関数適用によるギャップを考慮した コードクローンの検出および除去に関する研究

松下 翼, 篠埜 功

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

ma14099@shibaura-it.ac.jp

sasano@sic.shibaura-it.ac.jp

概 要　コードクローンとはソースコード内の同一、または類似した部分を指す。コードクローンの存在により、一般にソフトウェアの保守性や可読性が低下する。コードクローン中の不一致部分をギャップと呼ぶ。ギャップによってコードクローンが分断され、単純な検出手法ではコードクローンが発見されない場合や、分断の前後で別々のコードクローンとして検出される場合がある。ギャップを考慮したコードクローン検出手法として字句や構文を使った手法がこれまでにいくつか提案、実装された。本研究では関数適用により生じたギャップに着目した検出手法を提案する。提案手法に基づいて Standard ML で書かれたプログラムのコードクローンの検出および除去を行うツールを実装し、実験を行った。

1　はじめに

コードクローンとはソースコード中の同一、あるいは類似した部分を指す。コードクローンはプログラミング時に主にコピーアンドペーストにより数多く生み出されている。オープンソースソフトウェアに含まれるコードクローンの調査 [14] では、ソースコード全体の字句数に対するコードクローンに含まれる字句数の割合 (CRate) が平均で 11.3%あることが示されている。一般に、ソースコードがコードクローンを多く含む場合、一貫した変更を行うのは手間がかかるため、ソフトウェアの保守性が低下すると言われている。そのため、自動生成されたコードを除き、一般にコードクローンは少ないのが望ましい。

Bellon ら [4] はコードクローンを 3 種類に分類した。コピー元のソースコードに空白、改行、コメントの追加、削除によって得られるコードクローンが type1 である。type1 の変更に加え、定数や識別子の変更によって得られるコードクローンが type2 である。type2 の変更に加え、文や式の変更、追加、削除によって得られるコードクローンが type3 である。type3 のコードクローン対間の差異をギャップと呼ぶが、ギャップを含んだコードクローンの検出に関する研究はそれほど多くない。大規模なソースコードから type3 を含むコードクローンの検出を行う手法として、AGM アルゴリズムを用いる手法 [17]、Smith-Waterman アルゴリズムを用いる手法 [16] が提案された。しかし、これらの手法はあらかじめ定めた大きさ以上のギャップを含んだコードクローンが検出できないなどといった問題がある。

コードクローンの検出、除去に関する研究は盛んに行われているが、関数型言語を対象とした研究は少ない。Erlang を対象とした検出手法 [10] では、字句の比較によりコードクローンを検出するため、対象とするコードクローンが type2 までである。Haskell を対象とした検出手法 [5] では、抽象構文木の各部分木を anti-unification により比較することでコードクローンを検出する。これらは特定の関数型言語を対象としてはいるが、関数型言語に特化した検出手法とまでは言えない。また、筆者が調査した限り、Standard ML を対象とした研究は存在しない。

関数型言語のプログラムの一つの特徴として、関数適用が多用されることがあり、関数適用式によるギャップが多いと予想される。そこで本論文では関数適用によるギャップに着目し、ギャップが関数適用式になっているコードクローンを検出、除去する手法を提案する。関数適用式であるかどうかを判別するために構文解析が必要であるが、Haskell を対象とした検出手法 [5] のようにすべてのソースコードを構文解析するのではなく、字句単位での検出手法をもとに、必要な場合にのみ構文解析を行う。提案手法に基づき、Standard ML で書かれたソースコードを対象としてコードクローンの検出及び除去を行うツールを実装した。実装したツールを用いてコードクローン検出実験を行い、実験結果について考察を行った。

本論文の構成は以下の通りである。2 節で本論文で用いるコードクローンの定義について述べる。3 節でコードクローンの検出について述べる。4 節でコードクローンの除去について述べる。5 節で Standard ML を対象としたコードクローン検出および除去の実装について述べる。6 節で実験とその結果について述べる。7 節で関連研究について述べる。8 節で将来の課題と本論文のまとめについて述べる。

2　コードクローンについて

ここではコードクローンが発生する要因について概観したのち、本論文で用いるコードクローンの定義について述べる。

2.1　コードクローンの発生要因

コードクローンが発生する要因としては様々なものが挙げられる [15] が、最も多いのはコピーアンドペーストである。プログラミング時に、必要な機能に似た機能が記述されている部分をコピーアンドペーストして必要に応じた修正を加えることが一般によく行われる。また、コードを自動生成する場合、コードクローンが大量に発生する場合が多い。Lex や Yacc などで生成されるソースコードにはコードクローンが大量に含まれる。また、定形処理部分では小さなコードクローンが生み出されることが多い。例えばデータベースの操作やデータ構造アクセス処理などは同じようなコードになってしまう場合が多いので、小さなコードクローンが生成されやすい。

