RustOwl: Rustにおける所有権とライフタイムの可視化

岡本 祐希 [1], 篠埜 功 [2]

芝浦工業大学

[1] ma21029@shibaura-it.ac.jp [2] sasano@sic.shibaura-it.ac.jp

概 要　Rustは所有権とライフタイムを導入することにより、メモリ安全性やデータ競合がないことを保証しつつ効率の良いプログラムを記述できるようにした。所有権とライフタイムはRust初学者にとって難しく、所有権やライフタイムの可視化に関する研究が行われてきたが、その多くは初学者向けのものであり、評価も初学者を対象に行われていた。本研究では、借用検査器Poloniusの出力を用いてライフタイムを可視化するアルゴリズムを提案する。中間表現MIRとこのアルゴリズムを用いて、所有権とライフタイムをエディタ上で可視化するツールRustOwlを実装した。このツールの学習効果を測定する実験を72名のプログラミング経験者を対象に行い、既存の可視化ツールAquascopeと比べて有意に学習効果があった。また、主観評価としてこのツールがメモリ最適化に使えるという結果が得られた。

キーワード：Rust、所有権、ライフタイム、可視化

1　はじめに

プログラミング言語Rustは実行効率を犠牲にすることなくメモリ安全でデータ競合の起きないプログラムを記述できる言語として、近年注目を集めている。Rustにおいてコンパイル時にこうした安全性を保証する仕組みとして、所有権とライフタイムが導入されている。コンパイラは所有権とライフタイムを検査することで、`unsafe`コードを除いて不正なメモリ参照を実行前に検査することができる。

一方、所有権とライフタイムはRust初学者に限らず誤った理解をされることが多い[1]。Rustでは2022年にNon-Lexical Lifetime (NLL)と呼ばれる新しいライフタイムの定義が安定版に導入されており、これも所有権とライフタイムの理解を難しくしている。このような課題は、学習のみならず、バグの原因になっていることも指摘されており、IDEなどでの開発支援の必要性が提唱され[2]、所有権の移動や借用の可視化が研究されてきた[3, 1, 4, 5, 6]。これらの研究は、NLLへの対応がなされていないものが多く、また所有権の教育に特化しており、ライフタイムを理解したデバッグやメモリ管理といった実用からは遠い。

本研究では、所有権とライフタイムをテキストエディタ上で可視化するツールであるRustOwlを提案する。可視化は色付きの下線をソースコード上に引くことで行い、下線の範囲を計算するためにRustでLSPサーバを実装し、VS Code、Neovim、Emacsと通信をして可視化する。また、第三者によって、RustRoverやSublime Textといったその他のエディタ向けの拡張も実装されている。可視化のための計算はファイル保存時に行うこととし、ユーザーが任意の時点でローカル変数、関数呼び出し式、およびメソッド呼び出し式にテキストカーソルを合わせると、当該ローカル変数またはその関数呼び出し式の結果について、ライフタイム、有効であるべき範囲、それらの差分に色付きの下線がつく。また、同じ操作で、不変借用、可変借用、所有権の移動も色付き下線で可視化する。所有権のエラーはライフタイムの過不足よって起こるため、これらを両方表示することで、所有権のエラーの原因について理解してデバッグすることができる。さらに、マウスカーソルが下線上に置かれたら、その下線の意味を説明するポップアップウインドウを表示する。この機能により、ユーザーは文章で下線の意味を確認することができる。RustOwlは広く用いられている既存の

LSP サーバーである rust-analyzer などとも併用できる。RustOwl は Rust コンパイラ `rustc` の内部 API と、借用検査器である Polonius を利用して実装しており、Rust 1.89.0 のフルセットに対応し、NLL、`async`、`unsafe` を含む実用的なコードにも対応する。また、serde や tokio などの広く使われているライブラリや RustOwl 自身でも利用できる。我々の知る限りでは、所有権とライフタイムを同時に、実用的なエディタ上で可視化をするツールは他に存在しない。

RustOwl の実用性を示すため、RustOwl の使用により、所有権およびライフタイムに関するユーザーの理解を促進するかどうかについて、既存の Rust フルセットに対応した可視化ツールである Aquascope[1] との対照実験を行った。この実験結果から、RustOwl は Aquascope に比べて、被験者全体で所有権およびライフタイムに関する理解を有意に促進することが示された。習熟度別では Rust 中級者で総得点に有意差があり、また主観評価からはメモリ最適化への利用の可能性も示唆された。

本研究の貢献は以下のとおりである。

- 借用検査器である Polonius の出力を基に、ソースコード上でローカル変数が有効でなければならない範囲を求めるアルゴリズムを提案した。
- RustOwl と Aquascope を用いてユーザーの所有権とライフタイムの理解度を問う対照実験を実施し、RustOwl が所有権とライフタイムの理解をより促進することが有意に示された。
- 従来 Rust の可視化ツールは教育目的で研究、評価されてきたが、RustOwl は特に中級者の理解を促進し、主観評価ではメモリ最適化へ利用できることを示した。
- フルセットの Rust を対象に、所有権とライフタイムの両方を実用的なエディタ上で可視化するツールを初めて実装した。

本論文は以下のような構成となっている。2 節では所有権とライフタイムについて、現状の Rust コンパイラ内部の解析アルゴリズムに触れ、本研究に必要な前提について述べる。3 節で RustOwl の可視化の対象や可視化手法について述べる。4 節でローカル変数が有効であるべき範囲を求めるアルゴリズムを提案する。5 節では 4 節で示したアルゴリズムを実装したツールである RustOwl の実装の詳細および利用方法について述べる。6 節で評価実験とその結果について述べる。7 節で既存研究と本研究の関連について議論する。8 節でまとめと今後の課題について述べる。

2 所有権とライフタイム

Rust[7, 8] ではオブジェクトは変数によって所有される。所有権はコンパイル時に検査される。オブジェクトの所有権の移動はムーブ（move）とも言う。借用（borrowing）は参照（reference）を作成することである [9, 4.2 節]。借用においては所有権（ownership）の移動を伴わない。借用は不変借用（immutable borrowing）と可変借用（mutable borrowing）の 2 種類がある。不変借用によって作成された参照は不変参照（immutable reference）と呼ばれ、この参照を経由して参照先のオブジェクトを読めるが書き換えられず、可変借用によって作成された参照は可変参照（mutable reference）と呼ばれ、この参照を経由して参照先のオブジェクトを読むことも書き換えることもできる。また、1 つのオブジェクトに対する不変参照は同時に複数生存できるが、1 つのオブジェクトに対する可変参照は同時に 1 つまで生存でき、あるオブジェクトに対する可変参照が生存している間はそのオブジェクトに対する不変参照は 1 つも生存できない。

