Rustを対象としたType-3のコードクローンの検出手法の提案および評価用データセットの構築

周 隆澤　篠埜 功

コードクローンはソフトウェアの保守性低下を招く恐れがあり，主に Java，C，C++を対象としてコードクローンの検出についてこれまで多くの研究が行われてきたが，Rust に関しては，ほとんど研究がなく，評価用のデータセットもなかった．そこで本研究では，まず，Rust の関数とメソッドを対象とし，トークンのバッグの類似度によりクローンの検出を行った後，判定が困難な候補に対しては抽象構文木のハッシュ値および特徴ベクトルを用いて構文構造の比較を行う．また，独自に Rust のコードクローンのデータセットを構築し，これを用いて CCFinderSW および MSCCD との比較を行う．その結果，Type-3 に関し，再現率は CCFinderSW は 0.52，本研究のツールは 0.92，MSCCD は 0.89 となった．また，クローン対の違いの種類ごとに，再現率の違いについて分析した．

1　はじめに

ソフトウェア開発では，既存のコードの一部をコピーし，貼り付けた後に修正して再利用することがある．このような作業により，ソースコード内に同一または類似したコード断片が発生することがあり，これらはコードクローンと呼ばれる．コードクローンは多くのソフトウェアシステムに存在し，ソフトウェアシステム全体の約 5〜20%がコードクローンであると報告されている [10]．コードクローンは開発の効率化に寄与する一方で，コード欠陥の伝播やコード行数の増加を招き，保守性の低下につながる可能性がある [6] [7]．

コードクローンは，Type-1，Type-2，Type-3，Type-4 に分類される [9]．Type-1 は，空白，改行，コメントを除いて一致するコードクローンである．Type-2 は，Type-1 の違いに加えて，識別子名やリテラルの違いを含むコードクローンである．Type-3 は，Type-2 の違いに加えて，文や式の追加，削除，変更を含むコードクローンであり，Type-4 は，構文上は異なるが同様の機能を実現するコードクローンである．

コードクローン検出に関する研究は，主に C/C++ や Java を対象として行われてきた．Rust については，Type-1 および Type-2 のクローン検出に関する研究 [8] があるが，文や式の追加，削除，変更を含む Type-3 クローンに関する研究は筆者らの知る限りこれまで十分に行われていない．また，Rust 用の評価用データセットもない．

そこで，本研究では，トークン情報と抽象構文木情報を組み合わせ，Rust 言語における Type-3 コードクローンを対象とした検出手法を提案する．提案手法では，Rust コードから関数およびメソッドを抽出し，まずトークン情報に基づいて候補となる関数対を絞り込む．その後，トークン情報のみでは判定が困難な候補対に対して，抽象構文木に基づく構造的な類似度を用いてクローンかどうか判定する．

提案手法の有効性を評価するために，17 個の Rust オープンソースプロジェクトから条件を満たす関数およびメソッドを抽出した．さらに，抽出した関数およびメソッドに対して幾つかの規則を適用するこ

---

* Proposal of a Type-3 Code Clone Detection Method for Rust and Construction of an Evaluation Dataset. This is an unrefereed paper. Copyrights belong to the Author(s).

Longze Zhou, Isao Sasano, 芝浦工業大学大学院理工学研究科電気電子情報工学専攻, Dept. of Electrical Engineering and Computer Science, Graduate School of Engineering and Science, Shibaura Institute of Technology.

とにより，Type-1 から Type-3 までのクローン対を含む評価用データセットを構築した．構築した評価用データセットを用いて，提案手法と既存ツールである CCFinderSW [12] および MSCCD [17] の再現率を比較した．その結果，Type-3 の再現率について，提案手法は 0.92，CCFinderSW は 0.53，MSCCD は 0.89 となった．

本論文の構成は以下の通りである．まず 2 節で，コードクローン検出手法，Rust 言語を対象としたコードクローン検出研究，および評価用データセットの構築に関する関連研究について述べる．次に 3 節で，提案手法の全体構成と，トークン情報および抽象構文木情報を用いた判定方法について述べる．4 節では，評価用 Rust コードクローンデータセットの構築方法について述べる．5 節では，コードクローン検出ツールおよびデータセット構築システムの実装について述べる．6 節では，評価実験の結果を示し，Type-1 から Type-3 までのクローン種別ごとの再現率を比較する．また，Type-3 クローンについては，変換種別ごとの再現率の違いを分析する．最後に 7 節で，本研究のまとめと今後の課題について述べる．

2 関連研究

本節では，本研究に関連するコードクローン検出手法，Rust 言語を対象としたコードクローン検出研究，および評価用データセットについて述べる．

2.1 コードクローン検出手法

Roy らは，コードクローン検出手法を，ソースコードを文字列として比較する手法，字句解析により得られるトークン列に基づく手法，抽象構文木に基づく手法，プログラム依存グラフに基づく手法などに分類した [9]．また，一つの表現のみを用いるのではなく，トークン列や抽象構文木など，複数の表現を組み合わせてコードクローンを検出する手法も提案されてきた [5] [16]．Wang らは，Java を対象とし，トークンに基づく処理と抽象構文木に基づく処理を組み合わせたコードクローン検出手法を提案した [16]．この手法では，まずトークンに基づく処理により候補となるコード断片の対を抽出し，その後，抽象構文木に基づく処理により，抽出されたコード断片の対がコードクローンであるかを判定する．

