一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

Javaへの所有権システム移植によるRust学習支援

竹重　拓輝†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{h-takesg,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　メモリ安全かつ処理が高速なソフトウェアを実現するプログラミング言語として Rust が注目されている．Rust はコンパイラがメモリの参照を検査し安全性を保証し，また適切なタイミングでメモリ解放処理を挿入しメモリ使用を効率化している．これら安全性と高効率性は言語設計に所有権という概念を取り入れ実現されており，所有権の理解は Rust の特長を利用する上で重要である．一方で所有権は多くのプログラミング言語には存在せず，Rust 学習者にとってその習得を妨げる大きな要因となっている．本研究では他言語既習者の Rust 学習支援を目的とし，他言語に所有権概念を移植し所有権概念の学習環境を提供する．所有権概念の学習を Rust の文法や開発環境の学習と切り離し，既習言語で習得できるようにし，学習効率の向上を目指す．

キーワード　Rust，学習支援，機能移植，所有権，ライフタイム

1. はじめに

近年高パフォーマンスかつ高信頼性を実現するプログラミング言語として Rust が注目されている．Rust はコンパイラがプログラム内のメモリ操作の安全性を保証するという特徴を持つ．この特徴によりガベージコレクションを不要とし，高速な処理を実現している．パフォーマンスを犠牲にせずとも安全なソフトウェアを開発できる点が評価され，発表から 10 年程度という比較的新しい言語ながら大規模プロジェクトでも採用されつつあり，Linux カーネルの開発においても C 言語に加え第 2 の言語として導入されている[1]．

一方で Rust 普及を妨げる要因として，言語習得の難しさが挙げられている [1]．コンパイラによる厳格な検査によりメモリ操作の安全性を保証しており，その検査において重要となる所有権とライフタイムという要素の理解習得が難しいとされている [2]．このため，パフォーマンスや信頼性の高さといった利点があるにもかかわらず，開発言語として採用されにくいと考えられる．

本研究の目的は Rust の言語移行における学習支援である．既になんらかの言語を習得している開発者が効率的に Rust に移行できるようにする．本研究では文法と所有権の学習の分離により Rust の学習を支援する．新たにプログラミング言語を習得するには，まずその言語の文法を理解する必要があるが，Rust の場合はさらに所有権といった独自の概念も学ぶ必要がある．これらを同時に学ぶことは容易ではないため，既に文法を習得している言語を利用して所有権の概念を学習できるようにする．それぞれの要素を個別に学ぶことで、学習の複雑さを低減し，より効率的な言語移行を可能とする．

この実現のため，所有権を Java に移植する．Java 既習者に対して所有権を考慮したプログラミングが必要な環境を提供し，所有権に対する理解を深めさせる．この環境では，Java では問題ないコードに対しても Rust における所有権に関連するエラーと同様のエラーが発生する．このエラー修正を通じて所有権に対する理解を深めさせる．また，そのエラーメッセージでは提案環境で得られた知見を Rust で活かしやすいよう，Rust のエラーと対応づけ，Java から Rust への知識の転用を促進する。

移植による再現度を確かめるため，Java のコードに対する適用実験を実施した．その結果 8 件中 3 件で再現ができており，残りのコードに対しても実装の進展により対応可能だと考える．

2. 準　　備

コンパイラがメモリ操作の安全性を検証するため，Rust では所有権という概念が採用されている．ソースコード中で操作する各データに対して所有権を持つただ一つの変数をそれぞれ設定し，この変数またはこの変数から貸し出された参照を通じてのみデータへのアクセスを許可する．このアクセス制限により，解放後メモリの参照やデータ競合を防いでいる．

所有権を持つ変数は，他の変数に対し所有権を譲渡できる．このとき，所有権を渡した変数はその時点で無効となり，以降データにはアクセスできなくなる．所有権を受け取った変数が以降の処理においてデータと紐づけられる．

所有権を持つ変数から貸し出される参照には存在条件が設定されている．参照には可変参照と不変参照の 2 種類が存在する．可変参照は参照先のデータの書き換えが可能な参照，不変参照は不可能な読み取り専用の参照である．1 つのデータに対し可変参照は高々 1 つまで，可変参照と不変参照は同時存在禁止といった制限がある．この制限により，不意なデータの書き換えを防いでいる．