ある関数の本体において、あるオブジェクトに対する参照が生存するソースコード中（正確には MIR 中）の範囲について、Non-Lexical Lifetime (NLL) [10] で定義され、2022 年に `rustc` の安定版に導入された。ローカル変数に代入された参照が関数の本体内において生存する範囲は従来その変数に参照が代入されてからそのローカル変数が宣言されたブロックを抜けるまでであった。NLL は借用検査を通る場合を増やすため、参照が代入されたローカル変数の利用状況に応じて、関数の本体内で参照が生存する範囲を縮小するために導入された。NLL では参照が最後に用いられた場所

```
p0    let n = Box::new(1);
p2    let mut r = &'r1 n;
p4    if **r == 2 {
p6        let m = Box::new(2);
p8        r = &'r2 m;
      }
p10   println!("{}", r);
```

図 1. プログラム中の位置を表すポイントと参照のライフタイムを付与した Rust ソースコードの例

までを参照が生存している範囲とすることで、可変参照と他の参照が同時に生存することによるエラーを削減する。借用検査（borrow check）は、制御フロー解析が容易に行える Rust の中間表現 Mid-Level IR (MIR) に対して行われる。MIR では各文にプログラムの位置を表すポイントが振られており、借用検査で用いられる。

Rust においてライフタイム（lifetime）という用語は Rust 特有の意味を持ち、文脈に応じて以下の意味で用いられる。

1. 参照のライフタイムは、その参照が用いられる範囲を表す。この範囲は、リージョン推論（region inference）の結果、MIR 中のポイントの集合として求める [11, 12]。これは上で述べた、NLL で導入された定義である。
2. オブジェクトのライフタイムは、そのオブジェクトが生成されてから解放されるまでの期間を表し、これを Rust 特有の表現としてオブジェクトのスコープとも呼ぶ [11, 12]。
3. ローカル変数のライフタイムは、その変数にオブジェクトが代入されてから、その変数が宣言されたブロックを抜けるか、あるいはその変数に格納されたオブジェクトがムーブされるまでの範囲を言う [13]。この範囲を、本研究では**ローカル変数の有効範囲**とも呼ぶこととする。

1 において、参照が用いられる範囲とは、ローカル変数に参照が代入されてからその変数が最後に使われるまでの範囲を指す。なお、`rustc` では、グローバル変数にはライフタイムとして`'static` が割り当てられており、借用検査で用いられる。Rust においてラムダ式に相当するものはクロージャと呼ばれ、MIR では新たな関数定義を生成してコンパイルされ、クロージャの外のローカル変数は引数として借用されるかムーブされて渡されるような形にコンパイルされる。

3 において、変数に格納されているオブジェクトのムーブは、そのオブジェクトが他の変数に代入されるか、その変数が関数の引数として渡されるときに起こる。変数のオブジェクトがムーブされた後に、変数にオブジェクトが再代入された場合、変数のライフタイムが再開する。借用検査ではライフタイムの範囲を MIR の制御フローグラフを使って検査しており、if 式の片方のブランチで不正な値の参照が起きるなどの場合はコンパイルエラーとしている。なお、ローカル変数のリソースを解放するドロップ処理は、その変数が宣言されたブロックを抜けるか、`drop` 関数にその変数が引数として渡されてムーブされる場合に行われる。

ライフタイム変数はプログラマが名前をつけて導入するか、コンパイラがコンパイル時に割り当てる。借用検査では各関数内でライフタイム制約を解いている。ライフタイムは MIR 中のポイントの集合である。Rust ソースコードにプログラム中の位置を表すポイント $p_0, \ldots, p_{10}$ と、参照のライフタイムを表す $'r_1, 'r_2$ を付与した例を図 1 に示す。実際の Rust コンパイラでは、MIR の文（statement および terminator）に対してポイントを付与するが、ここでは説明のためにソースコード中にポイントを付与している。文の開始位置のポイントが p_i のとき、それぞれの文の種類に応じた副作用を持つ直前のポイントが p_{i+1} である。図 1 の Rust コードは借用検査でエラーになる。`rustc` でのコンパイル結果を図 2 に示す。

数年前から `rustc` では借用検査は NLL[12] に従って、Polonius という借用検査ライブラリを用い

```
error[E0597]: ‘m‘ does not live long enough
 --> error-rust.rs:6:13
  |
5 |         let m = Box::new(2);
  |             - binding ‘m‘ declared here
6 |         r = &m;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - ‘m‘ dropped here while still borrowed
8 |     println!("{}", r);
  |                    - borrow later used here

error: aborting due to 1 previous error
```

図 2. 図 1 の Rust ソースコードをコンパイルした時のコンパイルエラー

て、ライフタイム制約を解くことによって行われている。以下では、この概略について述べる。

ライフタイム制約は 2 つのポイント集合 P_1, P_2 とポイント p の間の関係であり、P_2 が P_1 中のポイントのうち p から到達可能なポイントを全て含んでいなければならないという制約である。この制約を本論文では以下の表記で表す。

$$(P_1 \subseteq P_2) \ @ \ p$$

MIR において、任意の参照は、何らかの型 T および何らかのライフタイム変数 $'r$ について、`&'r` T 型を持ち、$'r$ はこの参照のライフタイムを表す [12]。Polonius は、検査中の関数内で宣言された参照型の各変数 v について、以下のようにライフタイム制約を生成する。変数 v の型が `&'r`T のとき、この変数 v に値が代入されたポイントの次のポイントから、その値が最後に使われるポイントまでのすべてのポイント p について、ライフタイム制約 $(\{p\} \subseteq 'r) \ @ \ p$ を追加する [12]。各変数についてライフタイム制約を生成したのち、それらを満たすような最小の解を求める。この解を利用して、可変参照と他の参照のライフタイムに共通部分があるかや、オブジェクトのライフタイムが参照のライフタイムを超えていないかを検査する。

例えば、図 1 のプログラムにおけるライフタイム制約を全て列挙すると、次のようになる。

$$\begin{aligned}
(\{p_3\} \subseteq 'r_1) \ &@ \ p_3 \\
(\{p_4\} \subseteq 'r_1) \ &@ \ p_4 \\
(\{p_9\} \subseteq 'r_2) \ &@ \ p_9 \\
('r_1 \subseteq 'r_2) \ &@ \ p_9 \\
(\{p_{10}\} \subseteq 'r_1) \ &@ \ p_{10}
\end{aligned}$$

制約 $('r_1 \subseteq 'r_2) \ @ \ p_9$ の生成については、説明を省略する。

上記で得られた 5 つの制約の解は以下のようになる。

$$\begin{aligned}
'r_1 &= \{p_3, p_4, p_{10}\} \\
'r_2 &= \{p_9, p_{10}\}
\end{aligned}$$