また，多言語に対応可能なコードクローン検出ツールとして，CCFinderSW [12] や MSCCD [17] がある．Semura らは，多言語に対応可能なトークンベースのコードクローン検出ツールである CCFinderSW を提案した [12]．Zhu らは，多言語に対応可能なコードクローン検出ツールである MSCCD を提案した [17]．MSCCD は，構文解析結果を用いてトークンバッグの比較により Type-3 コードクローンを検出する．6 節で述べる評価実験では，これらのツールを比較対象として用いる．

2.2 Rust 言語を対象としたコードクローン検出

Rust 言語を対象としたコードクローン検出研究として，Pizzolotto らは，Rust のソースコードおよび中間表現を対象として，CCFinderSW を用いたコードクローン検出を行った [8]．この研究では，ソースコード上で検出されたクローンと中間表現上で検出されたクローンを比較し，中間表現を用いた場合のクローン検出結果の違いを分析した．ただし，分析対象は Type-1 および Type-2 コードクローンに限定されており，Type-3 コードクローンは対象外である．

2.3 評価用データセット

コードクローン検出手法の評価には，クローン種別などの情報を含む評価用データセットが用いられる．代表的な評価用データセットの一つに，Java ソースコードを対象とした大規模なコードクローンベンチマークである BigCloneBench がある [14]．BigCloneBench は，Type-1 から Type-4 までのクローン種別や，コード断片間の類似度に関する情報を含む．なお，BigCloneBench におけるクローン対はメソッド単位である．

一方で，Rust 言語を対象とした Type-3 コードクローン検出の評価に用いることのできるデータセットは筆者らの知る限り見当たらない．評価用データセットを構築する方法の一つとして，既存のコード断片に対して規則に基づく変換を適用し，評価対象となるクローンを生成する方法がある．Roy と Cordy は，既

存のコード断片に変更を加えて人工的なクローンを生成することによりコードクローン検出ツールを評価する方法を提案した[11]．この研究では，コメントの追加，削除，変更，空白，改行の追加，削除，識別子名の変更，行内へのコード断片の追加および行の一部の削除あるいは変更，行単位の追加，削除，変更など，クローン種別に対応する変更が加えられた．

3 提案手法

この節では，トークン情報と抽象構文木情報を組み合わせる考え方[16]を参考にし，Rust 言語を対象とした Type-3 コードクローン検出手法を提案する．

本研究の提案手法は，トークンに基づく分類と，抽象構文木に基づく分類の 2 段階から構成される．提案手法の処理の流れを図 1 に示す．図 1 の中の「トークンに基づく分類」と「抽象構文木に基づく分類」の処理の詳細を図 2 と図 3 にそれぞれ示す．まず，Rust ソースコードから関数および `impl` ブロック内のメソッドを抽出し，各関数およびメソッドに対してトークン列を生成する．次に，各関数およびメソッドにおいて，元のコードの N 行ごとのハッシュ値と各トークンの出現頻度を用いて，すべての関数およびメソッドの組み合わせについて，クローンか，クローンでないか，あるいは不明の 3 通りに分類する．ここで，不明に分類されたコード断片の対は，図 1 における「不明」に対応する．不明に分類されたコード断片の対については，それぞれのコード断片の抽象構文木を構築し，抽象構文木から得られるハッシュ値および特徴ベクトルを用いてクローンかどうかの判定を行う．これにより，トークンの出現頻度だけでなく，構文構造も考慮したクローン判定を行う．

3.1 トークンに基づく分類

次に，Wang らの手法[16]と同様に，トークンに基づく分類は，関数あるいはメソッド単位で行う．そのために，Rust のソースコードを構文解析し，関数および `impl` 構文内のメソッドを抽出する．抽出した関数およびメソッドについて，N 行ブロックごとのハッシュ値およびトークン重複度により，2 つの関数あるいはメソッドの間の類似度を計算する．

まず，各 N 行ブロックのハッシュ値を用いた類似度について，関数またはメソッド f と g の類似度を以下のように定義する．

$$\mathrm{SR}(f,g) = \frac{\mathrm{match}_N(f,g)}{\max(B_N(f), B_N(g))}$$

ここで，$\mathrm{match}_N(f,g)$ は，関数またはメソッド f と g 内の N 行ブロックの中でハッシュ値が一致する N 行ブロックの数を表す．また，$B_N(f)$ および $B_N(g)$ は，関数またはメソッド f, g から生成される N 行ブロックの数を表す．

次に，トークン重複度による類似度については，$\mathrm{SR}(f,g)$ が 6.1 節で述べるしきい値を満たす関数またはメソッドの対 (f,g) について，f, g のトークン列を $S(f)$，$S(g)$ とし，$S(f)$, $S(g)$ をそれぞれソートし，2 つのトークン列内で一致するトークン数 $\mathrm{match}_T(S(f), S(g))$ を計算する．2 つのトークン列 A, B について，それらの間のトークン重複度を以下のように定義する．

$$\mathrm{Overlap}(A,B) = \frac{\mathrm{match}_T(A,B)}{\max(|A|,|B|)}$$

列 L について，$|L|$ は L の長さを表す．Overlap による類似度においては，トークン列に含まれる同一トークンの重複を取り除かずに，一致するトークン数を計算する．そのため，各トークンの出現頻度を反映した比較となる．

