JVM上の動的言語のための抽象解釈の実装

馬谷 誠二[1,a)]

概要：本論文では，現在我々が開発を行なっている Clojure 言語用の抽象解釈器 OPAL/Clj の実装について説明する．OPAL/Clj は，JVM バイトコードレベルの抽象解釈と Clojure コードの「具象解釈」を混ぜながら実行することにより，動的で複雑な言語機能を使用する現実的なコードであっても，その振舞いをある程度正確に解析することが可能である．バイトコードの抽象解釈には既存のフレームワークを用いているが，混ぜながらの実行を実現するには，単純にフレームワークの機能を呼び出すのではなく，Clojure コンパイラにも修正を加える必要がある．本論文では，Clojure コンパイラにどのような修正を行ったかを中心に実装について詳述する．

キーワード：抽象解釈，Clojure，JVM バイトコード，コンパイラ

1. はじめに

抽象解釈 [1] は，プログラムを静的に解析する方法の一つである．特に，解析対象のプログラミング言語の意味が，何らかの形の解釈として定義されている場合に有効な解析手段広く用いられている [2], [7], [10].

一般に，あるプログラミング言語の抽象解釈器を実現する場合，その言語のための通常の解釈方法に基づいて，内部で扱う様々なデータ型を抽象化し，解析したい性質を導き出すのに必要な情報だけが含まれるよう適宜修正することにより構築する．しかしながら，実際には，動的言語やスクリプト言語の処理系であってもソースコードを直接解釈実行することは稀であり，ほとんどの場合，より低レベルなコードへのコンパイラを用いている．たとえば，Clojure 言語 [4] のための抽象解釈器が欲しいとしても，処理系は JVM バイトコード [6] へのコンパイラとして実装されており，ソースコードのインタプリタなどどこにも存在しない．ラムダ計算のような小さな言語に対するインタプリタを一から定義することは容易いが，現実のプログラミング言語のインタプリタを一から構築し，リファレンス実装（コンパイラ）と厳密に等しい実装をつくるのは開発コストが大き過ぎ，抽象解釈器を構築するための現実的なアプローチとは言い難い．Clojure に限って言えば，そもそも厳密な正確さで書かれた言語仕様は存在しない．また，Clojure ライブラリ中には，（Java の）ASM[9] を呼び出して，直接バイトコードを生成するような特殊な言語機能も存在し，JVM レベルの実行モデルを考慮せずソースコードレベルの解釈によってそのような言語機能に意味を与えることは困難である．

Clojure プログラムを解析するための代替手段として，コンパイラが生成するバイトコードに既存のバイトコード解析ツールを適用することが考えられるが，残念ながら，動的言語のコンパイラが生

[1] 京都大学大学院情報学研究科
[a)] umatani@kuis.kyoto-u.ac.jp

成するバイトコードは静的言語（たとえば，JavaやScala）と比べ非常に複雑であり，上手く解析できないことが先行研究によって明らかとなっている [5].

そこで，本研究では既存のバイトコードレベルの抽象解釈器を用いながら，動的言語の振舞いを正確に把握できるための抽象解釈手法の開発を目指す．本論文で提案する手法では，バイトコードレベルの抽象解釈と対象プログラムの通常の実行を適宜切り替えながら同時に実行することにより，解析を行う．バイトコードレベルの解析に抽象解釈を用いることで通常の実行とシームレスに切り替えることが特徴と言える．

本稿の残りの構成は次のとおりである．まず，2節では提案フレームワーク OPAL/Clj の設計について述べる．次に，3節では，OPAL/Clj の実装を理解するのに必要となる知識である Clojure 処理系の内部構造について簡単に説明する．その後，4節でフレームワークの実装のうち，特に重要な部分について述べる．最後に5節でまとめと今後の課題について述べる．

2. OPAL/Clj

本節では，提案する抽象解釈フレームワーク OPAL/Clj について簡単に紹介する．より詳しくは，[13](PRO 発表資料) の発表資料を参照して貰いたい．[*1]

2.1 OPAL

OPAL[3], [12] は JVM バイトコードを解析対象とする抽象解釈フレームワークである．単なる抽象解釈器の一つではなく，ソフトウェアプロダクトライン手法を用いることで，豊富な再利用/拡張可能なコンポーネントを柔軟に組み合わせることによって，目的に応じた高水準のプログラム解析をバイトコードの抽象解釈器として構築することが可能である．

バイトコードレベルの抽象解釈では，メソッド本体中のバイトコード命令が先頭から順に実行される．操作対象となるオペランドスタックとレジスタ群（メソッドローカル変数）には，実際の値ではなく抽象値が含まれ，抽象解釈の結果は各命令の前後で取り得るオペランドスタックとレジスタ群の状態として表現される．

```
1  public class Demo {
2    public static int compute(
3        boolean b, int x, int y) {
4      int r;
5      if(b) r = x + y; else r = x * y;
6      return r;
7    }
8    public static void main() {
9      int r = compute(true, 3, 4);
10   }
11 }
```

図 1 解析対象サンプル Java コード

表 1 public static void Demo#main() の解析結果

PC	Instruction	Stack	Registers
0	ICONST_1		0. UNUSED
1	ICONST_3	{ 1: int }	0. UNUSED
2	ICONST_4	{ 3: int } { 1: int }	0. UNUSED
3	INV.S Demo#compute	{ 4: int } { 3: int } { 1: int }	0. UNUSED
6	ISTORE_0	{ 7: int }	0. UNUSED
7	RETURN		0. { 7: int }

たとえば，**図 1** の簡単な Java コードに対し，OPAL上に構築されたバグ検出ツール BugPicker[11] の内部で実際に用いられている抽象解釈器を用いると，**表 1**，**表 2** のような解析結果が得られる．

各行の Stack と Registers には，同じ行の Instruction を実行する直前のオペランドスタックとレジスタ集合の状態が書かれている（Stack 中のスタック表現は，紙面の上側がスタックトップである）．抽象値 { v: int } は int 型の実際の値 v だけを含むシングルトンセットを意味する．また，表中には含まれていないが，{ $v_1, \ldots, v_n$: int } で複数の値を含む抽象値を意味する．BugPicker の抽象解釈器では，int 集合の大きさが7を超えると，正確な値を保持することをあきらめ，（Scala のシング

