実行時オーバーヘッドなくアクセスできる循環参照を構築するための Rust 言語の拡張にむけて

坂本 洸亮　山崎 徹郎　千葉 滋

本論文では，実行時オーバーヘッドなくアクセスできる循環参照を構築するための Rust 言語の拡張を提案する．現状の Rust では，循環参照の構築と安全なゼロコストアクセスの両立が困難であり，循環参照の構築を妨げる静的検査を回避するために動的検査を用いる既存手法では，構築後もそのコストを受け入れる必要がある．提案手法では，構築した循環参照を動的検査等による安全性保証の下で静的検査の管理下に移すことで目的を実現する．ユーザーは静的検査を回避する新しい種類の参照を利用できるサンドボックス内で循環参照を自由に構築でき，構築した循環参照を外に取り出して静的検査の下で利用する．取り出しに際し，静的検査の管理下で扱えるようにするための動的検査を行い，以降のアクセスの安全性を静的に保証できるようにする．構築した循環参照は不変参照のみでアクセスできるようにすることで，Rust の静的検査の下で安全かつ実行時オーバーヘッドなく扱える．

1　はじめに

Rust は，安全で高速なシステムの開発に適したプログラミング言語として近年注目を集めている．Rust は AXM 規則やライフタイム規則と呼ばれる独自の規則体系に基づく静的検査によって，ダングリング参照や不正なメモリアクセスを防ぐことができ，プログラマの注意に依存しない安全性保証を実現する．一方，Rust の通常の参照を用いた循環参照の構築には，すでに参照されているオブジェクトの書き換えや，循環を構成する参照への共通のライフタイムの割り当てが必要となり，Rust の静的検査が柔軟な構築を妨げる．`RefCell` 等の利用によって検査を実行時に移す方法では安全に循環参照を構築できるが，構築後にもアクセスごとの動的検査が残り，実行時オーバーヘッドがかかる．また，生ポインタを用いて静的検査を回避する方法では，安全性の保証がプログラマに委ねられる．

本研究は，一度構築した循環参照を含むオブジェクトグラフを繰り返し読み取る処理を対象とし，循環グラフの柔軟な構築と，安全かつアクセスごとの実行時オーバーヘッドを伴わない利用を両立させることを目的とする．このような処理では，構築中の操作に Rust の通常の規則を適用しない場合でも，読み取り処理の開始前にグラフ全体の安全性を一度だけ検査し，その後のグラフに対する危険な操作を禁止することができれば，アクセスごとに動的検査を繰り返す必要はない．

これに基づき，本論文では通常の Rust とは異なる規則とメモリ管理を適用するサンドボックスを Rust に導入する言語拡張を提案する．構築中は Rust 規則を適用しない新しい種類の参照を用いるサンドボックス内で循環グラフを構築する．また，構築したグラフをサンドボックス外に取り出す操作を提供し，取り出したグラフをアクセスごとの動的検査なしに走査できるようにする．取り出し時はグラフ全体を走査して安全性を動的に検査し，検査を通過したグラフを通常の不変参照を介して扱えるようにする．

本論文の貢献は以下のとおりである．

- AXM 規則とライフタイム規則を適用しない `&ind` 参照とガベージコレクションを備え，循環

Towards Extending Rust to Construct Cyclic References Without Runtime Access Overhead

Kosuke Sakamoto, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

参照を柔軟に構築できるサンドボックスを導入した Rust 言語拡張を設計した．

- サンドボックスからのグラフの取り出し時の動的検査とガベージコレクション，および Rust の静的検査を組み合わせることで，構築した循環グラフを通常の不変参照で安全に利用し，アクセスごとの動的検査を不要にする方式を示した．
- 提案する言語拡張の動作を実行可能な形で検証するための，改造した Miri，プリプロセッサ，および専用ライブラリからなるプロトタイプの実装方針を示した．

2 Rust における循環参照の構築とアクセス

以下では，メモリ上に配置され参照の対象となりうる値をオブジェクトと呼ぶ．また，オブジェクトを頂点，オブジェクトが保持する別のオブジェクトへの参照を辺とみなした構造をオブジェクトグラフと呼ぶ．このグラフに，あるオブジェクトから参照を順に辿ることで再び同じオブジェクトへ戻る経路が含まれるとき，その経路を構成する参照関係を循環参照と呼び，循環参照を含むオブジェクトグラフを単に循環グラフと呼ぶ．

通常の Rust において，オブジェクトを通常の参照で結ぶことにより循環参照を含むオブジェクトグラフを柔軟に構築することはできない．これは，循環参照の構築に Rust の静的検査が課す規則を満たせない操作が必要になるためである．この問題を回避するため，静的検査による安全性保証を動的検査によって置き換える手法が存在し，この機能を提供する API を利用することで循環を含むデータ構造の実装が可能となっている．

しかし，一度構築したグラフに頻繁かつ高速にアクセスする必要があるアプリケーションでは，通常の不変参照では不要な，参照先への各アクセスで行われる動的検査による実行時オーバーヘッドが問題となる．一方で，不変参照の参照先の書き換えやダングリング参照といった危険な状態が発生しないことも求められる．例えば，正規表現から循環を含む有限オートマトンを一度構築し，同じ状態遷移グラフを繰り返し辿って複数の入力文字列を照合する正規表現エンジンのようなケースでは，一度構築したオートマトンを多数の入力文字列に適用するため，グラフの構築よりも照合処理が頻繁に実行される．また，正規表現照合は入力値の検証やフィルタリング等の判定処理として利用されるため，誤った照合結果や照合途中での処理の中断は，その結果を利用する後続の処理にも影響する．