上で定義した関数 SR と Overlap を用いることにより，関数またはメソッドの対を，クローン，非クローン，不明の 3 通りに分類する．まず，$\mathrm{SR}(f,g)$ がしきい値 θ_1 未満の場合，対 (f,g) を非クローンとし，除外する．また，$\mathrm{SR}(f,g)$ が θ_1 以上であっても，トークン列 $S(f)$, $S(g)$ 間のトークン重複度 $\mathrm{Overlap}(S(f), S(g))$ がしきい値 θ_2 未満である場合は，非クローンとし，除外する．つまり，以下の式 (1) を満たす対がクローンの候補として残る．

$$\mathrm{SR}(f,g) \geq \theta_1 \quad \wedge \quad \mathrm{Overlap}(S(f), S(g)) \geq \theta_2 \quad (1)$$

式 (1) を満たす対 (f,g) のうち，$\mathrm{SR}(f,g)$ または $\mathrm{Overlap}(S(f), S(g))$ が特に大きい対は，この時点でクローンとする．この判定のためのしきい値としては，$\theta_3 > \theta_1$，$\theta_4 > \theta_2$ を満たす θ_3，θ_4 を用いる．つまり，以下の式 (2) を満たす対 (f,g) はこの時点でクローンとする．

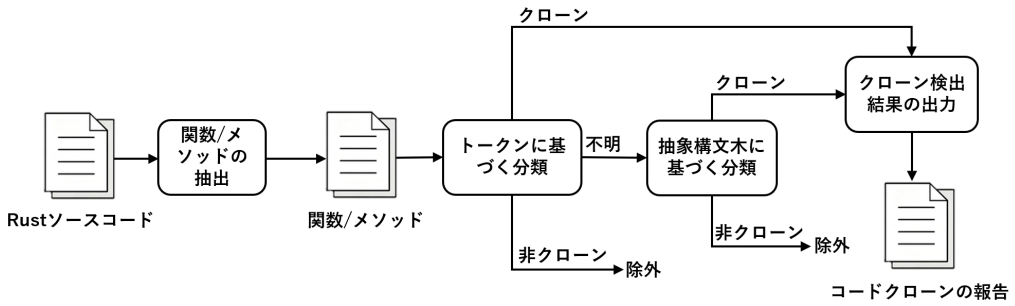

図 1　提案手法におけるコードクローン検出の流れ

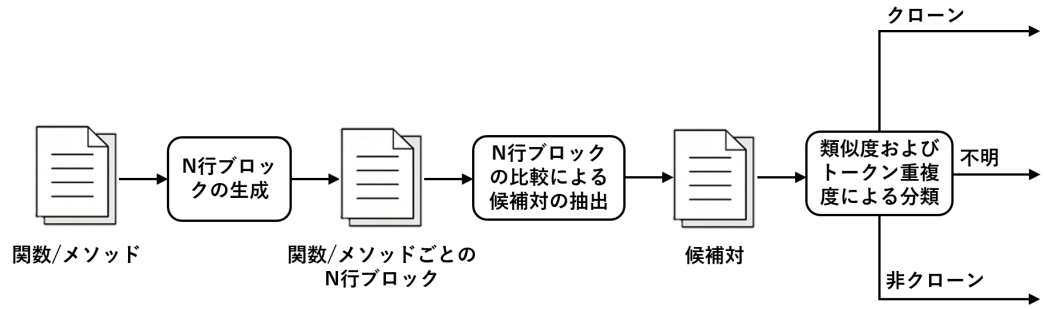

図 2　図 1 の「トークンに基づく分類」の処理

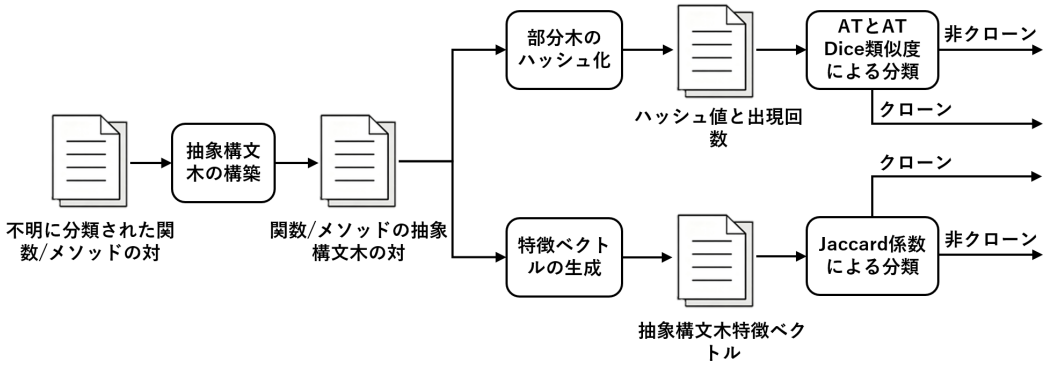

図 3　図 1 の「抽象構文木に基づく分類」の処理

$$\mathrm{SR}(f,g) \geq \theta_3 \quad \vee \quad \mathrm{Overlap}(S(f),S(g)) \geq \theta_4 \quad (2)$$

一方，式 (1) を満たすものの，式 (2) を満たさない組み合わせは，この時点ではクローンかどうかの判定をせず，不明に分類する．不明に分類された関数またはメソッドの対は抽象構文木を用いた判定の対象とし，これについては次節で述べる．