[*1] 著者に直接連絡して貰えれば，PDF ファイルをお渡しします．

表 2 `static int Demo#compute(boolean,int,int)` の解析結果

PC	Instruction	Stack	Registers
0	ILOAD_0		0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
1	IFEQ 11	{ 1: int }	0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
4	ILOAD_1		0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
5	ILOAD_2	{ 3: int }	0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
6	IADD	{ 4: int } { 3: int }	0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
7	ISTORE_3	{ 7: int }	0. { 1: int } 1. { 3: int } 2. { 4: int } 3. UNUSED
8	GOTO 15		0. { 1: int } 1. { 3: int } 2. { 4: int } 3. { 7: int }
11	ILOAD_1	n/a	n/a
12	ILOAD_2	n/a	n/a
13	IMUL	n/a	n/a
14	ISTORE_3	n/a	n/a
15	ILOAD_3		0. { 1: int } 1. { 3: int } 2. { 4: int } 3. { 7: int }
16	RETURN	{ 7: int }	0. { 1: int } 1. { 3: int } 2. { 4: int } 3. { 7: int }

ルトン・オブジェクトで表される）真に抽象的な値 AIntValue に置き換えられる．

OPAL には，他にも，C クラスの何らかの値を表す抽象値 {_: C}，S クラス（インタフェース）のサブクラスの何らかの値であるか，または null である値を表す抽象値 {_ <: S, null} 等が用意されている．

表 1 のリターン命令直前のレジスタ 0 に含まれている抽象値から分かるとおり，OPAL による解析は十分な精度[*2]を確保できている．また，メソッド間解析が適切に行われていることも，表 2 から見て取ることができる．

次に，同じ抽象解釈器を図 1 と同様の計算を行う Clojure プログラム：

```
(defn compute [b x y] (if b (+ x y) (*
    x y)))
(fn&a [] (compute true 3 4))
```

に適用してみる．解析結果を**表 3** に示す．

見て分かるとおり返り値は {_ <: Object, null } となっており，まったく上手く解析できていない．これは，トップレベル変数 compute をあらわす Var クラスのオブジェクトから，関数オブジェクトを取り出す命令 3 の INVOKEVIRTUAL を上手く扱えていないことに原因がある．その後の 1 (true), 3, 4 の評価結果の精度は落ちていないが，抽象解釈中に compute 関数の正確な値を取得できなければどうしようもない．

なお，いくつかの言語とバイトコードレベル静的解析ツールの組み合わせに関する報告 [5] によると，Clojure 言語に限らず，動的言語とそれらの解析ツールの組み合わせでは，上記とよく似た原因によりまったく上手く解析できないことが分かっている．

動的言語をコンパイルすることによって得られるバイトコードには，ソース言語レベルでしか正確な意味を掴めない部分が含まれており，そのため，低水準なバイトコードレベルの解析ツールを適用するには本質的に困難な点があると言える．具体的には，たとえば上記の例のように，トップレベル環境中の変数を参照するには，通常複雑なデータ構造（巨大なマップ）により実装されてい

[*2] この場合，対象コードが非常に単純なため，Clojure 処理系による通常評価と同じ正確な値が得られている．

表 3 compute 関数の解析結果

PC	Instruction	Stack	Registers
0	GETSTATIC fn__13475.const__0		0. {_: fn__13475}
3	INVOKEVIRTUAL Var#getRawRoot	{_ <: Var, null}	0. {_: fn__13475}
6	CHECKCAST IFn	{_ <: Object, null}	0. {_: fn__13475}
9	GETSTATIC Boolean.TRUE	{_ <: IFn, null}	0. {_: fn__13475}
12	GETSTATIC fn__13475.const__2	{_ <: Boolean, null} {_ <: IFn, null}	0. {_: fn__13475}
15	GETSTATIC fn__13475.const__3	{_ <: Long, null} {_ <: Boolean, null} {_ <: IFn, null}	0. {_: fn__13475}
18	INVOKEINTERFACE IFn#invoke	{_ <: Long, null} {_ <: Long, null} {_ <: Boolean, null} {_ <: IFn, null}	0. {_: fn__13475}
23	ARETURN	{_ <: Object, null}	0. {_: fn__13475}

るトップレベル環境を扱う複雑なバイトコードを解析する必要があり，なおかつ，トップレベル環境は新たな関数等を定義する度に段階的に膨らんでいくという扱いにくさも備えている．原理的には，処理系のランタイム（実装 Java クラス）全体を適切に抽象化してやるという方法も考えられなくはないが，その作業は抽象的な言語処理系を一から構築するようなものであり，あまり現実的とは言えない．

2.2 OPAL/Clj の仕様

本節では，前節の最後で述べた問題点を考慮し Clojure コードもある程度正確に解析可能なよう拡張した OPAL フレームワークである OPAL/Clj について説明する．まず，問題点を解決するための基本的なアイディアについて述べ，その後，それを実際に実現している Clojure 側 API を詳しく説明する．

2.2.1 基本方針

OPAL による抽象解釈で解析精度が落ちるのは，Clojure の「動的な」機能を使用しているいくつかの箇所であるため，そのような部分式の評価だけ精度を上げる別の解釈手段を提供するというのが，提案手法の基本的なアイディアである．Clojure を含む動的言語では，コンパイラは言語処理系の実行環境の上で動作していることが多い．そこで，精度を上げる別の解釈手段として，ここでは**実行環境上で普通に評価する**ことにする．以降，通常の評価のことを抽象解釈と対比的に**具象解釈**と呼ぶことにする．

OPAL/Clj では，具象解釈をどこにどの程度混ぜるかの決定を自動では行わない．どの程度の解析精度が必要となるのかを理解しているのは，基本的には，応用に関する知識を持っているフレームワークの利用者（高水準プログラム解析の実装者）であり，OPAL/Clj は，利用者が具象解釈する範囲を自由に調整可能なよう，API の柔軟性を確保するだけに留めている．明らかに具象解釈するべき箇所[*3]についてはある程度自動化する方が便利であるが，その実現は今後の課題とする．