以降では，Rust で通常の参照を用いて循環参照を構築する際の具体的な課題，および現状の回避策の課題について述べる．

2.1 Rust の静的検査による循環参照構築の制限

通常の参照を用いて循環参照を構築する際，循環を閉じるためにすでに参照されているオブジェクトを書き換える必要があるが，この操作は Rust の Aliasing XOR Mutability（AXM）規則によって拒否される．AXM 規則は，同じオブジェクトに対して一つ以上の有効な不変参照が存在するか，単一の有効な可変参照が存在するかのいずれか一方だけを許可する [3]．すなわち，不変参照が存在する間に可変参照を介した書き込みを行うことは禁止される．これにより，不変参照を介してオブジェクトが利用されている最中に，別の経路からその内容が書き換えられることを防ぐ．

例えば図 1 は，通常の参照を用いて循環参照を構築する Rust プログラムの例である．二つのオブジェクトを相互に参照させる場合，一方のオブジェクトへの不変参照を作成して他方に格納した後（13 行目），最初のオブジェクトを書き換えて逆向きの参照を格納する必要がある（14 行目）．この書き換えには最初のオブジェクトへの可変参照が必要だが，そのオブジェクトへの不変参照はすでに他方のオブジェクトに格納されており，両者は同時に存在できない．Rust コンパイラは AXM 規則に従い，循環を閉じるためのオブジェクトの書き換えを拒否するため，この方法で循環参照を構築することはできない．

また，循環グラフを構築する典型的な方法として，異なる関数が返す部分グラフを相互に接続することが考えられる．通常の参照を用いる限りこの実装にも AXM 規則に違反する操作が必要となるが，仮にこの問題を回避できたとしても，Rust のもう一つの静的検査基盤であるライフタイム規則に起因する別の制

```
1  struct Node<'a> {
2    value: i32,
3    next: Option<&'a Node<'a>>,
4  }
5
6  fn main() {
7    let mut a = Node {
8      value: 0, next: None
9    };
10   let mut b = Node {
11     value: 1, next: None
12   };
13   a.next = Some(&b);
14   b.next = Some(&a);
15 }
```

図 1　通常の Rust ではコンパイルエラーになる循環参照を構築するプログラムの例

約が残る．Rust においてライフタイムは参照が有効でありうるプログラム上の範囲，すなわち参照が作成されてから最後に利用されるまでの期間を指す．ライフタイム規則は，この期間中に参照先のオブジェクトが生存し続けることを要求し，無効な参照先を指し続けるダングリング参照の発生を防ぐ [3]．この条件を満たすかどうかは，オブジェクトの所有者の破棄がメモリの解放に対応することから静的に計算できる．

循環参照がライフタイム規則を満たすためには，それを構成する参照へ共通のライフタイムを割り当てる必要があるが，それをプログラムに反映させるためには複雑な設計が要求され，循環グラフの柔軟な構築が妨げられる．循環グラフを構成するオブジェクト i がオブジェクト $i+1$ を参照するとき，その参照のライフタイムを L_i と表すことにする．オブジェクト $i+1$ が L_i の間有効ならば，オブジェクト $i+1$ が保持するオブジェクト $i+2$ への参照も L_i の間有効でなければならない．したがって，$L_i \subseteq L_{i+1}$ が要求される．この関係を循環参照に対して繰り返し適用することにより

$$L_1 \subseteq L_2 \subseteq \cdots \subseteq L_n \subseteq L_1$$

が要求されることになる．したがって，循環参照を構成するすべての参照を有効に保つには，それらが同一のライフタイムを持つ必要がある．

この制約から，部分グラフの相互接続による循環グラフの構築について，各部分グラフを構成する参照のライフタイムはすべて同一でなければならず，異なるライフタイムの参照を含む部分グラフをそのまま接続することはできない．また，各部分グラフの参照先が共通のライフタイム全体で有効になるようにする必要があるため，部分グラフの型，関数のシグネチャ等のライフタイムパラメータを適切に設計しなければならない．加えて，オブジェクトの確保方式や所有者等のメモリ管理の側面についても併せて設計する必要があり，プログラムが煩雑になる．

2.2　静的検査を回避する既存手法

Rust の静的検査を回避することで循環参照を構築できるようにする既存手法として，`RefCell` を用いて借用検査を実行時に行う方法と，生ポインタを `unsafe` コード内で操作する方法がある．`RefCell` は，内部に保持した値への読み取りと書き込みの整合性を実行時に管理する型である [5]．内部の値が読み取り中または書き込み中であるかを記録し，複数の読み取りは同時に許可する一方，書き込みは他の読み取りや書き込みが行われていない場合にのみ許可する．この条件を満たさないアクセスが要求された場合には実行時エラーを発生させることで，同じ値に対する読み取りと書き込みが同時に行われることを防ぐ．

