特別研究報告

題目

ソースコード中の繰り返し部分に着目した
コードクローン検出手法の実装

指導教員
楠本 真二 教授

報告者
村上 寛明

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

ソースコード中の繰り返し部分に着目した コードクローン検出手法の実装

村上 寛明

内容梗概

コードクローン (ソースコード中の同一あるいは類似するコード片) の存在は，あるコード片に修正すべき箇所が見つかった場合，そのコード片とコードクローン関係にあるすべてのコード片について同様の修正を検討しなければならないため，ソフトウェアの修正に要する作業量を増大させるおそれがあると考えられている．そのため，これまでに多くのコードクローン検出手法が提案され，コードクローンを自動的に検出するツールが多数開発されている．しかし，既存の検出手法にはソースコード中で同じ文が繰り返し記述された箇所（以下，繰り返し部分と呼ぶ）において，小さなコードクローンが多く検出されるという問題点がある．そのようなコードクローンの検出はソフトウェア保守の観点において，必ずしも有益ではない．その理由は 2 つある．1 つ目は，繰り返し部分はプログラミング言語の性質上コードクローンとして検出されやすいからである．例えば，連続した変数宣言はクラスや関数の先頭で行われることが多く，それゆえ，クラスや関数の先頭で繰り返し部分が多くなる．よって，クラスや関数の先頭では小さなコードクローンが多く検出されやすくなるが，そのようなコードクローンは検出する必要がない．2 つ目は，繰り返し部分内の一部に対してリファクタリングなどの保守作業を行うことはないからである．

本研究では既存手法の問題点を解決するため，繰り返し部分を折りたたむという前処理を提案する．繰り返し部分を折りたたむことで，着目する必要のない冗長なコードクローンの検出を抑止するとともに，把握すべきコードクローンが検出できるようになる．さらに，提案手法を用いたコードクローン検出ツールを作成し，複数のオープンソースソフトウェアに対して，折りたたみを行う場合と行わない場合の 2 通りの検出を行った．その結果，折りたたみを行うことで繰り返し部分の検出数が減少したことを確認した．特に繰り返し部分を多く含むソフトウェアに対しては，提案する折りたたみ機能が効果的に働いたため，検出数の大幅な減少に繋がった．

主な用語

コードクローン
プログラム解析
ソフトウェア保守

目 次

1 まえがき **1**

2 準備 **3**

2.1 コードクローン 3

2.1.1 定義 3

2.1.2 発生の原因 4

2.2 コードクローン検出ツール 5

2.2.1 コードクローン検出ツールの分類 5

2.2.2 Dup 8

2.2.3 CloneDR 8

2.2.4 CCFinder 8

2.2.5 Duplix 9

2.2.6 CLAN 9

2.2.7 Duploc 9

2.2.8 Deckard 9

3 研究の動機 **10**

4 提案手法 **11**

5 実装 **13**

5.1 概要 13

5.2 処理の流れ 13

5.2.1 ステップ 1 : 字句解析・正規化 14

5.2.2 ステップ 2 : 変形処理 14

5.2.3 ステップ 3 : 繰り返し部分の折りたたみ 16

5.2.4 ステップ 4 : 一致部分文字列の検索 17

5.2.5 ステップ 5 : 出力整形処理 17

6 評価実験 **19**

6.1 準備 19

6.1.1 正解クローン 19

6.1.2 評価基準 21

6.2 折りたたみの有無によるコードクローン検出数の比較実験 23

6.2.1 目的 24
6.2.2 実験結果 24
6.3 既存手法との比較実験 26
6.3.1 目的 26
6.3.2 比較した検出手法 26
6.3.3 実験結果 26

7 考察 **27**
7.1 折りたたみを用いた検出例 27
7.2 折りたたみによる検出数の減少 27
7.3 既存手法で検出できず提案手法で検出できた新正解クローン 28
7.4 既存手法で検出できて提案手法で検出できなかった新正解クローン 30

8 実験結果の妥当性について **32**
8.1 正解クローンの作成法 32
8.2 検出手法におけるコードクローンの定義の違い 32
8.3 対象ソフトウェア 32

9 関連研究 **33**

10 あとがき **34**

謝辞 **35**

参考文献 **36**

1　まえがき

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェアの保守に要するコストが増加している．ソフトウェアの保守を困難にさせる要因の 1 つとしてコードクローンへの関心が高まっており，これまでにコードクローンに関する様々な研究が盛んに行われている [1].

コードクローンとは，ソースコード中に存在する同一あるいは類似したコード片のことであり，主にコピーアンドペーストの操作等によって発生するといわれている．あるコード片に修正すべき箇所が見つかった場合，そのコード片とコードクローン関係にある他のすべてのコード片に対して同様の修正の是非を検討しなければならない．このため，コードクローンの存在はソフトウェアの修正に要する作業量を増大させるおそれがあると考えられている. この問題に対処するため，コードクローンの情報を開発者が把握することは有益であると考えられている．しかし，人間がすべてのコードクローンを認識しておくことは現実的ではないため，これまでにコードクローンを検出する様々な手法が提案されている．またそれらの手法を実装したコードクローンを自動的に検出するツールも多数開発されている [2, 3, 4, 5, 6, 7, 8, 9].

しかし，既存の検出手法には，繰り返し部分で多くのコードクローンを見つけてしまうという問題がある．繰り返し部分の例として，switch 文の各 case エントリや連続した変数宣言，及び連続した同一関数の呼び出しなどが挙げられる．そのようなコードクローンの検出はソフトウェア保守の観点において，必ずしも有益ではない．その理由は，繰り返し部分はプログラミング言語の性質上コードクローンとして検出される可能性が高く，また，繰り返し部分の一部に対してリファクタリングなどソフトウェアの保守作業をすることは無いからである．例えば，類似した処理を行っている case エントリをもつ switch 文の場合，1 つの switch 文内において case エントリのみで構成されたコード片をコードクローンとして検出することは有益ではない．また，類似した処理を行っている case エントリをもつ switch 文が複数あり，かつ，各 switch 文内の case エントリの数が異なるならば，case エントリのみで構成されたコードクローンが多数検出されるが，このようなコードクローンは有益ではない．これらの例においてユーザにとって有益な情報とは，1 つの switch 文の場合はコードクローンを持たないという情報で，複数の switch 文の場合は switch 文全体が 1 つのコードクローンであるという情報である．

そこで本研究では，より有益なコードクローン検出結果を得るために，ソースコード上の繰り返し構造を折りたたむという検出の前処理を提案する．繰り返し部分を折りたたむことで，着目する必要のない冗長なコードクローンの検出を抑止するとともに，把握すべきコードクローンを検出できるようになる．また，提案手法を用いたコードクローン検出ツールを作成し，複数のオープンソースソフトウェアに対して検出を行った．その結果，折

りたたみ前に比べて折りたたみ後の検出数が減少した．特に繰り返し部分を多く含むソフトウェアに対しては効果が大きく，検出数が半数以下になることを確認できた．また，既存手法との比較を通じて評価を行った．評価項目は，既存手法，提案手法を用いた各ツールが検出したコードクローンの再現率と適合率である．代表的なコードクローン検出ツールであるCCFinder[4] と比較したところ，再現率は 1〜1.2 倍，適合率は 1.5〜4 倍程度上昇したという結果が得られた．

以降，2 節ではコードクローン，コードクローンが発生する原因，及び，コードクローン検出ツールについて述べる．3 節では研究の動機を説明する．4 節で提案手法を説明し，5 節で実装について説明する．6 節では折りたたみの有無による検出数の違いを計測する実験と，既存手法との検出精度の比較実験を行う．7 節で 2 つの実験結果の考察を行う．8 節で実験の妥当性について述べ，9 節で関連研究について述べる．最後に 10 節で本研究のまとめと今後の課題について述べる．

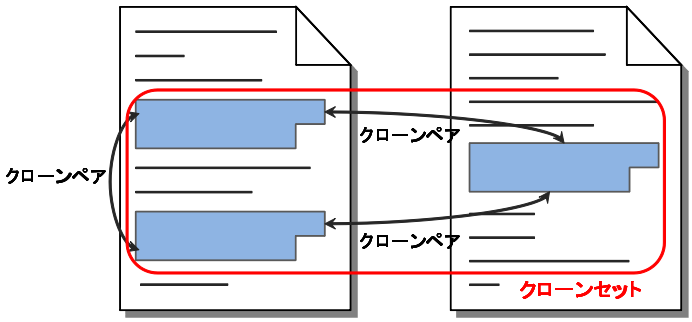


図 1: クローンペアとクローンセット

2 準備

2.1 コードクローン

2.1.1 定義

コードクローンとはソースコード中に存在する同一あるいは類似するコード片のことである．図 1 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [10].

ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

また，コードクローン間の類似の度合いに基づきコードクローンを次の 3 つのタイプに分類することができる [11][12].

Type-1 空白やタブの有無，括弧の位置などのコーディングスタイルに依存する箇所を除いて，完全に一致するコードクローン．

Type-2 変数名や関数名などのユーザ定義名，また変数の型など一部の予約語のみが異なるコードクローン．

Type-3 Type-2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン．

2.1.2 発生の原因

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものが挙げられる [4][3][13].

既存コードのコピーアンドペーストによる再利用

近年のソフトウェア設計手法を利用することにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンドペーストによる場当たり的な既存コードの再利用が多く行われるようになった．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである．

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用いた処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは，あらかじめ決められたコードをベースにして自動的に生成される．このため，類似した処理を目的としたコードを生成した場合，識別子名等の違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.2 コードクローン検出ツール

2.2.1 コードクローン検出ツールの分類

コードクローンを検出する手法はこれまでに多数提案されている．またそれらを実装した，コードクローンを自動的に検出するツールも多数開発されている．これらの検出技術はコードクローンをどの単位で検出するかによって，大まかに以下の 5 つに分類することができる [1]．

行単位の検出

行単位の検出は，ソースコードを行単位で比較してコードクローンを検出する手法であり，閾値以上連続して一致する行をコードクローンとして検出する．他の手法と異なり，ソースコードに対する事前処理を必要としない．このため，他の手法と比べて高速にコードクローンを検出可能である．しかし，同じ処理を行っているコードであっても，例えば長い行を複数行に分割した場合と分割しなかった場合など，コーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

字句単位の検出

字句単位の検出は，ソースコードを字句単位に分割し，閾値以上連続して一致する字句の部分列をコードクローンとして検出する手法である．行単位の検出と異なり，コーディングスタイルのみ違う場合などもコードクローンとして検出することが可能である．ソースコー

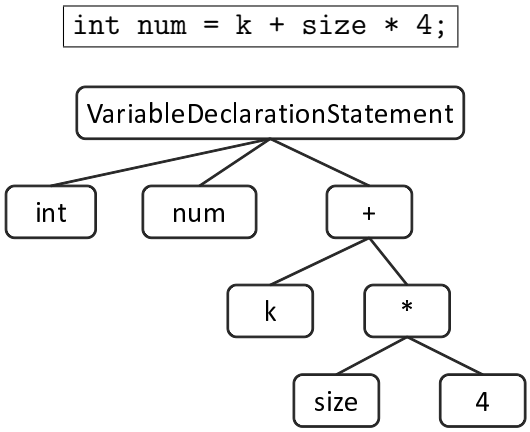


図 2: 抽象構文木の例

ドを検出用の中間表現に変換する必要がないため，高速にコードクローン検出を行うことができるという利点もある．また，字句に事前処理を行うことで変数名などのユーザ定義名のみ異なるコードクローンなども検出可能となる．

抽象構文木を用いた検出

抽象構文木 (図 2) を用いた検出は，ソースコードに対して構文解析を行い，抽象構文木を構築した後，その抽象構文木を用いてコードクローンを検出する手法であり，抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を構築するという事前処理を要するため，行単位の検出や字句単位の検出と比べ，時間的，空間的コストが高くなるという欠点がある．ある関数定義の終わりから次の関数定義の先頭まで検出された類似部分など，プログラムの構造を無視したコードクローンを検出しないという特徴を持つ．

プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3，[14]) を用いた検出は，ソースコードに対して意味解析を行い，ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後，そのプログラム依存グラフを用いてコードクローン検出を行う手法である．プログラム依存グラフ上の同形部分グラフがコードクローンとして検出される．抽象構文木を用いた検出と同様に事前処理を必要とするため，時間的，空間的コストが高くなるという欠点を持つ．ソースコードの順番が入れ替わっていても意味的に同一であるコードクローン (順序入れ替わりコードクローン) などは意味的な処理を考慮しなければ検出できないが，この手法はこれらのコードクローンを検出することができるという点が特徴として挙げられる．

```
1: void sample() {
2:   for ( int i = 0 ; i < 10 ; i++ ) {
3:     System.out.println(i);
4:   }
5: }
```

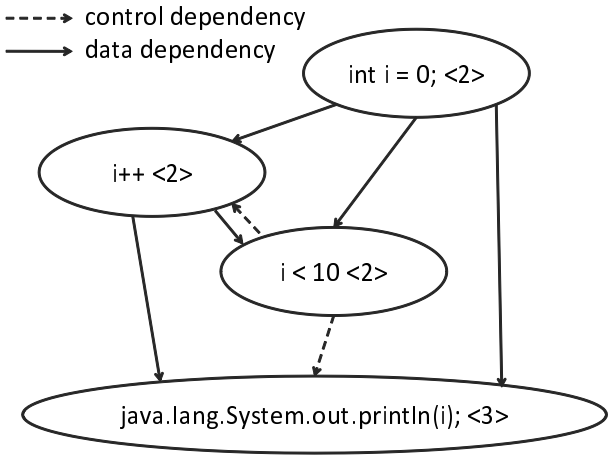


図 3: プログラム依存グラフの例

```
  fp = lookaheadset + tokensetsize;
  for (l = lookaheas(state) ; l < k ; i++) {
%     fp1 = LA + i * tokensetsize;
%     fp2 = lookaheadset;
%     while (fp2 < fp3)
%           *fp2++ |= fp1++;
  }
```

(a) コード片 1

```
  fp3 = base + tokensetsize;
  ...
  if (rp) {
      while ((j = *rp++) >= 0) {
          ...
#             fp1 = lookaheadset;
#             fp2 = LA + j * tokensetsize;
#             while (fp1 < fp3)
#                   *fp1++ |= *fp2++;
      }
  }
```