変数 `m` のライフタイムは、変数 `m` が宣言されたポイントの直後のポイントからブロックの終わりまでのポイントの集合 $\{p_7, p_8, p_9\}$ であり、$'r_2$ が `m` の参照のライフタイムであることから、$'r_2 \subseteq \{p_7, p_8, p_9\}$ が成り立つ必要があるが、成り立っていないため、借用検査が失敗し、コンパイルがエラーで終了する。

なお、MIR 上のポイントは、`rustc` でエラーメッセージに用いるために、ソースコード上の位置と紐づいており、ポイントからソースコード上の位置を取得することが可能であり、本研究のツールで用いている。

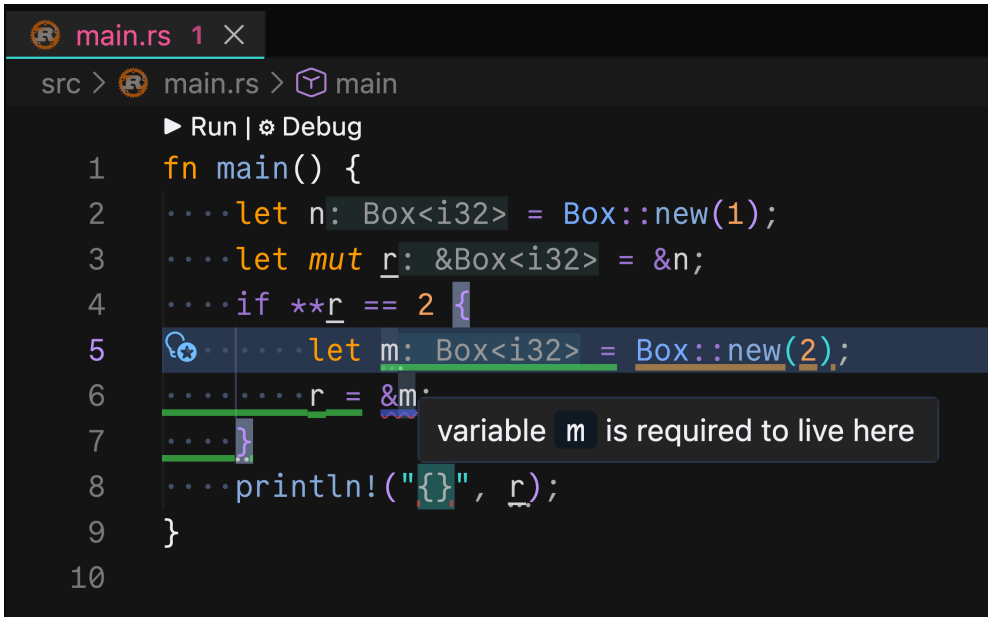

図 3. RustOwl の動作画面

3 可視化の対象

本研究では所有権とライフタイムを可視化するため、VS Code などのエディタで動作するツールである RustOwl を実装した。

RustOwl はエディタ上のカーソルの位置にローカル変数や関数呼び出し式がある場合に、その変数や関数呼び出し式の結果が格納される MIR 中の変数について、以下のような範囲をソースコード中の位置に対応させて色付きの下線を用いて可視化する。

1. 緑色：ローカル変数の有効範囲
2. 青色：ローカル変数を不変借用する式の範囲
3. 紫色：ローカル変数を可変借用する式の範囲
4. 橙色：ローカル変数に格納されたオブジェクトがムーブしている式の範囲
5. 赤色：ローカル変数が有効であるべき範囲とその変数の有効範囲との差分

1、2、3、4 は Polonius で求めた MIR 中のポイントの集合をソースコード上の位置に対応させている。5 のローカル変数が有効であるべき範囲とは、ローカル変数の有効範囲がこの範囲を含んでいなかった場合にコンパイルエラーとなる範囲を表す。ローカル変数が有効であるべき範囲を赤色で下線を引くために、4 節で提示する Algorithm 2 で求める MIR 中のポイントの集合を、MIR 中にあるデバッグ情報を用いてソースコード上の位置に対応させる。これらの可視化を行う具体例を図 3 に示す。図 3 では、if 式のブロック内部で宣言された `Box<i32>`型の値を所有する変数 `m` の参照が、if 式の外で用いられていることを 8 行目の赤色の下線で示している。if 式のブロック内で宣言された変数の有効範囲はそのブロック内のみである。変数 `m` の有効範囲は実際は緑色の下線で示された部分までであり、この有効範囲を超えて変数 `m` の参照を用いることはできないことが可視化されてる。これにより、ユーザの有効範囲のエラー解決に役立つ。また、ヒープ領域などのローカル変数のリソースは、その変数が宣言されたブロックを抜けるタイミングでリソースの解放処理が行われる。そのため、ローカル変数の有効範囲はリソースが確保されている範囲に等しく、ユーザのリソース管理に役立つ。これらに加え、下線部にマウスカーソルを合わせた際に、その可視化の意味を小さいウインドウ上に表示する。関数呼び出し式の内側のローカル変数や、複数のメソッド呼び出し式がある場合は、カーソルの位置している範囲のなかで最も内側のものについて可視化する。

なお、四則演算などのプリミティブな関数やグローバル変数はこれらの可視化の対象外とする。可

視化は MIR の関数呼び出し式に対して行っており、プリミティブな関数は MIR 中の呼び出し位置が異なる。ローカル変数に格納されないプリミティブな関数の実行結果の型はドロップ処理が行われないものが多く、ライフタイムを可視化する需要が少ないと考えられる。グローバル変数はムーブが起きず、有効範囲もプログラム全体にわたるため可視化の対象外とする。

4 ローカル変数が有効であるべき範囲を求めるアルゴリズム

本節では 1 つの関数内において、ローカル変数の有効範囲と、ローカル変数が有効であるべき範囲の可視化のために、MIR 上でローカル変数が有効であるべきポイントの集合を計算するアルゴリズムを提案する。なお、不変借用、可変借用、ムーブしている式の範囲は MIR から求まり、ローカル変数の有効範囲は借用検査器である Polonius の出力に含まれているため、可視化においてそれらを用いる。

本節ではメタ変数として、プログラム中のポイント p とポイントの集合 P、ライフタイム変数（lifetime variable）$'r$ とライフタイム変数の集合 R、ローカル変数 v とローカル変数の集合 V、参照 b と参照の集合 B を用いる。また、解析対象の関数について Polonius が生成したライフタイム制約全ての集合を C で表す。

本節で提案する Algorithm 1, 2 において、Polonius が計算する以下の写像を用いる。

- ポイント p を受け取り、ポイント p をそのライフタイムに含む参照のライフタイムを表すライフタイム変数全ての集合 R を返す写像 *ValidRegions*
- ローカル変数 v を受け取り、ローカル変数 v を借用した参照全ての集合 B を返す写像 *Borrows*
- ポイント p とライフタイム変数 $'r$ を受け取り、$R = \{'r_1 \mid ('r_1 \subseteq 'r) \ @ \ p \in C\}$ を返す写像 $\mathit{PointSubsets}(p)('r) = R$
- ポイント p とライフタイム変数 $'r$ を受け取り、ポイント p をそのライフタイムに含む参照のうち、$'r$ **に関連した参照**全ての集合 B を返す写像 $\mathit{PointRegionBorrows}(p)('r) = B$