(注1)：https://www.kernel.org/doc/html/next/rust/index.html

```rust
fn validateId(id: String) {
    /* some process */
}
fn registerUser(id: String) {
    /* some process */
}

fn main() {
    let userId = String::from("ist2024");

    validateId(userId);
    registerUser(userId); // error!
}
```

```
error[E0382]: use of moved value: `userId`
```

図 1　エラーを含む Rust のコード例

```java
void validateId(@Owner String id) {
    /* some process */
}
void registerUser(@Owner String id) {
    /* some process */
}

void main() {
    @Owner String userId = "ist2024";

    validateId(userId);
    registerUser(userId); // error!
}
```

```
error[E0382]: use of moved value
  Since the variable has lost ownership, the memory area
  referenced by the variable may have already been freed.
```

図 2　エラーを含む Rust のコードを Java で再現した例

また，ガベージコレクションを用いずに安全なメモリ操作を実現するためライフタイムという概念を採用している．Rust において所有権を持つ変数や参照を格納する変数にはそれぞれライフタイムが設定される．このライフタイムは多くの場合変数のスコープに対応する．ライフタイムによって参照の貸し出しは制限されており，所有権を持つ変数より長く生きる変数には参照を貸し出せない．ライフタイムが終了した，すなわち変数のスコープが終了すると，所有権を持つ変数ならその変数のデータを格納していたメモリ領域が解放され，参照を格納する変数であればその参照が消滅し，参照の存在条件の判定から除外される．参照の貸し出し制限により解放後メモリに対する参照を経由したアクセスを防げ，また，ライフタイムを用いたメモリ解放タイミングの管理により不要となったメモリを即時解放できる．

例えば図 1 のコードでは所有権を失った変数の再利用としてエラーが起こる．このコードでは 9 行目で変数 `userId` を宣言している．`"ist2024"`という文字列データの所有権は `userId` が持つ．11 行目で `userId` は関数 `validateId()` の呼び出しに使用される．このとき，`validateId()` はその引数 `id` に所有権を要求する．よって 11 行目の実行時に `userId` が持っていた所有権は関数 `validateId()` の `id` に譲渡される．その後，12 行目で関数 `registerUser()` の呼び出しに `userId` を使用すると，所有権を失った変数の使用としてエラーが発生する．この制限により，所有権を失いライフタイムがメモリの解放と紐づかなくなった変数を通したメモリアクセスを禁止し，不正なメモリ領域へのアクセスを防いでいる．

Rust は高いパフォーマンスと安全性を実現する一方で，開発言語として採用する際の問題点として急な学習曲線が挙げられている [1]．その大きな原因として所有権とライフタイムが挙げられている [2]．所有権は他のメジャーな言語では導入されておらず，多くの開発者にとって馴染みのない概念である．よって Rust 習得には通常の言語習得プロセスと並行して所有権概念の習得が必要であり，他の言語と比べて習得難度は高いと言える．

3. 提案手法

3.1 概　　要

提案手法では Java 経験者を対象として Rust 言語の習得を支援する．一般に既に何らかの言語を習得しているプログラミング経験者は，新たな言語の学習において既習概念を基にした類推によって効率的に言語を学習する．例えば言語によらない共通の概念として繰り返しのような構造や，パターンマッチングなどがある．これらの概念について既に習得済みの開発者は学習中の言語のにおいてそれら概念がどのように記述されるかのみに集中して学べる．

Rust 学習者に既習言語が存在する場合，先にその言語で所有権概念のみ習得できれば，通常の言語移行と同様に，既習概念を基にした類推によって学習を促進できる．よって所有権概念の他言語への移植により，当該言語既習者が所有権概念のみを習得できる環境を作成する．

提案手法を導入した Java コードを 図 2 に示す．このコードは 図 1 の Rust コードを Java で再現したコードである．Annotation によって変数の役割を示しており，`@Owner` は所有権を持つ変数である．このコードをコンパイルすると，13 行目において所有権を失った変数を使用しているとしてエラーが発生する．