3.2　抽象構文木に基づく分類

Wang らの手法 [16] では，抽象構文木をハッシュ値に変換して類似度を計算するが，本研究における類似度の計算とは異なる．具体的には，Wang らは AT

類似度を用いたが，本論文では AT 類似度と ATDice 類似度を用いて類似度を計算する．

前節で抽象構文木を用いた判定対象とした関数またはメソッドの対については，抽象構文木を用いてクローンかどうか判定する．まず，対象となる関数またはメソッドの対のソースコードから抽象構文木を構築する．この抽象構文木の特徴ベクトルおよび各部分木のハッシュ値を用いてクローンかどうか判定する．

抽象構文木のハッシュ値に基づく判定では，抽象構文木の各ノードについて，ノードの種類を表す文字列からハッシュ値を計算する．また，子ノードを持つノードについては，自身のノード種類に基づくハッシュ値と子ノードのハッシュ値を組み合わせることで，部分木のハッシュ値を計算する．各部分木のハッシュ値とその出現回数を用いて，2 つの抽象構文木の類似度を計算する．

2 つの関数またはメソッド f, g の抽象構文木において，ハッシュ値 h の出現回数をそれぞれ $H_f(h)$, $H_g(h)$ とする．また，関数またはメソッド f の抽象構文木に現れる部分木のハッシュ値の集合を $\mathcal{H}(f)$ とする．2 つの関数またはメソッド f, g の抽象構文木においてハッシュ値が一致する部分木の数を $\mathrm{Match}(f, g)$ で表すこととし，以下のように定義する．

$$\mathrm{Match}(f, g) = \sum_{h \in \mathcal{H}(f) \cup \mathcal{H}(g)} \min(H_f(h), H_g(h))$$

また，関数あるいはメソッド f の抽象構文木に含まれる部分木の数は，以下のように定義される関数 Size を用いて $\mathrm{Size}(f)$ と表される．

$$\mathrm{Size}(f) = \sum_{h \in \mathcal{H}(f)} H_f(h)$$

2 つの関数あるいはメソッド f, g について，抽象構文木のハッシュ値に基づく類似度 $\mathrm{AT}(f, g)$ は以下のように定義される [16]．

$$\mathrm{AT}(f, g) = \frac{\mathrm{Match}(f, g)}{\max(\mathrm{Size}_H(f), \mathrm{Size}_H(g))}$$

$\mathrm{AT}(f, g)$ の値が大きいほど，2 つの関数またはメソッドは類似した構文構造を持つと判断できる．ただ，AT のみを用いた場合は，大きい関数が小さい関数の全体を含む場合に，AT のしきい値を大きくするとクローンと判定されない場合がある．また，AT のしきい値を小さくすると，非クローンがクローンとして判定される偽陽性が増加する．そこで，2 つの抽象構文木の大きさを考慮する以下の ATDice という類似度関数も用いることとする．

$$\mathrm{ATDice}(f, g) = \frac{2\mathrm{Match}(f, g)}{\mathrm{Size}(f) + \mathrm{Size}(g)}$$

次に，抽象構文木から特徴ベクトルを生成する．特徴ベクトルは，抽象構文木に含まれるノードの種類を用いて計算する．この特徴ベクトルにおいてはノードの種類が特徴であり，各特徴の値は，抽象構文木中におけるその種類のノードの出現回数を表す．

2 つの関数またはメソッド f および g から得られる特徴ベクトルをそれぞれ V_f，V_g とする．V_f，V_g における特徴 p の値をそれぞれ $V_f(p)$，$V_g(p)$ と書き，特徴ベクトル f，g の特徴の集合をそれぞれ $\mathcal{P}(f)$, $\mathcal{P}(g)$ とする．

2 つの特徴ベクトルの類似度は，以下で表される Jaccard 係数により求める．

$$\mathrm{Jaccard}(f, g) = \frac{\sum_{p \in \mathcal{P}(f) \cup \mathcal{P}(g)} \min(V_f(p), V_g(p))}{\sum_{p \in \mathcal{P}(f) \cup \mathcal{P}(g)} \max(V_f(p), V_g(p))}$$

以上の 3 つの類似度を用いて，抽象構文木判定対象となった関数またはメソッドの対について，以下の式 (3) を満たす場合にクローンと分類する．

$$\begin{aligned} &\mathrm{AT}(f, g) \geq \theta_5 \quad \vee \quad \mathrm{ATDice}(f, g) \geq \theta_6 \quad \vee \\ &\mathrm{Jaccard}(f, g) \geq \theta_7 \end{aligned} \tag{3}$$

ここで，θ_5，θ_6，θ_7 は，それぞれ AT，ATDice，Jaccard 係数に関するしきい値である．式 (3) を用いることにより，トークン情報のみでは判定しにくい関数またはメソッドの対についても，構文構造を考慮したクローン判定を行えるようになる．

4 評価用データセットの構築

既存研究では，Java などを対象としたコードクローン検出ツールの評価用データセットが広く利用されているが，Rust を対象としたデータセットは筆者らが調べた限り見当たらない．そこで本研究では，Rust コードを対象とした関数あるいはメソッド単位の評価用データセットを構築した．以下では，Rust 用コードクローンデータセットの構築方法について述べる．