(b) コード片 2

図 4: 順序入れ替わりコードクローン

順序入れ替わりコードクローンの例を図 4 に示す．この例の場合，%で表されているコード片と，#で表されているコード片が順序入れ替わりコードクローンとなる．

その他の技術を用いた検出

その他の技術を用いた検出手法として，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法であるメトリクスを用いた検出や，プログラムの盗用の検出やプログラムの作者を特定することを目的とした，フィンガープリントやバースマークを用いた検出手法などがある．

以下の小節では，本研究に用いたコードクローン検出ツールについて述べる．

2.2.2 Dup

Dup[2][15][16] は，Baker によって開発されたコードクローン検出ツールであり，行単位での検出手法に分類される．前処理としてユーザ定義名をパラメータ化している．マッチング処理には Suffix Tree を用いているため線形時間での検出が可能である．

2.2.3 CloneDR

CloneDR[3] は，Baxter らによって開発されたコードクローン検出ツールであり，抽象構文木を用いた検出手法に分類される．入力されたソースコードから抽象構文木を作成した後，抽象構文木の各ノードの比較にハッシュ値を使用する．また，同形部分木が含んでいるトークンの類似度を計算し，類似度が閾値以上ならばコードクローンとして検出するという設定を行うことも可能である．

2.2.4 CCFinder

CCFinder[4][17] は，Kamiya らによって開発されたコードクローン検出ツールであり，字句単位の検出手法に分類される．ソースコードに対し事前処理を施すことで，ユーザ定義名，定数，名前空間，コンパウンドブロックの中括弧表記などの違いを吸収している．大規模ソフトウェアに対して実用的な時間とメモリ消費量で検出できることも特徴である．また，GUI フロントエンドである Gemini[18] により，コードクローンの散布図や実際のソースコードを見ることができる．

2.2.5 Duplix

Duplix[5] は，Krinke によって開発されたコードクローン検出ツールであり，プログラム依存グラフを用いた検出手法に分類される．このツールはプログラム依存グラフに細かい情報付け加えた Fine-grained PDG を定義し，それを利用してコードクローンを検出する．

2.2.6 CLAN

CLAN[6] は，Merlo らによって開発されたコードクローン検出ツールであり，メトリクスを用いた検出手法に分類される．ソースコード中の関数に対して 21 種類のメトリクスを計測し，その類似度を比較することでコードクローンの検出を実現している．

2.2.7 Duploc

Duploc[7] は Ducasse らによって開発されたコードクローン検出ツールであり，行単位での検出手法に分類される．行単位で表検索を用いた比較によって，コードクローンの検出を実現している．また，Duploc はコードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行う．

2.2.8 Deckard

Deckard[8] は，Jiang らによって開発されたコードクローン検出ツールであり，抽象構文木を用いた検出手法に分類される．抽象構文木の各部分木を配列表現に変換し，局所感度ハッシュアルゴリズム [19] を用いて類似配列を求めることによって，コードクローンを検出する．

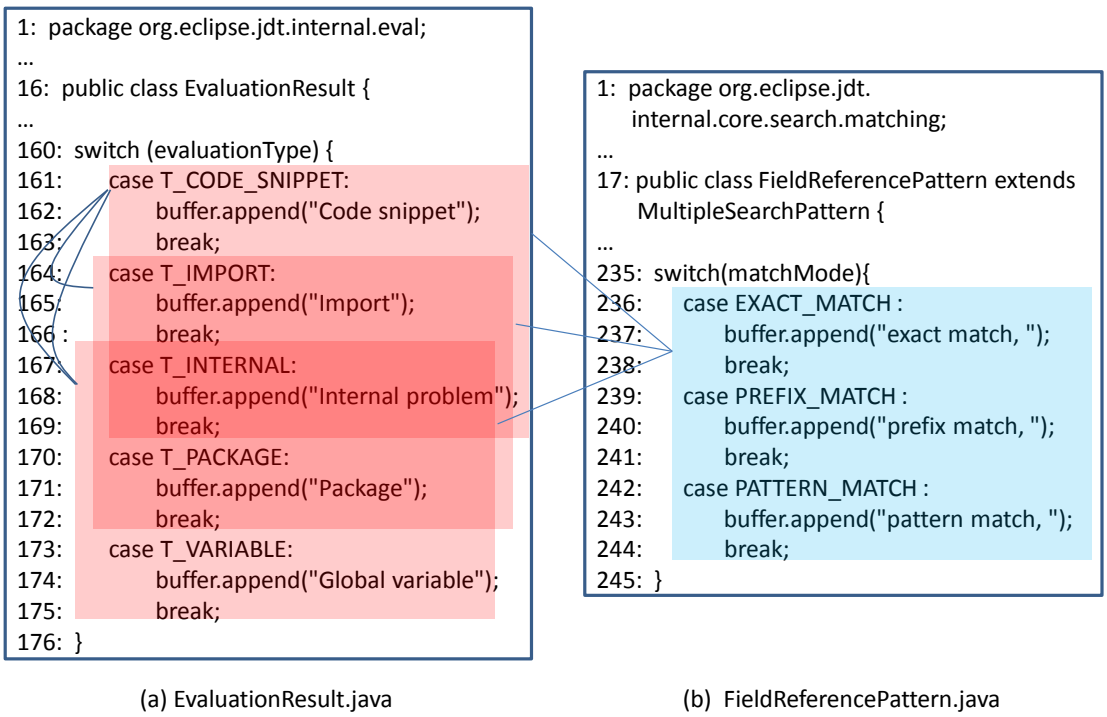


図 5: 繰り返し部分における既存手法の検出例

3 研究の動機

1 節で述べたように，既存手法は繰り返し部分で多くのコードクローンを検出するという課題点を抱えている．図 5 は冗長なコードクローン検出の例を表している．図 5(a) と図 5(b) の switch 文は，それぞれ 5 つと 3 つの case エントリを含んでいる．これら 8 つの case エントリは，リテラルが違うのみの繰り返し構造となっている．この switch 文に対して既存の検出ツールを適用すると 6 つものコードクローンのペアを検出してしまう．しかし，このソースコードにおいて，ユーザが必要とする情報は，図 5(a) と図 5(b) の switch 文は共に StringBuffer を用いた文字列の追加処理を行なっているということであると考えられる．よって，switch 文全体をコードクローンのペアとして検出することが望ましい．

コードクローン検出ツール CCFinder は既存ツールの中でも，必要なコードクローンを漏れなく検出できている．しかし，繰り返し部分で多くのコードクローンを検出するという課題点を抱えている．よって，CCFinder と同じ程度の検出精度を保ちつつ繰り返し部分の折りたたみを行えば，より有用なコードクローン検出ツールを作ることができる．

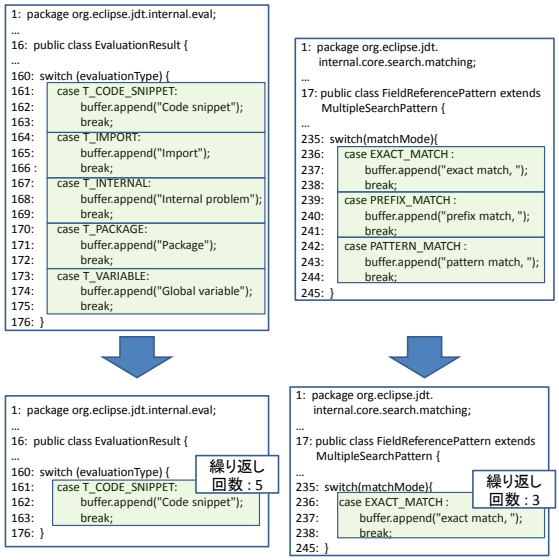


図 6: 折りたたみの様子

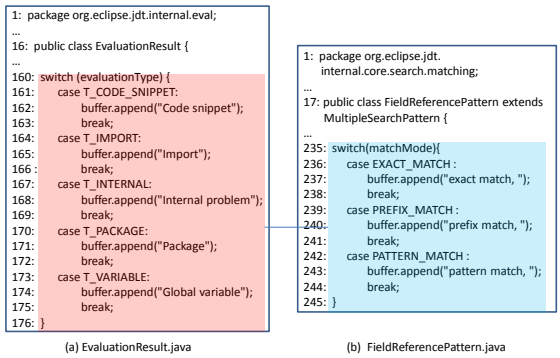


図 7: 繰り返し部分における提案手法の検出例

4 提案手法

この節では，本研究の提案手法である繰り返し部分の折りたたみについて，3 節で挙げた例を元に説明する．この例では switch 文の各 case エントリが繰り返されている．この繰り返し部分を折りたたむと図 6 のように 2 つの switch 文が 1 つの case エントリのみをもつ構造になる．繰り返し部分の繰り返し回数を記憶し，ユーザ定義名を特別な同一の字句に置き換えると，単純な字句単位の解析で switch 文全体をコードクローンとして検出することができる．検出例を図 7 に示す．図 7(a) の 160 行目〜176 行目と図 7(b) の 235 行目〜245 行目をクローンペアとして検出できる．

折りたたみのアルゴリズムを Algorithm1 に示す．このアルゴリズムを適用する前の段階で，ソースコードは正規化され，1 つの文字列となっている．引数の str は折りたたむ文字列，repeatCount は折りたたむ文の数を表す．例えば，repeatCount が 2 であれば「文 1，文 1」と「文 1，文 2，文 1，文 2」といった繰り返しを折りたたみ,「文 1，文 2，文 3，文 1，文 2，文 3」のような繰り返しは折りたたまない．アルゴリズム中の $isSentenceEnd()$ は，与えられた文字が”;”，”{”，”}”のいずれかであれば true を，そうでなければ false を返す手続きである．また，$length()$ は，与えられた文字列の長さを返す手続きである．6-18 行目と 21-30 行目で隣接する 2 つの文を取得する．31-37 行目で隣接する 2 つの文が同じだったときの折りたたみの処理を行う．repeatCount の数だけ文の折りたたみを行うと，最後に 40 行目で折りたたまれた文字列を返す．

Algorithm 1 fold Repetition

Input: $str, repeatCount(\geq 1)$
Output: str after folded

```
1: strlen ← length(str)
2: for i = 0 to repeatCount do
3:    left ← 0
4:    loop
5:       flg ← true; index ← left; tmpleft ← left; count ← 0;
6:       while count ≤ i and index < strlen do
7:          if isSentenceEnd(str[index]) then
8:             if flg then
9:                k ← index + 1; flg ← false
10:            end if
11:            count ← count + 1
12:         end if
13:         index ← index + 1
14:      end while
15:      if index > strlen then
16:         break
17:      end if
18:      tmp ← str[left..index − 1]
19:      count ← 0
20:      left ← index
21:      while count ≤ i and index < strlen do
22:         if isSentenceEnd(str[index]) then
23:            count ← count + 1
24:         end if
25:         index ← index + 1
26:      end while
27:      if index > strlen then
28:         break
29:      end if
30:      tmp2 ← str[left..index − 1]
31:      if tmp = tmp2 then
32:         str ← str[0..left − 1] + str[index..strlen]
33:         strlen ← length(str)
34:         left ← tmpleft
35:      else
36:         left ← k
37:      end if
38:   end loop
39: end for
40: return str
```

図 8: 処理の流れ

5 実装

5.1 概要

ツールは Java を用いて実装した．現在のところ，コードクローンの検出が可能な言語は Java と C である．提案手法を組み込んだコードクローン検出ツールは，検出対象のソースコードを入力とし，コードクローンの情報を出力とする．また，5.2 で述べる各ステップにおいて，それぞれのステップ内の処理は互いの処理内容に影響を及ぼさないので並列処理が可能である．実装したツールでは，使用する計算機のプロセッサの数だけのスレッドを作成し，それらを並列に処理させることで高速化を図っている．

5.2 処理の流れ

提案手法を組み込んだコードクローン検出ツールは，検出対象のソースコードを入力とし，コードクローンの情報を出力とする．提案手法の全体の流れを図 8 に示す．コードクローン検出は以下の 5 ステップで行われる．

ステップ 1：**字句解析・正規化**

入力されたソースファイルに対して字句解析と正規化を行い，トークン列に変換する．この文字列は入力ファイルの数だけ得られる．正規化により，ユーザが定義した変数名，関数名，型名は同一文字に置換される．また，文字と行番号とのマッピングも行う．

ステップ 2：**変形処理**

ステップ 1 で得られた文字列を変形ルールに従い変形する．変形処理は主にソフトウェアの保守に有用でないコードクローンを取り除く．

ステップ 3：**繰り返し部分の折りたたみ**

文字列内に繰り返し部分があった場合，繰り返されている文字列の先頭だけ残し，残りの文字列を取り除く．取り除いた後の文字列を新しい文字列とする．

ステップ 4：**一致部分文字列の検出**

文字列の中から指定された長さ以上の一致部分文字列を検出する．検出には Suffix Array[20] を用いる．

ステップ 5：**出力整形処理**

ステップ 4 で得られた一致部分文字列をクローンペアとして出力する．

以下，それぞれのステップについて，その処理内容を詳細に述べる．

5.2.1 ステップ 1：字句解析・正規化

まず，字句解析によって入力ソースコードをトークン列に変換する．この際，コメントや改行記号，タブ，空白は取り除かれる．次に，ユーザ定義名を”$”に変換する．同時に，中括弧のネストを認識することで，メソッドや関数の境界を識別し，トークン列内におけるそれらの境界に”#”を埋め込む．また，それぞれの文字が何行目にあるかを記録しておく．以下，各ステップにおいて文字列を変形させるが，各文字と行番号との整合性は常に保っておく．図 9 に字句解析・正規化の例を示す．

5.2.2 ステップ 2：変形処理

このステップでは字句解析・正規化によって得られた文字列を変形ルールに従って変形する．Java と C の変形ルールを以下に示す．

・パッケージ名を取り除く

PackageName + ’.’ + ClassName → ClassName

パッケージ内のクラスやインターフェースを呼び出す命令があった場合，パッケージ名を省略する．なお，この変形ルールは Java にのみ適応する．

・テーブルの初期化を取り除く

’=’ ’{’ InitalizationList ’}’ → ’=’ ’{’ ’}’