エラーは 15,16 行目の内容である．15 行目では Rust と同様のエラーメッセージを表示し，Rust のエラーと対応づけている．16 行目ではエラー内容の説明として，なぜそれが禁止されているのか説明している．このコードにおいては，`userId` が所有権を失っているため，参照先のメモリが解放されている可能性がある旨を説明している．

上記エラーから，開発者は所有権を考慮したコード作成方法を学ぶ．エラー発生箇所の確認により，所有権により制限さ

れる操作や，その操作がソースコード上のどこで起こりやすいのか理解を深められる．また，エラーに付与される説明から所有権による操作制限の意図を理解し，より容易にエラー解消するようソースコードを修正できる．

このコードの場合，まずメソッド呼び出しに対してエラーが発生していることからメソッド呼び出しでは所有権を意識しなければならないことがわかる．また，エラーの原因が所有権を失った変数の再利用であることから所有権を保持するように書き換える必要があることがわかる．以上の理解から，このメソッド呼び出し時点において所有権を保持するために，所有権の譲渡が発生する箇所を探す．このコードでは 12 行目 `validateId()` の呼び出しが該当する．よって，この `validateId()` が引数の所有権を奪わないよう，引数の型宣言を`@Borrow` に変更する．この修正によりエラーは解消される．この経験から開発者はメソッド呼び出しにおける所有権操作の注意点や引数宣言の重要性を学べる．この知見は Rust の学習においても活用でき，その習得を効率化できる．

提案手法は 3 つの要素から構成される．所有権の再現にあたり簡易化し定義された操作制限，所有権に係る操作の Java 内での記述方法，所有権を再現する検査である．以降ではこれらの要素について説明する．

3.2 所有権による操作制限

Rust 所有権の再現にあたり，まず Rust において所有権によって制限される操作を整理する．所有権はソースコード中での操作に制限を加え，メモリ安全性を高める機構である．開発者がこの制限を誤って理解していたとき，コンパイラエラーが多発し Rust の習得を妨げる．よってこの制限を再現する．

ただし，厳密に Rust コンパイラによる検査と同等の処理を実装するには多大な開発コストが必要である．よって，制限を実現する簡易的な制限を定義し Java に実装する．

Rust において所有権によって次のような制約が存在する．

- 参照の存在条件
- ライフタイム制約
- 不変参照から破壊的メソッドの呼び出しの禁止
- 所有権を持たず参照でもない変数の使用禁止

参照の存在条件とライフタイム制約は 2. 章で説明したとおりである．不変参照から破壊的メソッド呼び出しの禁止は不変参照のアクセス権限による制約である．不変参照は参照先の変数に対して変更を加える権限を持たない．よって変数に付帯する破壊的メソッドへの不変参照を経由したアクセスは禁止される．また，Rust において所有権を持たず参照でもない変数の使用は禁止されている．変数が持つ所有権は他の変数への譲渡が可能である．この処理を Rust においてはムーブと呼ぶ．ムーブによって所有権を失った変数は以降の処理において使用が禁止される．

3.3 所有権操作の記述方法

所有権によってソースコード中の操作が制限される箇所は参照の作成や，メソッド呼び出しによる引数の受け渡しである．提案手法では参照の作成を代入文で行うこととする．このとき，所有権による制限は型検査による変数の使用制限と類似した処理と考えられる．よって，型検査の拡張により所有権による操作制限を再現する．

型検査の拡張にあたり，ソースコード中の変数に対し開発者に標準の型に加えて追加の型の宣言を求める．追加の型は 3 種類あり，所有権を持つ変数 `Owner`，可変参照である `MutBorrow`，不変参照である `Borrow` である．開発者は変数に対していずれかの型を追加で宣言する．宣言がない変数がソースコード中で使用された場合は警告を発生させる．エラーではなく警告とした理由は，所有権を考慮した型検査の適用範囲を段階的に広げていくような導入を可能とするためである．これにより，開発者はソースコードの一部分のみを対象として学習を始められ，その理解に応じて対象とする範囲を広げられる．最終的には警告も発生させないよう，ソースコード全体の変数に対し追加の型を宣言し，かつ所有権に係るエラーも発生させないことを目指す．

