修士学位論文

題目

Rust 学習支援を目的とした Java への所有権システムの移植

指導教員

楠本 真二 教授

報告者

竹重 拓輝

令和 6 年 2 月 1 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 5 年度 修士学位論文

Rust 学習支援を目的とした Java への所有権システムの移植

竹重 拓輝

内容梗概

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 の特徴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．提案環境の有効性を評価するため被験者実験を実施し，所有権理解の促進に寄与することを確認した．

主な用語

Rust，学習支援，機能移植，所有権，ライフタイム

目次

図目次

表目次

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[1] が注目されている [2]．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 [3]．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 [4]．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 カーネルや Android の開発においても導入されている [5, 6].

一方でその普及は進んでおらず，言語の人気を表す TIOBE index では 2024 年 2 月時点で C や C++ が 2 位と 3 位である一方 Rust は 19 位である [7]．Rust の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 [8, 9]．Rust は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 [10, 11]．開発者が言語を選択する際には信頼性よりも開発速度や表現力を重視する [12] とされており，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 への言語移行における学習支援である．すなわち，既に何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上で課題となるのはその言語の文法を理解することである [13]．しかし，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 2 種類の知識を同時に学習することとなり容易ではないと考えられる．よって，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．

この目的の実現のため，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，提案環境から Rust への知識の転用を促進する．

提案手法による学習支援効果を評価するため，被験者実験を行った．その結果 Rust を使用して学習した被験者と比較し，提案手法による学習を行った被験者がより高い成績を獲得した．

以降，2 節では所有権について例を示しながら説明し，3 節ではこの所有権を他言語に移植する提案手法について述べる．4 節では Java 上での Rust 所有権の再現度評価のため実施した適用実験について

述べ，5 節では学習支援効果を評価するため実施した被験者実験とその結果に対する考察について述べる．6 節では本研究における妥当性の脅威について述べ，7 節では提案手法の制約やその発展について議論する．8 節では関連研究を示し，最後に 9 節で本研究のまとめと今後の課題について述べる．

2 準備

プログラミング言語の設計においてメモリ安全性を高めることは重要である．これは，解放済みメモリへのアクセスが未定義動作を引き起こすなど，その言語で作成されるソフトウェアの品質に影響するためである [14]．Java や C#といった言語はガベージコレクションによってメモリ安全性を高めている．ガベージコレクションではメモリ上のオブジェクトへの参照を定期的に確認し，参照されなくなったオブジェクトを解放する．しかし，参照の確認処理を定期的に実行する際，ソフトウェアの処理が停止するというパフォーマンス上の問題が存在する [15]．

ガベージコレクションのようなプログラム実行時の安全機構に依らずメモリ安全を保証する手段として型システムによるメモリ管理が提案された．Linear Types[16] や Affine Types[17]，c[18, 19] である．これらの型システムは変数のメモリ解放が必ず 1 回または高々 1 回となるよう型システムによって保証すること，および変数の生存期間を型として捉え，生存期間で階層構造を定めることが特徴である．この特徴により，解放済みメモリへのアクセスや dangling ポインタの発生を静的解析で検知できるようにし，メモリ安全性を保証する．Rust においてメモリ安全性を保証する仕組みである所有権はこれらの概念を取り入れている．

Rust ではソースコード中で操作する各オブジェクトに対して所有権を持つただ一つの変数がそれぞれ設定される．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オブジェクト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，以降オブジェクト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オブジェクトと紐づけられる．

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オブジェクト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 つのオブジェクト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，不意なオブジェクトの書き換えを防いでいる．

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オブジェクト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

```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

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"`という文字列オブジェクトの所有権は `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()` の引数 `id` に譲渡される．この時点で `userId` は所有権を失い，オブジェクトへアクセスできなくなる．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，開発言語として採用する際の問題点としてその習得が困難であることが挙げられている [8]．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 [10]．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．

```
1  void validateId(@Owner String id) {
2      /* some process */
3  }
4  void registerUser(@Owner String id) {
5      /* some process */
6  }
7
8  void main() {
9      @Owner String userId = "ist2024";
10
11     validateId(userId);
12     registerUser(userId); // error!
13 }
```

```
⊗ 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

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 [20]．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．アノテーション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，12 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 図 2 の下部の内容である．1 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．2,3 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．

このソースコードの場合，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 11 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，コンパイルエラーが多発し 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，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 節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．この譲渡処理を 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，オブジェクト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オブジェクトと参照を格納する変数を記録する．テーブルでは各オブジェクト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．使用とはオブジェクト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アノテーションを使う．アノテーションはソース

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．

アノテーションを使う理由は 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アノテーション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）[*1]を使う．CF はアノテーション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．