提案手法を組み込んだ OPAL/Clj フレームワークの全体構成を**図 2** に示す．

OPAL/Clj はモジュールの多層構造からなり，図 2 は，各層が隣接するすぐ上の層の機能を呼び出しているという関係を表わしている．最上層の OPAL 層は Scala で書かれたライブラリであり，本研究では一切手を加えていない．抽象ドメインの

[*3] たとえばトップレベル変数の参照は時間のかかる処理ではなく停止性の心配もないため，必ず具象解釈するのが良さそうである．

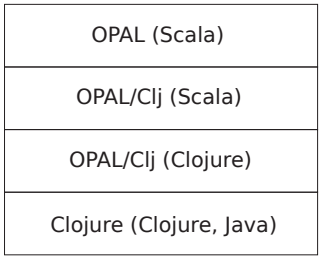

図 2 OPAL/Clj の構成（括弧内はその層の実装言語）

定義の追加などは OPAL/Clj (Scala) 層で実現されている．

先に述べたとおり，具象解釈を行うには Clojure の実行環境が必要であり，それが最下層に位置している．最上層の抽象解釈と最下層の具象解釈との橋渡しを行うのが，OPAL/Clj (Scala) 層と OPAL/Clj (Clojure) 層である．これら二つの層が具体的に何を行っているのかについては，4 節で触れる．

抽象解釈の起動は，OPAL/Clj (Clojure) で定義された API (2.2.2 節を参照) を呼び出すことにより行う．バイトコードへのコンパイルもその層で行われ，結果のバイトコードが上の層へと渡される．

最上層での抽象解釈中，式の一部を具象解釈することになると，OPAL/Clj は上から下に向けて具象解釈の依頼を投げる．依頼を受け取った Clojure 実行環境はその時点での（トップレベル環境を含む）実行環境を用いて通常どおりに式を評価する．評価結果の具体的な値（以降，抽象値と対比的に**具象値**と呼ぶことにする）は，下から上へと返される過程において，正確な値を一つだけ含む ConcreteValue クラスにラップされる

2.2.2 Clojure 側 API

解析対象の Clojure プログラム中に，抽象解釈の実行を指示する構文，および，具象解釈する箇所を指定するための構文について説明する．

まず，OPAL/Clj の構文中で頻繁に用いられる Clojure のメタデータについて簡単に説明する．Clojure のメタデータとは，大まかには Java のアノテーションに相当する機能である．Clojure のマップ型の値を任意のデータ[*4]に対しメタデータとして付加することができる．具体的には，関数 `(with-meta` ⟨*data*⟩ ⟨*map*⟩`)` によりデータ ⟨*data*⟩ にメタデータ ⟨*map*⟩ を付加し，関数 `(meta` ⟨*data*⟩`)` により ⟨*data*⟩ に付けられたメタデータを取り出せる．また，Clojure コード中のフォームにメタデータを付加するための簡便な記法として「`^`⟨*map*⟩ ⟨*form*⟩」と書くことのできるリーダマクロも用意されている．さらに，「`^`⟨*class-name*⟩ ⟨*form*⟩」は「`^{:tag` ⟨*class-name*⟩`}` ⟨*form*⟩」の省略記法であり，「`^`⟨*keyword*⟩ ⟨*form*⟩」は「`^{`⟨*keyword*⟩ `true}` ⟨*form*⟩」の省略記法である（たとえば，⟨*class-name*⟩ には `String`，⟨*keyword*⟩ には `:foo` などと書ける）．

[*4] 厳密には，整数値など一部のプリミティブなデータには付加することができない．

(a) 抽象解釈の実行

[構文] `(fn&a [{`⟨*abs-desc*⟩ ⟨*param*⟩`}...]` ⟨*body*⟩`...)`

[意味] fn 式（ラムダ式）と同様，関数オブジェクトを生成するが，それに加え，関数本体式の抽象解釈も行う．解析結果は関数オブジェクトにメタデータとして付けられている．

関数本体式の抽象解釈時，各パラメータが含んでいる抽象値は以下のいずれかの記法によるプレフィックス ⟨*abs-desc*⟩ で指定できる：

- 無指定: `Object` クラスの何らかの値
- `^`⟨*class*⟩: ⟨*class*⟩ クラスの何らかの値
- `^{:conc` ⟨*expr*⟩`}`: ⟨*expr*⟩ を評価（具象解釈）した結果の値（`ConcreteValue`）

現在の実装では，解析結果メタデータには次の二つの情報が含まれている：

- `:opal-full-result`
  図 1 等の表形式と全く同じデータを含む S 式
- `:opal-summarized-return-value`
  関数の返り値が取り得る抽象値．複数のリターン命令がある場合は，すべての返り値の抽象ドメイン上の最小上限（join）とする．

(b) 具象解釈パートの指定

[構文] `^:conc` ⟨*expr*⟩

[意味] fn&a 式の本体中，具象解釈したい部分式を指定．fn&a 式の本体以外でこのアノテーションをつけた場合，何の意味も持たない．

先述の (compute true 3 4) の評価は：
(1) 式 compute の評価
(2) 式 true，3，4 の評価
(3) 手順 (1) によって得られる関数オブジェクトを手順 (2) で得られる引数に適用
の順に実行される．OPAL の抽象解釈も，当然のことながら同様の順序で実行を進めるが，たとえば手順 (1) だけを具象解釈したい場合には：

```
(fn&a [] (^:conc compute true 3 4))
```

と書けば良い．

2.3 クロージャの扱い

最後に，自由変数への参照を含む関数（クロージャ）についても，適切にアノテーションを加えることで上手く扱うことができることを示す．

簡単な例として，ラムダ式 (fn [x] (fn [] x)) を考えることにする．まず，試しに^:conc アノテーションをまったくつけず：

```
(def c1 (fn&a [] (((fn [x] (fn [] x))
    3.14))))
```

を抽象解釈してみると，返り値は{_ <: Object, null}となる．

Closure コンパイラがクロージャを JVM 上でどのように実現しているかの詳細は省略するが，基本的に自由変数の値は，関数オブジェクトの生成時，フィールド中に格納される．本体コード実行中の自由変数への参照は，対応するフィールドからの読み出しとなる．上記コードの抽象解釈中，自由変数 x に対しそららの処理を行うバイトコードを OPAL がどのように解析したかを**図 3** に示す．PUTFIELD 命令によって格納している抽象値{_: double }が，GETFIELD 命令によって取り出す時には{_ <: Object, null}になってしまっていることが，図から容易に読み取れる．

解決方法の一つは，OPAL の抽象ドメイン定義を修正し，フィールドアクセス関連の解析精度を上げることである．しかし，関数オブジェクトのフィールドだけを特別扱いし一般のデータ構造と区別するのは，若干アドホックな方法であり，実装も煩雑となってしまう．

PC	Instruction	Stack	Regs
5	...	...	...
6	PUTF fn__13278.x	{_ <: Double, null} {_: fn__13278 }	...
9	RETURN		...