4.1 評価用データセットの構築の概要

評価用データセットの構築の全体像を図 4 に示す．まず，複数の Rust オープンソースプロジェクトから関数および `impl` ブロック内のメソッドを抽出し，

4.2 節で述べる一定の条件を満たすものを seed 関数として選定する．seed 関数は関数あるいはメソッドである．

さらに，選定した seed 関数に対して幾つかの規則を適用することにより，Type-1 から Type-3 までのクローン対を生成する．また，正例のクローン対に加えて，明らかにクローンではない関数またはメソッドの対を負例として用意する．負例は，異なる seed 関数の組み合わせから，4.4 節で述べる一定の条件を満たすものを選定する．

これらの正例クローン対および負例対にラベルを付与し，評価用データセットとして用いる．

4.2 Seed 関数の抽出と選定

まず，複数の Rust オープンソースプロジェクトから Rust のファイルを収集し，構文解析を行って関数およびメソッドを抽出する．抽出した関数およびメソッドについては，プロジェクト名，ファイル名，関数名，開始行，終了行などの情報を記録する．

次に，抽出した関数およびメソッドのうち，一定の条件を満たすものを seed 関数として選定する．コードクローン検出ツールの評価では，短すぎるコード断片を評価対象から除外するために，最小行数や最小トークン数を設定する場合がある [4] [15]．これを参考に，本研究では，コメントや空白を除いて計算したトークン数が 50 以上 300 以下であり，有効行数が 6 行以上である関数およびメソッドを seed 関数として用いる．有効行は，コメントと空白以外のトークンが 1 つ以上ある行である．また，規則に基づく変換の適用可能性を考慮し，関数またはメソッドの本体を構成する文や式が 3 つ以上であることを条件とした．上記のように選択した seed 関数の中で重複があると無駄なので，完全に一致する場合，またはコメントや空白を除いた後のトークン列が完全に一致する場合は，1 つを残して他は除外した．以上のようにして選択した seed 関数を用いて，次節で述べる正例クローン対を生成する．

表 1　正例クローン対の生成に用いた変換

種別	変換対象	変換内容
Type-1	コメント	追加，削除，変更
	空白，改行	追加，削除
Type-2	識別子名，リテラル	変更
Type-3	文や式の一部	追加，削除，変更
	1 つの文や式	追加，削除，変更
	複数の文や式	追加，削除，変更
	複数箇所における文や式	追加，削除，変更

4.3 正例クローン対の生成

選定した seed 関数に対して，クローン種別に応じた規則を適用することにより，Type-1 から Type-3 までの関数またはメソッドのクローン対を生成した．作成した正例クローン対には，適用した変換規則に応じて，Type-1，Type-2，Type-3 のいずれかを表すクローン種別ラベルを付与した．また，Type-3 クローン対については，追加，削除，変更などの変換内容に基づいて，変換種別ラベルも付与した．

Type-1 では，コメントの追加，削除，変更，および空白，改行の追加，削除を行った．Type-2 では，識別子名やリテラルの変更を行った．Type-3 では，文や式の追加，削除，変更を行った．なお，Type-3 の変換では，文や式の一部，1 つの文や式，複数の文や式，および複数箇所における文や式を対象とした追加，削除，変更も含めた．

式の変更により生成した Type-3 クローン対の例を図 5 および図 6 に示す．この例では，図 5 の 3 行目の式 `n * 10` を if 式に変更することで，図 6 の関数を生成しており，Type-3 のクローン対としている．

表 1 に，正例クローン対の生成に用いた変換について示す．本研究では，Type-1 ではコメント，空白，改行に対する変換を行い，Type-2 では識別子名やリテラルの変更を行った．Type-3 では，文や式を対象として，追加，削除，変更を行った．

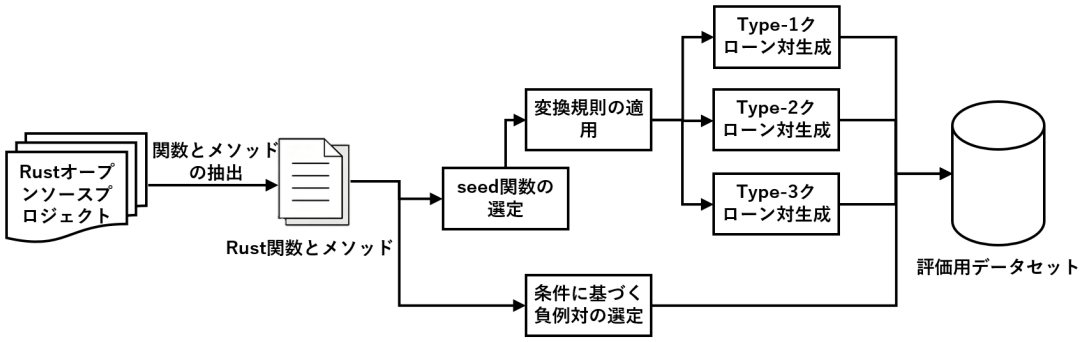

図 4　本研究における評価用データセットの構築の全体像

```
pub fn price(p: i32, n: i32) -> i32 {
  let mut total = p * n;
  let fee = n * 10;
  total = total + fee;
  if total > 10000 {
    total = total - 500;
  }
  if total < 0 { 0 } else { total }
}
```

図 5　式の変更による Type-3 クローン生成の例 (変換前)

```
pub fn price(p: i32, n: i32) -> i32 {
  let mut total = p * n;
  let fee = if n > 0 { n * 10 }
            else { 0 };
  total = total + fee;
  if total > 10000 {
    total = total - 500;
  }
  if total < 0 { 0 } else { total }
}
```