追加の型が宣言された変数同士の代入処理において，ムーブや借用を表現する．例えば `Owner` 同士の代入であればムーブと解釈し，`Borrow` に対して `Owner` を代入する場合，不変参照の借用とする．

また，メソッド呼び出しにおいても実引数が仮引数に代入されていると捉えムーブや借用を処理する．仮引数に対する追加の型はメソッド宣言時の仮引数の型宣言と併せて記述する．

3.4 検　　査

検査においては 3 つのテーブルを使用する．各変数のライフタイムを記録するテーブル，データ毎の参照の存在状況を記録するテーブル，開発者以外が作成したメソッドに対する追加の型宣言テーブルである．

変数が宣言された際，その変数のスコープをその変数のライフタイムとしてテーブルに記録する．Rust のライフタイムは厳密には変数のスコープと一致しない場合があるが，実装の簡易化のためスコープで代用する．

参照の存在状況を管理するテーブルは検査を進めながら構成する．参照の作成時点で参照先のデータと参照を格納する変数を記録する．テーブルでは各データについて参照を持たない，不変参照を持つ，または可変参照を持つという状態のいずれであるか記録する．また，参照を格納する変数のライフタイムが終了していればその参照を削除できるようにする．

追加の型宣言テーブルは著者らが事前に作成する．メソッドの引数がいずれの追加の型を要求するかはメソッド宣言時に記述する．よって，標準ライブラリなどの開発者以外が作成したメソッドには追加の型は付与されていない．これらのメソッドに対しても所有権を意識した操作を強制するため，追加の型を著者らが事前に定義する．定義にあたってはそのメソッドの処理内容を鑑み，適切であると著者らが考える型を付与する．例えば，引数として受け取った変数を書き換える破壊的メソッドに対しては，書き換えられる引数に `MutBorrow` を付与する．

提案手法によって検査が必要な箇所は変数の使用箇所である．使用とはデータへのアクセスまたは所有権の移動，参照の作成処理を指す．よってこれらの操作が行われる代入文，お

よびメソッドの呼び出し文を検査対象とする．

代入文では右辺から左辺へ所有権のムーブまたは参照の貸し出しを行う．メソッドの呼び出し文ではインスタンスと各引数について，実引数から仮引数への代入と見て，それぞれ代入と同様に処理する．

処理内容は制約違反の検出とテーブルの更新である．制約違反の検出には前述の各テーブルを使用する．例えば，所有権を失った変数を使用した場合，各変数の役割を記録するテーブルの情報からこれを検出する．また，ムーブや参照の貸し出しに応じて参照の存在状況テーブルを更新する．

また，ムーブにより所有権を失った変数は追加の型 `Unusable` で表現する．`Owner` 同士の代入によってムーブが発生したとき，代入元の変数の型 `Owner` を削除し，`Unusable` に付け替える．以降，`Unusable` が付与された変数が使用された場合，エラーを発生させる．

3.5 実　　装

本研究では移植対象の言語として Java を用いる．Java を対象とする理由は 3 つある．広く利用されている言語であるため，Rust が得意とするシステム開発で使用されており移行ユーザーが見込めるため，メモリ管理機構が大幅に異なるためである．

特にメモリ管理については Rust と Java は大きく異なると考える．Java はメモリ管理の手法としてガベージコレクションを採用している．よって，Java 開発者は基本的にメモリの確保や解放を意識的には操作していない．このためメモリ管理に対する考え方が Rust 開発者とは大幅に異なり，Rust の習得にあたり所有権によるメモリ管理に躓きやすいと考えられる．故に，提案手法による所有権学習のメリットが大きいと考えられる．

Java ソースコード中において所有権操作の記述には Annotation を使う．Annotation はソースコード中に記述できる，Java コンパイラに対する追加処理の指示である．Java 標準機能であり，例えば `@Override` はメソッドのオーバーライドを必須とする指示として広く使用されている．

Annotation を使う理由は Java 標準機能であるためである．提案手法において独自の記法を導入すると，開発者は所有権概念を学ぶためにその記法を習得しなければならない．これは記法の習得と所有権の習得を分離し，所有権のみを学ばせるという目的を達成できない．よって標準機能であり，一般の開発においても使用されている Annotation によって記述させることで記法の習得を容易にする．