(a) 自由変数の格納（fn__13278 クラスのコンストラクタ中)

PC	Instruction	Stack	Regs
0	...	...	...
1	GETF fn__13278.x	{_: fn__13278 }	...
4	ARETURN	{_ <: Object, null}	...

(b) 自由変数の参照（fn__13278 クラスの invoke メソッド中)

図 3 クロージャの自由変数アクセスの解析結果

そこで，ここでは OPAL/Clj の機能だけを使って改善を試みることにする．クロージャ中の自由変数アクセスが問題なため，クロージャ生成式を具象化する次のコード：

```
(def c2 (fn&a [] ((^:conc (fn [x] (fn []
    x)) 3.14))))
```

が良さそうに思えるが，これではクロージャの本体コードを単に具象解釈するだけであり，その解析は行われていない．そこで，以下のように書いて外側の関数の**引数を束縛するという処理だけ**を具象解釈することにする．

```
(def c4 (fn&a []
        ((^:conc (fn [x]
                   (fn&a [] ^:conc x))
          3.14))))
```

具象解釈される関数の本体を fn&a 式とすることで，再び抽象解釈を起動している．抽象解釈中，具象関数の抽象値（ここでは 3.14 を抽象化した{_: double }）への適用が起こっており，内側の fn&a 式は変数 x がこの抽象値に束縛された状態で評価される．そのため，その本体中で変数 x を参照する際には，**Clojure 環境中で変数 x に幾許されている抽象値**を正しく参照するため^:conc アノテーションが必要となっている．

以上をまとめると，OPAL/Clj におけるクロージャの正確な解析は：

- 具象関数を抽象値に適用するこにより，正確な束縛をつくる
- 抽象解釈中の自由変数参照は^:conc により，その正確な束縛を参照する

により可能となる．

3. Clojure 処理系

本節では，OPAL/Clj の実装を理解するのに必要となる Clojure 処理系に関する前提知識について述べる．まず，コンパイラの全体構成を概観し，各フェーズで行われる処理を簡単に説明する．次に，OPAL/Clj の実現にとって重要となる関数閉包の実装方法について詳述する．

なお，通常の方法でインスールした Clojure 処理系中のコンパイラは Java で書かれているが，OPAL/Clj では互換性のある[*5]Clojure 自身で書かれたコンパイラ [8] を利用している．

3.1 全体構成

Clojure 処理系は JVM バイトコードへのコンパイラとして実装されている．これは，対話環境 (REPL) に入力される式の評価についても同様であり，バイトコードを含むクラス定義へと一旦コンパイルされた後，そのクラスを JVM にロード・実行することとなる．

また，動的言語の大きな特徴と言える eval 関数の実装自体が Clojure 処理系全体を含んでいると言うこともできる．たとえば：

```
user=> (+ 1 2)
3
```

のような対話環境上の式の評価と：

```
user=> (eval '(+ 1 2))
3
```

における eval 中の実行はまったく同じである．そこで，ここでは，eval 関数が引数式を評価する手順を追うことにより，Clojure コンパイラの実装の詳細について説明する．

Clojure コンパイラは，5 つのフェーズからなる．以下，式 (eval '(cons 1 [])) の実行（[1] に評価される）を例に，各フェーズで行われる処理について簡単に説明する．

(1) マクロ展開

引数に渡された式がマクロの呼出しであれば，マクロ展開を行い，その結果の式に対し再帰的に eval を呼び出す．特に，結果の式が (do $e_1 \cdots e_n$) であった場合，部分式を左から順に eval の再帰呼出しにより評価し，最後の式 e_n の結果を返す．

cons はマクロではないので，上記の例では手順 (2) に進む．

(2) analyze

マクロ呼出しでなければ，analyze 関数によって抽象構文木（AST）へ変換する．ただし，Clojure コンパイラでは Clojure の各関数定義を一つのクラス定義へとコンパイルするため，eval 関数の引数を無引数関数（サンク）にラップしてから analyze に渡される．また，analyze 関数は自由変数の情報を含む環境を追加の引数として受け取る必要がある．環境の詳細については後述するが，ここでは必ず，単なる空環境を表す (empty-env) が渡される．

上記の例であれば：

```
(analyze
  '(fn [] (cons 1 []))
  (empty-env))
```

が呼び出される．

生成される AST について簡単に説明する．まず，トップレベルのノードは：

```
{:op :fn, :arglists ([]),
 :internal-name "fn__12228",
 :methods <METHODS>,
 :closed-overs {}, :env ..., ...}
```

のようなマップデータ構造となっている（以降，特に断わらない限り，AST を含む Clojure コンパイラが用いる中間表現の各ノードはマップである）．先程述べたとおり，評価される式全体がサンクで囲われており，:op や:arglists からそのサンクを表す fn 式のノードとなっていることが分かる．また，:internal-name は Clojure コンパイラによって自動生成されたユニーク ID であり，最終

[*5] 完全に 100%互換ではないらしいが，OPAL/Clj に関する限りまったく同じである．

的にはこの fn 式に対応するクラス定義の名前となる．:closed-overs，:env は自由変数に関するフィールドであるが，これについては次節で詳しく述べる．

<METHODS>には関数本体式に対応する AST が含まれている[*6]．

```
{:op :fn-method,
 :params [], :env ...,
 :body {:op :do,
        :statements [],
        :ret {:op :invoke,
              :fn {:op :var,
                   :var #'clojure.core/cons
                        , ...},
              :args [...]}}, ...}
```

