細についての情報もご提供いただいた笹田耕一氏に感謝します．抽象的な実行による型情報抽出アプローチについての着想をいただいた三浦英樹氏に感謝します．また，本論文に関する有益なコメントを頂いた査読者に感謝します．

本研究の一部は，東北大学電気通信研究所共同プロジェクト研究 採択番号 H28/B07 「産業的プログラミング言語開発とプログラミング言語基盤研究の技術融合」として実施されたものです．また，本研究の一部は JSPS 科研費 15K15964，15H02681 の助成をそれぞれ受けたものです．

参考文献

[1] プログラミング言語 Ruby リファレンスマニュアル. https://docs.ruby-lang.org/ja/2.6.0/doc/.

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pp. 459–472, New York, NY, USA, 2011. ACM.

[3] Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, Vol. 1, No. ICFP, pp. 41:1–41:28, August 2017.

[4] クックパッド開発者ブログ, Ruby 2.6 新機能：本番環境での利用を目指したコードカバレッジ計測機能. https://techlife.cookpad.com/entry/2018/12/26/103330. 2019 年 1 月 6 日閲覧. 本記事の執筆は本論文の第 1 著者による.

[5] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pp. 283–300, New York, NY, USA, 2009. ACM.

[6] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pp. 1859–1866, New York, NY, USA, 2009. ACM.

[7] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pp. 303–315, New York, NY, USA, 2015. ACM.

[8] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for Ruby. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pp. 269–290, Cham, 2018. Springer International Publishing.

[9] Yukihiro "Matz" Matsumoto. Coming Soon... RubyKaigi 2014, keynote speach, `http://rubykaigi.org/2014/presentation/S-YukihiroMatzMatsumoto/`.

[10] Yukihiro "Matz" Matsumoto. Ruby3 typing. RubyKaigi 2016, keynote speach, `http://rubykaigi.org/2016/presentations/yukihiro_matz.html`.

[11] Microsoft. TypeScript - JavaScript that scales. `https://www.typescriptlang.org`.

[12] miura1729/mruby-meta-circular: mruby by mruby. `https://github.com/miura1729/mruby-meta-circular`.

[13] Dmitry Petrashko, Paul Tarjan, and Nelson Elhage. Gradual typing of Ruby at scale. Strange Loop 2018, `https://www.thestrangeloop.com/2018/gradual-typing-of-ruby-at-scale.html`.

[14] Dmitry Petrashko, Paul Tarjan, and Nelson Elhage. A practical type system for Ruby at Stripe. RubyKaigi 2018, `https://rubykaigi.org/2018/presentations/DarkDimius.html`.

[15] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, Vol. 22, No. 1, pp. 1–44, January 2000.

[16] Brianna M. Ren and Jeffrey S. Foster. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 462–476, New York, NY, USA, 2016. ACM.

[17] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The Ruby type checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pp. 1565–1572, New York, NY, USA, 2013. ACM.

[18] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[19] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pp. 7:1–7:12, New York, NY, USA, 2008. ACM.

[20] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pp. 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[21] soutaro/steep: Gradual Typing for Ruby. `https://github.com/soutaro/steep`.

[22] T. Stephen Strickland, Brianna M. Ren, and Jeffrey S. Foster. Contracts for domain-specific languages in Ruby. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pp. 23–34, New York, NY, USA, 2014. ACM.

[23] 笹田耕一, 松本行弘, 前田敦司, 並木美太郎. Ruby 用仮想マシン YARV の実装と評価. 情報処理学会論文誌（PRO）, Vol. 47, No. SIG2(PRO28), pp. 57–73, 2006.

[24] 三浦英樹. 私信.