この仕組みにより，同じオブジェクトが複数の箇所から参照されている場合でも，そのオブジェクトがアクセス中でなければ内部の値を変更できるため，オブジェクト間を相互に接続して循環参照を構築することができる．一方，`RefCell` を用いて構築したグラフには，構築後も読み書きのための専用のメソッドを介してアクセスする必要がある．そのため，グラフを不変に走査するだけの処理でもアクセスのたびに借用状態の検査が行われ，実行時オーバーヘッドが発生する．

生ポインタを用いる方法では，オブジェクト間の参照関係を `*const T` や `*mut T` として保持し，`unsafe` コード内で参照先を読み書きする．生ポインタには AXM 規則やライフタイム規則が適用されないため，複数の生ポインタでオブジェクトを相互に接続して循環を構築することができる．また，`RefCell` のよう

な借用状態の記録やアクセスごとの動的検査を必要としないため，構築後のグラフを追加の実行時オーバーヘッドなく辿ることができる．

ただし，生ポインタの参照先が有効であることや，生ポインタを介した読み書きが競合しないことはRustコンパイラによって保証されない．参照先のオブジェクトがムーブや破棄等で無効になった後も生ポインタが残っている場合にはダングリングポインタとなり，そのデリファレンスは未定義動作となる．そのため，プログラマがオブジェクトの所有者や確保先，破棄のタイミング等を管理し，生ポインタを利用するすべての箇所で安全性を保証しなければならない．

3 サンドボックスによる循環参照の構築と不変参照によるアクセス

循環参照を含むオブジェクトグラフについて，その構築と読み取り専用での利用を明確に分離できる場合を考える．このとき，利用開始前にグラフ全体が安全性に必要な条件を満たすことを一度だけ検査して，その条件を損なう利用中の変更を禁止できれば，構築中はRustの規則に従わなくても利用中は安全性を保ちながらアクセスごとの動的検査を不要にすることができる．この着眼点に基づき，本論文では循環参照を構築し不変参照でアクセスできるようにするRust言語の拡張を提案する．これにより，実行時オーバーヘッドなくアクセスできる循環参照を柔軟に構築することが可能となる．

3.1 サンドボックスと `&ind` 参照によるオブジェクトグラフの構築

提案手法では，通常のRustが要求する規則に従わない特別なメモリ空間の中で循環グラフを柔軟に構築できる．本論文では，この特別なメモリ空間をサンドボックスと呼び，サンドボックスの外はRustの規則に従う通常のメモリ空間であると考える．

オブジェクトグラフの構築中は，サンドボックス内のオブジェクトは新しい種類の参照 indeterminate reference（`&ind T` と表記するが，以降では単に `&ind` 参照とも表記する）でのみ参照することができる．サンドボックス内のオブジェクトへのアクセスは，サンドボックス外に保持された参照を起点として，到達したオブジェクトが保持する参照を順に辿ることによってのみ行える．

ただし，サンドボックス内に確保されるオブジェクトの型定義では `&ind` 参照ではなく不変参照を用いる．`&ind` 参照を介してオブジェクトのフィールドへアクセスする場合，フィールドの型に含まれる通常の参照は再帰的に `&ind` 参照として扱う．すなわち，型定義上は不変参照で表現されているフィールドでも，サンドボックス内からのアクセスでは `&ind` 参照を用いた表現に読み替える．例えば `Option<&'a U>` 型のフィールドは，`&ind` 参照を介したアクセスでは `Option<&ind U>` 型として扱われる．一方，後述のようにオブジェクトをサンドボックスから取り出し不変参照を介してアクセスする場合は読み替えを行わない．この規則はフィールドの読み出しと書き込みの双方に適用され，型定義上は不変参照を含むフィールドに `&ind` 参照を介して `&ind` 参照を書き込むことができる．また，そのフィールドを `&ind` 参照を介して読み出した結果も `&ind` 参照となり，不変参照として取り出すことはできない．

Rustの通常の参照と異なり `&ind` 参照にはAXM規則が適用されないため，それを用いて循環参照を構築することができる．図2は実際に `&ind` 参照を利用して循環参照を構築するプログラムの例である．`ind_alloc` はサンドボックス内にオブジェクトを確保し，それを指す `&ind` 参照を返す関数である．16行目の代入操作では有効な参照が存在する状態での書き込みが必要となるが，`&ind` 参照に対してAXM規則に関する静的検査は行われないためコンパイルエラーとならない．

また，`&ind` 参照にはライフタイム規則も適用されないため，循環グラフの柔軟な構築が可能である．2.1項で述べたように，通常の参照を用いて複数の部分グラフから循環グラフを構築する場合，部分グラフの接続には制限があるほか，循環参照を構成する参照に共通のライフタイムを持たせる必要があり，型や関数およびメモリ管理の複雑な設計が要求される．本手法では，ライフタイムに制限のない `&ind` 参照を利用することで，循環参照を構成する参照のライフ

```
struct Node<'a> {
  value: i32,
  next: Option<&'a Node<'a>>,
}

let a: &ind Node = ind_alloc(Node {
  value: 0,
  next: None,
});
let b: &ind Node = ind_alloc(Node {
  value: 1,
  next: None,
});

a.next = Some(b);
b.next = Some(a);

// サンドボックスからグラフを取り出す
let agent = publish(a);

// agent が保持する不変参照を得る
let a_: &Node = agent.get();

let b_: &Node = a_.next.unwrap();
let a_re: &Node = b_.next.unwrap();
assert!(a_.value == a_re.value);

// サンドボックスへグラフを戻す
unpublish(agent);
```