:body フィールドの下に（do で囲まれた）cons 呼出し式の AST が含まれていることが見てとれる．

以上のように，このフェーズの出力である AST においては，関数等の Clojure 言語における概念はまだ直接的に表現されており，Java のオブジェクト指向モデルとは独立している．

(3) emit-classes

analyze の出力結果である AST を引数に：

```
(emit-classes ast {})
```

を呼ぶことで，Clojure コンパイラは Java のクラス定義を生成する．ただし，JVM 上に直接読み込むことの可能な形式ではなく，詳細をある程度抽象化した中間表現（以降，クラス AST と呼ぶ）を出力する．

また，eval 対象のコードの内部に入れ子の fn 式が含まれる場合，別個のクラス AST へと変換されるため，emit-classes は正確には複数のクラス AST のベクタを返す．一番外側の fn 式（サンク）に対応するクラス AST が必ずベクタの末尾に格納されることになっている．

クラス AST がどのような構造となっているかの簡単な例として，(eval '(cons 1 [])) の場合のトップレベルのサンクに対応するクラス AST を以下に示す：

```
{:op :class, :name "fn__12228",
 :super :clojure.lang.AFunction,
 :interfaces nil,
 :fields ({:op :field,
           :attr #{:public :static :final},
           :name "const__1",
           :tag java.lang.Long}...),
 :methods (<METHOD>.. )}
```

AST 中の各 fn ノードに対応し，同じ名前のクラスが定義されている．AFunction は Clojure ランタイム中の関数を表現する抽象クラスであり，各関数はそれを実装するサブクラスとなっている．

:fields 中の static フィールドには，主にコード中のリテラル値を格納するために用いられ，クラス初期化時に適切な値に初期化される．たとえば，上の clas クラス AST 中の const__1 は，cons 呼出しの第 1 引数に書かれている整数リテラル 1 のためのフィールドである．さらに，:fields には static ではないフィールドが含まれていることもあるが，その用途については次節で説明する．

:methods にはクラス AST におけるメソッド表現が含まれている．たとえば (cons 1 []) 呼出しを含む関数本体は，fn__12228 クラスの invoke メソッドとして次のように表現される（バイトコード列は紙面の都合により少し簡単化している）：

```
{:op :method,
 :method [[:invoke] java.lang.Object],
 :code
 [...
  [:get-static "fn_12228" "const__0"]
  [:invoke-virtual [:Var/getRawRoot] :
      Object]
  [:check-cast :IFn]
  [:get-static "fn__12228" "const__1"]
  [:get-static "fn__12228" "const__2"]
  [:invoke-interface
   [:IFn/invoke :Object :Object]
   :Object]
  [:return-value]]}
```

const__0 に含まれているるトップレベル変数オブジェクトから cons 関数を取り出し，:invoke-interface 命令によって呼び出している．このように，JVM のバイトコードにかなり近

[*6] Clojure では引数シグネチャに応じたオーバロード定義が可能であるため，一般には複数の本体が存在するが，本論文では簡単のため一つだけ含まれているものとする．

い表現へ変換されていることが分かる．

上記の変換は，AST の各ノードをクラス AST の対応するノードへと変換する Clojure のマルチメソッド emit により実装されている．たとえば，AST 中の:invok ノードの変換は次のようなメソッド定義により処理される：

```
(defn emit-args-and-invoke
  ([args {:keys [to-clear?] :as frame}]
   (let [frame (dissoc frame :to-clear?)]
     `[~@(mapcat #(emit % frame) (take 19
          args))
       ~@(when-let [args (seq (drop 19
           args))]
          (emit-as-array args frame))
       [:invoke-interface [:IFn/invoke
          ... :Object)] :Object]])))

(defmethod -emit :invoke
  [{:keys [fn args env ...]} frame]
  `[~@(emit fn frame)
    [:check-cast :IFn]
    ~@(emit-args-and-invoke args ...)])
```

:fn，:args フィールドに含まれる AST から再帰的にコードを emit し，それらの結果がオペランドスタックに積まれた状態で:invoke-interface 命令を実行するようなコードが生成されている．(20 番目以降の引数は 1 つの配列オブジェクトにして渡している．)

他の種類の AST ノードについても同様のメソッド定義により処理されており，全体として，いわゆる Visitor パターンのように木変換処理が行われる．

(4) compile-and-load

手順 (3) で生成された全てのクラス AST を，ASM[9] ライブラリを使って本物の JVM バイトコード表現へと変換し，それらを Java 標準のクラスロード機能（Class#defineClass()）を使って JVM 上にロードする．クラス AST から本物のバイトコードへの変換は，一般のアセンブリ言語のように，ほぼ一対一対応の素直な変換となっている．

(5) 関数オブジェクト生成＆起動

Java のリフレクション機能を用いる以下の Clojure コード：

```
public final class fn__12367 extends AFunction {
  public Object invoke(Object f) {
    return new fn__12368(f);
  }

  class fn__12368 extends AFunction {
    Object f;

    public fn__12368(Object obj) {
      f = obj;
    }

    public Object invoke(Object x) {
      return ((IFn)f).invoke(x);
    }
  }
}
```

図 4 関数閉包のためのクラス定義（一部簡略化）

```
((.newInstance ^Class (last classes)))
```

により，サンクに相当する関数クラスのインスタンスを生成し，即座に呼び出す．classes には手順 (4) でロードされた全ての Class オブジェクトが含まれており，末尾はサンクである．このようにして呼出したサンクの返り値がそのまま eval の評価結果となる．

なお，4 節で詳しく述べるが，OPAL/Clj では，以上の 5 つのフェーズのうち主に (2) と (3) に対し修正を行っている．

3.2 関数閉包

OPAL/Clj の fn&a 式の実装では，通常の Clojure 処理系とは異なる方法で変数参照（とくに関数閉包中における自由変数参照）が扱われている．その理由や具体的な方法については次節で詳しく説明するが，まず，本節では，通常の Clojure 処理系における関数閉包（クロージャ）の実装方法，特に（自由変数を含む）変数参照がどのように実現されているのかについて説明する．

まず，簡単な関数閉包を例に考えてみる．式 (fn [f] (fn [x] (f x))) において，内側の fn 式は自由変数 f への参照を含む関数閉包オブジェクトへ評価される．