図 6　式の変更による Type-3 クローン生成の例 (変換後)

4.4　負例対の選定

正例クローン対に加えて，非クローンである負例対もデータセットに含めた．負例対の候補は，正例クローン対と同様，Rust オープンソースプロジェクトから抽出した関数および `impl` ブロック内のメソッドを用いた．このうち，コメントや空白を除いたトークン数が 50 以上 300 以下であり，有効行数が 6 行以上である関数およびメソッドを対象とした．これらの条件を満たす候補のうち，トークン重複度，N 行ブロックに基づく類似度 SR，および連続一致率を用いて，以下の 2 つの条件のいずれかを満たすものを負例対として選んだ．1 つ目は，トークン重複度および N 行ブロックの類似度がともに低い組み合わせである．具体的には，2 つの関数あるいはメソッド (f, g) のトークン重複度 $\mathrm{Overlap}(S(f), S(g))$ が 0.25 未満かつ $\mathrm{SR}(f, g)$ が 0.10 未満である場合に，その組み合わせを負例対とした．2 つ目は，トークン重複度や N 行ブロックでは一定の類似性を持つものの，連続して一致する部分が短い組み合わせである．具体的には，2 つの関数あるいはメソッド (f, g) のトークン重複度 $\mathrm{Overlap}(S(f), S(g))$ が 0.25 以上 0.55 未満，$\mathrm{SR}(f, g)$ が 0.10 以上 0.30 未満であり，さらに以下で定義する連続一致率 $\mathrm{CR}(f, g)$ が 0.25 未満である場合に，その組み合わせを負例対とした．

$$\mathrm{CR}(f, g) = \frac{\mathrm{LC}(f, g)}{\min(|S(f)|, |S(g)|)} \tag{4}$$

$\mathrm{LC}(f, g)$ は，2 つの関数あるいはメソッド f, g のトークン列 $S(f)$, $S(g)$ で一致する最大長の連続部分列の長さである．

2 つ目の条件は，明らかに類似度が低い組み合わせに加えて，トークン列や行単位では一定の類似性を持つが，連続一致率が小さい組み合わせを負例対として

含めるために用いた．

5 実装

ここでは，コードクローン検出ツールおよびデータセット構築システムの実装について述べる．実装したコードクローン検出ツールおよびデータセット構築システムのソースコードは，`https://github.com/sasano-lab-papers/Rust-type3-clone-detector` および `https://github.com/sasano-lab-papers/Rust-type3-clone-dataset` で公開する．

分析対象および評価用データセット構築用の関数およびメソッドは複数の GitHub のオープンソースプロジェクトから収集し，構文解析を行って抽出する．Rust の構文解析には，`syn`[3] version 2.0.111 を用いた．また，Rust の字句解析には `rustc_lexer`[2] version 0.1.0 を用いた．抽出した関数およびメソッドについては，プロジェクト名，ファイル名，関数名，開始行，終了行，トークン列を，後続の処理で利用するため，JSONL 形式で保存した．評価用データセットの規則に基づく変換を適用する場合は構文解析ライブラリ `tree-sitter-rust`[13] version 0.23.2 を用いた．

6 評価実験および考察

本節では，本研究で構築した評価用データセットを用いて，提案手法の検出性能を評価する．また，既存のコードクローン検出ツールとの比較を行い，クローン種別ごとの再現率および Type-3 変換種別ごとの再現率について考察する．

6.1 評価実験

4 節で述べた評価用データセットを用いて，3 節で提案したクローン検出手法について評価した．比較対象として，既存のコードクローン検出ツールである CCFinderSW[12] および MSCCD[17] を用いた．

本研究では，以下の 17 個のオーブンソースコードから Rust の関数およびメソッドを抽出した．抽出した関数およびメソッドは評価実験で用いる評価用データセット構築のために用いた．

- arrayvec: `https://github.com/bluss/arrayvec`
- bitvec: `https://github.com/ferrilab/bitvec`
- bstr: `https://github.com/BurntSushi/bstr`
- bytemuck: `https://github.com/Lokathor/bytemuck`
- Bytes: `https://github.com/tokio-rs/bytes`
- cgmath-rs: `https://github.com/rustgd/cgmath`
- generic-array: `https://github.com/fizyk20/generic-array`
- hashbrown: `https://github.com/rust-lang/hashbrown`
- indexmap: `https://github.com/indexmap-rs/indexmap`
- Itertools: `https://github.com/rust-itertools/itertools`
- ndarray: `https://github.com/rust-ndarray/ndarray`
- num-rational: `https://github.com/rust-num/num-rational`
- parking_lot: `https://github.com/Amanieu/parking_lot`
- petgraph: `https://github.com/petgraph/petgraph`
- Slab: `https://github.com/tokio-rs/slab`
- rust-smallvec: `https://github.com/servo/rust-smallvec`
- uuid: `https://github.com/uuid-rs/uuid`

上記 17 個のプロジェクトの Rust のソースコードは合計 121,076 行である．これらの Rust のソースコード中の関数およびメソッド 7,320 個のうち，4.2 節で述べた抽出条件を満たす関数およびメソッドが 628 個あり，これらを seed 関数とし，各 seed 関数に対して 4.3 節で述べた規則に基づく変換を適用することで，Type-1 から Type-3 までの正例クローンを生成した．表 2 に，評価用データセットの構成を示す．