テーブルを初期化するときに要素が指定されていれば，それを取り除く．

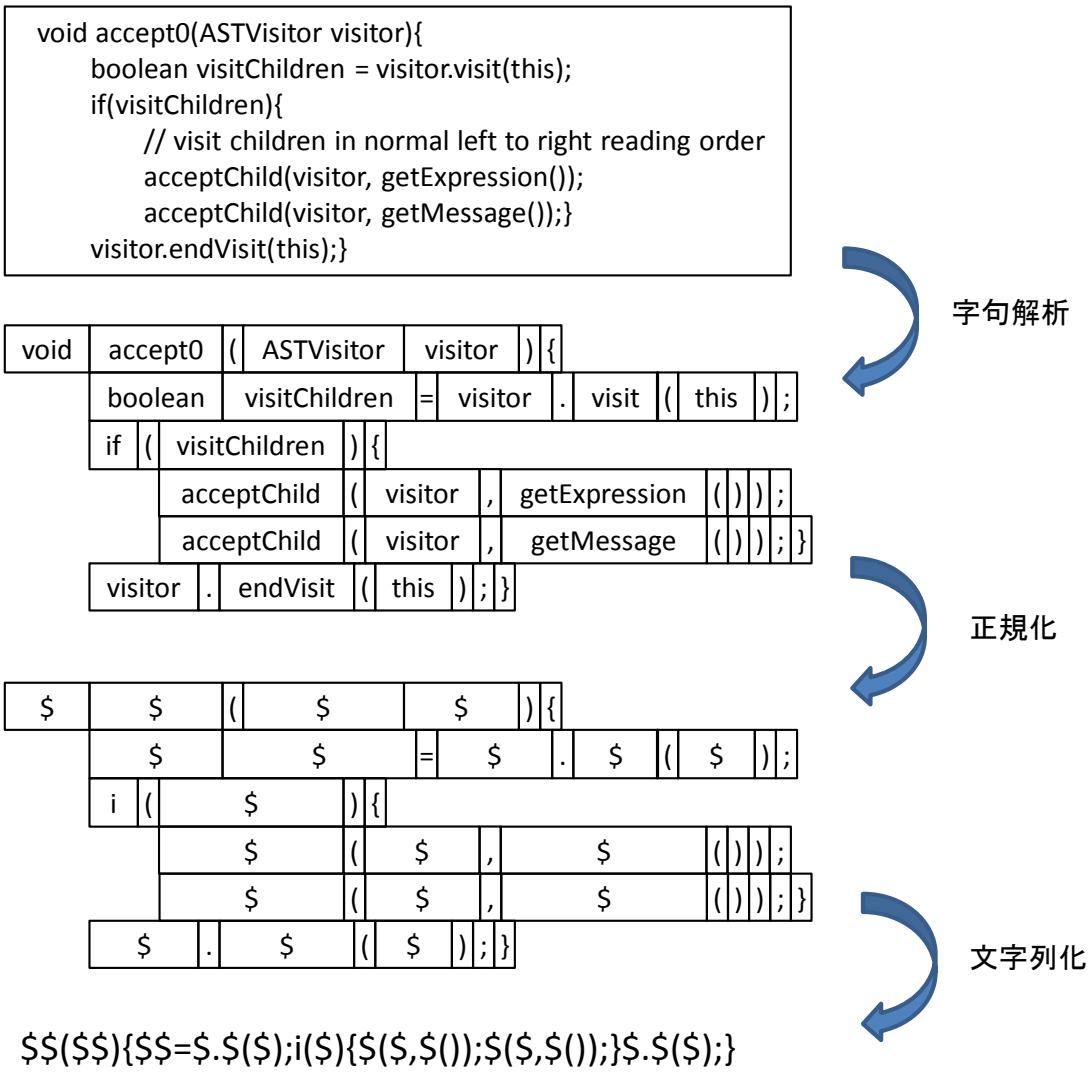


図 9: 字句解析・正規化の例

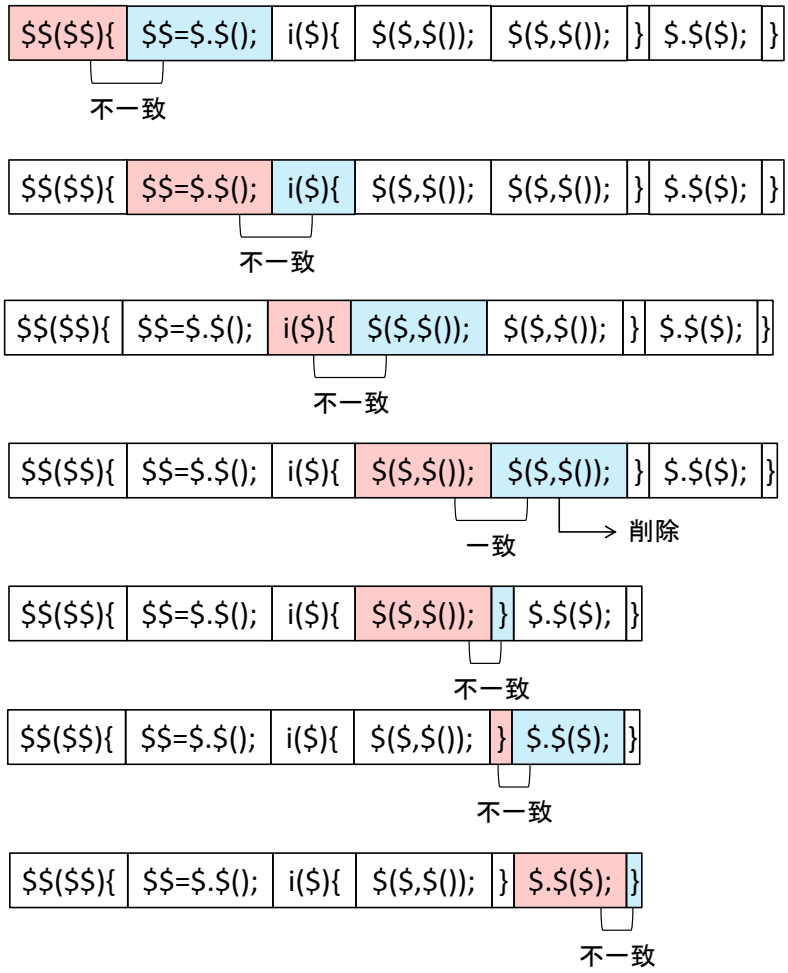


図 10: 単一の文に対する折りたたみの例

5.2.3 ステップ 3：繰り返し部分の折りたたみ

変形処理によって得られた文字列の中で繰り返している部分を取り除く．まず，”;”，”{”，”}”を「区切り文字」と定義し，区切り文字で終わる文字列を「文」と定義する．次に，変形処理で得られた文字列を文単位で区切る．隣接する文が同じであれば同じ文が繰り返されているということであり，繰り返されている後ろの文を取り除く．折りたたみの例を図 10 に示す．また，複数の文の繰り返し（「文 1，文 2，文 1，文 2」のような繰り返し）を含む文字列に対しても同様の処理を行う．例を図 11 に示す．

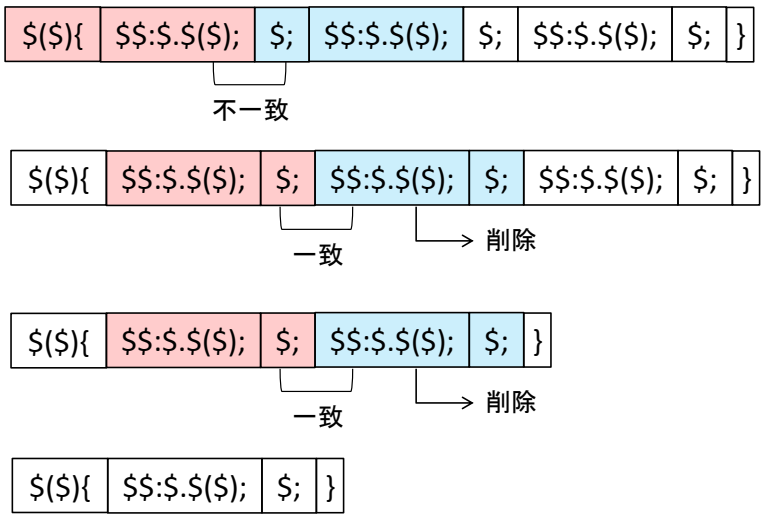


図 11: 複数の文に対する折りたたみの例

5.2.4 ステップ 4：一致部分文字列の検索

このステップでは，折りたたみによって得られた文字列の中から一致部分文字列を検索する．検索には Suffix Array[20] を用いる．Suffix Array の構築，および，Suffix Array による一致部分文字列検索アルゴリズムを示す.

Suffix Array の構築アルゴリズム

Suffix Array を構築するアルゴリズムを Algorithm2 に示す．アルゴリズム中の変数 suffixarray は String 型のリストを表している．*sort*() は，与えられたリストを昇順にソートする手続きである．通常の Suffix Array は各文字の接尾辞を配列に格納するが，提案手法では各文の接尾辞を格納する.

Suffix Array による検索アルゴリズム

A.2.1 節で構築した Suffix Array は各文の接尾辞が辞書順に並べられているので，高速な検索アルゴリズムである二分探索を使用することができる.. Suffix Array による検索のアルゴリズムを Algorithm3 に示す.

5.2.5 ステップ 5：出力整形処理

検索した 2 つの一致部分文字列の開始行と終了行をクローンペアの開始行と終了行として出力する.

Algorithm 2 make Suffix Array

Input: *str*

Output: *suffixarray* made of *str*

```
1: strlen ← length(str)
2: initialize suffixarray
3: add str to suffixarray
4: for index = 0 to strlen − 1 do
5:     if isSentenceEnd(str[index]) then
6:         add str[index + 1..strlen] to suffixarray
7:     end if
8: end for
9: sort(suffixarray)
10: return suffixarray
```

Algorithm 3 find Suffix Index

Input: *str*

Output: *index of suffixarray*

```
1: strlen ← length(str)
2: low ← 0
3: high ← strlen − 1
4: while low < high do
5:     middle ← low + (high − low)/2
6:     if suffixarray[middle] > str then
7:         high ← middle
8:     else
9:         low ← middle + 1
10:    end if
11: end while
12: if suffixarray[middle] startwith str then
13:    return len − length(suffixarray[high])
14: end if
15: return −1
```

6 評価実験

6.1 準備

これまでに，コードクローン検出ツールの性能評価実験がいくつか行われている [12, 21, 22]. 今回の実験は，これらの評価実験の中で最も大規模である Bellon らの実験に沿って行う．

6.1.1 正解クローン

Bellon らの実験により得られたデータ [23] を検出すべきコードクローンとした．このコードクローンの情報は下記の手順で作成された．

(1) Bellon が検出対象ソフトウェアを決め，6 人のコードクローン検出ツールの開発者に検出を依頼した．Bellon はコードクローン検出ツールの開発者ではないため，中立的な立場で検出対象ソフトウェアを選んでいると述べている [12].

(2) 各開発者は，自身が開発した検出ツールを用いて，対象ソフトウェアからコードクローンを検出した．所定のフォーマットに従って，検出したコードクローンの位置情報を Bellon に送付した．

(3) Bellon は各開発者から送られたコードクローンの全ペア（325,935 個）のうちの 2%を無作為に選択し，それらが本当にコードクローンであるかを手作業により判定した．ツールが検出したコードクローンがそのままコードクローンと判断される場合もあるが，Bellon が必要に応じてコードクローンを加工して 1 つのコードクローンと判断した場合もある．この結果 4 つの対象ソフトウェアから 4,789 個のコードクローンのペアが抽出された．

以降，この実験では，ツールが検出したクローンペアの集合を**クローン候補**，Bellon が抽出したクローンペアの集合を**正解クローン**と呼ぶ．しかし，正解クローンの中にコードクローンとして適切でないものが多数見られた．例えば，複数のメソッドにまたがってコードクローンとしている例（図 12）や，メソッドがないクラスのみをコードクローンとしている例（図 13）である．今回の実験では，正解クローンの中からこれらのコードクローンを取り除いたものを新たな正解クローン（以降，**新正解クローン**と呼ぶ）とした．対象ソフトウェアにおける正解クローンと新正解クローンの個数を表 1 に示す．以降，対象ソフトウェアの表記にあたり，表 1 に記載されている省略名を使用する．

```
7:  package javax.swing.plaf.multi;
…
23: public class MultiButtonUI extends ButtonUI {
…
90:  public Dimension getPreferredSize(JComponent a) {
91:    Dimension returnValue =
92:        ((ComponentUI) (uis.elementAt(0))).getPreferredSize(a);
93:    for (int i = 1; i < uis.size(); i++) {
94:        ((ComponentUI) (uis.elementAt(i))).getPreferredSize(a); }
95:    return returnValue; }
96:  /**
97:   * Invokes the <code>getMinimumSize</code>
       method on each UI handled by this object.
98:   *
99:   * @return the value obtained from the first UI, which is
100:  * the UI obtained from the default <code>LookAndFeel</code>
101:  */
102: public Dimension getMinimumSize(JComponent a) {
103:     Dimension returnValue =
104:         ((ComponentUI) (uis.elementAt(0))).getMinimumSize(a);
105:     for (int i = 1; i < uis.size(); i++) {
106:         ((ComponentUI) (uis.elementAt(i))).getMinimumSize(a); }
107:     return returnValue; }
```

(a) MultiButtonUI.java

```
7:   package javax.swing.plaf.multi;
…
23: public class MultiLabelUI extends LabelUI {
…
114: public Dimension getMaximumSize(JComponent a) {
115:     Dimension returnValue =
116:         ((ComponentUI) (uis.elementAt(0))).getMaximumSize(a);
117:     for (int i = 1; i < uis.size(); i++) {
118:         ((ComponentUI) (uis.elementAt(i))).getMaximumSize(a); }
119:     return returnValue; }
120:  /**
121:   * Invokes the <code>getAccessibleChildrenCount</code>
       method on each UI handled by this object.
122:   *
123:   * @return the value obtained from the first UI, which is
124:   * the UI obtained from the default <code>LookAndFeel</code>
125:   */
126: public int getAccessibleChildrenCount(JComponent a) {
127:     int returnValue =
128:         ((ComponentUI) (uis.elementAt(0))).getAccessibleChildrenCount(a);
129:     for (int i = 1; i < uis.size(); i++) {
130:         ((ComponentUI) (uis.elementAt(i))).getAccessibleChildrenCount(a); }
131:   return returnValue; }
```