Clojure コンパイラが生成するクラス定義を**図 4** に示す．本当はバイトコードを出力しているが，

図では可読性のため同等の Java コードにより表現している．

外側の fn 式に対応するクラスが fn__12367 であり，内側の fn 式に対応するクラスはその内部クラス fn__12368 として定義されている．後者には，自由変数 f の値を格納するためのフィールドがあり，外側の関数の invoke メソッド（そこでは束縛変数 f はメソッド引数である）から呼ばれるコンストラクタによって初期化される．内側の関数の invoke メソッド本体では，このフィールド f にアクセスすることによって間接的に自由変数参照を実現している．

上記のようなバイトコードを出力するため，Clojure コンパイラの各フェーズではそれぞれの中間表現中に適切な情報を含めている．

まず，AST では外側の fn 式は次のノード：

```
{:op :fn, :arglists ([f]),
 :internal-name "fn__12367",
 :closed-overs {}, :env {:local {},
     ...}, ...}
```

により表現されるが，:env フィールド（に含まれるマップの:local フィールド）には fn 式を取り囲むレキシカル環境，:closed-overs フィールドには fn 式本体から参照される全ての自由変数の情報がそれぞれ含まれている．この場合，どちらも空であるが，内側の fj 式に対応するノード：

```
{:op :fn, :arglists ([x]),
 :internal-name "fn__12368",
 :closed-overs {f__#0 {:op :local,
     ...}},
 :env {:local {f {:name f__#0, :arg-id
     0, ...}},
       ...},
 ...}
```

のように自由変数 f に関する情報がどちらにも含まれていることが分かる．f__#0 はコンパイラが自動生成したユニークな名前であり，また，:arg-id *i* はその束縛変数が *i* 番目の引数による束縛であることを表している．

このような AST を出力するため，analyze 関数の第 2 引数は，変数束縛に関する情報を含むパラメータ env を受け取るようになっている．eval 関数では引数として:locals フィールドが空マップである (empty-env) を使って analyze 関数を呼び出す．

クラス AST を生成する emit-class 関数についても，基本的には analyze 関数と同様，変数束縛に関する情報を含むパラメータ frame を管理しながら AST を再帰的に処理する．eval 中から呼び出される際には，frame には空のマップが渡される．

以上のようなコンパイル手順を経ることにより，Clojure の通常の eval 関数ではその呼出しの外側の文脈中の変数を対象コード中からは参照できない点に注意して欲しい．たとえば：

```
(let [f (fn [x] x)] (eval '(fn [x] (f x)
    )))
```

を実行するとコンパイラ時エラー「Unable to resolve symbol: f in this context」となる．

4. OPAL/Clj の実装

4.1 全体構成

OPAL/Clj の実装は主に，OPAL 側で具象値を適切に用いるための抽象ドメインである ConcreteValue クラスの定義（図 2 の上から 2 番目の層）と，Clojure 側での具象解釈を要求するためのバイトコードを生成する層（図 2 の下から 2 番目の層）からなる．前者については，OPAL の拡張可能な抽象ドメイン機構を用いることで，OPAL フレームワークに備わっている抽象値クラスを継承するクラスとして実現できる（ConcreteValue クラスの詳細については [13] を参照）．

後者については，Clojure コンパイラの内部に直接修正を行っている．その他にも，OPAL/Clj の 2 つの層には，fn&a 構文を実現するための Clojure マクロ定義，コールバックコードを OPAL 中で適切に処理するための専用の抽象解釈モード等が含まれている．

4.2 抽象解釈用コードの生成

ここでは以下の簡単なコード：

```
(fn&a [] (* ^:conc (+ 1 2) 3))
```

PC	Instruction	Stack
0	INVOKESTATIC OPALClj#D	
3	IFNULL 24	{ "DUMMY": String }
6	NEW fn__13681	
9	DUP	{ false: Boolean }
10	INVOKESPECIAL fn__13681#<init>	{ false: Boolean } { false: Boolean }
13	CHECKCAST IFn	{_: fn__13681 }
16	INVOKEINTERFACE IFn#invoke	{_: fn__13681 }
21	GOTO 31	{ 3: Long }
24	LCONS_1	n/a
25	LLOAD 2	n/a
28	INVOKESTATIC Numbers#add	n/a
31	LLOAD 3	{ 3: Long }
34	INVOKESTATIC Numbers#mult	LongSet(3) { 3: Long }
37	LRETURN	ALongValue

(a) 具象解釈のコールバックを含むバイトコード

```
.class public final super fn__13681
.super clojure/lang/AFunction
.implements clojure/lang/IFn

.method public <init> : ()V
L0:    aload_0
L1:    invokespecial clojure/lang/AFunction <init> ()V
L4:    return
.end method

.method public invoke : ()Ljava/lang/Object
L0:    lconst_1
L1:    ldc2_w 2L
L4:    invokestatic clojure/lang/Numbers add (JJ)J
L7:    invokestatic clojure/lang/RT box (J)Ljava/lang/Number;
L10:   areturn
.end method
.end class
```

(b) ^:conc 部分式評価のためのサンク（一部簡略化）

図 5 具象評価の実装

を例に，抽象解釈中の具象解釈をどのように実現しているか，つまり，^:conc アノテーションに対してどのようなバイトコードを生成するを簡単に説明する．

OPAL/Clj 用に修正した Clojure コンパイラは上記の Clojure コードに対し，**図 5** (a) のようなバイトコードを生成する[*7]．さらに，^:conc アノテーションのついた各部分式に対し，図 5 (b) のようなサンク（無引数関数）の定義が生成される．たとえば，この例では (fn [] (+ 1 2))のサンク（fn__13681 クラス）が定義される．

[*7] OPAL/Clj による解析結果も一緒に示しておく．ただし，レジスタには this 関数オブジェクトが 0 番に格納されているだけなため省略している．

式 (+ 1 2)を普通にコンパイルした場合のコードが命令 24–28 に生成されている．一方，上述のサンクを生成し，その invoke メソッドを呼び出すコードが命令 6–21 に生成されており，この部分が抽象解釈中の具象解釈を実現している．