図 2　&ind 参照を利用して循環グラフを構築しサンドボックスから取り出すプログラムの例

タイムやオブジェクトの確保先をあらかじめ考慮することなく循環グラフの構築を容易に行える．なお，この性質から &ind 参照は通常の参照と異なりライフタイムパラメータを構文上持たない．

サンドボックス内のオブジェクトはガベージコレクションで管理される．通常の Rust では，オブジェクトの所有者がスコープを抜けると自動的にメモリが解放される．一方，&ind 参照には所有権やライフタイムがなく，同じオブジェクトへの &ind 参照を複数のオブジェクトが保持できるため，変数がスコープを抜けたことを根拠としてオブジェクトを解放すると，別のオブジェクトに残った &ind 参照がダングリング参照になりうる．本手法では，このような &ind 参照の利用を制限せず，かつプログラマに明示的な解放を要求せずにオブジェクトを回収するため，到達可能性に基づくガベージコレクションを採用する．ガベージコレクションでは，サンドボックス外に保持されている有効な &ind 参照をルートとして，オブジェクト間の参照を再帰的に辿る．これにより，有効な &ind 参照から到達可能なオブジェクトを生存させる一方，いずれのルートからも到達不能になったオブジェクトを循環参照の有無にかかわらず回収できる．また，循環グラフの構築でグラフ全体に同じ寿命を持たせる必要はなく，到達不能になった部分グラフを回収できる．

3.2　publish と Agent によるオブジェクトグラフの取り出し

サンドボックス内で構築したオブジェクトグラフは，不変参照を通じてサンドボックス外から利用可能にすることができる．この操作を publish と呼び，対象となるオブジェクトグラフの根への &ind 参照を指定して実行する．グラフを publish すると，グラフへの不変参照を直接得る代わりに，それを保持するオブジェクト Agent が得られる．Agent はサンドボックス外でグラフを扱うためのハンドルとして振る舞い，そこからグラフへの不変参照を得ることができる．なお本論文では publish によってサンドボックス外からオブジェクトグラフを利用できるようにすることを，サンドボックスからのグラフの取り出しと表現する．また，それによってグラフ中のすべてのオブジェクトが同時に取り出されると考える．ただし publish はオブジェクトのメモリ上の移動やコピーを伴わず，グラフの取り出しはメモリ空間の間のグラフの移動を意味しない．

図 2 では，提案手法が提供する組み込み関数 publish を用いて循環グラフをサンドボックスから取り出す例を示す．この例では，サンドボックス内で構築した循環グラフを publish して取り出し，Agent から得た不変参照を介してグラフ内部にアクセスしている．3.1 項で述べたように，サンドボックス内では型定義中の不変参照を &ind 参照に読み替えるが，Agent から得た不変参照を介したフィールドアクセスでは読み替えは行われず，グラフの走査には &ind 参照を利用しない．

publish されたオブジェクトグラフのメモリは

publish 後もガベージコレクションによって管理される．ガベージコレクションでは，publish 時に作られた Agent をルート集合に追加することで，そこから得られた不変参照が利用されうる間はグラフ全体を生存させる．これは，ガベージコレクションのルートとしてサンドボックス外に保持されている有効な &ind 参照の参照先のみを考えると，Agent が利用されている間に &ind 参照が破棄され，publish されたグラフが到達不能と判断され解放されてしまう可能性があるためである．

Agent が破棄されるとグラフはガベージコレクションのルートから外れるが，他の &ind 参照から到達可能なオブジェクトは引き続き保持され，いずれのルートからも到達不能になったオブジェクトだけが回収される．したがって Agent は対応するグラフを排他的に所有するものではなく，それが有効な期間とグラフの生存期間は必ずしも一致しない．例えば部分グラフを publish して得た Agent が破棄されても，元のグラフへの有効な &ind 参照が残っていればその部分グラフは回収されず，再び元のグラフ全体の publish 等を行うことができる．

また，提案する言語拡張では一度サンドボックス外に取り出されたオブジェクトグラフを再びサンドボックス内に戻す操作 unpublish も提供する．図 2 の 29 行目のように，unpublish は対象のグラフに対応する Agent を消費して所有権を移動（ムーブ）し，それに紐付けられたグラフをサンドボックス内に戻すことができる．unpublish によってグラフは再び変更可能となるため，グラフへの不変参照が利用中の場合安全性が損なわれる可能性がある．このため，unpublish の実行時には Agent をムーブすることでグラフへの利用中の不変参照がないことを静的に検査する．Agent から得た不変参照が利用中の場合は Rust の静的検査がムーブを拒否するため，利用中のグラフが変更可能になることを防ぐことができる．なお Agent の破棄時にはこの操作に相当する処理が行われる．すなわち，Agent が破棄され不変参照を介した利用が終了すると，グラフはサンドボックス内に戻り再び &ind 参照を介した書き込みができるようになる．