実装には The Checker Framework（CF）を使う．CF は Annotation を用いて型システムを拡張し，代入やメソッド呼び出しにおいて追加の検査を実行するフレームワークである．この検査部分で所有権による制限を再現する．

4. 適 用 実 験

提案手法が Java においてどの程度 Rust の制限を再現できるか検証するため，Java ソースコードを提案手法によって検査する．

```
1 // Returns the n-th largest element in a slice
2 fn find_nth<T: Ord + Clone>(elems: &[T], n:
      usize) -> T {
3     elems.sort();   // error!
4     let t = &elems[n];
5     return t.clone();
6 }
```

```
error[E0596]: cannot borrow `*elems` as mutable,
              as it is behind a `&` reference
```

図 3　Ownership Inventory の find_nth

4.1 題　　材

対象とするソースコードは Crichton らが作成した Ownership Inventory [3] を元に作成する．Ownership Inventory は所有権を学ぶ際に発生しやすい誤解を明らかにするために作成された Rust のソースコード集である．各ソースコードは所有権に対する誤解を原因としたエラーを含んでおり，このエラーに対する対処を開発者に問うことで開発者の理解度を測るために使用された．

例えば，Ownership Inventory に含まれる 図 3 のコードは 3 行目においてエラーが発生する．仮引数の `elems` は配列の不変参照として受け取られる．3 行目では `elems` から `sort()` を呼び出している．このとき，`sort()` は呼び出し元のインスタンスについて可変参照を要求する．これは `sort()` はその実行において元のインスタンスに対して変更を加える破壊的メソッドであるためである．よって不変参照からは `sort()` は実行できないためエラーが発生する．

本研究では Ownership Inventory を所有権に起因する代表的な Rust のコンパイラエラーと捉え，そのエラーをどれだけ再現できるか検証する．Ownership Inventory は所有権に対する誤解を基に設計されている．よってそのコードとエラーを Java 上で再現できれば，Java 上でも所有権について理解を深められると考える．

4.2 適　　用

Ownership Inventory のコードを対象として，提案手法が Java 上でエラーを再現できるか検証した．その結果，8 件のソースコードのうち，3 件でエラーを再現できた． 表 1 にソースコードの特徴と再現の可否をまとめる．

例えば 図 4 は 図 3 のコードを Java で再現したコードである．このコードをコンパイルすると 3 行目において，不正なメソッド呼び出しだとしてエラーが発生する．このメソッド呼び出しは引数として渡された不変参照の `elems` を引数として呼び出されている．しかし，`Arrays#sort()` は破壊的メソッドであり，その引数に不変参照は使用できない．このような Rust の制約に沿わない変数の使用に対してエラーを発生させる．

一方で残りの 5 件ではエラーを再現できなかった．主な原因はメソッド呼び出しによる参照作成やライフタイム宣言などの機能の未実装である． 図 5 は再現できなかったコードの

表 1　適用実験結果

タイトル	処理内容	エラー原因	再現可否	再現できない理由
make_separator	引数が空文字列であればセパレーターを，そうでなければ引数をそのまま返す	ローカル変数の参照の返却	O	N/A
get_or_default	引数の Option が値を持っていればその値を，そうでなければ空文字列を返す	不変参照から破壊的メソッドの呼び出し	O	N/A
find_nth	引数の配列から昇順で n 番目の値を返す	不変参照から破壊的メソッドの呼び出し	O	N/A
remove_zeros	引数の配列から 0 を取り除く	参照の存在ルール違反	X	メソッド呼び出しによって生成される参照は管理できない
get_curve	値が指定されていれば配列にその値を加算し，指定されていなければ何もしない	1 つの構造体にフィールドを通して不変参照と可変参照を作る	X	this の所有権を管理できない
reverse	与えられた配列の逆順の配列を返す	同じ配列に対して複数の参照を生成している	X	メソッド引数での参照作成が未実装
concat_all	イテレータで与えられた配列の各文字列に指定の別の文字列を追記する	返り値のライフタイムが解析不能なため指定が必要	X	ライフタイムの宣言が未実装
add_displayable	値をヒープに格納し，ポインタを配列に格納する	ヒープに格納する変数のライフタイムが不明	X	ライフタイムの宣言が未実装

```
1  @Owner String findNth(@Owner String @Borrow[]
       elems, @Owner int n) {
2      Arrays.sort(elems); // error!
3      @Borrow String t = elems[n];
4      return t;
5  }
```

```
error[E0596]: cannot borrow `elems` as mutable
Arrays#sort() is a destructive method,
requiring @MutBorrow, whereas elems is immutable
as it is @Borrow, and thus cannot be modified.
```