[*1] `https://checkerframework.org/`

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

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

図 3　Ownership Inventory の find_nth

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．

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[21] 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 のソースコードは 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 のコンパイルエラーと捉え，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．

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	○	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	○	N/A
find_nth	引数の配列から昇順で引数 `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	○	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	×	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	×	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	×	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	×	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	×	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

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 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．

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

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

図 5　Ownership Inventory の remove_zeros

5 被験者実験

提案手法による学習支援効果を確認するため被験者実験を行う．被験者に提案手法を用いた学習を行わせ，Rust 所有権に対する理解を Rust での学習と比較して早期に獲得できるか調査する．

5.1 実験概要

本実験では被験者を 2 グループに分けた対照実験を実施する．両グループに所有権について学ばせ，その後，所有権に対する理解度を問うためにテストを解かせる．学習に用いる教材は 2 グループで同一である．ただし，学習した内容を試す実行環境として一方のグループは提案環境を使わせ，もう一方のグループには Rust を使わせる．

また，テスト終了後にアンケートを実施する．学習過程において感じた難しさを調査するほか，提案環境使用者に使用した感想を求め，提案環境の学習ツールとしての使用感を確認する．

5.2 実験設定

被験者は大阪大学大学院情報科学研究科の修士学生 6 名および大阪大学基礎工学部情報科学科の学部生 3 名である．被験者の募集にあたっては応募者に対し Rust の開発経験がないこと，および Java での開発経験があることを求めた．これは提案環境が想定する使用者層と合わせるためである．

被験者を 2 グループに分けるにあたって事前アンケートを実施した．アンケートでは Java の開発経験について 5 段階で，新言語の習得経験について 3 段階で自己評価させた．Java の経験は提案環境が Java の知識の活用を前提とするため調査した．また，新言語の習得経験は被験者自身の学習能力を表すと考え調査した．グループ分けにおいてはこの 2 点に偏りが発生しないよう注意した．

実験ではまず被験者に Rust の所有権およびライフタイムについて学習させる．学習の制限時間は 40 分とする．学習教材には The Rust Programming Language 日本語版 [22] の 4.1 節，4.2 節，10.3 節を使用する．また，教材で示されたコードや改変を行ったコードを被験者がコンパイルおよび実行できる環境を配布する．提案環境を使用するグループにはブラウザから利用可能な開発環境である GitHub Codespaces[*2]を用いる．Rust を使用するグループにはブラウザから利用可能な Rust の実行環境である The Rust Playground[*3]を用いる．提案環境を実行する Codespaces には教材内で示される Rust コードを，提案手法を用いて所有権操作を記述した Java コードに事前に変換したコードを含める．これは教材内で示された Rust コードをコンパイル，実行する際，Rust を使用するグループではコピー&ペーストのみで実行可能であるのに対し，提案環境では Java コードへの変換が必要となるためである．な

[*2] https://github.co.jp/features/codespaces
[*3] https://play.rust-lang.org/

```rust
// userNameに対する挨拶文を出力し，bobならAdministratorと出力する．
fn greeting(name: String) {
    print!("Hello␣{}!", name);
}

fn main() {
    let userName = String::from("alice");
    greeting(userName);
    if userName == "bob" {
        print!("Administrator");
    }
}
```

Q. 以上のコードはコンパイル時にエラーが発生する．エラー原因は何であり，どのように修正すればよいか．

図 6　問題 4 で使用したコンパイルエラーを含む Rust コードと問題文

お，単に標準出力への出力のみを行うなど，所有権理解に寄与しないと判断したコードは変換していない．以降，事前に変換で作成した Java コードを事前変換コード，事前変換の対象となった Rust コードを事前変換対象コードとする．

学習終了後，被験者に所有権に対する理解度を測るテストを解かせる．テストの制限時間は 20 分とする．テストに取り組む間，被験者には学習で用いた実行環境および Web 検索など外部情報の使用を禁止する．テストで使用する問題は 10 問あり，問題 1 から問題 6 は著者が作成し，問題 7 から問題 10 は Ownership Inventory[21] から引用した．問題の例を 図 6 に示す．各問題では所有権に係るコンパイルエラーを含む Rust コードが示される．このコンパイルエラーに対し，原因の特定およびプログラムの振る舞いを変えない修正を求める．原因と修正の解答を分割し，適切な理解の上で修正を検討しているかおよび場当たり的な修正を行っていないかを検証する．採点は著者が行う．結果の公平性のため，採点は解答者を隠した状態で行う．配点は原因と修正それぞれ 2 点ずつとし，採点基準は次のようにする．

原因　1 点　コンパイルエラーを引き起こす行や操作を特定している
　　2 点　前述に加えてコンパイルエラーを引き起こす処理の理由を説明している
修正　1 点　コンパイルエラーの原因を取り除く方針である
　　2 点　コンパイルエラーを解決する