(b) MultiLabelUI.java

図 12: 複数のメソッドにまたがったコードクローン

```
1: /*
2: * @(#)SliderUI.java   1.10 01/12/03
3: *
4: * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
5: * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6: */
7: package javax.swing.plaf;
8: /**
9:  * Pluggable look and feel interface for JSlider.
10: *
11: * @version 1.10 12/03/01
12: * @author Hans Muller
13: */
14: public abstract class SliderUI extends ComponentUI { }
```

(a) SliderUI.java

```
1:/*
2: * @(#)TableUI.java    1.10 01/12/03
3: *
4: * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
5: * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6: */
7: package javax.swing.plaf;
8: /**
9:  * Pluggable look and feel interface for JTable.
10: *
11: * @version 1.10 12/03/01
12: * @author Alan Chung
13: */
14: public abstract class TableUI extends ComponentUI { }
```

(b) TableUI.java

図 13: メソッドが無いクラスのみのコードクローン

表 1: 対象ソフトウェアにおける正解クローンと新正解クローンの個数

ソフトウェア	言語	省略名	行数	正解クローン数	新正解クローン数
netbeans-javadoc	Java	netbeans	14,360	55	39
eclipse-ant	Java	ant	34,744	30	11
eclipse-jdtcore	Java	jdt	147,634	1,345	1,160
j2sdk1.4.0-javax-swing	Java	swing	204,037	777	420
weltab	C	weltab	11,460	275	261
cook	C	cook	70,008	440	306
snns	C	snns	93,867	1,036	1,030
postgresql	C	postgresql	201,686	555	551

6.1.2 評価基準

この実験では，クローン候補が正解クローンにどれだけ類似しているかを good 値と ok 値を用いて判断し，再現率と適合率を算出する．good 値と ok 値の計算式を式 (1)，式 (2) に示す．ただし，説明にあたりクローンペア CP_1 を構成するコード片 CF_1 および CF_2 を $CP_1.CF_1$ および $CP_1.CF_2$ と表記し，$lines(CF_1)$ を CF_1 に含まれる行の集合とする．

$$\begin{aligned} good(CP_1, CP_2) := min(\quad & overlap(CP_1.CF_1, CP_2.CF_1), \\ & overlap(CP_1.CF_2, CP_2.CF_2)) \end{aligned} \tag{1}$$

$$\begin{aligned} ok(CP_1, CP_2) := min(\quad & max(\quad contained(CP_1.CF_1, CP_2.CF_1), \\ & \qquad contained(CP_2.CF_1, CP_1.CF_1)), \\ & max(\quad contained(CP_1.CF_2, CP_2.CF_2), \\ & \qquad contained(C\_2.CF_2, CP_1.CF_2))) \end{aligned} \tag{2}$$

ただし，

$$overlap(CF_1, CF_2) := \frac{|lines(CF_1) \cap lines(CF_2)|}{|lines(CF_1) \cup lines(CF_2)|} \tag{3}$$

$$contained(CF_1, CF_2) := \frac{|lines(CF_1) \cap lines(CF_2)|}{|lines(CF_1)|} \tag{4}$$

good 値はクローン候補の要素と正解クローンの要素がどれだけ重なっているかを示す割合である．ok 値はクローン候補の要素と正解クローンの要素がどれだけ包含しているかを示す割合である．クローン候補の要素 c と正解クローンの要素 r の good 値が閾値以上であれば,「c と r は good である」とする．クローン候補の要素 c と正解クローンの要素 r の ok 値が閾値以上であれば,「c と r は ok である」とする．Bellon の実験 [12] では閾値として 0.7 が用いられており，本実験でもそれを用いた．

図 14 を用いて good 値と ok 値の算出例を示す．図 14 には 2 つのクローンペア (CP_1 と CP_2) が存在している．このときの good 値は，

$$good(CP_1, CP_2) = min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8} \leq 0.7 \tag{5}$$

となるため，CP_1 と CP_2 は good でない．また，ok 値は，

$$ok(CP_1, CP_2) = min(max(\frac{5}{6}, \frac{5}{7}), max(\frac{6}{6}, \frac{6}{8})) = \frac{5}{6} \geq 0.7 \tag{6}$$

となるため，CP_1 と CP_2 は ok である．

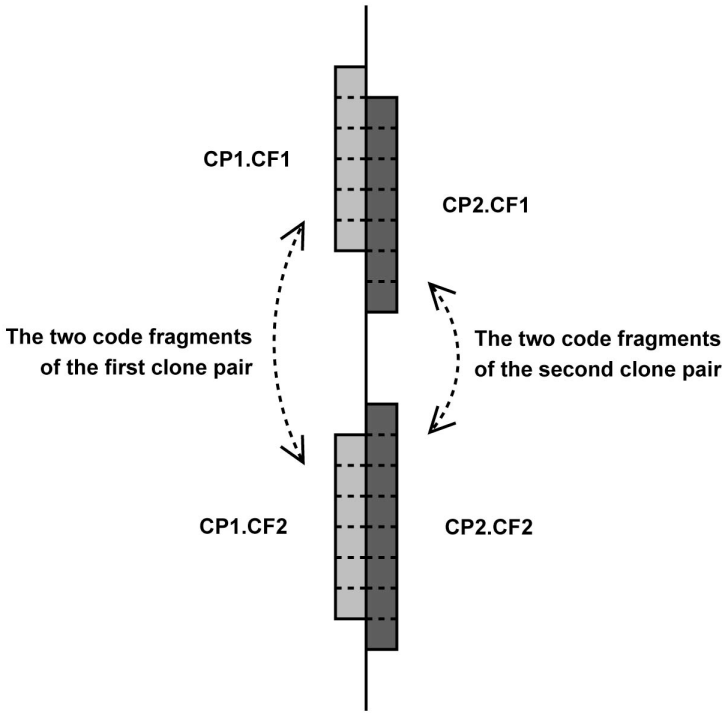


図 14: 2 つのクローンペアの重なり

ここで，クローン候補の要素が，どの正解クローンの要素に最も類似しているかを求める．つまり，ok 値もしくは good 値が最大となる正解クローンとクローン候補のマッピングを求める．Algorithm4 にそのアルゴリズムを示す．アルゴリズム中の Reference は正解クローン，Candidates はクローン候補，BestReferenceOf[c] はクローン候補の要素 c にマッピングされている正解クローンの要素を表す．ok(c, r) はクローン候補の要素 c と正解クローンの要素 r の ok 値を求める手続きである．good(c, r) はクローン候補の要素 c と正解クローンの要素 r の good 値を求める手続きである．また，手続き better は以下のいずれかを満たすならば true となり，どれも満たさなければ false となる．p は閾値であり，今回の実験では 0.7 とする．

1. $good \geq p \wedge good > good\_max$
2. $good = good\_max \wedge ok > ok\_max$
3. $ok \geq p \wedge ok\_max < p$

Algorithm4 により，正解クローンとクローン候補のマッピングが求まる．つまり，各正解クローンに一番類似しているクローン候補がそれぞれ求まる．次に，recall と precision の

Algorithm 4 mapping from candidates to reference

```
1: for all r References do
2:    for all c in Candidates do
3:       ok_max := ok(c, BestReferenceOf[c])
4:       good_max := goode(c, BestReferenceOf[c])
5:       ok := ok(c, r)
6:       good := good(c, r)
7:       if better then
8:          BestReferenceOf[c] := r
9:       end if
10:   end for
11: end for
```

計算式を式 (3)，式 (4) に示す.

対象プログラム P，検出ツール T，クローンタイプ τ に対し，

$$recall(P, T, \tau) := \frac{|DetectedRefs(P, T, \tau)|}{|Refs(P, \tau)|} \tag{7}$$

$$precision(P, T, \tau) := \frac{|DetectedRefs(P, T, \tau)|}{|Cands(P, \tau)|} \tag{8}$$

ただし，

$$DetectedRefs(P, T, \tau) := OKrefs(P, T, \tau) \cap GoodRefs(P, T, \tau) \tag{9}$$

OKrefs(P, T, τ) はプログラムが P でクローンタイプが τ である正解クローンの中で，検出ツール T のクローン候補と ok の基準で等しいものの集合を表す.

Goodrefs(P, T, τ) はプログラムが P でクローンタイプが τ である正解クローンの中で，検出ツール T のクローン候補と good の基準で等しいものの集合を表す.

Refs(P, τ) はプログラムが P でクローンタイプが τ である正解クローンの集合を表す.

$Cands(P, T, \tau)$ は検出ツール T がプログラム P に対して検出したコードクローンのうち，クローンタイプが τ であるものの集合を表す.

6.2 折りたたみの有無によるコードクローン検出数の比較実験

実装したツールを用いて，複数のオープンソースソフトウェアに対して実験を行った．表 1 のソフトウェアが対象となるソフトウェアである.

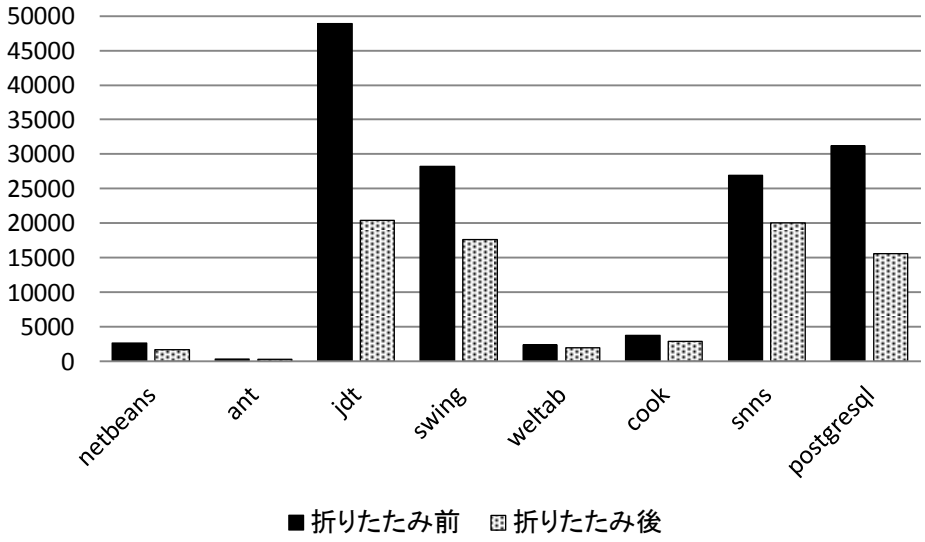


図 15: 折りたたみの有無におけるコードクローンの検出数

6.2.1 目的

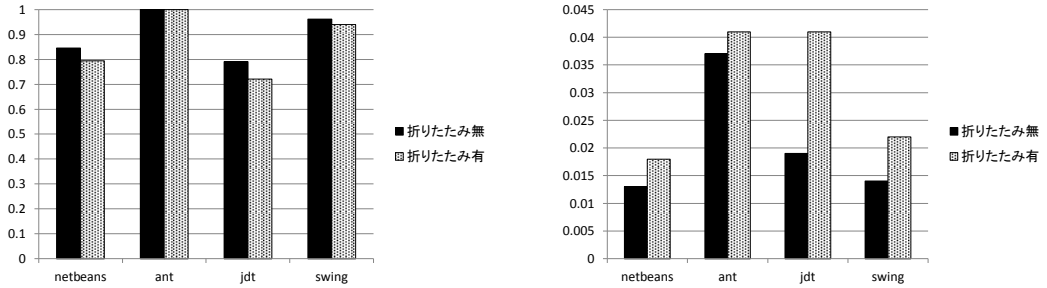
この実験の目的は，折りたたみの有無でコードクローン検出数・精度の変化を調べることである．具体的な評価項目は以下の点である．

- 折りたたみの有無におけるコードクローンの検出数
- 折りたたみの有無におけるコードクローンの recall（再現率）
- 折りたたみの有無におけるコードクローンの precision（適合率）

6.2.2 実験結果

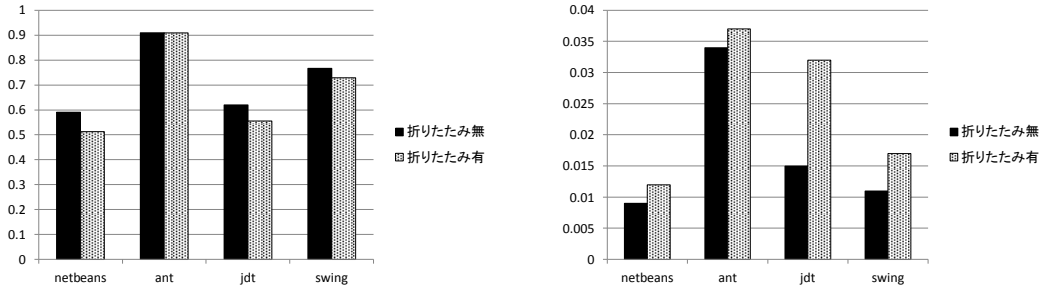
検出数についての実験結果を図 15 に示す．図 15 に示すように，すべてのソフトウェアにおいてコードクローンの検出数が減少した．

recall と precision についての実験結果を，Java プログラムは図 16 に，C プログラムは図 17 にそれぞれ示す．繰り返し部分を折りたたむことで大半のソフトウェアについて recall は減少し，precision は上昇した．