なお、ライフタイムがライフタイム変数 $'r$ で表される参照、あるいはその参照を直接、あるいは間接的に参照する参照を、**ライフタイム変数** $'r$ **に関連した参照**と呼ぶこととする。

Algorithm 1 では、以下の 4 つの写像を求める関数を与える。

- GET_REGION_INCLUDING_POINTS 関数は、写像 *ValidRegions* を受け取って、ライフタイム変数 $'r_i$ を受け取り、ライフタイム $'r_i$ が含むポイントの集合 P を返す写像 *RegionPoints* を返す。
- GET_REF2VAR 関数は、MIR の関数内のすべてのローカル変数 *AllVars* およびローカル変数 v を受け取って参照の集合 B を返す写像 *Borrows* を受け取り、参照 b を受け取って b が借用しているローカル変数 v を返す写像 *Ref2Var* を返す。
- GET_ALL_SUBSETS 関数は、写像 *PointSubsets* を受け取って、ライフタイム変数 $'r$ を受け取って $'r$ が含むライフタイム変数全ての集合を返す写像 *Subsets* を返す。
- GET_REGION_REQUIRED_POINTS 関数は、上記の GET_REGION_INCLUDING_POINTS 関数と GET_ALL_SUBSETS 関数を使って計算された写像 *RegionPoints* と写像 *Subsets* を受け取り、ライフタイム変数 $'r$ を受け取ってライフタイム変数 $'r$ に関連した参照のいずれかのライフタイムに含まれるポイント全ての集合を返す写像 *RegionRequired* を返す。

Algorithm 2 は、上記 4 つの関数を用いて、ローカル変数 v を受け取って、その変数 v が有効であるべきポイントの集合 P を返す写像 *VarRequired* を返す。

なお、Algorithm 1, 2 において、各種写像を解析学等で標準的な対の集合として表している。また、写像の更新を

$$M[b \mapsto v] = \begin{cases} M \setminus \{(b, M(b))\} \cup \{(b, v)\} & \text{if } M(b) \text{ が定義されている} \\ M \cup \{(b, v)\} & \text{otherwise} \end{cases}$$

Algorithm 1 MIR 中でローカル変数が有効であるべき範囲を求めるために必要な写像を求めるための関数

```
function GET_REGION_INCLUDING_POINTS(ValidRegions)
    RegionPoints = { }
    for all (p, R) ∈ ValidRegions do
        for all 'r ∈ R do
            RegionPoints = RegionPoints['r ↦ RegionPoints('r) ∪ {p}]
        end for
    end for
    return RegionPoints
end function
function GET_REF2VAR(AllVars, Borrows)
    Ref2Var = { }
    for all v ∈ AllVars do
        for all b ∈ Borrows(v) do
            Ref2Var = Ref2Var[b ↦ v]
        end for
    end for
    return Ref2Var
end function
function GET_ALL_SUBSETS(PointSubsets)
    Subsets = { }
    for all (_, S) ∈ PointSubsets do
        for all ('r, R) ∈ S do
            Subsets = Subsets['r ↦ Subsets('r) ∪ R]
        end for
    end for
    return Subsets
end function
function GET_REGION_REQUIRED_POINTS(Subsets, RegionPoints)
    RegionRequired = { }
    for all ('r, R) ∈ Subsets do
        for all 'r_sub ∈ R do
            RegionRequired = RegionRequired['r ↦ RegionRequired('r) ∪ RegionPoints('r_sub)]
        end for
    end for
    return RegionRequired
end function
```

Algorithm 2 MIR 中でローカル変数が有効であるべき範囲を求めるアルゴリズム

```
1: function GET_VARIABLES_MUST_LIVE(AllVars, PoloniusOut)
2:     {ValidRegions, Borrows, PointSubsets, PointRegionBorrows} = PoloniusOut
3:     RegionPoints = GET_REGION_INCLUDING_POINTS(ValidRegions)
4:     Ref2Var = GET_REF2VAR(AllVars, Borrows)
5:     Subsets = GET_ALL_SUBSETS(PointSubsets)
6:     RegionRequired = GET_REGION_REQUIRED_POINTS(Subsets, RegionPoints)
7:     // ローカル変数 v から、その変数 v が有効であるべき MIR 中のポイントの集合 P への写像を求める
8:     VarRequired = { }
9:     for all (p, r2B) ∈ PointRegionBorrows do
10:        for all ('r, B) ∈ r2B do
11:            for all b ∈ B do
12:                v = Ref2Var(b)
13:                VarRequired = VarRequired[v ↦ VarRequired(v) ∪ RegionRequired('r)]
14:            end for
15:        end for
16:    end for
17:    return VarRequired
18: end function
```

で表す。

1 つの MIR の関数を対象とした Algorithm 1 の各関数の計算量について、その MIR の関数中の全てのポイントの集合を P、全てのライフタイムの集合を R、全てのローカル変数の集合を V、全ての借用の集合を B、全てのライフタイム制約の集合を S として考える。集合や写像のデータ型として B 木を用いることとすると、GET_REGION_INCLUDING_POINTS 関数の計算量は $O(|P||R|\log|R|\log|P|)$、GET_REF2VAR 関数の計算量は $O(|V|\log|V|)$、GET_ALL_SUBSETS 関数の計算量は $O(|P||S|(\log|R|)^2)$、GET_REGION_REQUIRED_POINTS 関数の計算量は $O(|R|^2(\log|R|)^3)$ である。

これらを利用した Algorithm 2 の計算量は、

$$O(|P||R|\log|R|\log|P| + |V|\log|V| + |P||S|(\log|R|)^2 + |R|^2(\log|R|)^3 + |P||R||B|\log|B|\log|V|\log|R|\log(|P||R|))$$

である。

5 実装

4 節に示したアルゴリズムを Rust コンパイラの一部として実装し、その出力を利用する形で LSP サーバを実装した。Rust コンパイラへのアルゴリズムの実装は、Rust コンパイラ rustc の内部 API を用いて実装され、所有権とライフタイムの解析を行い出力する以外、通常の Rust コンパイラとして振る舞う。解析結果は標準出力に JSON 形式で出力され、LSP サーバはこれを読み、サーバ内に結果を保存する。

5.1 RustOwl の構成

RustOwl の構成の概略を図 4 に示す。VS Code などのエディタはこの LSP サーバに、LSP の定義にはない独自のメソッドであるカスタムメソッドの呼び出しをリクエストし、リクエストを受け