3.3 publish 時およびその後の動的検査による安全性保証

提案する言語拡張では，publish 時にオブジェクトグラフに対して行う動的検査，および publish 後のグラフへの書き込みに対する動的検査によって，publish 後の循環参照を通常の Rust 規則に従いながら利用できるようにする．publish 時の検査では対象のグラフを走査し，安全性を保証できないような参照やオブジェクトがグラフに含まれていないこと，およびすでにサンドボックス外に取り出されたオブジェクトが含まれていないことを確認する．検査を通過した場合，グラフに含まれるすべてのサンドボックス内オブジェクトに対し，その後の &ind 参照経由での書き込みを動的検査により禁止する．

publish 時に対象のオブジェクトグラフ中にサンドボックスの中から外への参照が含まれる場合は，ダングリング参照の発生を防ぐため原則実行時エラーを発生させる．&ind 参照にはライフタイム規則が適用されないため，サンドボックス内のオブジェクトがサンドボックス外のオブジェクトを参照する際に制約は課されない．そのため，サンドボックス外の参照先が Agent の利用中を通して有効であることを保証できず，そのまま publish するとグラフ中にダングリング参照が生じうる．

ただし，サンドボックスの中から外への参照について，対応する不変参照を publish の実行時に明示的に指定した場合は実行時エラーを発生させない．これは，指定された不変参照のライフタイムを Agent に引き継ぐことで，Agent がその参照より長く利用されないことを Rust のライフタイム規則によって保証できるためである．Agent を利用できる期間は渡されたすべての不変参照が有効な期間に制約され，その間の参照先のムーブや破棄は Rust の静的検査によって拒否される．

例えば図 3 において，20 行目でコメントアウトされている publish は実行時エラーとなる．これは，root から辿れるオブジェクト child がサンドボックス外のオブジェクト out への参照を保持しているためである．対して 23 行目の publish では out への不変参照を引数として渡しており，Rust の静的検査

```
1  struct StrNode<'a> {
2    string: Option<&'a String>,
3    next: Option<&'a StrNode<'a>>,
4  }
5
6  // サンドボックス外のオブジェクト
7  let mut out = String::from("out");
8
9  let root: &ind StrNode =
       ind_alloc(StrNode {
10   string: None,
11   next: None,
12 });
13 let child: &ind StrNode =
       ind_alloc(StrNode {
14   string: Some(&out),
15   next: None,
16 });
17 root.next = Some(child);
18
19 // この publish は実行時エラー：
20 // let agent = publish(root);
21
22 // サンドボックス外への参照を明示的に渡す
23 let agent = publish(root, &out);
24
25 // この操作はコンパイルエラー：
26 // out = String::from("dangling");
27
28 let root_: &StrNode = agent.get();
29 let child_: &StrNode =
       root_.next.unwrap();
30 println!("{}",
       child_.string.unwrap());
```

図 3　サンドボックス外への参照を含むグラフを publish するプログラムの例

によりその後の Agent の利用中に out に変更を加える操作が禁止される．

また，publish するオブジェクトグラフに内部可変性パターンに従う型のオブジェクトが含まれる場合も実行時エラーを発生させる．Rust において，あるオブジェクトに不変参照が存在する場合でもそのデータを変更できるようにするデザインパターンを Interior Mutability（内部可変性）と呼び [5]，Cell や RefCell などはこれを使用している．以下では，内部可変性パターンを使用して実装されている型を単に内部可変な型と呼ぶ．

内部可変な型のオブジェクトでは不変参照を通じて書き込みを行うことが可能であり，publish 後に Agent から得た不変参照を介してグラフの辺を削除することができる．このため，publish 後に Agent から到達不能となりガベージコレクションによって解放されてしまうオブジェクトが生じる可能性があり，このオブジェクトへの不変参照はダングリング参照となる．このように，内部可変な型のオブジェクトを含むグラフの publish はその後の安全性を保証することができないため禁止される．なお，オブジェクトのフィールドに生ポインタが含まれる場合も不変参照を介した書き込みが可能となるが，生ポインタ経由での書き込みは安全性の保証されない Unsafe Rust 上での操作であり，本言語拡張が提供する安全性保証の対象としない．

publish するオブジェクトグラフに，すでに publish されたグラフ中のオブジェクトが含まれる場合も実行時エラーを発生させる．すなわち，publish した二つの異なるグラフが同じオブジェクトを含むことは禁止される．これを許すと，一方のグラフを unpublish することでそのオブジェクトに対する書き込みを行うことができることになり，上述の内部可変性の議論と同じ問題が発生しうる．例えば図 3 のプログラムでは，23 行目の publish の後に child を指定して publish を行うことはできない．これは root を指定した publish により取り出されるグラフに child が含まれるためである．この性質は，一度サンドボックスから取り出したオブジェクトを再び取り出すことはできない，というメタファーで表現することができる．

オブジェクトグラフの publish 後は，そのグラフ中のオブジェクトを指す &ind 参照を介した書き込みを動的に検知し，実行時エラーを発生させる．publish 後のグラフは不変参照を通じて利用されるため，グラフを構成するオブジェクトに変更を加える操作を禁止する必要がある．サンドボックス内のオブジェクトを参照できるのは &ind 参照と Agent から取り出した不変参照のみであるため，この動的検査のみによって publish されたグラフの不変性を保証することができる．例えば図 3 のプログラムでは，23 行目

の publish の後は root や child を介した書き込みは禁止される．

3.4 publish 済みグラフへのアクセスの安全性と実行時オーバーヘッド

publish されたオブジェクトグラフには不変参照でアクセスでき，通常の Rust 規則に従って実行時オーバーヘッドなく循環グラフを扱うことができる．グラフ中の各オブジェクトへのアクセスは Agent から取り出した不変参照を介してのみ行われ，読み書きの競合が起こらないことが保証されるため，AXM 規則に従いながらグラフの走査が可能である．