(a) okrecall

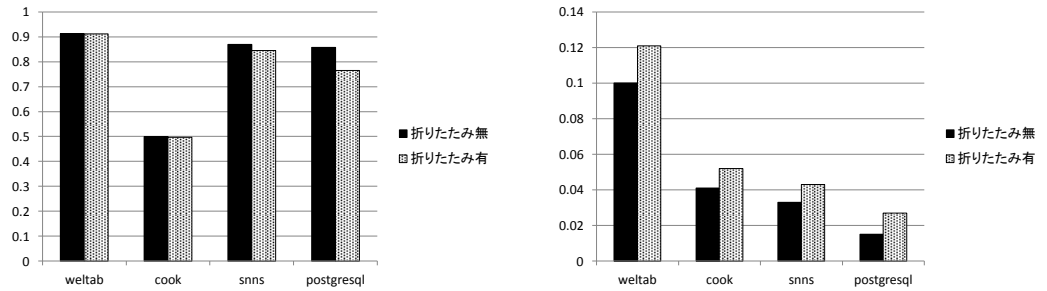
(b) okprecision



(c) goodrecall

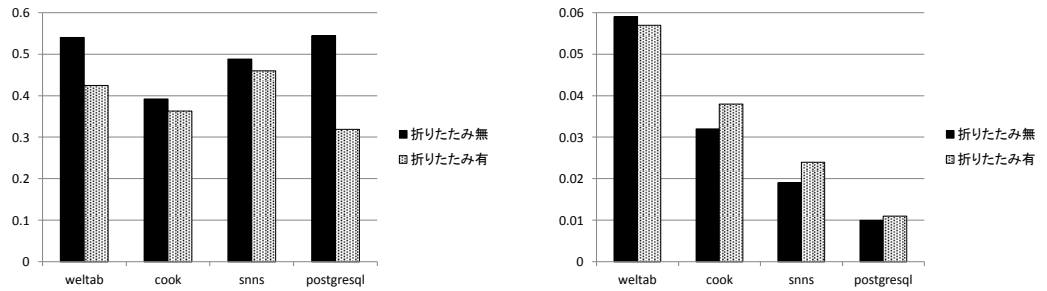
(d) goodprecision

図 16: 折りたたみの有無における Java プログラムの recall と precision



(a) okrecall

(b) okprecision



(c) goodrecall

(d) goodprecision

図 17: 折りたたみの有無における C プログラムの recall と precision

6.3 既存手法との比較実験

実装したツールを用いて，複数のオープンソースソフトウェアに対して実験を行った．6.2で行った実験と同じく，表 1 に示す対象ソフトウェアに対して実験を行った．

6.3.1 目的

この実験の目的は，提案手法と既存手法の検出精度を比較することである．具体的な評価項目は以下の点である．

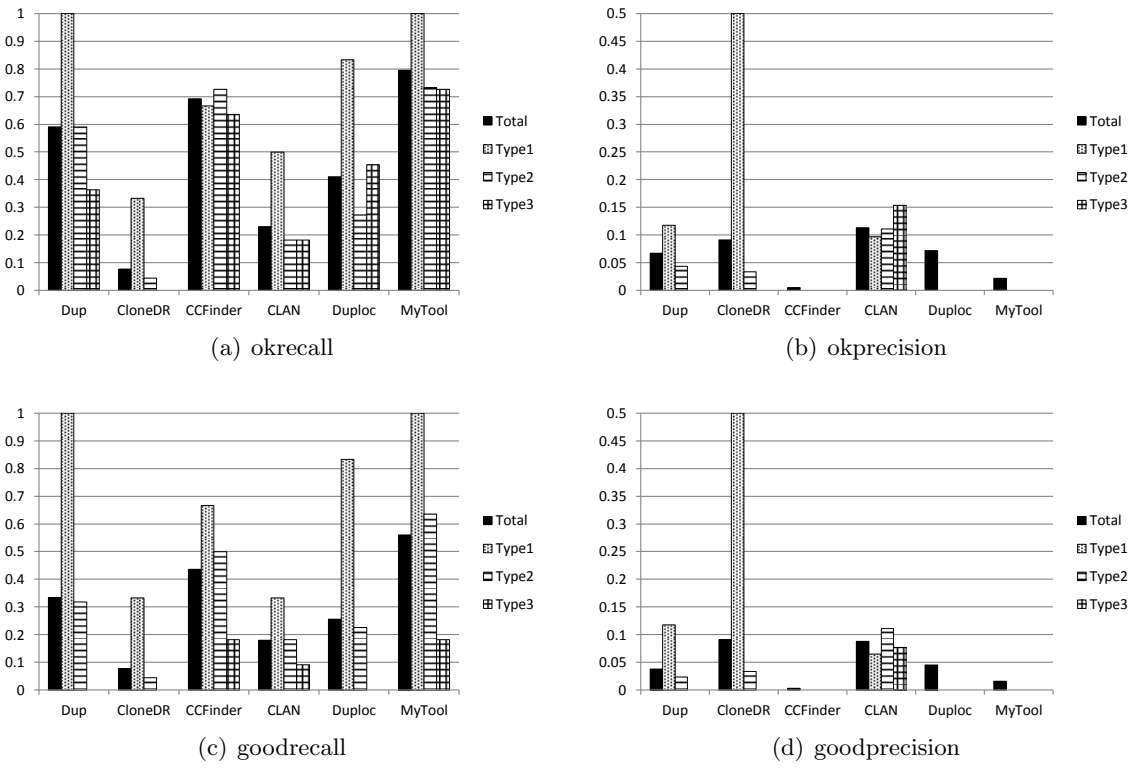
- 提案手法と既存手法を用いて検出されたコードクローンの recall（再現率）
- 提案手法と既存手法を用いて検出されたコードクローンの precision（適合率）

6.3.2 比較した検出手法

Bellon は Baker の手法 [16]，Baxter らの手法 [3]，Kamiya らの手法 [4]，Krinke の手法 [5]，Mayrand らの手法 [6]，Ducasse らの手法 [7] の 6 つを比較した．今回の実験は，これらの 6 つの既存手法と提案手法の検出精度を比較する．

6.3.3 実験結果

図 18 に netbeans を対象にした実験結果を示す．Kamiya のツール”CCFinder”，Ducasse のツール”Duploc”，提案手法を実装したツールは出力結果においてクローンタイプを載せていないので，全体（Type-1，Type-2，Type-3 の合計）の結果のみとなっている．その他のソフトウェアに対する検出結果は付録に示す．提案手法は CCFinder に対して再現率は 1〜1.2 倍，適合率は 1.5〜4 倍程度上昇したという結果を得た．



(a) okrecall (b) okprecision

(c) goodrecall (d) goodprecision

図 18: netbeans の実験結果

7 考察

7.1 折りたたみを用いた検出例

提案手法による繰り返し部分の検出結果の 1 つを図 19 に示す．繰り返し部分において，各メソッド呼び出しを別々にコードクローンとして検出することなく，メソッド呼び出し全体をまとめて 1 つのコードクローンとして検出できている．すなわち，冗長なコードクローンを検出せず，把握すべきコードクローンを検出できていることが確認できた．

7.2 折りたたみによる検出数の減少

jdt との比較において，検出数が 48,888 から 20,463 と半数以下に減少した．jdt のソースコードを調査したところ，jdt のプログラム中に if 文，if-else 節，switch 文内の case エントリ，try-catch 節などの繰り返しが非常に多く見られた．このような繰り返し部分に対し，提案する折りたたみ機能が効果的に働いたため，検出数の大幅な減少につながったといえる．jdt のプログラムにおける case エントリと if 文の繰り返し例を図 20，図 21 に示す．また，

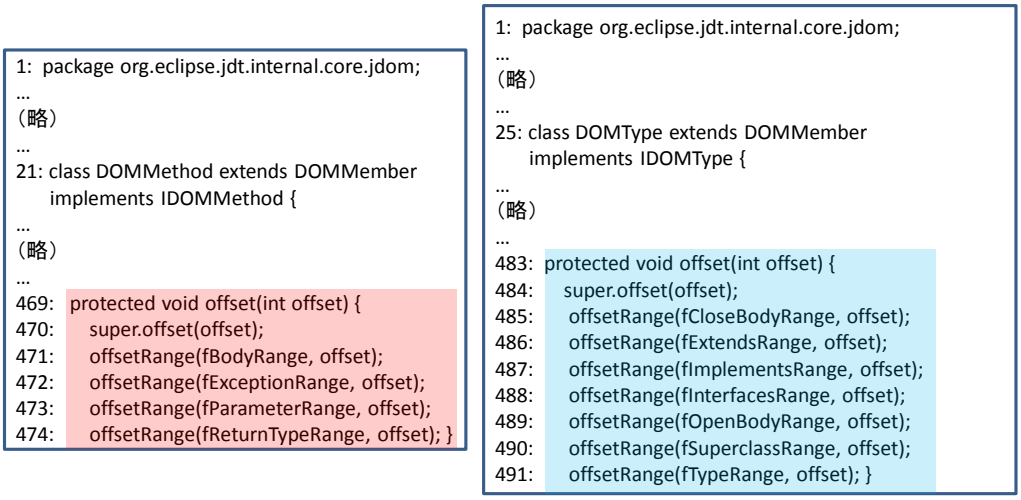


図 19: 繰り返し部分における提案手法の検出例

jdt の検出結果のうち正解クローンであったものの数は 918 個から 836 個に減少した．これは，折りたたみによって繰り返し部分内の新正解クローン（図 22）を検出しなくなったためである．繰り返し部分内のコードクローンは 1 節で述べたように検出する利点がないため，新正解クローンの検出数の減少は問題ないといえる．

7.3 既存手法で検出できず提案手法で検出できた新正解クローン

既存手法では ok 値でしか検出できないが，提案手法では good 値で検出できたコードクローンが多く見られた．すなわち，これらのコードクローンについては，既存手法より提案手法の方がより正解クローンに類似した形でコードクローンを検出しているといえる．なぜなら，正解クローンは Bellon がコードクローン検出ツールの出力結果を実際に見て決めたものであり，意味的にまとまってるものが多いためである．既存手法は純粋な行単位，字句単位の比較をしていて意味的なまとまりを認識していないため，ok 値では検出できているが good 値で検出できてないという事例が多い．図 23 はメソッドの境界における検出例で図 23(a) は既存手法（CCFinder），図 23(b) は提案手法である．メソッドの抽出や引き上げを考慮すると，意味的なまとまりを認識してコードクローンを検出する方が適しているので，提案手法が優れているといえる．

```
1:  package org.eclipse.jdt.internal.compiler.parser;
…
（略）
…
15:  public class Parser implements BindingIds, ParserBasicInformation,
       TerminalSymbols, CompilerModifiers, OperatorIds, TypeIds {
…
（略）
…
3100:  case 400 : // System.out.println("Expressionopt ::=");
3101:         consumeEmptyExpression();
3102:         break ;
3103:  case 404 : // System.out.println("ImportDeclarationsopt ::=");
3104:         consumeEmptyImportDeclarationsopt();
3105:         break ;
3106:  case 405 : // System.out.println("ImportDeclarationsopt ::= ImportDeclarations");
3107:          consumeImportDeclarationsopt();
3108:          break ;
3109:  case 406 : // System.out.println("TypeDeclarationsopt ::=");
3110:          consumeEmptyTypeDeclarationsopt();
3111:          break ;
3112:  case 407 : // System.out.println("TypeDeclarationsopt ::= TypeDeclarations");
3113:          consumeTypeDeclarationsopt();
3114:          break ;
3115:  case 408 : // System.out.println("ClassBodyDeclarationsopt ::=");
3116:          consumeEmptyClassBodyDeclarationsopt();
3117:          break ;
```

Parser.java

図 20: jdt のプログラム中の case エントリの繰り返し

```
11: package org.eclipse.jdt.core.dom;
…
（略）
…
29: class ASTConverter {
…
（略）
…
449:   if (expression instanceof ArrayAllocationExpression) {
450:         return convert((ArrayAllocationExpression) expression); }
451:   if (expression instanceof QualifiedAllocationExpression) {
452:         return convert((QualifiedAllocationExpression) expression); }
453:   if (expression instanceof AllocationExpression) {
454:         return convert((AllocationExpression) expression); }
455:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.ArrayInitializer) {
456:         return convert((org.eclipse.jdt.internal.compiler.ast.ArrayInitializer) expression); }
457:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.PrefixExpression) {
458:         return convert((org.eclipse.jdt.internal.compiler.ast.PrefixExpression) expression); }
459:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.PostfixExpression) {
460:         return convert((org.eclipse.jdt.internal.compiler.ast.PostfixExpression) expression); }
461:   if (expression instanceof CompoundAssignment) {
462:         return convert((CompoundAssignment) expression); }
463:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.Assignment) {
464:         return convert((org.eclipse.jdt.internal.compiler.ast.Assignment) expression); }
465:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.CastExpression) {
466:         return convert((org.eclipse.jdt.internal.compiler.ast.CastExpression) expression); }
467:   if (expression instanceof ClassLiteralAccess) {
468:         return convert((ClassLiteralAccess) expression); }
469:   if (expression instanceof FalseLiteral) {
470:         return convert((FalseLiteral) expression); }
471:    if (expression instanceof TrueLiteral) {
472:         return convert((TrueLiteral) expression); }
```