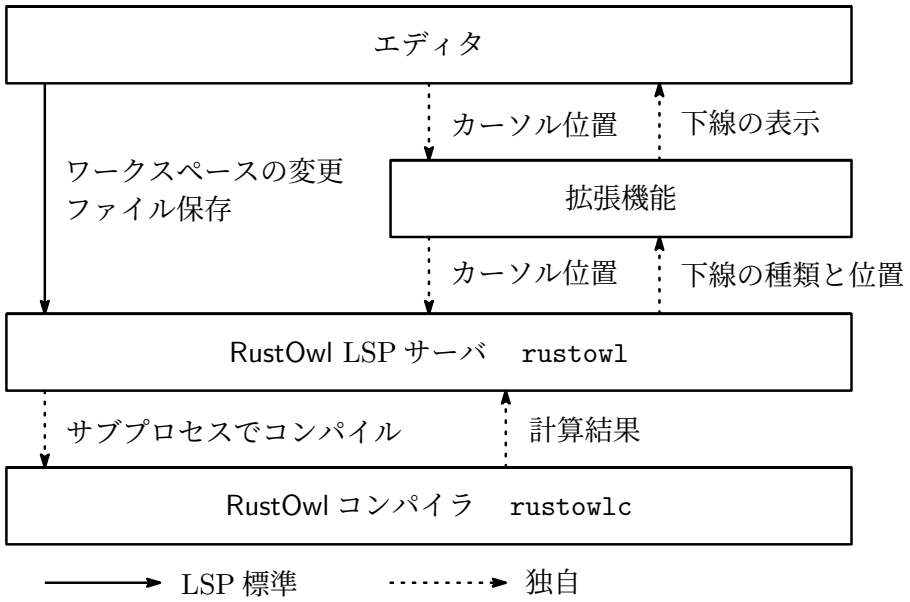

図 4. RustOwl の概略図

表 1. RustOwl が計算にかかった時間（秒）

クレート	RustOwl
bat (0.26.1)	9.36
ripgrep (15.1.0)	7.63
sccache (0.12.0)	18.2

表 2. `rust-analyzer` が計算にかかった時間（秒）

クレート	rust-analyzer
bat (0.26.1)	14.2
ripgrep (15.1.0)	8.43
sccache (0.12.0)	21.8

た LSP サーバがカーソルの位置から適切な解析結果を返し、エディタの拡張機能がこれをエディタ上に表示する。

Rust コンパイラおよび LSP サーバの実装は Rust で、VS Code の拡張機能は TypeScript で、Neovim の拡張機能は Lua で、Emacs の拡張機能は Emacs Lisp で実装した。GitHub の main ブランチで実装されている最新のコードは、2025 年 12 月 10 日現在、Rust コンパイラの実装行数は 1,085 行、LSP サーバの実装行数は 3,003 行、VS Code の拡張機能の実装行数は 564 行である。各エディタについて、3 に示した配色とは別に、ユーザが配色を設定するオプションが利用可能であり、色覚多様性にも配慮している。

これらの機能を持つソフトウェア全体を指して、ツール名を RustOwl としている。実装は `https://github.com/cordx56/rustowl` で公開している。

5.2 計算時間

広く利用されているパッケージリポジトリである `crates.io` において、それぞれライフタイムが異なる、ダウンロード数の多い実行可能クレート 3 つについて、計算にかかった時間を表 1、表 2 にまとめた。計測は zsh 標準の time コマンドを用いて、Ryzen 9 7950X3D、192GB RAM、Ubuntu 24.04 のマシン上で行った。RustOwl の行う計算は `rust-analyzer` が行う計算とは異なるが、計算にかかる時間は `rust-analyzer` と比較して短く、実用的な解析時間であると言える。

5.3 高速化手法

RustOwl では解析を高速化するため、マルチスレッディングとキャッシュを活用した。Rust ではスレッド間で安全に所有権の移動ができることを示す `Send` トレイトと、複数のスレッドからのアク

セスができることを示す Sync トレイトを型に実装することで、スレッド間で所有権を移動し、移動後のスレッド内でその変数にアクセスができるようになる。これはスレッド安全性を担保するための仕組みである。

RustOwl は Rust コンパイラ rustc の動的リンクライブラリである rustc_driver に依存している。現在、rustc では、マルチスレッド対応が部分的に進められているが、マルチスレッドに対応していない部分もある。このため、rustc 内部の一部のデータ型について、Send および Sync が実装されていても、他スレッドからアクセスした場合に異常終了することがある。RustOwl では、他スレッドで実行できない検査をスレッドローカルで実行し、これらの結果得られた値を RustOwl で定義したスレッド安全な型に変換する。4 節で提案したアルゴリズムを含む解析は、rustc で定義された型の値を RustOwl で定義した型の値に変換して、複数のスレッドで実行する。これにより、スレッド安全性に起因する問題を避け、全体の解析時間を短縮している。

コンパイル済みクレートのキャッシュは rustc で行われているが、ファイルに変更があった場合、そのファイルを含むクレートは再度コンパイルする必要がある。プログラマが編集しているクレートに依存しているクレートはキャッシュが利用できるが、プログラマが編集しているクレートはキャッシュされない。ある関数が記述されたファイルとその関数の MIR に変更がない場合は可視化を行う位置に変更は発生しないため、RustOwl ではファイルと MIR のハッシュ値をキーとして、ライフタイムの範囲の計算結果を JSON 形式にシリアライズしてファイルに保存し、キャッシュとして借用検査前に参照するようにし、実行時間を短縮した。

6 評価実験

本研究では RustOwl の有効性を示すために、既存の Rust 可視化ツールである Aquascope[1] との対照実験を、Web ブラウザ上でオンラインで行った。実験の参加者は 79 名で、そのうち中断が発生せず正常に実験が行われた有効な回答数は 72 件で、そのうち RustOwl を用いたグループが 36 名、Aquascope を用いたグループが 36 名であった。各グループへの参加者の割り当ては、両グループの数がほぼ同じになるよう、参加申込順で交互に振り分けた。各検定は RustOwl を用いたグループと Aquascope を用いたグループで得点の平均が変わらないことを帰無仮説とし、有意水準 5%で両側検定を行った。参加者の募集は SNS の X および Discord で日本語で行い、実験自体は英語で行った。参加者のうち 78 名が日本国内在住、1 名が南アフリカ在住であった。実験の参加者はまず Rust の習熟度、プログラミング経験年数、現在の職業等について回答し、その後問題に回答した。

6.1 実験手順

実験において用いた問題の Rust ソースコードと設問を図 5, 6 に示す。設問 q1 と q2 は所有権とライフタイムに関連するエラーの原因を問う問題、設問 q3、q4 はライフタイムに関係したデッドロックのデバッグに関して問う問題、設問 q5 はライフタイムに関係したメモリ使用量に関して問う問題である。なお、q4 は q3 が正答の場合は記入しないのが正解であるため、正答数の統計処理から外した。また、回答をスキップした場合は不正解として扱った。q1、q2 は rustc のエラー出力、q3、q5 は Ryzen 9 7950X3D、192GB RAM、Ubuntu 24.04 のマシン上での実行結果をもとに正答を決めた。