採点基準の意図としてはエラーを正確に解決するような完全な理解に達していない場合も評価を与えるためである．

5.3 結果の除外

一部の被験者の結果について実験結果から除外する．まず，不適切な実験設定により提案環境の評価に適さないと判断した結果を除外する．本実験では提案環境を使用した被験者のグループの結果について提案環境の使用による影響を受けたことを期待した．一方で，提案環境を使用したグループの被験者の一部は提案環境のコンパイル回数が少なかった．その原因として学習の制限時間が短く，教材の読み込みでその大半を使用してしまったことが挙げられる．このような被験者の結果は提案環境の使用による影響を受けていないと考えられる．よってコンパイル回数が 10 回未満の被験者の結果については除外する．除外対象となった被験者は 2 名である．

また，コンパイル回数を計測できなかった被験者の結果についても除外する．本実験では被験者各自が用意した環境において実験を実施させた．コンパイル回数やコンパイル対象ファイルは被験者自身で録画した操作画面の映像をもとに調査した．しかし，一部の被験者では学習時間の一部について映像が欠落していた．よってこの被験者の結果はコンパイル回数などの学習過程の記録を調査できず，前述のコンパイル回数による除外の対象となる可能性があるため除外する．除外対象となった被験者は 1 名である．

なお，以降で掲載する実験結果を表すグラフでは除外した被験者の結果も灰色で表記する．

5.4 実験結果

5.4.1 テスト全体を通した分析

学習過程におけるコンパイル数を 図 7 に示す．前述した通り除外した被験者は灰色で表示している．Rust を使用した被験者は提案環境を使用した被験者よりも多くコンパイルを実行する傾向があった．また，どの被験者も 1 回以上のコンパイルをしていた．よって，教材の読み込みと並行して，プログラムを実行しながら学ぶという学び方を全ての被験者が行っていたと言える．

提案環境において灰色で示された 2 名は 5.3 節で説明したように，学習の過程において教材の読解に多くの時間を消費した被験者である．この 2 名は提案環境をほぼ利用しておらず，その理解には提案手法の有無は影響しなかったと考えられる．

学習過程のコンパイル回数について大半の実行は教材からのコピー&ペーストしたコード，もしくは提案環境においてそれに相当する事前変換コードであった．前述の通り提案環境では教材に掲載されているコードのうち一部のみを事前変換している．よって Rust と提案環境ではコンパイル対象としたコード数に差がある．よってこのコード数を揃えるため，学習過程のコンパイル回数について，事前変換コードおよびその変換元コードとそれぞれに改変を加えたコードのコンパイルに限定すると 図 8 のようになった．変わらず，Rust を使用した被験者のグループのほうがコンパイル回数が多い一方で，提

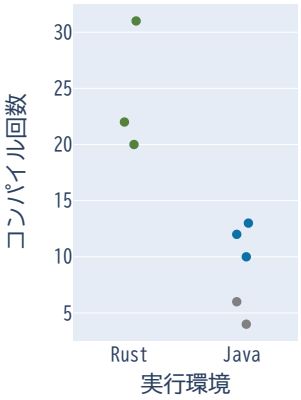

図 7　実行環境ごとのコンパイル回数. ただし, Rust を使用した被験者のうち 1 名はコンパイル回数が不明なため記載していない.

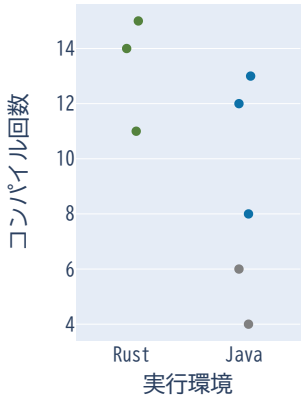

図 8　事前変換対象コードと事前変換コードのコンパイル回数. ただし，Rust を使用した被験者のうち 1 名はコンパイル回数が不明なため記載していない.