ASTConverter.java

図 21: jdt のプログラム中の if 文の繰り返し

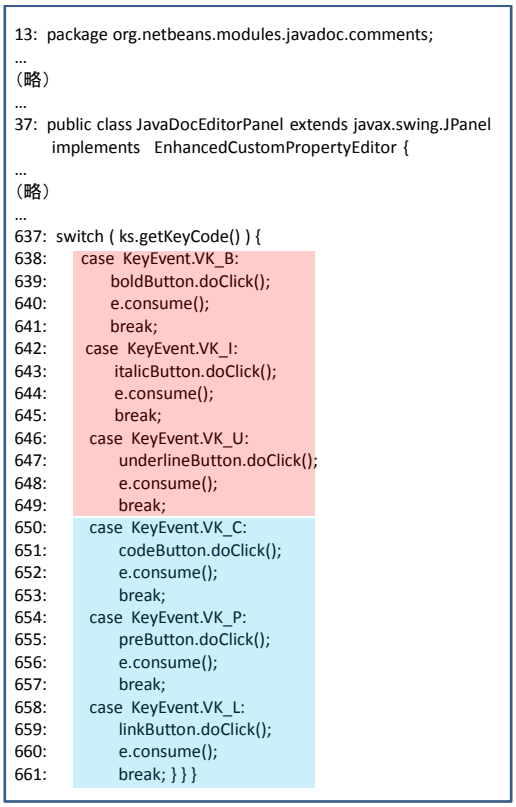
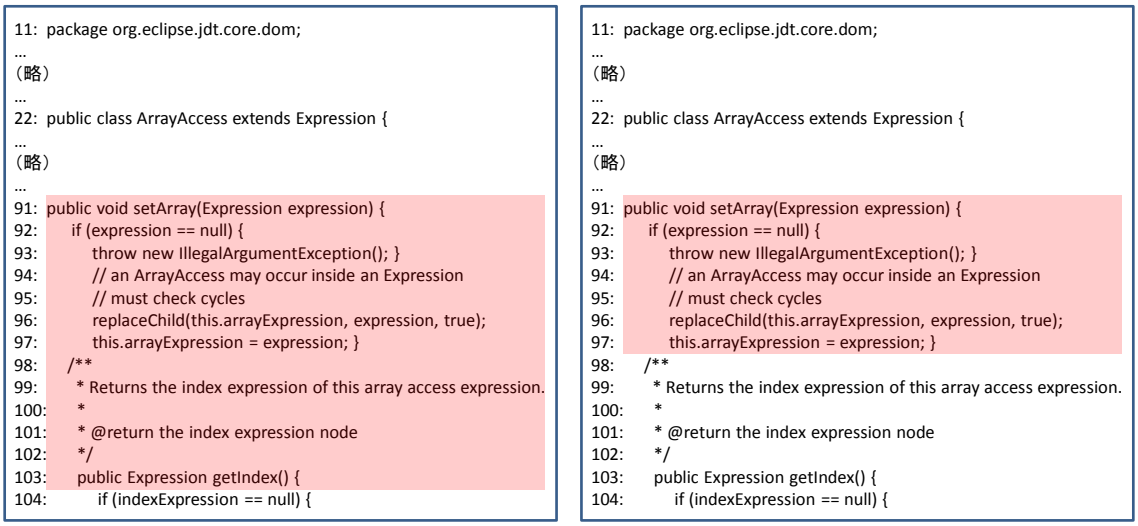


図 22: 繰り返し部分内のコードクローンの例

7.4 既存手法で検出できて提案手法で検出できなかった新正解クローン

既存手法で検出できて提案手法で検出できなかったコードクローンは数が多く，すべてを調べることはできなかった．そこで無作為に 100 個のクローンペアを抽出し，このようなコードクローンがいくつあるのかを調べた．その結果，提案手法で検出できないコードクローンとして，メソッドをまたいでいるコードクローン，繰り返し部分内のコードクローン，import 文や package 文を含むコードクローンの 3 つに分類できることが分かった．提案手法ではメソッドの境界を識別しているため，メソッドをまたいでいるコードクローンは検出されない．import 文や package 文は，正規化の段階で排除しているため検出されない．繰り返し部分内コードクローンは，提案手法で繰り返し部分を折りたたんでいるため検出されない．図 24 に，この 3 つの分類の割合を示す．繰り返し部分内コードクローンの例を図 22 に示す．



(a) 既存手法　　(b) 提案手法

図 23: メソッドの境界における検出例

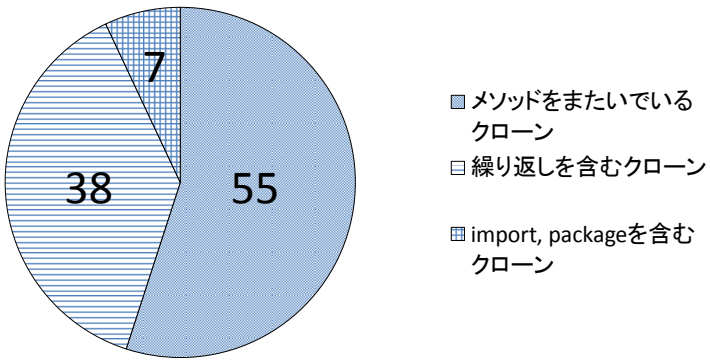


図 24: 既存手法で検出できて提案手法で検出できなかったコードクローンの割合

8　実験結果の妥当性について

本研究の結果の妥当性に関して，以下で挙げる点に留意する必要がある．

8.1　正解クローンの作成法

本論文の実験では，Bellon らが作成した正解クローンを用いて新正解クローンを作成した．また，新正解クローンを用いて，提案手法を実装したツールの性能について調査を行った．正解クローンは対象ソフトウェアに含まれるすべてのコードクローンではなく，それを用いて作成した新正解クローンも同様である．そのため，すべてのコードクローンを対象にして同様の実験を行った場合は，異なる実験結果が得られる可能性がある．しかし，ソースコード中に存在するすべてのクローンを対象にすることは現実的ではないので，正確な recall，precision の値を求めることは難しい．recall，precision の値は相対的な評価にのみ用いることができる．

8.2　検出手法におけるコードクローンの定義の違い

2.2 節で述べたようにコードクローン検出ツールは字句や行単位のものだけでなく，抽象構文木やプログラム依存グラフを用いたものなど様々である．各コードクローン検出ツールによってコードクローンの定義が異なるので，他のツールを用いて比較をした場合，異なる実験結果が得られる可能性がある．

8.3　対象ソフトウェア

本研究では，計測対象をオープンソースソフトウェアのみに限定している．しかし，一般的に商用ソフトウェアはオープンソースソフトウェアと比べてコードクローン含有率が高いといわれているため，商用ソフトウェアに対して計測を行った場合，本研究とは異なる結果が導かれる可能性がある．

9 関連研究

肥後らは，クローンペアの集合に含まれるコード片がどの程度繰り返し要素を含まないか表すメトリクス RNR を提案している [24]．このメトリクスを使うことで関数や代入文などの羅列，switch 文のように同じ構造になりやすい文などを取り除くことができる．RNR は，CCFinder[4] の後続ツールである CCFinderX[17] でも採用されており，閾値を定めることで検出結果のフィルタリングができる．RNR は，繰り返し要素から構成されるコードクローンをある程度自動的にフィルタリングすることは可能だが，把握すべきクローンを検出することができない点においては提案手法に対して劣っている．

今回の実験では，既存手法と提案手法の比較に Bellon らの実験を用いた．他にもコードクローン検出手法・ツールの比較実験は行われているが，Bellon らの実験は比較ツールの数，対象プログラムの規模共に最大であることから，本研究における比較実験として Bellon らの実験方法を採用した．Burd らの実験 [21]，Rysselberghe らの実験 [22] を以下に示す．

Burd らは Kamiya らの手法 [4]，Baxter らの手法 [3]，Mayland らの手法 [6]，Prechalt らの手法 [25]，Aiken らの手法 [26] の 5 つの比較をしている．各検出手法で検出されたコードクローンを実際に見て，本当にコードクローンであったものを正解クローンとし，各検出手法の再現率と適合率を求めている．

Rysselberghe らは，行単位，字句単位，メトリクス計測の 3 つの検出技法を比較している．Rysselberghe らはツールを比較するのではなく検出技法をするため，各検出技法を用いたツールを作成し，実験を行った．行単位，字句単位の検出は各プログラミング言語用に解析器を作る必要があるが，それほどコストは高くないと述べている．また，メトリクス計測による検出はソースコードから様々な情報を得なければならないので，解析器を作るコストは高いと述べている．

10　あとがき

本研究では，ソースコード中の繰り返し部分を折りたたんで検出する手法を提案した．提案手法を用いたコードクローン検出ツールを実装し，オープンソースソフトウェアに対して検出を行った．その結果，折りたたみ無しの場合に比べて，折りたたみ有りの場合でコードクローンの検出数が減少したことを確認した．さらに既存の検出手法との比較を通じて，検出されたコードクローンの評価を行った．その結果，繰り返し部分の折りたたみにより，既存手法の問題点を解消したことを示した．

今後の課題は以下の通りである．

- 検出対象の言語を増やす．
- インクリメンタルな検出に対応する．インクリメンタルな検出とは，検出処理で得た情報をデータベースに登録し，それ以降の検出処理においてデータベースを活用することである．これにより，検出時間を短縮できる．
- 現在は，コードクローンの情報をテキストファイルに出力している．より実用性を高めるためには，表示方法の改良や Eclipse プラグインとして実装するなど，ユーザインターフェースの充実も必要であると考えられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に多大なるご助言およびご指導を頂きました 井垣 宏 特任准教授に深く感謝致します.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究を行うにあたり, 多大なるご助言, ご助力を頂きました, 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博士前期課程 2 年の 堀田 圭佑 氏に深く感謝申し上げます.

本研究を行うにあたり, 適切なご助言, ご助力を頂きました, 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博士前期課程 1 年の 佐々木 唯 氏に深く感謝申し上げます.

その他，楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌 D, Vol. 91-D, No. 6, pp. 1465–1481, June 2008.

[2] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, Vol. 24, pp. 49–57, 1992.

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of the 14th IEEE International Conference on Software Maintenace*, pp. 368–377, March 1998.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.

[5] J. Krinke. Identifying similar code with program dependence graphs. In *In Proc. the 8th Working conference on Reverse Engineering*, pp. 301–309, October 2001.

[6] J. Mayland, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the 12th IEEE International Conference on Software Maintenace*, pp. 244–253, 1996.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the 15th IEEE International Conference on Software Maintenance*, pp. 109–118, August 1999.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *Proc. of the 29th International Conference on Software Engineering*, May 2007.

[9] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Indexbased code clone detection: incremental, distributed, scalable. In *Proc. of the 32th IEEE International Conference on Software Engineering*, pp. 1–9, 2010.

[10] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

[11] S. Bellon. Detection of software clones. *Technical Report, Institute for Software Technology, University of Stuttugart*, 2003. available at http://www.bauhaus-stuttgart.de/clones/.

[12] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Software Engineering*, Vol. 31, No. 10, pp. 804–818, October 2007.

[13] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, April 2005.

[14] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2009, 9 2009.

[15] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of the 2nd Working Conference on Reverse Engineering*, pp. 86–95, July 1995.

[16] B.S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, Vol. 26, pp. 1343–1362, 1997.

[17] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.

[18] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. of the 8th International Symposium on Software Metrics*, pp. 67–76, 2002.

[19] M.Datar, N.Immorlica, P.Indyk, and V.S.Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *In Proc. of the 20th Symposium on Computational Geometry*, pp. 253–262, 2004.

[20] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–27, 1990.

[21] E.Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36–43, 2002.

[22] F. V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proc. of the 19th IEEE International Conference on Automated Software Engineerin*, pp. 336–339, 2004.

[23] Detection of Software Clones. `http://bauhaus-stuttgart.de/clones/`.

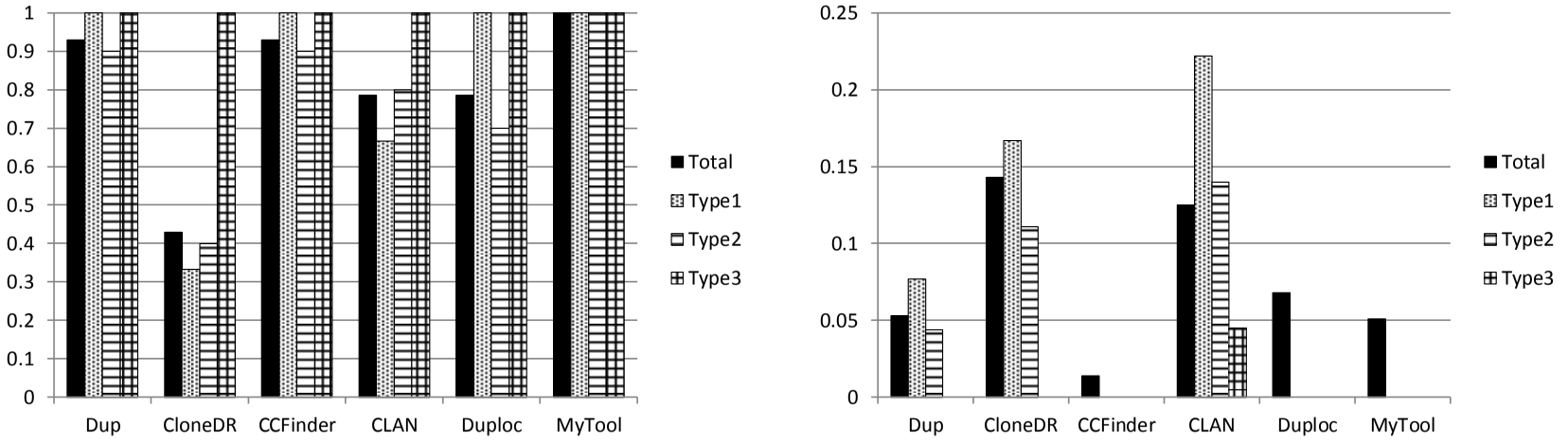
[24] 肥後芳樹, 吉田則裕, 楠本真二, 井上克郎. 産学連携に基づいたコードクローン可視化手法の改良と実装. 情報処理学会論文誌, Vol. 48, pp. 811–822, 2007.

[25] L.Prechelt, G.Malpohl, and M.Phlippsen. Jplag: Finding plagiarisms among a set of programs. *Technical report, University of Karlsruhe, Department of Informatics*, 2000.

[26] Moss: A system for detecting software plagiarism. `http://www.cs.berkeley.edu/~aiken/moss.html`.