6.2 設問の得点による定量的な評価

実験に用いた Web ページや実験手順の詳細は A 節に示す。実験の結果について、ツール毎の正答数の合計を示すグラフを図 7 に、ツール毎に各問の正答率を示すグラフを図 8 に示す。また、RustOwl と Aquascope のそれぞれについての各問の正答率と、Cohen の d、および Welch の t 検定の p 値を表 3 に示す。表 3 から、RustOwl が Aquascope に比べて全設問の合計得点に有意差があった。この

```
fn split<'a>(src: &'a str, del: &str) ->
    (&'a str, &'a str) {
    let p = src.find(del).unwrap();
    src.split_at(p)
}

fn main() {
    let s = String::from("Hello, world!");
    let (head, mut tail) = split(&s, ",");
    if head.len() < 5 {
        let rep = s.replace("!", ".");
        tail = split(&rep, ",").1;
    }
    println!("{head} {tail}");
}
```

q1. Why does this program cause an error?

1. The variable `s` does not live long enough.
2. The variable `rep` does not live long enough.
3. The variable `tail` does not live long enough.
4. `&s` is an invalid reference.

正答：2

q2. On which line is the variable dropped?

正答：11 または 12

```
use std::sync::Mutex;

fn add(r: &mut i32, n: i32) {
    *r += n;
}

fn dead_lock() {
    let mutex = Mutex::new(0);
    let mut write = mutex.lock().unwrap();
    add(&mut *write, 3);
    drop(write);
    if 0 < *mutex.lock().unwrap() {
        println!("value: {}", *mutex.lock
            ().unwrap());
    }
}
```

q3. Does this program cause a deadlock?

正答：No

q4. If so, on which line does the program get stuck?

図 5. 実験において用いた問題とソースコードの一覧 1

表 3. ツールごとの各問および合計の正答率 (%)、Cohen の d、および Welch の t 検定の p 値

設問	Aquascope	RustOwl	d	p
q1	75.0	94.4	0.55	0.049
q2	69.4	86.1	0.40	0.156
q3	55.6	75.0	0.41	0.137
q5	69.4	88.9	0.49	0.082
合計	67.4	86.1	0.63	0.010

```rust
use std::{thread::sleep, time::Duration};

fn function1() {
    // make a large vector
    let v1: Vec<_> = (1..100000000).collect();
    // sleep 10 seconds
    sleep(Duration::from_secs(10));
    // make a large vector
    let v2: Vec<_> = (1..100000000).collect();
    // sleep 10 seconds
    sleep(Duration::from_secs(10));
    drop(v2);
    drop(v1);
}

fn function2() {
    // make a large vector
    let v: Vec<_> = (1..100000000).collect();
    // sleep 10 seconds
    sleep(Duration::from_secs(10));
    drop(v);
    // make a large vector
    let v: Vec<_> = (1..100000000).collect();
    // sleep 10 seconds
    sleep(Duration::from_secs(10));
    drop(v);
}
```

q5. Which function uses more memory?
正答：`function1`

図 6. 実験において用いた問題とソースコードの一覧 2

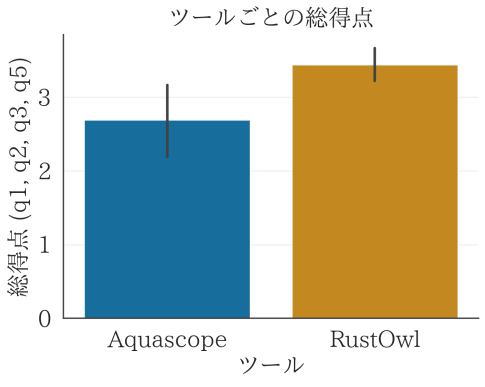

図 7. ツールごとの正答数（全 4 問）

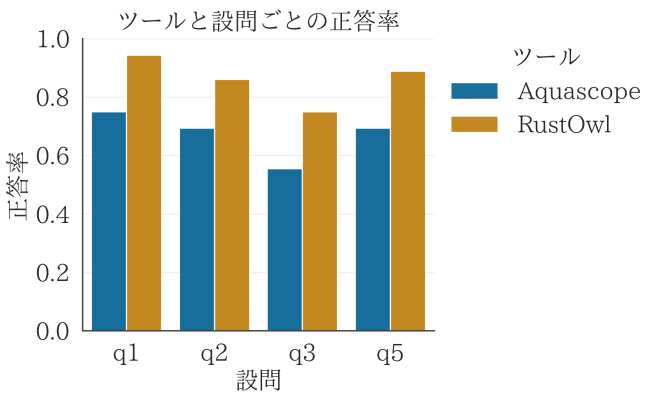

図 8. ツールごとの各問の正答率

表 4. ツール別の Rust 習熟度別の正答率 (%)、人数 N、Cohen の d、および Welch の t 検定の p 値

習熟度	Aquascope (N)	RustOwl (N)	d	p
First time	50.0 (8)	79.2 (6)	0.81	0.126
Beginner	77.5 (10)	80.8 (13)	0.13	0.778
Intermediate	67.3 (13)	93.2 (11)	0.87	0.037
Proficient	75.0 (5)	93.8 (4)	0.56	0.400
Expert	– (0)	87.5 (2)	–	–

表 5. 各設問の回答にかかった平均時間 (秒)、スキップせずに回答した人数 N、Cohen の d、および Welch の t 検定の p 値

設問	Aquascope (N)	RustOwl (N)	d	p
q1、q2	194 (31)	247 (36)	0.23	0.349
q3	180 (31)	208 (36)	0.22	0.374
q5	115 (32)	111 (36)	-0.03	0.910
合計	746 (36)	1020 (36)	0.40	0.091

ことから、RustOwl が所有権とライフタイムについて理解を促進すると言える。設問別では、設問 q1 で有意差があることが示され、RustOwl が所有権とライフタイムの両方が関連するタスクについて理解を促進する効果があると考えられる。

Aquascope と RustOwl のそれぞれについて、習熟度別の全問の正答率と、Cohen の d、および Welch の t 検定の p 値を表 4 に示す。Rust の習熟度が Expert レベルの参加者は極端に少なかったため、統計処理を行わなかった。表 4 から、RustOwl は Aquascope と比べて、中級者（Intermediate）の所有権とライフタイムについての理解度に有意差があり、またその効果も大きいことが示された。Rust を全く経験していない初心者であるグループ（First time）と実務レベルのグループ（Proficient）については有意差はなかった。

この実験では各タスクの回答にかかった時間を測定した。この測定結果を表 5 に示す。なお、q1 と q2 は同じソースコードに関する設問であり、2 つの設問の合計の回答時間を測定した。回答をスキップした場合は回答時間に含めていない。表 5 から、RustOwl と Aquascope の間で解答にかかる時間に有意差はみられなかった。

6.3 主観による評価

また、この実験ではツールに関する主観に基づく実験も行った。主観評価に関する設問と選択肢は以下の通りである。

- q6. How often do you use the tool when writing programs?
  1. I did not use it at all.
  2. I used it a little.
  3. I used it frequently.
- q7. Can the tool be used for debugging?
  1. Not at all.
  2. It worked sometimes.
  3. It was helpful for debugging.
- q8. Can the tool be used for memory optimization?
  1. Not at all.