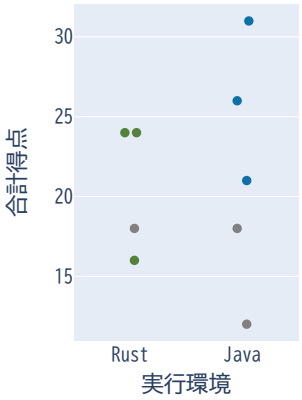

図 9　実行環境毎の得点散布図

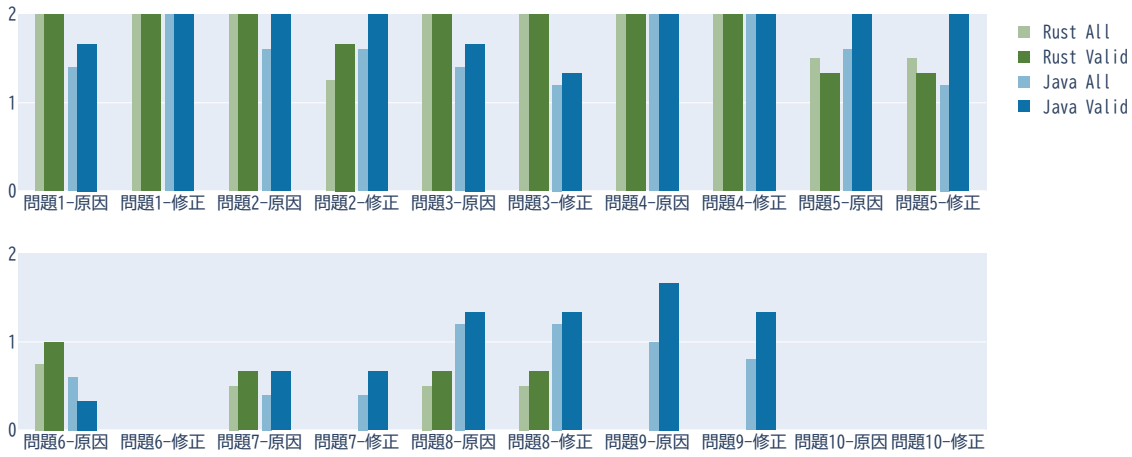

図 10　タスク毎の各グループの平均得点．All は全被験者，Valid はコンパイル回数が 10 回未満，または不明な被験者を除いた被験者を表す.

案環境のグループとの差は小さくなった．このことから Rust を使用した被験者のコンパイル回数が多かった理由の一つとして，所有権理解に寄与しないと判断し Java コードに事前変換しなかったコードのコンパイルが多かったと言える.

提案環境を使用したグループと Rust を使用したグループのそれぞれの被験者の合計点数を 図 9 に示す．最高得点を獲得した被験者は提案環境を使用したグループであり，同グループの他の被験者につい

```rust
fn main() {
    let mut x = String::from("Hello");
    let y = &x;
    y.push_str("World");
    print!("{}", y);
}
```

Q. 以上のコードはコンパイル時にエラーが発生する．エラー原因は何であり，どのように修正すればよいか．ただし，5 行目を書き換えてはならない．

図 11　問題 3 で使用したコンパイルエラーを含む Rust コードと問題文

ても Rust を使用した被験者と比べて得点が高い傾向がある．

5.4.2　問題別の分析

各問題毎のそれぞれのグループの平均得点を 図 10 に示す．問題 1 から問題 4 までの比較的簡単な問題では Rust を使用した被験者の平均得点のほうが高い．例として提案環境を使用した被験者の平均得点が低い問題 3 を 図 11 に示す．この問題の原因の模範解答は「4 行目で不変参照である `y` から破壊的メソッドである `push_str()` を呼び出している．`y` の内容を不変参照から書き換えようとするためエラーが発生する」である．また，修正の模範解答は「3 行目で `y` を可変参照とする」である．この問題では破壊的メソッドの呼び出しにおいて，その呼び出し元の変数に可変性が必要であることに気づき，呼び出し元変数の `y` に可変性を与えるべく可変参照に変更する修正する必要があった．

提案環境を使用した被験者の一部では，この可変性の操作に対する誤解が生じていた．変数の代入による所有権の移動と同時に，可変性も保存されると考えた点である．提案環境を使用した被験者の修正に対する解答として次のものがあった．「`let y = &x;` を `let y = x;` として所有権を渡す」この修正では `y` が不変参照である点は解消されているが，`y` は可変性を持たない．よってエラーの完全な解決には至らない．この誤解の原因としては提案環境において変数の可変性を考慮していなかった点がある．Java において `final` を付与せずに宣言した変数は常に可変である．よって提案環境では可変性については参照を通じた操作についてのみ制限を設けた．一方で Rust においては `mut` を付与せずに宣言した変数は不変である．このため変数の可変性について提案環境では Rust と異なる実装であったため提案環境使用者に誤解が生じた可能性がある．

問題 7 から問題 9 では提案環境を使用した被験者の平均得点のほうが高い．これらの問題は序盤と比べて構文的にも複雑かつ解答難易度も高く設定した．例として問題 9 を 図 12 に示す．問題 9 では `Option` 型が値を内包しており，`unwrap()` で値を取り出すために呼び出し元の変数の所有権が要求されることに気づき，呼び出し元の変数が所有権を持つように書き換える必要があった．値に紐づく所

```rust
/// Gets the string out of an option if it exists,
/// returning a default otherwise
fn get_or_default(arg: &Option<String>) -> String {
    if arg.is_none() {
        return String::new();
    }
    let s = arg.unwrap();
    s.clone()
}
```