評価用データセットのコードは，表 2 に示すように，Rust ファイル 6,356 個，Rust コード 165,766 行

表 2　評価用データセットの構成

項目	数
seed 関数	628
Type-1 変換後関数	1,908
Type-2 変換後関数	911
Type-3 変換後関数	2,909
Rust ファイル	6,356
Rust コード行	165,766

から構成される．本研究では，Type-1 から Type-3 までの正例クローン対に対する検出結果に基づき，再現率を評価指標として各手法を比較した．再現率は，生成された正例クローン対のうち，クローンと判定されたクローン対の割合であり，以下の式で計算する．

$$\frac{TP}{TP+FN}$$

ここで，TP は検出された正例クローン対の数を表し，FN は検出されなかった正例クローン対の数を表す．

評価実験における 3 つのツールの設定は，ツールごとに行った．CCFinderSW では，対象言語を Rust に指定し，検出するクローンの最小長を 50 とした．その他の設定はすべてデフォルト値を用いた．MSCCD では，対象言語を Rust とし，クローン検出に用いるトークンバッグの最小サイズを表す `minTokens` を 50 に設定した．その他の設定は，GitHub リポジトリ [1] の設定ファイルで定義されているデフォルト値に準拠した．本研究の提案手法では，関数またはメソッドを単位とし，トークン数が 50 以上の関数またはメソッドを対象とした．また，N 行ブロックの N を 3 とした．トークンに基づく分類では，式 (1) における SR のしきい値 θ_1 を 0.15，トークン重複度のしきい値 θ_2 を 0.50 とした．また，式 (2) における SR のしきい値 θ_3 を 0.70，トークン重複度のしきい値 θ_4 を 0.70 とした．抽象構文木に基づく分類では，式 (3) における AT のしきい値 θ_5 を 0.65，ATDice のしきい値 θ_6 を 0.70，Jaccard のしきい値 θ_7 を 0.75 とした．

以上の設定で各手法を実行し，各手法の検出結果を評価用データセットに記録された正例クローン対と照合した．

CCFinderSW については，クローンの単位は決まっていないため，検出結果のコード断片と，正例クローン対を構成する関数またはメソッドの範囲を比較した．対を構成する両方の関数またはメソッドが検出されたコード断片と 70%以上重なる場合に，その正例クローン対を検出したとみなした．

表 3　各手法のクローン種別ごとの再現率

手法	Type-1	Type-2	Type-3
提案手法	1.00	0.94	0.92
CCFinderSW	1.00	0.99	0.53
MSCCD	1.00	0.92	0.89

6.2　クローン種別ごとの評価結果

Type-1，Type-2，Type-3 について，各手法のクローン種別ごとの再現率を表 3 に示す．

表 3 より，Type-1 ではいずれの手法も再現率が 1.00 となった．Type-2 でも，いずれの手法も高い再現率を示した．一方，Type-3 では手法によって再現率に差が見られた．CCFinderSW [12] の再現率は 0.53 であり，Type-1 および Type-2 と比べて低くなった．これに対して，提案手法は 0.92，MSCCD [17] は 0.89 の再現率を示した．この結果から，提案手法および MSCCD [17] では，文や式の追加，削除，変更を含む Type-3 クローンに対しても，0.9 前後の再現率が得られた．

6.3　Type-3 の変換種別ごとの考察

Type-3 クローンについて，変換種別ごとの再現率を表 4 に示す．Type-3 のクローン生成のための変換では，変換対象となる文や式の範囲と操作の種類によって，各手法の再現率に差が見られた．表中の「一部」は文や式の一部を対象とした変換を表し，「1 つ」は 1 つの文や式を対象とした変換を表す．また，「複数」は複数の文や式を対象とした変換を表し，「複数箇所」は複数箇所における文や式を対象とした変換を表す．

表 4 より，Type-3 では変換の種類によって再現率に差が見られた．追加型および削除型では，提案手法が MSCCD [17] より大きくなる場合や，両手法の差が小さい場合が多く見られた．一方，変更型に着目すると，一部変更では提案手法と MSCCD [17] の再

表 4　Type-3 変換種別ごとの再現率

変換種別	提案手法	CCFinderSW	MSCCD
一部追加	1.00	0.89	0.97
一部削除	0.95	0.68	0.94
一部変更	0.95	0.67	0.95
1 つ追加	0.96	0.55	0.92
1 つ削除	0.85	0.61	0.70
1 つ変更	0.82	0.42	0.80
複数追加	0.98	0.49	0.99
複数削除	0.69	0.39	0.57
複数変更	0.92	0.68	0.99
複数箇所追加	0.92	0.34	0.92
複数箇所削除	0.99	0.79	0.99
複数箇所変更	0.91	0.69	0.99

現率がいずれも 0.95 となり，1 つ変更では提案手法が 0.82，MSCCD [17] が 0.80 となった．これに対して，複数変更および複数箇所変更では，MSCCD [17] がいずれも 0.99 の再現率となり，提案手法より大きい値を示した．このことから，MSCCD [17] は，複数の文や式，または複数箇所を対象とした変更を含む Type-3 クローンに対して高い再現率を示した．また，CCFinderSW [12] では，多くの Type-3 変換種別において，提案手法および MSCCD [17] よりも低い再現率となった．