また，Agent から取り出せる各オブジェクトへの不変参照はすべて Agent の借用期間に紐づく共通のライフタイムを持ち，それらの利用中は参照先が有効であることが保証されるため，循環参照が存在していてもライフタイム規則に自動的に従う．図 2 において，agent.get() によって a_ を得ると（22 行目），a_ が利用される間は Agent が借用された状態となる．a_ からフィールドを辿って得られる b_ や a_re も同じ借用期間に紐づき，この期間を超えて利用することはできない．この制約はフィールドを辿るたびに自動的に適用され，プログラマは循環を構成する参照へ共通のライフタイムを明示的に割り当てる必要がない．

オブジェクトグラフへの参照の利用中に参照先が有効であることは，Rust の静的検査およびガベージコレクションの挙動によって保証される．Agent から得た参照が有効な間は Rust の静的検査によって Agent のムーブや破棄が拒否されるため，unpublish 等の参照先を無効にしうる操作が禁止される．図 2 では，グラフへの参照の最後の利用である assert!（26 行目）より後に unpublish を行っているが，この順序を入れ替えると利用中の参照を残したまま Agent をムーブすることになるため，Rust の静的検査によってコンパイルエラーとなる．また，サンドボックス内のオブジェクトを管理するガベージコレクションでは Agent が保持する参照からグラフ内部の参照を再帰的に辿るため，グラフの利用中は参照先のオブジェクトが先に回収されることはない．なお publish 済みグラフへの &ind 参照を介した書き込みは実行時エラーとなるため，グラフ内部の到達可能性が利用中に変更されることはない．サンドボックス外への参照についても，publish 時の検査と Agent に引き継がれたライフタイムによって参照先の有効性が保証される．このように，publish 後の利用では通常のフィールドアクセスを記述するだけで，各参照のライフタイム中は参照先が有効であり続けるというライフタイム規則が満たされる．

これらの安全性を保証するためのアクセス時の動的検査は不要であり，循環グラフに実行時オーバーヘッドなく安全にアクセスすることができる．グラフの構築後の走査が処理全体の大部分を占めるケースでは，この言語拡張の恩恵を最大限に受けることができる．

4 Miri による言語拡張の実装

提案する言語拡張は，Rust が公式に提供する MIR インタプリタである Miri [2] を改造したものの上で動作する．MIR は Rust コンパイラが型検査後に生成する中間表現であり，型情報を保持する単純な制御フローグラフとしてプログラムを表現する．Miri は MIR を解釈，実行するインタプリタであり，本来は Rust プログラム中の未定義動作を検出するために用いられる．Miri は実行時の各メモリアロケーションとポインタの参照先を内部で管理しながら，AXM 規則やライフタイム規則に基づくメモリアクセスや読み書き操作の妥当性について Rust コンパイラと同等の検査を行う．

本実装では，提案する言語拡張に必要な機能を Miri に追加することで提案手法の規則や挙動を再現する．具体的には，各アロケーションへのメタデータの追加やメモリアクセスへのフック等により，オブジェクトへの &ind 参照を介したアクセスの制御，publish 時およびその後の動的検査，およびガベージコレクションを実装する．ただし，本実装は Miri による MIR の解釈，実行を前提としており，通常の Rust コンパイラが生成するネイティブコードでは提案手法の検査等を利用することはできない．本実装の目的は提案する言語拡張の妥当性を実行可能な形で示すことであり，ネイティブ実行時の性能を評価することは含まない．

4.1 Miri 上でのサンドボックスの再現と publish 時の動的検査

Miri を改造することで，サンドボックスへのオブジェクトの確保やサンドボックスからのオブジェクトグラフの取り出しといったサンドボックス上の操作を実現する．3 節で導入したサンドボックスは，通常のメモリ空間とは異なるアクセス規則とメモリ管理が適用されるメモリ領域であるが，その具体的な実装までは規定しない．例えば Rust コンパイラへ直接実装する場合には，サンドボックスをガベージコレクションの対象となる独立したヒープ領域として実現することが考えられる．一方，本実装では独立したメモリ領域を設けず，Miri が管理する各メモリアロケーションにメタデータを付加することで，サンドボックス内外のオブジェクトを区別する．Miri は各アロケーションに対してメタデータを保持しており，各オブジェクトがサンドボックス内に確保されたものであるかをメタデータとして追加しておくことで，その後のアクセス時に確保先を判定できる．加えて，各アロケーションに対応するオブジェクトが `publish` 済みであるかもメタデータに含め，その後の動的検査等で利用する．

また，3.3 項で述べた `publish` 時の動的検査について，内部可変な型を持つオブジェクトがグラフ中に含まれないことを検査するため，`publish` の対象となる各オブジェクトについて，その具体的な型が `Freeze` トレイトを実装していることを要求する．`Freeze` は `UnsafeCell` を内部に直接含まないことを表す Rust トレイトであり，これを実装した型は内部可変でないと判断できる．この検査により，通常の不変参照を介して `publish` 済みのオブジェクトを書き換える経路がないことを保証する．ただし，オブジェクトの参照先の型は `Freeze` の判定では考慮されないため，走査によって到達したすべてのオブジェクトについてこの検査を行う．