表 6. 主観評価の平均値、Cohen の d、および Mann-Whitney の U 検定結果

設問	Aquascope	RustOwl	d	p
q6	1.89	1.83	-0.07	0.768
q7	2.11	2.33	0.30	0.192
q8	1.69	2.08	0.53	0.028
q9	1.17	1.08	-0.25	0.293

2. It worked sometimes.
3. It was helpful for memory optimization.

- q9. Do you find the tool helpful for understanding ownership or lifetimes?
  1. Yes
  2. No

各設問において、選択肢が等間隔であると仮定し、選択肢の番号を数値として、平均値、Cohen の d、および Mann-Whitney の U 検定を行った結果を表 6 に示す。表 6 から、設問 q8 では RustOwl と Aquascope で有意差があった。この結果から、RustOwl は Aquascope と比べてメモリ最適化においては RustOwl が有効であることが示された。設問内容と表 3 で示された結果からは、所有権とライフタイムの理解の促進について有意差があり、これは主観評価と異なる結果となった。一方、この主観評価からは RustOwl がこれまで初心者の教育目的とされてきた他の可視化ツールとは異なり、メモリ最適化という実務への応用が期待できる。

実験結果をまとめると、以下のようなことが言える。

- RustOwl は Aquascope と比べて、所有権とライフタイムの理解を促進すると考えられる。
- 習熟度別では中級者において RustOwl は Aquascope と比較し、所有権とライフタイムの理解を促進する。
- 主観評価では、RustOwl はメモリ最適化に有効である。

7 関連研究と議論

Rust の開発支援において、所有権については主に学習目的で、ライフタイムについては主にデバッグ目的で議論されてきた。

ライフタイムの可視化による VRLifeTime[3] は MIR を用いて、VS Code 上で変数のライフタイムを可視化するツールとして実装されている。この可視化はデッドロックの解決に重点を置いており、ハイライトとは別に、デッドロックが発生しうる箇所に警告を表示する。Qin らは Rust の排他ロックのバグについて、Rust のライフタイムに関する理解の欠如が原因であることを指摘した [2]。また、Qin らは将来の IDE では排他ロックの解除をハイライトする機能を備えるべきであると主張している。

Dominik は Polonius API を用いてライフタイムの制約を有向グラフで表現する手法を考案し実装した [4]。この研究では NLL が考慮されている。また、色付きの縦棒をソースコードの左側に表示させる可視化についても考案している。RustViz[5] は借用やムーブを、色付きの矢印をソースコード上に表示して可視化するツールとして教育目的で提案、実装されている。RustViz は教育者が Rust のコードに独自の DSL で注釈を記述することで可視化を行っている。この可視化はムーブの可視化を明確に行うことで、所有権の移動を追いやすくした。Christian は Rust 学習者向けに所有権の動きをコード上に色付きの矢印を追加して可視化する手法を提案し、BORIS というツールとして実装した [6, 14]。この研究では定性的な評価も行っており、Rust 初心者に高く評価される傾向があることが示された。また、このツールの現状の制約として、ライフタイム注釈、クロージャ、内部

可変性、async コードなどへの対応が十分でないことが挙げられている。Rust においては変数が可変であるかどうかが借用検査において重要であるが、内部可変性はそれらの検査を動的に行い、コンパイル時に行わないようにするための手法である。Crichton らは Rust における所有権と借用の可変性、およびメモリ上のデータの可視化を行い、Rust の所有権をモデル化し、また所有権がどのようにして不正なメモリ参照を防ぐかに関する学習を支援するツールである Aquascope を実装し、効果があることを示した [1]。このツールは 6 節の対照実験において用いた。

既存の実用的なエディタ上での可視化ツールとして、JetBrains 社の RustRover[15] が挙げられる。RustRover はエラー発生時にエラーの原因となる参照の範囲をソースコードの左に縦の色付き棒で可視化する。

VRLifeTime[3] では、明示的にデッドロックの警告を行っていたが、本研究ではロックオブジェクトを持つ変数のライフタイムのみを表示し、判断はプログラマに任せた。6 節の実験では、Rust 初学者には排他ロックの解放がロックオブジェクトがドロップされるタイミングで行われるということが分からなかったという趣旨のコメントが寄せられており、デッドロックに起因するバグの修正では VRLifeTime のような明示的な警告が必要かもしれない。

6 節の実験では、Aquascope の可視化が所有権の理解に分かりやすいという意見が寄せられたが、RustOwl の可視化は初心者には分かりづらいという意見も寄せられた。RustOwl はエディタ上で実用的な可視化を行うために色付きの下線を用いたが、RustViz や BORIS、Aquascope に比べて分かりづらく、学習には適さない可能性も考えられる。

RustRover における可視化は、借用検査の結果コンパイルエラーが発生するようなコードのライフタイムに限られる。6 節の実験では、RustOwl が所有権とライフタイムの両方の理解を問うタスク q1 において Aquascope との有意差があるが、RustRover については Web ブラウザ上で実行できないので、今回の実験の対象にしなかった。RustOwl の RustRover プラグインは開発コミュニティによって開発されており、`https://github.com/siketyan/intellij-rustowl` で公開されている。

8 まとめと将来課題

本研究では、Rust におけるライフタイムのコード上の範囲を求めるアルゴリズムを提案し、所有権とライフタイムを実用的なエディタ上で可視化するツールである RustOwl を実装した。

RustOwl は既存の可視化ツールとの対照実験で所有権とライフタイムの理解を促進することを示した。また、Rust の習熟度別では Rust 中級者で効果が示され、従来初学者向けであった所有権とライフタイムの可視化が中級者にも有効であることを示した。

今後の課題として、以下に挙げるようなものが考えられる。

- ライフタイム範囲を求めるアルゴリズムの妥当性評価
- 実用的な Rust プロジェクトでの利用についての評価
- RustOwl の RustRover プラグイン利用者へのヒアリング