なお，複数の文や式，または複数箇所を対象とした変換では，その変換を適用できる文や式を含む seed 関数が対象となる．特に削除や変更では，削除または変更の対象となる文や式が存在する必要がある．そのため，これらの変換では，比較的規模の大きい関数またはメソッドが対象となりやすく，変換後にも未変更の文や式が一定数残る場合がある．この場合，変換の規模が大きくても，変換前後で一致するトークン列や構文構造が多く残り，類似度計算に影響することが考えられる．したがって，変換種別ごとの再現率の違いは，変換内容だけでなく，生成対象となった関数またはメソッドの規模や未変更部分の量にも影響を受けると考えられる．

また，複数削除では，いずれの手法においても再現率が相対的に低かった．複数の文や式を削除する場合，変換前後で一致する文や式の数が減少し，連続一致部分の長さや類似度計算に影響する可能性がある．そのため，複数の文や式の削除を含む Type-3 クローンは，他の変換種別と比べて検出が難しくなったと考えられる．

6.4　本研究の制約

本研究にはいくつかの制約がある．まず，本研究で用いた評価用データセットは，実際の Rust オープンソースプロジェクトから抽出した関数およびメソッドをもとにしているが，正例クローン対は規則に基づく変換により生成したものである．そのため，6.2 節で述べた評価結果は，規則の適用により生成された Rust コードクローンに対する再現率を示すものであり，実際の開発過程で発生する多様なクローンに対する性能を直接示すものではない．

また，提案手法の実装では，字句解析に `rustc_lexer` を，抽象構文木の生成に `syn` を用いた．さらに，評価用データセットの構築では，変換対象となる文や式の位置の特定および変換後コードの構文の確認に `tree-sitter-rust` を用いた．したがって，本評価結果は，これらのライブラリのバージョンに影響を受ける可能性がある．

7　まとめと今後の課題

本研究では，Rust 言語における Type-3 コードクローンを対象として，トークン情報と抽象構文木情報を組み合わせた検出手法を提案した．また，Rust オープンソースプロジェクトから抽出した関数あるいはメソッドをもとに，Type-1 から Type-3 までのクローン対を含む評価用データセットを構築し，既存ツールとの比較を行った．さらに，Type-3 の変換種別毎に結果を分析し，変換の種類による傾向を確認した．

今後の課題としては，本研究の評価用データセットは規則に基づく変換により生成したため，実際の開発過程で自然に発生したクローンや，人手により確認されたデータセットを用いた評価を行うことが考えられる．また，Type-3 クローンでは，追加，削除，変更の程度によってクローンと非クローンの境界が曖昧になりやすい．そのため，自動生成により作成できる負例対は，明らかにクローンではないものに限られやす

く，クローンと誤検出されやすい負例対を十分に含めることは容易ではない．今後は，人手による確認を含めた負例対の構築方法を検討することが考えられる．

参 考 文 献

[1] : MSCCD, https://github.com/zhuwq585/MSCCD.

[2] : The Rust Programming Language, https://github.com/rust-lang/rust/tree/main/compiler/rustc_lexer.

[3] : syn, https://github.com/dtolnay/syn.

[4] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9(2007), pp. 577–591.

[5] Li, H. and Thompson, S.: Clone detection and removal for Erlang/OTP within a refactoring environment, *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, New York, NY, USA, Association for Computing Machinery, 2009, pp. 169–178.

[6] Mondal, M., Roy, B., Roy, C. K., and Schneider, K. A.: An Empirical Study on Bug Propagation through Code Cloning, *Journal of Systems and Software*, Vol. 158(2019), pp. 110407.

[7] Monden, A., Nakae, D., Kamiya, T., Sato, S.-i., and Matsumoto, K.-i.: Software Quality Analysis by Code Clones in Industrial Legacy Software, *Proc. 8th IEEE Symposium on Software Metrics*, IEEE, 2002, pp. 87–94.

[8] Pizzolotto, D., Matsushita, M., and Inoue, K.: Code Clone Detection in Rust Intermediate Representation, 研究報告ソフトウェア工学（*SE*）, Vol. 2022-SE-211, No. 26(2022), pp. 1–7.

[9] Roy, C. K., Cordy, J. R., and Koschke, R.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, *Science of Computer Programming*, Vol. 74, No. 7(2009), pp. 470–495.

[10] Roy, C. K. and Cordy, J. R.: A Survey on Software Clone Detection Research, Technical Report 2007-541, School of Computing, Queen's University at Kingston, 2007.

[11] Roy, C. K. and Cordy, J. R.: A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools, *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 157–166.

[12] Semura, Y., Yoshida, N., Choi, E., and Inoue, K.: CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization, *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 654–659.

[13] Sokolov, M.: tree-sitter-rust, https://github.com/tree-sitter/tree-sitter-rust.

[14] Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M.: Towards a Big Data Curated Benchmark of Inter-project Code Clones, *Proc. 2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 476–480.

[15] Svajlenko, J. and Roy, C. K.: Evaluating clone detection tools with BigCloneBench, *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 131–140.

[16] Wang, Y., Ye, Y., Wu, Y., Zhang, W., Xue, Y., and Liu, Y.: Comparison and Evaluation of Clone Detection Techniques with Different Code Representations, *Proc. 2023 IEEE/ACM 45th International Conference on Software Engineering*, IEEE, 2023, pp. 332–344.

[17] Zhu, W., Yoshida, N., Kamiya, T., Choi, E., and Takada, H.: MSCCD: Grammar Pluggable Clone Detection Based on ANTLR Parser Generation, *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 460–470.