4.2 プリプロセッサによるプログラムの変換

`&ind` 参照を Rust の参照型として直接扱うには，構文解析，型検査，借用検査，および MIR への変換を含む Rust コンパイラの各機能の改造が必要となる．Miri が実行する MIR は，Rust コンパイラによる構文解析と型検査を通過した後に生成される．そのため，Rust プログラム中で `&ind` 参照を言語組み込みの参照型として直接使用するためには，Rust コンパイラがそれを解析および検査できるようにする必要がある．

本実装では，Rust コンパイラを改造せずに `&ind` 参照型を含むプログラムを実行するため，それを Rust コンパイラが解釈できる形に変換するプリプロセッサを作成する．すなわち，`&ind` 参照型を含む表面言語で書かれたプログラムがプリプロセッサにより Rust プログラムに変換され，そこから生成される MIR を改造 Miri で実行する．プリプロセッサは `&ind` 参照型とそれを介した操作，および `publish` 等の組み込み関数を，生ポインタを利用した `unsafe` コード，および専用ライブラリの関数呼び出しへ変換する．改造 Miri では `&ind` 参照に相当する値を生ポインタとして表現し，`unsafe` な Rust コードによって `&ind` 参照に対する操作を再現する．この `unsafe` コードは Safe Rust では安全性を保証できない `&ind` 参照の挙動を改造 Miri 上で再現するための実装の内部に限定して用いる．

変換後のプログラムでは，サンドボックスへのオブジェクトの確保，および `publish` と `unpublish` 操作を実行する関数を提供する専用ライブラリを利用する．各ライブラリ関数は，改造 Miri 上の対応する処理を行う関数を呼び出し，サンドボックス上の操作を再現する．

5 関連研究

GhostCell [6] はグラフ中の各データへの読み書き権限を単一のトークンとして分離し，それに対する静的検査によって制御する．トークンに対する不変借用がグラフ中の任意のデータを読み出す権限，可変借用がデータを一つずつ書き換える権限の獲得に対応しており，トークンの借用状態を Rust の静的検査で管理することで，グラフ中のすべてのデータに AXM 規則を適用できる．すなわち，各データへの不変参照は複製できる一方，参照先へのアクセスにはトークンの借用が必要となるため，トークンが可変借用されている間に別の経路から同じデータが読み書きさ

れることはない．この仕組みにより GhostCell では，データ単位で読み書き権限の検査を行おうとすると AXM 規則に違反してしまう循環参照のようなデータ構造を，トークンに対する静的検査のみで安全に扱うことが可能である．

GhostCell を利用することで本研究と同様にアクセスごとの動的検査を避けながら安全に循環参照を扱うことができるが，安全性を確立するタイミングや利用時のデータ表現が異なる．GhostCell はグラフの構築・利用の区別なく同一の表現上で静的に読み書きを切り替えるのに対し，本研究は自由に変更できるサンドボックス内の状態と不変参照で利用する `publish` 状態との間でグラフを遷移させる．

GhostCell でもトークンを借用して取得した `&T` は既存の API へ渡せるが，データへの参照を取得するたびに対応するトークンを借用する必要があり，グラフを扱う関数等にトークンを受け渡さなければならない．この制約は，グラフへの不変借用を保持しながらその構造を変更する処理において顕在化する．図 4 は，グラフの根ノードへの参照を保持しながら `next` を辿って対象ノードを探し，そこに根へ戻る辺 `back` を追加するプログラムの例である．対象ノードのデータへの不変借用（21 行目）は書き込み前に終了しているが，`d_root` がトークンの不変借用を保持しているため，書き込みに必要な可変参照を取得するための `borrow_mut` は静的検査により拒否される（32 行目）．書き込み先のノードと借用中の根ノードが異なることが保証される場合，オブジェクト単位では読み書きが競合しないため，この操作は安全であるにもかかわらずコンパイルエラーとなる．この処理を実装するには，根ノードから走査に必要な情報を取り出したら不変借用を終了し，書き込み後に必要なら参照を取得し直すなど，単一のトークンがグラフ全体を一つの借用単位として扱うことを考慮する必要がある．

また，GhostCell はオブジェクトへのアクセス権限を管理するが，参照先のメモリ管理方式は規定しない．そのため，通常の参照を用いて循環グラフを構築する場合，プログラマはアリーナアロケータ等の特殊なメモリ管理手法を利用して共通のライフタイム割り当てを行いながら，所有者やライフタイムを意識したプログラミングを行う必要がある．一方，本研究ではサンドボックス内のオブジェクトをガベージコレクションによって管理するため，プログラマはこのような明示的なメモリ管理を行う必要がない．

```
1  type NodeRef<'a, 'id> = &'a
       GhostCell<'id, Node<'a, 'id>>;
2
3  struct Node<'a, 'id> {
4    value: i32,
5    next: Option<NodeRef<'a, 'id>>,
6    back: Option<NodeRef<'a, 'id>>,
7  }
8
9  fn add_back_edge<'a, 'id>(
10   token: &mut GhostToken<'id>,
11   root: NodeRef<'a, 'id>,
12 ) {
13   let d_root = root.borrow(token);
14   println!("before: {}",
         d_root.value);
15
16   let mut current = d_root.next;
17   let mut found = None;
18
19   while let Some(node) = current {
20     let (value, next) = {
21       let d = node.borrow(token);
22       (d.value, d.next)
23     };
24     if value == 0 {
25       found = Some(node);
26       break;
27     }
28     current = next;
29   }
30
31   let node = found.unwrap();
32   node.borrow_mut(token).back =
         Some(root);
33
34   println!("after: {}",
         d_root.value);
35 }
```