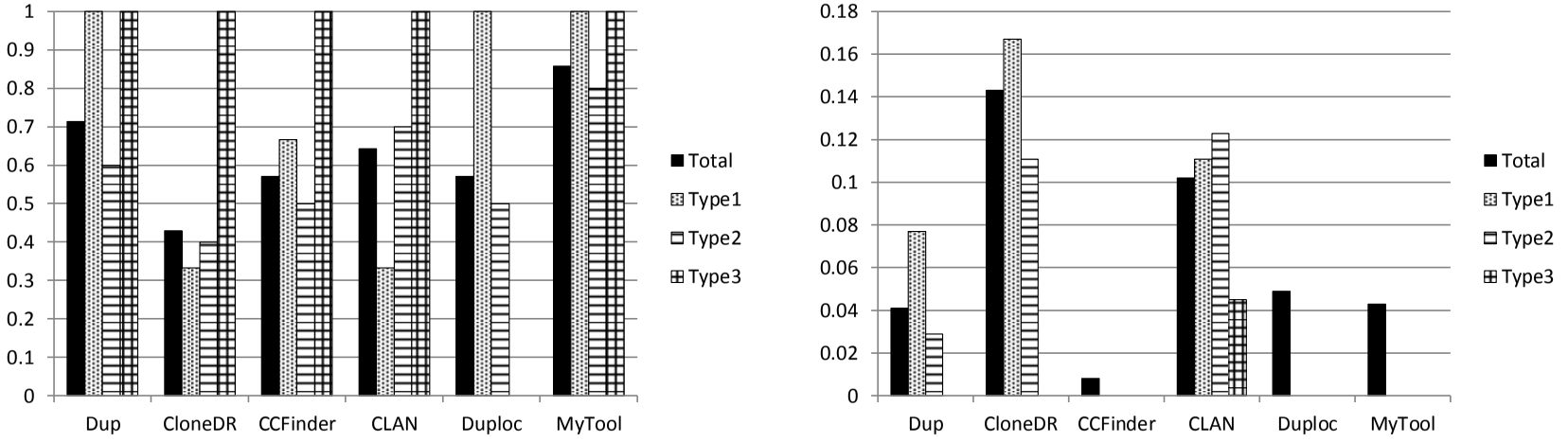
付録： 実験結果

図 10 が ant，図 26 が jdt，図 27 が swing，図 28 が weltab，図 29 が cook の，図 30 が snns の，図 31 が postgresql の実験結果である．Ducasse のツール”Duploc”は swing と postgresql の解析ができなかったため，結果が出ていない．また，Krinke のツール”Duplix”は Java プログラムの解析ができないので，Java プログラムの結果が出ていない．Kamiya のツール”CCFinder”，Ducasse のツール”Duploc”，提案手法を実装したツールは出力結果においてクローンタイプを載せていないので，全体（Type-1，Type-2，Type-3 の合計）の結果のみとなっている．