Q. 以上のコードはコンパイル時にエラーが発生する．エラー原因は何であり，どのように修正すればよいか．

図 12　問題 9 で使用したコンパイルエラーを含む Rust コードと問題文

有権の意図を深く理解し，その動作を推測することが必要であった．Rust を使用した被験者にとっても `Option` や，`<>`で宣言されるジェネリクスは Rust の学習では触れていない要素であったため理解が難しかったと考えられる．正答出来なかった Rust を使用した被験者のコメントとして，「学習したことのない構文や型の定義がでてきたため」問題 9 は難しかったとする意見があった．よってこの問題においてはソースコードの構文的な読解，処理内容の把握に係る難しさは学習時の使用環境に関わらず同程度であったと言える．読解の難しさが同程度でありながら提案環境を使用した被験者の平均得点が高いのは所有権に対する理解をより深められたためだと考えられる．

5.4.3　アンケートの分析

また，アンケートでは学習において難しいと感じた点として，提案環境と Rust のいずれの使用者からも共通してコード中で明示されない情報の管理の難しさが挙げられていた．

> （学習中難しかった点として）今どこに所有権があり，どの変数が所有権を持つのか，可変か不変かなど，コードを読むにあたって把握しておかなければならないことの多さ．（提案環境使用者）

> どのような場合に所有権が移譲され，どのような場合に移譲されないのかは不明瞭なままであった．（Rust 使用者）

提案環境を使用した被験者から提案環境に対する所感と発展の提案を受けた．提案環境に対して好意的な所感としては次のような回答が得られた．

> Rust と比べて所有権操作を明示的に記述する点がよかった

> 事前変換された Java コードを教材の Rust コードと見比べることで Rust の理解が深まった

一方で否定的な所感としては次のような回答が得られた．

> 提案環境でエラーが出ないことが Rust でもエラーが出ないことの保証にはならないため不安だった

> 教材との記述のマッピングに戸惑った

また，機能提案として次のような回答が得られた．

> 今回の実験で Java コードの横にコメントで Rust コードが載っていたように，Rust コードも Java コードの隣に記述して見比べながら学習できるとなお良いと思った

5.5 考察

被験者実験の結果より，提案環境使用者は Rust のみを使用した被験者と比較して所有権についてより理解を深められたと言える．難易度が高い問題において Rust 使用者を上回る成績を獲得しており所有権に起因する問題の特定，およびその解決能力を獲得できた．特に所有権上の仕様が教材などにいて明示されてない場合にも，その処理の意図から仕様を類推し，コンパイルエラーの原因を推測できた．

また，手法のメリットとして所有権操作を記述する際に，開発者の意図を明示的に記述できる点が挙げられる．Rust は&のような記号や mutable の短縮形 `mut`，暗黙的な参照の変換など非直感的な記法が多用されている．Coblenz らによると，*や&といった演算子の利用や Rust コンパイラによる暗黙的な所有権操作の挿入が Rust 学習者の理解を妨げると報告されている [23]．また，脳内でプログラムの処理を追跡することは難しいとされている [24]．一方で提案環境は明示的にアノテーションを使用したため学習者が意識的かつソースコード中に明示して操作でき，理解に貢献したと考えられる．実用上は文字数が増えコードが冗長となってしまうが，こと学習においては言語系による解釈ではなく自身の意図を直接記述できる点で理解しやすくなったと考えられる．

一方で提案環境の使用による副作用が存在すると言える．難易度が低い問題においては提案環境と Rust との仕様の差異により，Rust の使用に対する誤解が生じ，問題の解決に影響があった．また，提案環境でエラーが出ないことが Rust でもエラーが出ないことを保証しなかったため学習の過程で不安を生じさせた．被験者がこれらを提案環境の問題点として挙げたのは，提案環境の実行を通して学習を進めていく上で提案環境によるフィードバックを完全には信用できない点が，学習効率に悪影響を与えると考えた可能性がある．これらの点から機能移植における移植範囲や再現の限界を使用者に明示し，その限界より先でどのように Rust と接続していくかの道筋を提示する必要があったと言える．

本実験で得られた知見を基にした提案手法の発展として，学習支援手法として実行環境だけの提供ではなく，Rust コードから提案環境に適応した Java コードへのソースコード翻訳器の同時提供が考えられる．実験中，教材内で示される Rust コードを提案環境に適応させた Java コードを配布した．こ

れは，実行までの手順量の差を吸収するための措置であり，学習支援を意図した手順ではなかった．一方で被験者はこのコードと Rust コードを比較し，学習に活用していた．これは Rust が暗黙的に行っている所有権操作を提案環境が明示的に行っている点で，Rust コードの読解に寄与したと考えられる．よって任意の Rust コードを提案環境で再現できるソースコード翻訳の提供が考えられる．