図 4　Java で再現した Ownership Inventory の find_nth

```
1  /// Removes all the zeros in-place from a
       vector of integers.
2  fn remove_zeros(v: &mut Vec<i32>) {
3      for (i, t) in v.iter().enumerate().rev() {
4          if *t == 0 {
5              v.remove(i);    // error!
6          }
7      }
8  }
```

```
error[E0502]: cannot borrow `*v` as mutable
              because it is also borrowed as immutable
```

図 5　Ownership Inventory の remove_zeros

1 つである．このコードでは 3 行目において iter() を用い，イテレータとして v の不変参照を作成している．その後，5 行目で v から要素を削除するため remove() を呼び出している．このとき，remove() はインスタンスに対して可変参照を要求する．よって 5 行目において不変参照と可変参照が同時に存在するため，エラーが発生する．

このコードではメソッドを用い，イテレータとして不変参照を作成している．メソッドを用いた参照の作成には現段階の実装では対応していない．よってエラーの再現ができていない．しかし，メソッド呼び出しにおける検査処理の追加によりこの点は対応可能であり，提案手法の妥当性には影響はないと考える．

5. 議　　論

5.1 手法の制約

制限の定義において，簡易化するために考慮しなかった Rust の機能がある．例えば Rust の機能としてトレイトが存在する．このトレイトの使用により，ライフタイムや借用の挙動が変化する場合がある．この機能は Java には存在しないため，提案手法においてはその影響は考慮していない．提案手法は既習言語上での学習環境の提供であるため，このように既習言語に存在しない Rust 機能の再現には一定の制約がある．

また，開発者以外が定義したメソッドを含むコードに対する検査には強い制約がある．ソースコード全体に対して所有権を再現した検査を実行するには，事前に使用される全てのメソッドについて追加の型定義を用意しなければならない．このため，多くのライブラリが使用されうる，実アプリケーションに対する検査において完全な所有権の再現は難しい．

しかし，これらの制約によって提案手法の有効性が完全に失われることは無いと考える．提案手法の目的は Rust への言語移行における習得難度の易化である．よって提案手法によって所有権の学習全てを担う必要も無く，所有権の操作において

躓きやすい内容の理解支援であってもその目的は果たせると考える．また，任意のライブラリに対応する必要は無く，練習的な実装において使用されやすい Java 標準 API などのメソッドが用意されていれば十分であると言える．

5.2 効果的なエラーメッセージ

エラーメッセージの内容はエラー修正において重要である．Barik らの報告 [4] によると，エラー修正において開発者はエラーメッセージを情報源として活用しており，また，その読みにくさはエラー修正の難しさに影響するとしている．

提案手法ではエラー修正を支援するため，簡潔かつ原因の解消に役立つようなエラーメッセージを出力する．エラーメッセージにおいてそのエラーの発生原因となる制限とともに，なぜその制限が存在するかを説明する．ソースコード中の操作に対する制限は当該言語における安全性などの理念に基づいて設計されており，その理念の理解が早期の制限の理解に役立つと考える．

また，提案手法によって得られた知見を Rust で活かせるよう，提案手法が出力する各エラーメッセージを Rust コンパイラのエラーと対応づける．エラーメッセージにおいて Rust におけるエラーコードを記し，いずれの Rust コンパイラのエラーと対応するか明示する．さらに，エラー内容の表記についても Rust コンパイラが出力するメッセージと極力同等のメッセージを出力するようにし，エラー解消の経験を Rust コンパイラのエラーと紐付ける．これにより，Rust の使用開始後にコンパイラエラーが発生した際，それは提案手法において得た知見を活かせるのか，どのような修正が有効だったかを容易に判断できる．