また、評価実験は短いコードと短い利用時間での実験であり、実際の開発現場の利用についての効果が不明確であることと、初心者に対して高い難易度、上級者に対して低い難易度の設問であったことを考慮する必要がある。これらについて対策した実験の実施については将来課題とする。

謝辞

本研究は JST 次世代研究者挑戦的研究プログラム JPMJSP2168 の支援を受けたものである。Reddit ユーザーの sekhat 氏は、LSP サーバーの実装にあたりカスタムメソッドの存在を指摘され、LSP サーバーとして実装が可能となった。Reddit ユーザーの BoaTardeNeymar777 氏は、RustRover

IDE がライフタイム可視化を行うことを指摘していただいた。GitHub ユーザーの uhobnil 氏は、ライフタイムの範囲の解析において、全く同じ処理を行っているコードが 2 箇所あることを指摘し、また重複を除去するパッチを送っていただき、実行効率向上に繋がった。Avi Fenesh 氏、Muntasir Mahmud 氏はキャッシュおよびマルチスレッディングの適用範囲を拡大するための提案をされ、それらのアルゴリズムを実装することで実行効率向上に繋がった。Naoki Ikeguchi 氏は、RustOwl の RustRover プラグインを開発された。

参考文献

[1] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. A grounded conceptual model for ownership types in Rust. *Proc. ACM Program. Lang.*, Vol. 7, No. OOPSLA2, 2023.

[2] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, p. 763–779, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Ziyi Zhang, Boqin Qin, Yilun Chen, Linhai Song, and Yiying Zhang. VRLifeTime – an ide tool to avoid concurrency and memory bugs in Rust. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, p. 2085–2087, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Dietler Dominik. Visualization of lifetime constraints in Rust. Bachelor's thesis, ETH Zürich, 2018.

[5] Marcelo Almeida, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, and Cyrus Omar. RustViz: Interactively visualizing ownership and borrowing. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10, 2022.

[6] Schott Christian. Visualizing ownership and borrowing in Rust programs. Master's thesis, Julius-Maximilians-Universität Würzburg, 2024.

[7] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, p. 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, Vol. 2, No. POPL, dec 2017.

[9] The Rust programming language. `https://doc.rust-lang.org/book`. Referencing Dec 24, 2025.

[10] Non-lexical lifetimes (NLL) fully stable. `https://blog.rust-lang.org/2022/08/05/nll-by-default/`. Referenced on Aug 6, 2025.

[11] Validating references with lifetimes. `https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html\#the-borrow-checker`. Referencing Dec 15, 2025.

[12] 2094-nll - the Rust RFC book. `https://rust-lang.github.io/rfcs/2094-nll.html`. Referencing Nov 11, 2024.

[13] Lifetimes - Rust By Example. `https://doc.rust-lang.org/rust-by-example/scope/lifetime.html`. Referencing Dec 24, 2025.

[14] Christianschott/boris: Visualizing ownership and borrowing in Rust programs. `https://github.com/ChristianSchott/boris`. Referencing Jan 31, 2025.

[15] Rustrover: Rust ide by jetbrains. `https://www.jetbrains.com/rust/`. Referencing Dec 10, 2025.

A　実験ページの詳細

実験の参加者は図 9 に示した画面上で、画面右下に表示された RustOwl と Aquascope それぞれ 10 行程度のツールの説明を見ながら、画面左側の問題の指示に従い、画面右上のエディタを操作しながら、

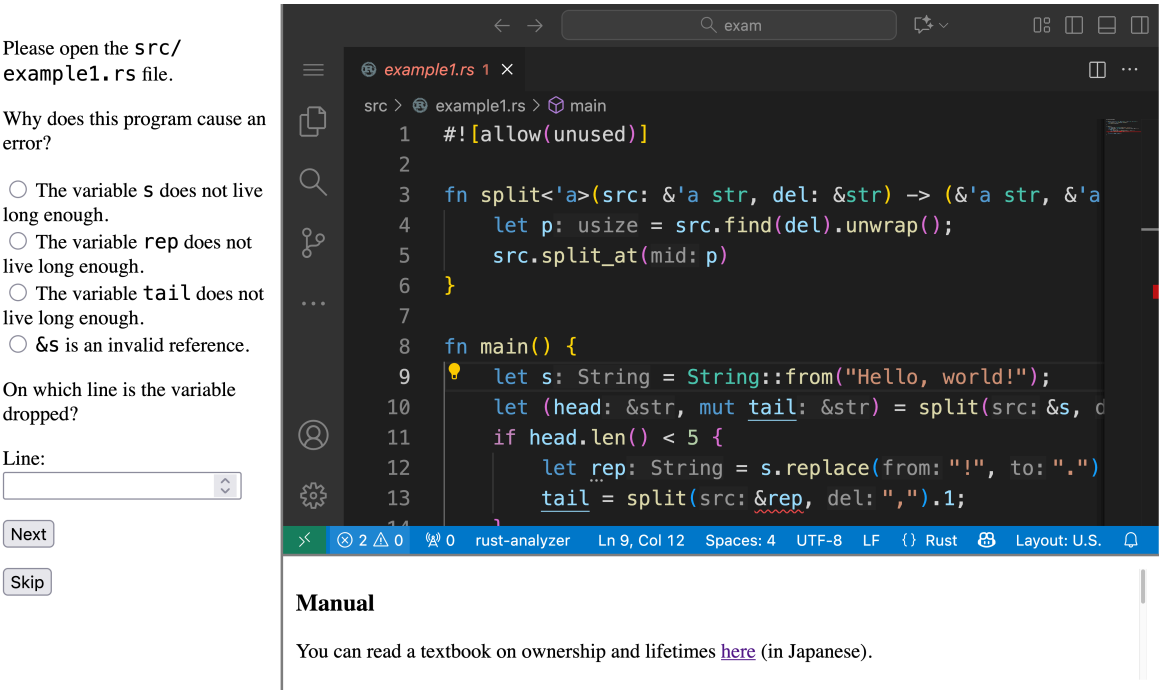

図 9. 実験ページの画面

問題に回答した。画面右下のツールの説明は、図 9 中では途切れているが、参加者はスクロールして説明を読むことができた。また、Rust に詳しくない参加者のために、日本語で所有権とライフタイムの教材を用意した。この教材は `https://github.com/56research/rustowl-experiment/blob/859edc0582e00cfe8ed1e4f75c47de1465695d33/book.md` で公開している。このエディタはブラウザ上で動作する OSS 版 VS Code である code-server を用いており、RustOwl グループは Aquascope が利用できず、RustOwl が利用可能な状態で回答を行った。Aquascope はエディタ上では動作しないため、Aquascope グループには Aquascope の playground のサイトにソースコードをコピーして貼り付ける形で利用するよう指示した。また、両グループとも、Rust 開発者に広く利用されている LSP サーバーである rust-analyzer もエディタ上で併用可能とした。この実験は 30 分を回答の上限時間とした。