6　妥当性の脅威

適用実験は Ownership Inventory[21] で示された 8 件のソースコードに対して行った．Ownership Inventory は Rust 所有権に対する誤解を明らかにするために作成されたソースコード集であり，所有権機能の再現度に対するベンチマークとして用いることは本来の目的から外れる．所有権機能の全体と比較した際に，提案手法の再現度はより小さく評価される可能性がある．

被験者実験は 10 問の問題と 9 人の被験者で実施した．これは統計的な分析を行うには規模が小さすぎる可能性がある．より一般化可能な分析とするには，問題数，被験者数を大きくする必要があり，その結果が今回と同様となるとは限らない．

被験者実験の分析においてコンパイル実行回数が一定数以上の被験者のみを対象とした．手法を想定通り使用した被験者に注目するため採った措置である．一方でこの制限により学習方法や被験者の能力に偏りが発生した可能性がある．

被験者実験で用いた問題 1 から問題 6 は著者の手作業により作成した．よって，その作成には所有権に対する著者の意識が影響している可能性があり，所有権理解を測る上での適切性や十分性は検討していない．より，客観的に設計された問題を用いた場合，その結果は今回の結果から大きく変わる可能性がある．

7　議論

7.1　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．

加えて，明確に Rust と Java で言語仕様が異なる点についても再現しなかった場合がある．例えば変数宣言時の可変性が存在する．Rust においては可変であると明示して宣言しない限り変数は不変である．一方 Java においては不変であると明示して宣言しない限り変数は可変である．この機能の移植については Java が本来持つ言語仕様に対する改変が必要であるため，提案手法では移植しなかった．このように既習言語と相反する仕様についてはその再現に制約がある．

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は無く，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．

7.2　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 [25] 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ル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．

7.3 Rust コードから提案環境に準拠した Java コードへのソースコード翻訳器

被験者実験において Rust コードと同等の処理，所有権操作を行う提案環境に適用させた Java コードを配布した．実験後のアンケートにおいてこのコードがが学習において役立ったという意見があった．新たな言語の学習において慣れない言語で記述された処理を追跡するより，見知った言語で処理内容をを予習してから学習対象の言語で記述されたコードを読むほうがより容易に記述内容を理解できると考えられる．また，本環境の利点として挙げられた所有権操作の明示的な記述が，所有権や参照といった Rust 上では一部暗黙的に操作される状態を読み取る上での助けになると考えられる．

このような Rust コードを元にした Java コードの提供の実現にあたっては自動ソースコード翻訳器が必要である．任意の Rust コードに対して手作業で変換したコードを用意するには多大な労力を要するためである．ソースコード翻訳には機械学習を用いた手法 [26, 27] が考えられるが，学習に大量のソースコードが必要である．提案環境を適用したソースコードは存在しないためこの手法は適用できない．よって翻訳を Rust から Java への変換，Java にアノテーションを追加の 2 段階に分けることを考える．ある言語から安全性を高めるよう拡張された言語に対する自動翻訳として Machiry らの 3C[28] がある．これは既存の C 言語のソースコードを C 言語の安全性を高めた拡張である Checked C[29] に自動で変換するツールである．変換は決定的アルゴリズムによる推論であるため学習元ととするソースコードは必要ない．このように提案環境で使用するアノテーションを推論する強力なアルゴリズムを開発できれば Rust から Java そして提案環境への変換を自動化できる．これにより実験と同様の提案環境のコードと Rust コードを見比べられる状況を任意の Rust コードから生成できるようになる．

8 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 [21]．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 [30]．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 [31]．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．本研究ではあくまで純粋な Rust の習得支援を目指した．

本研究では Java での開発経験を持つ開発者の Rust 習得を支援した．このように既にプログラミング言語を習得している開発者が新たな言語を習得することを言語移行と呼ぶ [32, 20]．この言語移行をプログラミング初学者の学習と区別し，言語移行に特有な問題を分析しその解決を目指す研究が存在する．Nischal らは言語間の移行において移行元言語の知識と移行先言語の知識の間で “促進” と “干渉” が発生すると報告 [32] した．促進と干渉は新たな知識を習得する際に事前に持っていた知識が影響することを表す．事前に持っている知識によって新たな知識の習得が助長されることを促進，阻害されることを干渉と呼ぶ [20]．言語移行においては移行元言語の仕様を移行先の言語でも同一であると誤認し，その仕様に基づいたプログラムを作成することがあると報告されている．また，Rust については所有権検査を行なうコンパイラの仕様が他の言語と大きく異なるため促進が発生しにくいとされている．本研究では Java コンパイラの仕様を Rust コンパイラの仕様に近づけることで促進の発生を図った．一方で実験においては仕様の差に起因する干渉も発生しており，干渉を抑えるような措置が必要である．Nischal ら [32] は言語移行に特化したドキュメントの作成が有効ではないかと考察している．