5.3 有効性の評価

提案手法の有効性を評価するには被験者実験が必要である．現状で評価している内容はあくまで再現度である．提案手法が目的とする学習支援に対する効果を評価するには，被験者に実際に提案手法を使用させ Rust の習得への貢献を測定する必要がある．

提案手法が前提とする文法などの概念と所有権の分離及び分離による所有権習得の容易化が達成できるか評価する．実際に Java 使用経験のある開発者に提案手法によって所有権を学習させ，その後 Rust 習得までの過程を調査する．提案手法を導入した Java 環境において文法を気にせず所有権のみを学習できたか，Java において得た知見を Rust 習得過程において活用できたかを評価する．

6. 関 連 研 究

Rust の習得難易度を改善しようとする研究が存在する．Crichton らは所有権学習における誤解を整理し，正しく理解するためのコンセプトモデルを作成，コンセプトモデルに基づいた教育法を提案した [3]．所有権学習過程において初学者が何を誤りやすいのかを明確化し，具体的な解決手法を提案した点で大きな貢献である．

Almeida らは静的解析によって Rust ソースコード中における所有権の移動や参照の状態を可視化する手法を考案した [5]．ソースコード中における所有権の状況把握として素直なアプローチであり，所有権操作の理解に貢献すると考えられる．提案手法のエラーメッセージ中に，この手法による可視化を取り入れられるとより理解しやすいエラーメッセージを出力できる可能性がある．

Coblenz らは Rust に対してガベージコレクションに相当する機能を導入し，Rust を簡単に利用できるようにした [6]．参照の存在条件の緩和などメモリ管理の難度が大幅に緩和され，初学者にとって Rust を活用する難易度を下げた．一方で所有権の習得には寄与しておらず，純粋な Rust の利用には更に学習が必要である．我々の提案手法はあくまで純粋な Rust の習得支援を目指す．

7. お わ り に

Rust の学習支援を目的とし，Rust 文法の習得と所有権概念の習得を分離するため，所有権を Java に移植した環境を作成した．この環境において開発者はソースコードを作成し，発生したエラーを修正する経験を通して所有権への理解を深め，Rust での学習を効率化する．

適用実験では題材としてコード 8 件中 3 件のコードで Rust コンパイラのエラーを再現できた．再現できていないコードについても実装の発展により再現可能であると考えている．

実際に Rust 学習において提案手法が効果を発揮するか評価するため，被験者実験を実施予定である．

謝辞 本研究の一部は，JSPS 科研費（JP21H04877, JP20H04166, JP21K18302, JP21K11829）による助成を受けた．

文　　献

[1] K.R. Fulton, A. Chan, D. Votipka, M. Hicks, and M.L. Mazurek, “Benefits and drawbacks of adopting a secure programming language: Rust as a case study,” In Proceedings of Symposium on Usable Privacy and Security, pp.597–616, 2021.

[2] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, “Learning and programming challenges of Rust: A mixed-methods study,” In Proceedings of International Conference on Software Engineering, pp.1269–1281, 2022.

[3] W. Crichton, G. Gray, and S. Krishnamurthi, “A grounded conceptual model for ownership types in Rust,” Journal on Proceedings of the ACM on Programming Languages, vol.7, no.OOPSLA2, pp.1224–1252, 2023.

[4] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, “Do developers read compiler error messages?,” In Proceedings of International Conference on Software Engineering, pp.575–585, 2017.

[5] M. Almeida, G. Cole, K. Du, G. Luo, S. Pan, Y. Pan, K. Qiu, V. Reddy, H. Zhang, Y. Zhu, et al., “Rustviz: Interactively visualizing ownership and borrowing,” In Proceedings of Symposium on Visual Languages and Human-Centric Computing, pp.1–10, 2022.

[6] M. Coblenz, M.L. Mazurek, and M. Hicks, “Garbage collection makes Rust easier to use: A randomized controlled trial of the Bronze garbage collector,” In Proceedings of International Conference on Software Engineering, pp.1021–1032, 2022.