これら二つの命令列の実行は，命令 0–3 の条件判定によって切り替えわる．スタティック変数 OPALClj#D の値が null かどうかを調べているが，その値は常に null としているため，この fn&a 式を Clojure 側で評価する際には，必ず命令 24–28 が実行されることになる．一方，OPAL/Clj の抽象解釈中ではこのスタティック変数を特別に扱い，文字列"DUMMY"が含まれているかのように振る舞う．そのため，抽象解釈中には必ず命令 6–21 が実行される（命令 24–28 のオペランドスタックが n/a となっているのはそのため）．

サンクを呼び出す命令列を参照しながら OPAL 側からサンクを実際に呼び出すのは，Java (Scala) のリフレクション機能を使えば難しくない．ただ，リフレクションではオブジェクトの生成とコンストラクタの呼出しは一つの API にまとめられており，バイトコードの命令 6 と命令 10 のように分かれていない．そのため，OPAL/Clj は命令 6 の NEW の実行では仮に適当な抽象値（ここでは false）をスタックに積んでおき，命令 10 の INVOKESPECIAL の実行によって，リフレクションで生成した fn__13681 オブジェクトと置き換えている．

以上のような抽象解釈用バイトコードを生成するため，OPAL/Clj では主にコンパイラの 2 つのフェーズに修正を行っている．以下，順に説明する．

(1) analyze

通常の Clojure コンパイラの analyze 関数は，各ノード毎の処理を実現するために内部でマルチメソッド-analyze-form を呼び出している．このマルチメソッドを上書きしてやることにより，任意のフォームに対する追加処理を加えることができる．今回の場合，通常の-analyze-form に処理を移譲することにより AST を生成することに加え，コンパイル対象コードである引数 form に:conc メタデータが付いているなら：

```
(-analyze-form
  `((fn* [] ~(vary-meta form
                        dissoc :conc)))
          env)
```

を呼び出すことにより，コールバック用サンクの生成&呼出しを行う AST を生成する．このようにうして生成した AST は，先程述べた form 本来の AST のルートノードに，:conc-fn フィールドとして保存しておく．

(2) emit-classes

emit-classes 関数中のコードに直接修正を施し，:conc-fn フィールドを含む全ての AST ノードに対し次の追加処理を行う．まず，(1) の analyze 関数が:conc-fn フィールドに格納した AST を取り出し，それを通常の emit-classes 関数によってクラス AST へ変換する．また，AST ノード自身も通常の方法によりクラス AST へ変換する．それぞれにより生成されたバイトコードを callback-bytecode, bytecode とし，最終的なバイトコードは：

```
`[[:invoke-static [:opalclj.ClojureAnalyzer/DUMMY]
                  :java.lang.Object]
  [:if-null ~null-label]
  ~@callback-bytecode
  [:go-to ~end-label]
  [:mark ~null-label]
  ~@bytecode
  [:mark ~end-label]]
```

とすることで，前述の DUMMY フィールドの値を条件とする分岐コードを生成する．

4.3 OPAL 側の抽象解釈

3 節で述べたとおり，標準の eval 関数ではコードを評価するために，関数定義に相当するクラス定義を基に，インスタンスの生成とその invoke メソッドの起動までを行っていた．一方，OPAL/Clj の fn&a 式では，生成されたクラス定義をそのまま OPAL に渡して抽象解釈を行う処理が追加される．また，invoke メソッドの起動は行わず，生成した関数クラスのインスタンスを，抽象解釈結果をメ

タデータとして付加した上でそのまま返す．

上記の処理を実現するため，OPAL/Clj では fn&a は次のようなマクロとして実装されている．（下記コードは，関数閉包の扱いに関する処理を省略した簡易版である．完全版は次節で示す．）

```
(defn fn&a* [params body]
  (let [pspecs (mapv meta params)
        form `(fn ~(vec params) ~body)]
    `(let [fn-cls# (gen-bytecode '~form)]
       [(fn [] (invoke-const fn-cls#))
        (fn [this#]
          (run-opal this#
                    (.getName fn-cls#)
                    ~pspecs))])))

(defmacro fn&a [[& params] & body]
  (let [code-gen (fn&a* params body)]
    `(let [[ctor# analyzer#] ~code-gen
           fn-obj# (ctor#)
           result# (analyzer# fn-obj#)]
       (with-meta fn-obj# result#))))
```

gen-bytecode 関数が eval 関数に相当し，引数に受け取ったサンクをコンパイルし，対応するクラスを返す．

run-opal 関数は，Scala で書かれた OPAL を利用するコード（図 2 の上から 2 番目）を呼び出して抽象解釈を行う関数である．解析結果がマップとして返される．抽象解釈中の具象解釈に必要となるため，関数オブジェクト（this#）自身も引数として渡されていることに注意して欲しい．invoke-const 関数で生成した関数オブジェクトは run-opal を呼び出した先の OPAL 側での処理については，[13] にもう少し詳しく書かれている．

4.4 関数閉包の拡張

2.3 節で取り上げた：

```
(fn&a []
  ((^:conc (fn [x]
             (fn&a [] ^:conc x))
           3.14)))
```

のようなコードを前節の fn&a マクロ定義に従って展開すると：

```
(... (gen-bytecode
      '(fn []
         ...
         (fn [x]
           ...
           (gen-bytecode '(fn [] x))
           ...))) ...)] ...)
```

のような入れ子の gen-bytecode 関数呼出しコードへと展開される．しかしながら，このコードはコンパイル時エラーとなる．先述のとおり，OPAL/Clj の gen-bytecode 関数は標準の eval 関数相当の処理を行い，内側の gen-bytecode 関数に引数として与えているコード'(fn [] x) 中の変数 x への参照が未束縛となるためである．

この場合，OPAL/Clj における「具象解釈中の変数束縛を内側の fn&a 式から参照する」を実現するためには，上の展開後コードにおける外側の gen-bytecode 呼出しに引数として与えられているコード中でつくられている x の変数束縛へと名前解決されることが望ましい．