9 おわりに

本研究では 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た Rust コード 8 件中 3 件のコードで Rust コンパイラのエラーを Java 上で再現できた．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．

被験者実験では実際に学習時に提案環境を使用させ，Rust を使用した被験者と比較して提案環境を使用した被験者が所有権についてより深い理解を獲得したことを確認した．一方で，提案環境による副作用として Rust と Java における言語仕様の差異に起因する誤解が発生したことも確認した．本研究の目的は Rust 習得の支援であるため，提案環境から Rust へより違和感なく移行するための支援が必要である．

今後の研究課題として，所有権再現度の向上，視覚的なエラーの表示および Java から Rust への接続支援が挙げられる．現状の所有権再現度は適用実験で示しているように再現できていない事項が複数存在する．この点の改善により開発者が学習に使用する上でコンパイラによるソースコードへのフィードバックに対する信頼度を向上させ，より学習支援効果を高められると考えられる．また，関連研究で示した RustViz[30] を用いた可視化を開発者に対するフィードバックに取り入れることで開発者により学習効果の高いフィードバックを与えられると考えられる．さらに，提案環境から Rust へ移行する際に再現されていない事項をより明確にするなどの接続支援が必要である．本研究では所有権概念のみを移植した Java 環境を学習環境として提供した．被験者実験において移植の対象外とした言語仕様について，Java の仕様が Rust でも同等であるとの誤解が発生した．よって提案環境での学習において習得すべき移植された仕様が何か，また，移植されず Rust への移行において再認識が必要な事項は何かを学習に併せて明示できる方法を取り入れる必要がある．

謝辞

本研究を遂行するにあたり，多くの方々に多大なご協力をいただきました．

楠本真二教授には研究遂行のための環境をご提供いただきました．場所としてだけでなく場として，和やかな研究室の雰囲気を作ってくださいました．研究がうまく進まないときも必要以上に思い詰めずいられたのはこの雰囲気のおかげです．深く感謝申し上げます．

柗本真佑助教には研究遂行の全てにおいて熱心にご指導いただきました．研究方針をはじめ，論文執筆や発表準備において幾度もご相談に乗っていただきました．また，一歩踏み出すか迷った際に幾度となく背中を押していただけたため，研究含め多くの事柄に挑戦できました．心より感謝申し上げます．

ソフトウェア工学講座の肥後芳樹教授には研究に対し新たな視点を与える助言をいただきました．また，海外発表の際には現地での活動のサポートをいただき大変お世話になりました．おかげさまでつつがなく発表を行えました．深く感謝申し上げます．

事務補佐員の橋本美砂子氏には研究に付随する作業を大いに助けていただきました．特に本研究の実験実施においては橋本氏のサポートをいただいたおかげで事務処理に煩わされることなく進行できました．深く感謝申し上げます．

楠本研究室の皆様には日々様々な会話を重ね，様々な刺激を頂きました．互いの研究について議論し新たな視点を得られたこともあれば，たわいない会話を楽しみ，研究に取り組む活力を得ることもありました．研究活動に楽しみながら取り組めたのは皆様のおかげです．誠にありがとうございました．

最後に，生活面でも精神面でも支えとなってくれた家族に感謝いたします．

参考文献

[1] Matsakis, N. D. and Klock, F. S.: The Rust Language, *SIGAda Ada Letters*, Vol. 34, No. 3, pp. 103–104 (2014).

[2] Stack Overflow, : Stack Overflow Developer Survey 2023, https://survey.stackoverflow.co/2023/#technology-admired-and-desired (Accessed at 2024-01-28).

[3] Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D.: Safe Systems Programming in Rust, *Communications of the ACM*, Vol. 64, No. 4, pp. 144–152 (2021).

[4] Bugden, W. and Alahmar, A.: Rust: The Programming Language for Safety and Performance, https://arxiv.org/abs/2206.05503 (2022).

[5] The kernel development community, : Rust, https://www.kernel.org/doc/html/next/rust/index.html (Accessed at 2024-02-01).

[6] Android Open Source Project, : Android Rust introduction, https://source.android.com/docs/setup/build/rust/building-rust-modules/overview (Accessed at 2024-02-01).

[7] TIOBE, : TIOBE Index, https://www.tiobe.com/tiobe-index/ (Accessed at 2024-02-01).

[8] Fulton, K. R., Chan, A., Votipka, D., Hicks, M. and Mazurek, M. L.: 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).

[9] Zeng, A. and Crichton, W.: Identifying Barriers to Adoption for Rust through Online Discourse, in *Proceedings of Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 1–6 (2019).