(a) okrecall

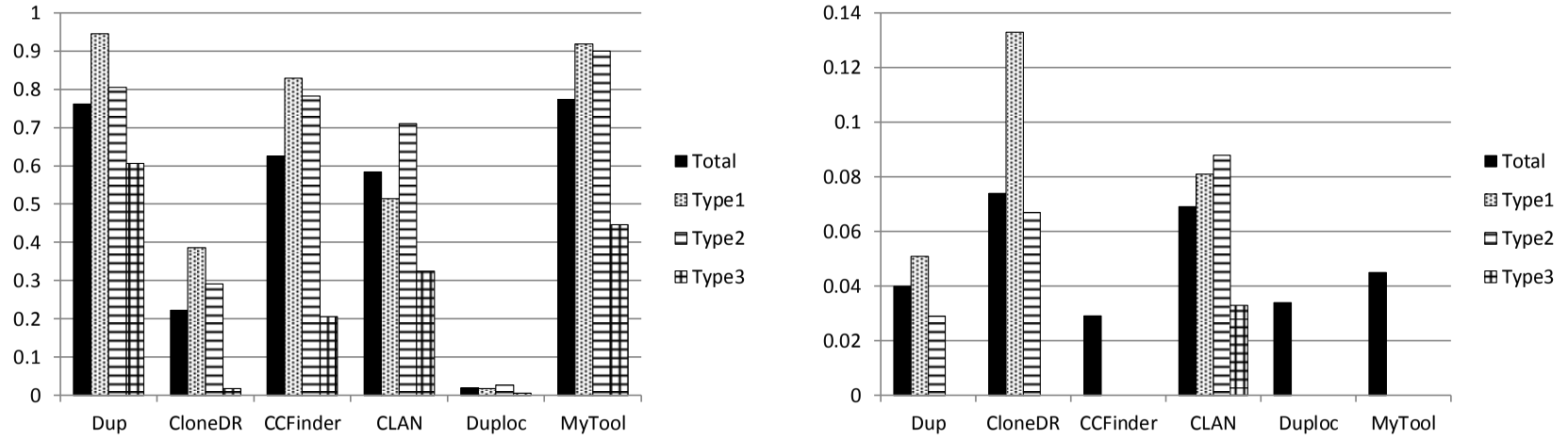
(b) okprecision



(c) goodrecall

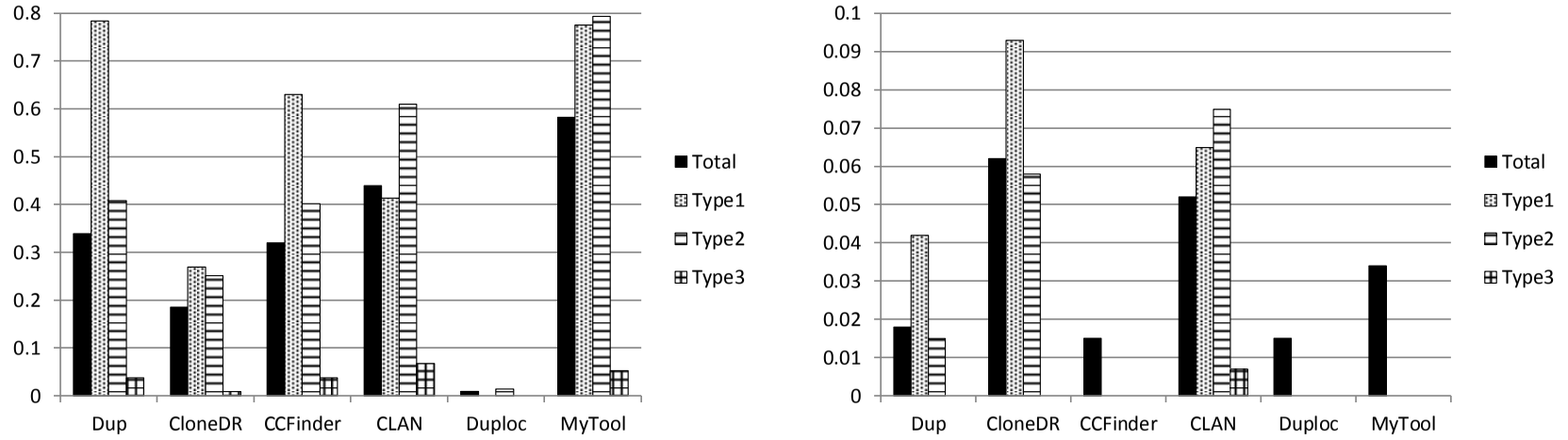
(d) goodprecision

図 25: ant の実験結果



(a) okrecall

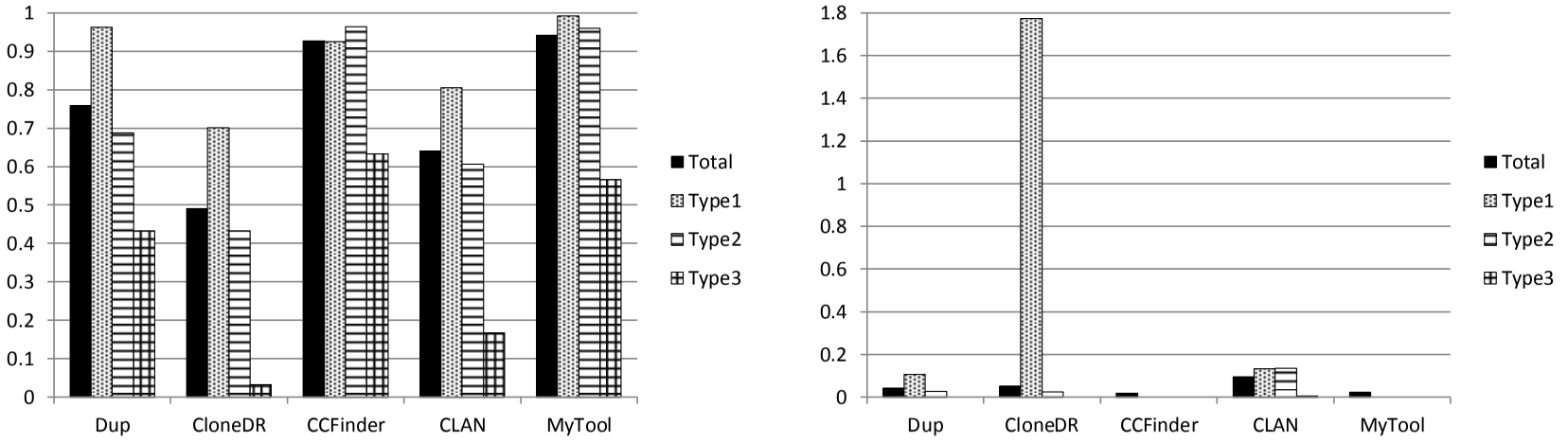
(b) okprecision



(c) goodrecall

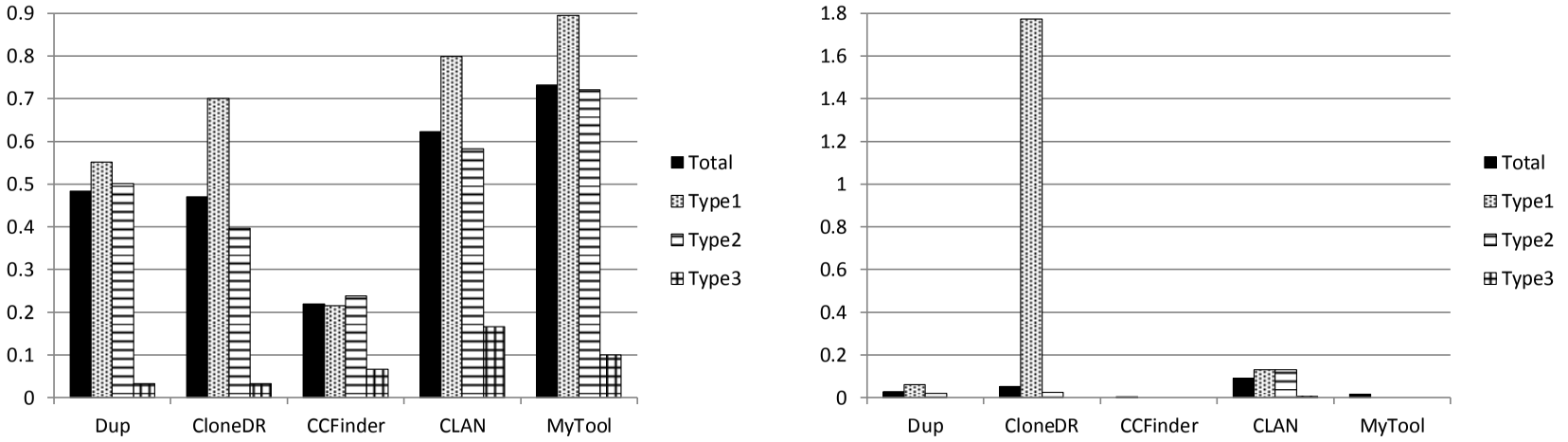
(d) goodprecision

図 26: jdt の実験結果



(a) okrecall

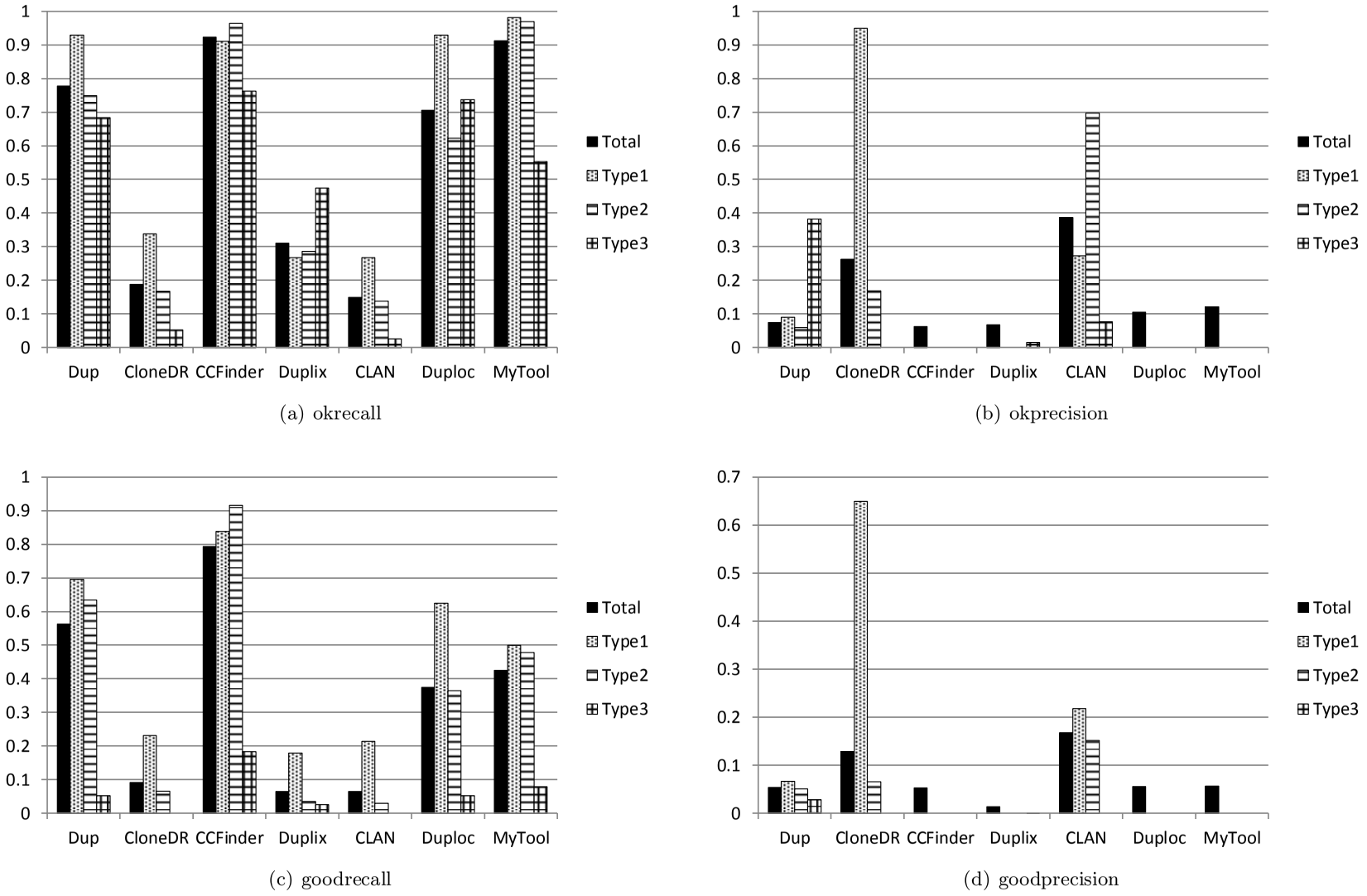
(b) okprecision



(c) goodrecall

(d) goodprecision

図 27: swing の実験結果



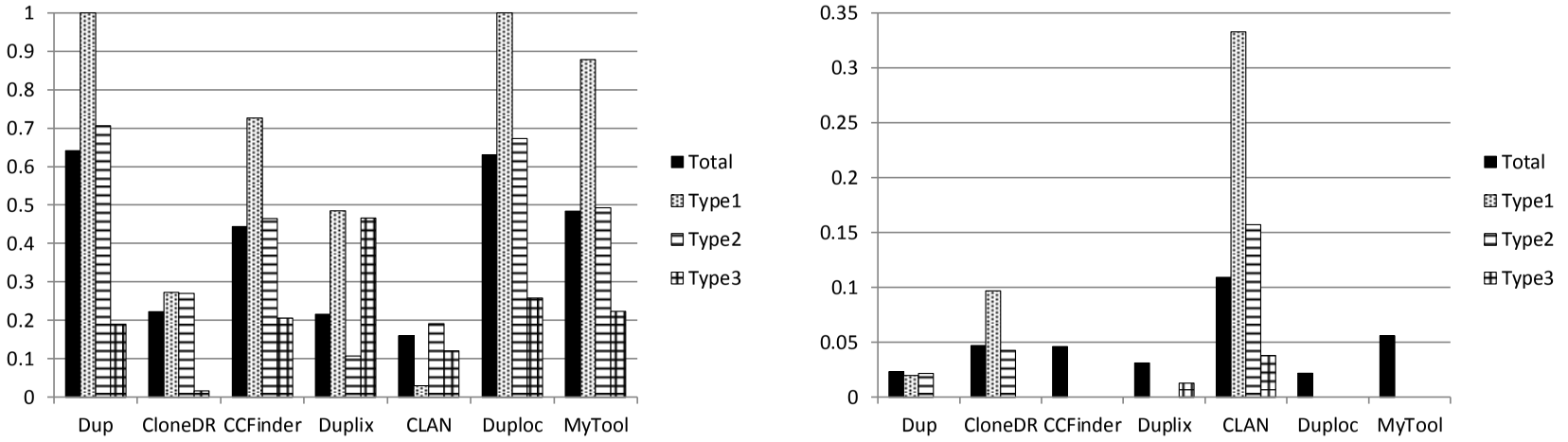
(a) okrecall

(b) okprecision

(c) goodrecall

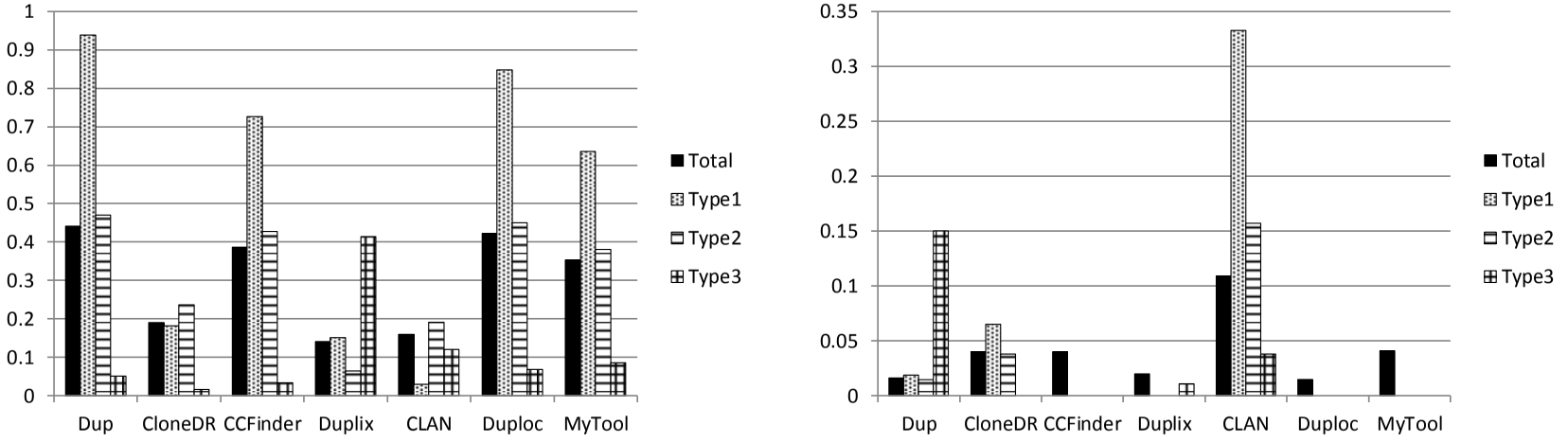
(d) goodprecision

図 28: weltab の実験結果



(a) okrecall

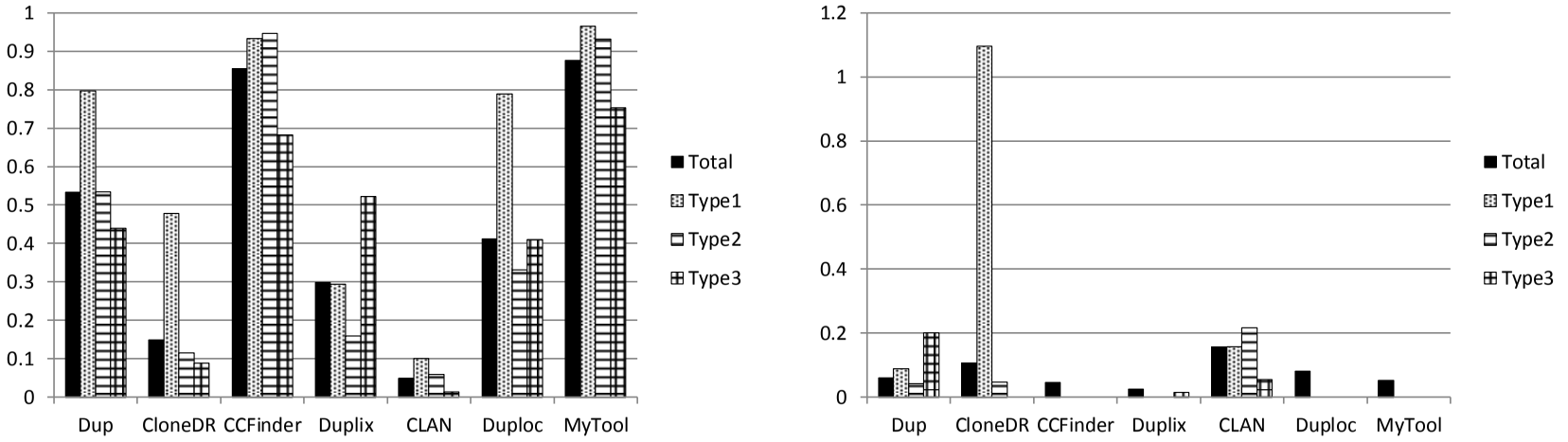
(b) okprecision



(c) goodrecall

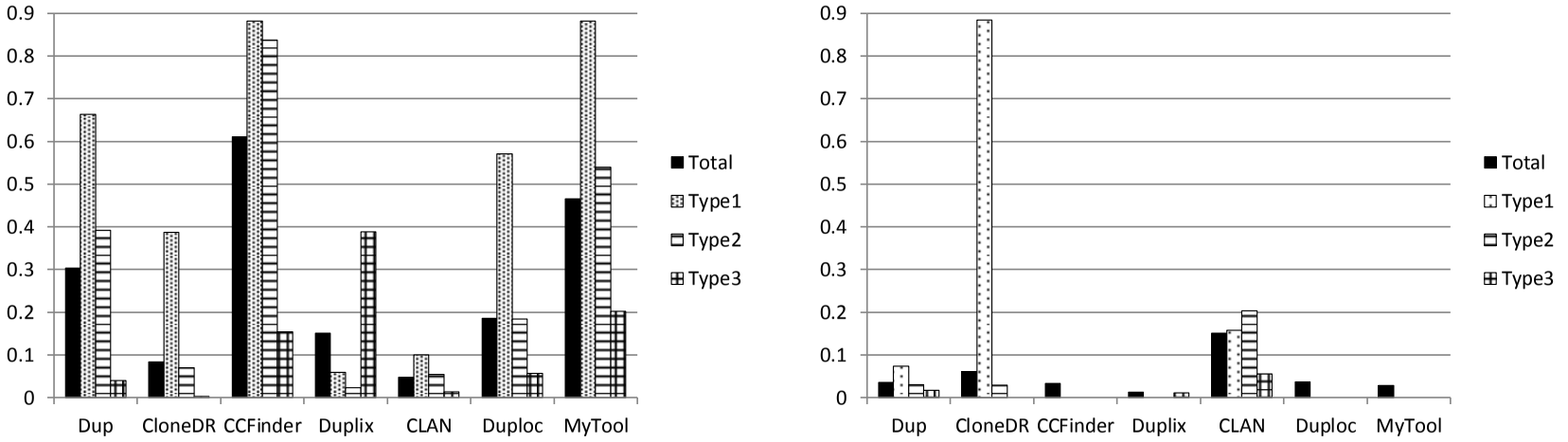
(d) goodprecision

図 29: cook の実験結果



(a) okrecall

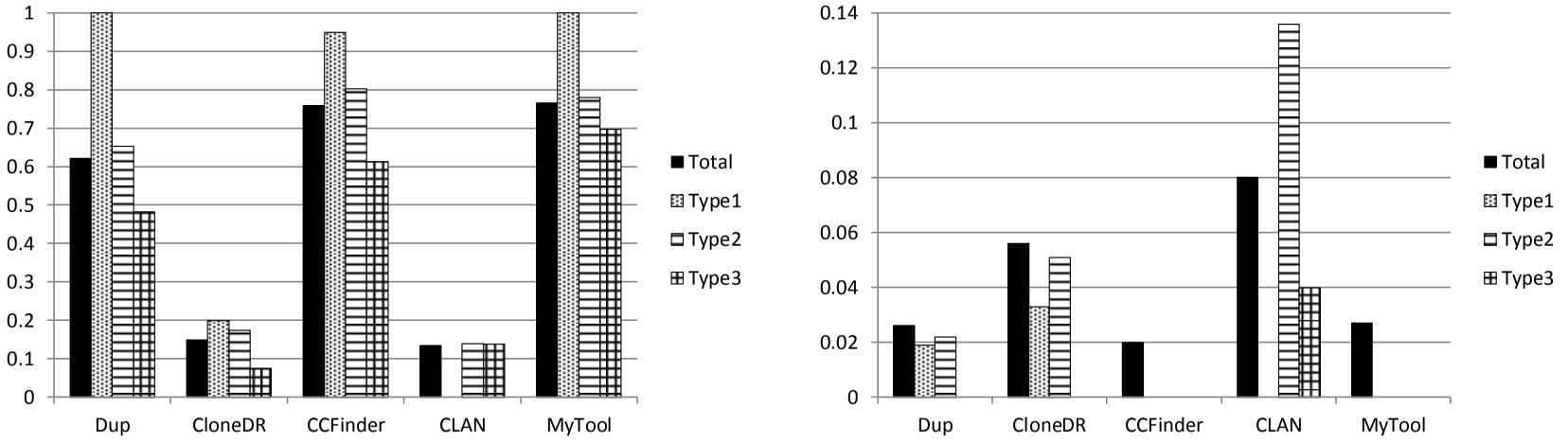
(b) okprecision



(c) goodrecall

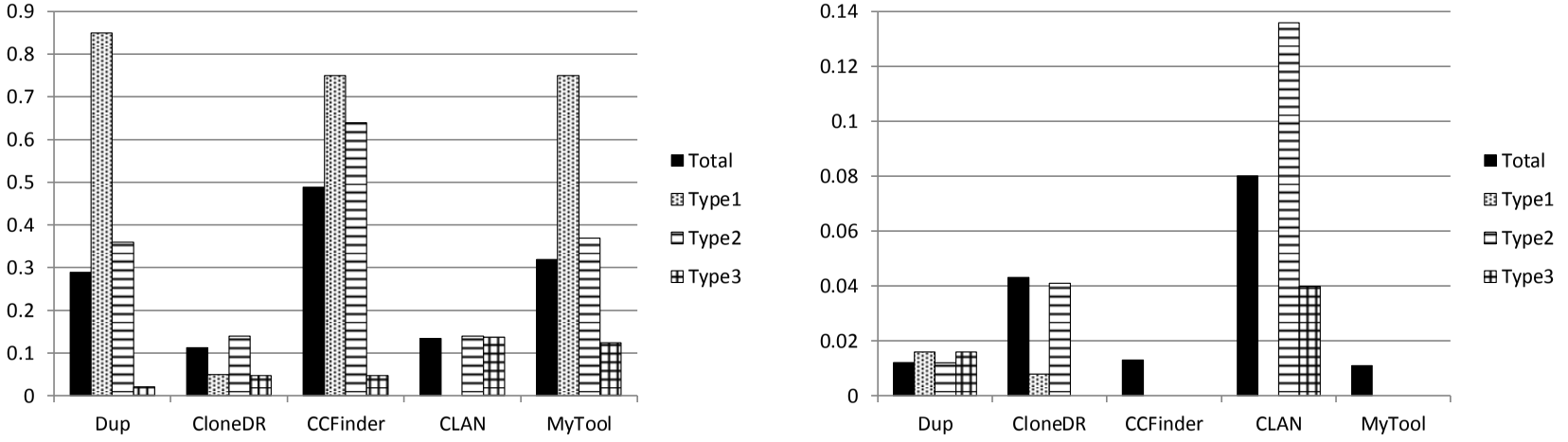
(d) goodprecision

図 30: snns の実験結果



(a) okrecall

(b) okprecision



(c) goodrecall

(d) goodprecision

図 31: postgresql の実験結果