図 4　GhostCell を利用したコンパイルエラーになるプログラムの例

言語へのサンドボックスの組み込みについて，本研究と同様に，サンドボックス内の処理が外部へ及ぼす影響を検査等により制限することで，外部における安全性を保証する手法について多くの研究が行わ

れている．Sammler らは，信頼されていない開発者が作成したライブラリ等の非信頼コードから，信頼コードが操作するメモリへのアクセスを禁止するサンドボックスを組み込んだ言語を定式化し，その安全性を形式的に示した [4]．この言語では，非信頼コードとのやり取りがすべて任意の値を表す `any` 型を介して行われる．Sammler らは，非信頼コードから受け取る値について何の仮定も置かず，すべて `any` 型として扱っても安全性が証明できれば，非信頼コードの振る舞いによらず信頼コードの安全性が保たれると主張している．また，サンドボックスの境界で安全に `any` との型変換を行う仕組みを提供し，信頼コード側ではより具体的な型を持つ値として利用できることを示している．

非信頼コードや機密データを扱う処理を特別な領域内で実行する手法は様々な目的で利用されている．Dak Albab らは，Rust で記述された Web アプリケーションにおいて，機密データへの直接のアクセスを限定されたコード領域に閉じ込める手法を提案した [1]．この手法では，機密データを直接扱える範囲を限定されたコード領域内に制限し，その処理結果を専用のオブジェクトでラップして保護したのちに領域外へ返すことで，領域外では機密データを直接扱えないようにする．加えて，領域内の処理が機密データを外部へ漏洩しないことを静的に検査し，静的に確認できない場合にはアプリケーション本体から隔離されメモリアクセスや外部入出力が制限された環境で実行することで漏洩を防ぐことができる．特別な領域内の処理を可能な場合には静的に検証し，失敗した場合にのみ実行時の隔離を用いる構成は，本研究においても，`publish` の安全性条件の一部を静的に検査し，実行時のオブジェクト間の参照関係に依存する条件だけを動的に検査するような実装を検討する上で参考となる．

これらの既存研究では，非信頼コードとのやり取りや機密データを扱う処理のたびに，サンドボックスの境界を越える操作を制限あるいは検査する必要がある．Sammler らの手法では，非信頼コードとの値の受け渡しのたびにサンドボックスの境界上での型変換操作が必要となる．また，Dak Albab らの手法では，機密データを領域外でも専用のオブジェクトで保護し，それを直接扱う処理を限定された領域内で実行する．これに対し本研究では，サンドボックス内で Rust の規則を一時的に保留して構築したオブジェクトグラフについて，サンドボックスの境界を越えるための検査を，その利用を開始する `publish` 時に一度だけ行う．検査を通過したグラフは Rust 規則の下で扱われるようになり，その後のアクセスではサンドボックスの境界上での操作や専用の表現を介さず，通常の参照のみで利用できる．

6 まとめ

本論文では，循環参照の柔軟な構築と，構築後の安全かつアクセスごとの実行時オーバーヘッドを伴わない利用を両立する Rust 言語の拡張を提案した．提案手法では，通常の Rust とは異なる規則とメモリ管理を適用するサンドボックスを導入し，そこで構築した循環グラフを動的検査による安全性保証のもとで外に取り出すことで，不変参照を介した実行時オーバーヘッドのないアクセスを可能にする．

今後の課題として，提案手法の操作的意味論と型付け規則を形式化し，`&ind` 参照と不変参照の共存，`publish` と `unpublish` 操作，およびガベージコレクションを踏まえた安全性の証明を与えることが挙げられる．また，本論文で示した方針に基づき Miri による提案手法の実装を完成させることも予定している．

参 考 文 献

[1] Dak Albab, K., Agvanian, A., Aby, A., Tiffany, C., Portland, A., Ridley, S., and Schwarzkopf, M.: Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 709–725.

[2] Jung, R., Kimock, B., Poveda, C., Muñoz, E. S., Scherer, O., and Wang, Q.: Miri: practical undefined behavior detection for rust, *Proceedings of the ACM on Programming Languages*, Vol. 10, No. POPL(2026), pp. 1383–1411.

[3] Li, R., Wang, B., Li, T., Saxena, P., and Kundu, A.: Translating c to rust: Lessons from a user study, *arXiv preprint arXiv:2411.14174*, (2024).

[4] Sammler, M., Garg, D., Dreyer, D., and Litak, T.: The High-Level Benefits of Low-Level Sandboxing, *Proceedings of the ACM on Programming Languages*, Vol. 4, No. POPL(2020), pp. 1–32.

[5] The Rust Programming Language: RefCell⟨T⟩ and the Interior Mutability Pattern - The Rust Programming Language — doc.rust-lang.org, https://doc.rust-lang.org/book/ch15-05-interior-mutability.html. [Accessed 06-08-2026].

[6] Yanovski, J., Dang, H.-H., Jung, R., and Dreyer, D.: GhostCell: separating permissions from data in Rust, *Proceedings of the ACM on Programming Languages*, Vol. 5, No. ICFP(2021), pp. 1–30.