[10] Zhu, S., Zhang, Z., Qin, B., Xiong, A. and Song, L.: Learning and Programming Challenges of Rust: A Mixed-Methods Study, in *Proceedings of International Conference on Software Engineering*, pp. 1269–1281 (2022).

[11] The Rust Survey Team, : Rust Survey 2020 Results, https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html (Accessed at 2024-01-28).

[12] Meyerovich, L. A. and Rabkin, A. S.: Empirical Analysis of Programming Language Adoption, in *Proceedings of International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 1–18 (2013).

[13] Stefik, A. and Siebert, S.: An Empirical Investigation into Programming Language Syntax, *ACM Transactions on Computer Education*, Vol. 13, No. 4, pp. 1–40 (2013).

[14] Xu, H., Chen, Z., Sun, M., Zhou, Y. and Lyu, M. R.: Memory-Safety Challenge Considered

Solved? An In-Depth Study with All Rust CVEs, *Transactions on Software Engineering and Methodology*, Vol. 31, No. 1, pp. 1–25 (2021).

[15] Carpen-Amarie, M., Marlier, P., Felber, P. and Thomas, G.: A Performance Study of Java Garbage Collectors on Multicore Architectures, in *Proceedings of International Workshop on Programming Models and Applications for Multicores and Manycores*, p. 20 – 29 (2015).

[16] Wadler, P.: Linear Types can Change the World!, in *Proceedings of Working Conference on Programming Concepts and Methods*, pp. 347–359 (1990).

[17] Pierce, B. C.: *Advanced Topics in Types and Programming Languages*, The MIT Press (2004).

[18] Tofte, M. and Talpin, J.-P.: Region-Based Memory Management, *Information and Computation*, Vol. 132, No. 2, pp. 109–176 (1997).

[19] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y. and Cheney, J.: Region-Based Memory Management in Cyclone, in *Proceedings of Conference on Programming Language Design and Implementation*, p. 282 – 293 (2002).

[20] Bower, M. and McIver, A.: Continual and Explicit Comparison to Promote Proactive Facilitation During Second Computer Language Learning, in *Proceedings of Conference on Innovation and Technology in Computer Science Education*, pp. 218–222 (2011).

[21] Crichton, W., Gray, G. and Krishnamurthi, S.: 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).

[22] Klabnik, S. and Nichols, C.: The Rust Programming Language, https://doc.rust-jp.rs/book-ja/ (Accessed at 2024-02-01).

[23] Coblenz, M., Porter, A., Das, V., Nallagorla, T. and Hicks, M.: A Multimodal Study of Challenges Using Rust (2023).

[24] Crichton, W., Agrawala, M. and Hanrahan, P.: The Role of Working Memory in Program Tracing, in *Proceedings of Conference on Human Factors in Computing Systems*, pp. 1–13 (2021).

[25] Barik, T., Smith, J., Lubick, K., Holmes, E., Feng, J., Murphy-Hill, E. and Parnin, C.: Do Developers Read Compiler Error Messages?, in *Proceedings of International Conference on Software Engineering*, pp. 575–585 (2017).

[26] Roziere, B., Lachaux, M.-A., Chanussot, L. and Lample, G.: Unsupervised Translation of Programming Languages, in *Proceedings of International Conference on Neural Information Processing Systems*, pp. 20601–20611 (2020).

[27] Chen, X., Liu, C. and Song, D.: Tree-to-tree Neural Networks for Program Translation, in *Proceedings of International Conference on Neural Information Processing Systems*, pp. 2552–2562 (2018).

[28] Machiry, A., Kastner, J., McCutchen, M., Eline, A., Headley, K. and Hicks, M.: C to Checked C by 3C, *Journal on Proceedings of the ACM on Programming Languages*, Vol. 6, No. OOPSLA1, pp. 1–29 (2022).

[29] Elliott, A. S., Ruef, A., Hicks, M. and Tarditi, D.: Checked C: Making C Safe by Extension, in *Proceedings of Cybersecurity Development*, pp. 53–60 (2018).

[30] Almeida, M., Cole, G., Du, K., Luo, G., Pan, S., Pan, Y., Qiu, K., Reddy, V., Zhang, H., Zhu, Y., et al.: Rustviz: Interactively Visualizing Ownership and Borrowing, in *Proceedings of Symposium on Visual Languages and Human-Centric Computing*, pp. 1–10 (2022).

[31] Coblenz, M., Mazurek, M. L. and Hicks, M.: 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).

[32] Shrestha, N., Botta, C., Barik, T. and Parnin, C.: Here We Go Again: Why is It Difficult for Developers to Learn Another Programming Language?, in *Proceedings of International Conference on Software Engineering*, pp. 691–701 (2020).