それを実現する完全な fn&a マクロを**図 6** に示す．

まず，gen-bytecode 関数は追加の引数 env を受け取るように変更する．gen-bytecode はそれを使って analyze 関数を呼び出す（analyze 関数は元々 env を第 2 引数として受け取るようになっていたが，eval ではただ空環境を渡していた）．gen-bytecode 関数に渡す環境データは，Clojure のマクロの機能により生成する．具体的には，マクロ定義中で特殊変数&env を参照することでマクロを呼び出している文脈における環境を取得できる．実際には，Clojure コンパイラが要求するフォーマットに変換した new-env を gen-bytecode 関数（さらにその先の analyze 関数）へ渡している．

また，fn&a 式中からの自由変数アクセスがあるため，対応する関数クラスのコンストラクタは自由変数の値を引数に受け取る必要がある．自由変数集合（ベクタ）を求める補助関数 free-vars-fn を使って自由変数（シンボル）のベクタを取得し，invoke-constructor 関数の引数とすることで，fn&a 式を評価する時点での自由変数の値を取得している．

```
(defn fn&a* [params body env]
  (let [gen-bytecode-id (gensym "gb")
        new-env (if (and env (map? env) (every? map? (vals env)))
                  (make-env env)
                  (let [lbs (vals env)
                        locals (zipmap (keys env) (map make-param-binding lbs))]
                    (make-env locals)))
        pspecs (mapv meta params)
        form  (list* 'fn* (vec params) body)
        fvars (free-vars-fn form new-env)
        ]
    [`(let [fn-cls# (gen-bytecode '~gen-bytecode-id '~form '~new-env)]
        [(fn [fvs#] (invoke-constructor fn-cls# fvs#))
         (fn [this#] (run-opal this# (.getName fn-cls#) ~pspecs))])
         fvars]))

(defmacro fn&a [[& params] & body]
  (let [[code-gen fvars] (fn&a* params body &env)]
    `(let [[ctor# analyzer#] ~code-gen
           fn-obj# (ctor# ~fvars)
           ana-result# (analyzer# fn-obj#)]
       (with-meta fn-obj# ana-result#))))
```

図 6 fn&a の実装

さらに，emit-classes 関数にも適切な変数束縛情報を含む frame を渡す必要がある．analyze 関数に渡す env は Clojure のマクロ機能を使って生成できる一方で，emit-classes 関数の frame はコンパイラ内部表現間のデータ表現であるため，Clojure のマクロ機能によって取得可能な環境情報だけでは不十分であり，簡単には生成できない．

そこで，OPAL/Clj では次のようにして gen-bytecode 関数の呼出しフォームが書かれている位置における frame 情報を内側のコードへ伝搬する．まず，gen-bytecode 関数の呼出し式フォーム毎にユニークな ID を生成し（図 6 の 2 行目），gen-bytecode 呼出しの追加引数とする（12 行目）．次に，emit-class から呼出されるマルチメソッド -emit が各関数呼出し式を処理する際，呼出されるメソッドの名前が gen-bytecode であれば，その時点の frame（-emit に引数として渡されている）を生成したユニーク ID をキーにしてグローバルなマップ *gen-bytecode-contexts* に保存する．最後に，gen-bytecode 関数中から emit-classes 関数を呼び出す際，従来であれば空の frame 情報を渡していたところを，*gen-bytecode-contexts* を検索して保存していた frame 情報が見つかれば，それを引数として呼び出すことで，外側の変数束縛情報を伝搬することができる．見つからなければ，従来どおりに空の frame 情報を渡す．

5. まとめ

本論文では，Clojure のための抽象解釈器 OPAL/Clj について，その実装方法に重点を置きつつ紹介した．OPAL/Clj は Clojure コンパイラによって生成された JVM バイトコードを既存の抽象解釈フレームワーク OPAL を使って解析する．OPAL の通常の方法では解析精度が失われる箇所について，具象解釈を適宜織り交ぜることを可能にすることにより，ユーザは柔軟に解析精度をコントロールすることができる．

今後の課題としては，動的な def の使用による変数の再定義の扱い，Clojure マクロ定義を適切に処理する方法の検討，アノテーションを自動的に付与する方法の検討などが挙げられる．後者の二つについては部分評価やマルチステージプログラミングにおける関連する先行研究が参考になると考えている．また，それ以外に，インライン関数や各種プリミティブ型などに関する処理の実装を完成させ，より現実的なコードに対しても利用可

能にすることが挙げられる.

謝辞 本研究は JSPS 科研費 JP16K00096 の助成を受けたものである.

参考文献

[1] Cousot, P. and Cousot, R.: Systematic Design of Program Analysis Frameworks, *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pp. 269–282 (1979).

[2] Darais, D., Labich, N., Nguyen, P. C. and Van Horn, D.: Abstracting Definitional Interpreters (Functional Pearl), *Proc. ACM Program. Lang.*, Vol. 1, No. ICFP, pp. 12:1–12:25 (2017).

[3] Eichberg, M. and Hermann, B.: A Software Product Line for Static Analyses: The OPAL Framework, *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pp. 1–6 (2014).

[4] Hickey, R.: Clojure Homepage, `https://clojure.org`.

[5] Lai, X., Luo, Z., Ali, K., Lhoták, O., Dolby, J. and Tip, F.: Evaluating Call Graph Construction for JVM-hosted Language Implementations, David R. Cheriton School of Computer Science Technical Report CS-2015-03 (2015).

[6] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A.: The Java Virtual Machine Specification, `https://docs.oracle.com/javase/specs/jvms/se11/html/index.html`.

[7] Might, M.: Abstract Interpreters for Free, *Proceedings of the 17th International Static Analysis Symposium*, SAS 2010, pp. 407–421 (2010).

[8] Mometto, N., Hickey, R. and contributors: tools.emitter.jvm (GitHub page), `https://github.com/clojure/tools.emitter.jvm/`.

[9] OW2 Consortium: ASM - Homepage, `http://asm.ow2.org`.

[10] Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D. and Piessens, F.: Monadic Abstract Interpreters, *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pp. 399–410 (2013).

[11] The OPAL Project: BugPicker, `http://www.opal-project.de/tools/bugpicker/index.php`.

[12] The OPAL Project: OPAL, `http://www.opal-project.de`.

[13] 馬谷誠二: JVM 上の動的言語のための抽象解釈, 情報処理学会 第 121 回プログラミング研究発表会 (2018).