2.2　コードクローンの定義

コードクローンの定義はコードクローンの研究やツールの実装ごとに異なり、本論文では Bellon らの定義 [4] を用いる。Bellon らの定義 [4] ではコードクローンは type1、type2、type3 の 3 種類に分類され、図 1 のプログラム断片を用いてそれぞれについて説明する。図 1 のプログラムの何箇所かに改行、インデントを加えたものが図 2 のプログラムである。図 1 と図 2 のような、空白、改行、コメントの追加、削除によって得られるコードクローンが type1 である。図 2 のプログラムに対してさらに変数名および定数の変更を行ったものが図 3 のプログラムである。図 1 と図 3 のプログラムのように、type1 の変更に加え、定数や識別子の変更によって得られる構文が同じコードクローンが type2 である。図 3 のプログラムに対し、関数名の変更、関数の引数の追加、条件式の変更、then パートの式の変更を行ったものが図 4 のプログラムである。図 1 と図 4 のように、type2 の変更に加え、文や式の変更、追加、削除によって得られるコードクローンが type3 である。type3 のコードクローン対の文や式の変更、追加、削除により生じる差異をギャップと呼ぶ。type2 のコードクローン対の差異もギャップと呼ぶ場合もあるが、本研究では type3 の差異のみを指すこととする。本研究では type3 までのコードクローンを対象とする。

```
fun eSum n = n + if (n=0) then 0 else eSum(n-2)
```

図 **1.** Standard ML のソースコード例

```
fun eSum n = n +
  if (n=0) then 0
  else eSum(n-2)
```

図 **2.** 図 1 のソースコードの type1 のコードクローン

```
fun oSum n = n +
  if (n=1) then 1
  else oSum(n-2)
```

図 **3.** 図 1 のソースコードの type2 のコードクローン

3 コードクローン検出手法

この節では本論文で提案するコードクローン検出手法を示す。既存の研究ではソースコードをブロックや行単位で分けたり、字句列や抽象構文木 (以降では AST と表記) のような形に変換することにより、コードクローンを検出する手法が提案されてきた。しかし、2 節で述べた type3 であるギャップを含んだコードクローンを検出できる手法は多くない。大規模なソースコードからギャップを含むコードクローンを検出する手法としては AGM アルゴリズムを用いた手法 [17] や Smith-Waterman アルゴリズムを用いた手法 [16] がある (7 節参照)。以下では字句単位でのコードクローン検出手法をもとに、構文木の情報を用いてギャップを考慮した検出を行う手法を提案する。まず 3.1 節で基本的な字句単位での検出手法とその問題点について述べ、3.2 節、3.3 節で問題点の解決案を提示する。

3.1 字句単位でのコードクローン検出

字句単位でのコードクローン検出は字句解析と検出の 2 つのステップからなる。まず字句解析により、文字の列であるソースコードを字句列に変換する。通常の言語においては空白やコメントは無視されるので、字句解析時に type1 の空白やコメントの有無による違いは解消される。また同時に識別子や定数もそれぞれに対応する字句になるので type2 のコードクローンも検出することができる。字句解析により type2 のコードクローン間の違いが吸収される様子を図 5 に示す。

コードクローンの検出には接尾辞配列 [11] を用いる。接尾辞とは文字列のある部分から末尾までの部分文字列である。接尾辞配列とはすべての接尾辞を辞書順にソートしたものであり、文字列を高速で処理するために用いられるデータ構造である。例として文字列 `mississippi` の接尾辞配列を表 1 に示す。

最長共通接頭辞 (longest common prefix、以降では LCP と表記) は接尾辞配列 S の n 番目の接尾辞 S_n と $n-1$ 番目の接尾辞 S_{n-1} の前方から一致する文字数である。コードクローン検出においては LCP がコードクローンの大きさであり、この値が検出するコードクローンの下限となる閾値を

```
fun oPro n m = n *
  if (n=m) then (n/10)
  else oPro(n-2) m
```

図 **4.** 図 1 のソースコードの type3 のコードクローン例

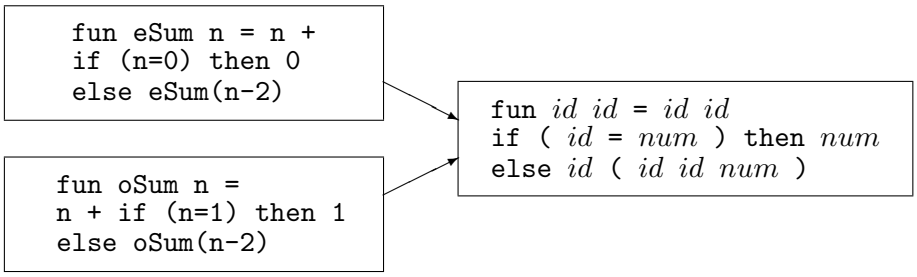

図 **5.** 字句単位での type1 と type2 のコードクローンの検出

表 **1.** 文字列 mississippi の接尾辞配列

n	start$[n]$	LCP$[n]$	S_n
0	10	0	i
1	7	1	ippi
2	4	1	issippi
3	1	4	ississippi
4	0	0	mississippi
5	9	0	pi
6	8	1	ppi
7	6	0	sippi
8	3	2	sissippi
9	5	1	ssippi
10	2	3	ssissippi

超えたものをコードクローンとして検出する。また 4 節で詳しく述べるが、検出したコードクローン対は関数宣言および関数適用式で置き換えることで除去を行う。その際関数の引数の数とコードクローンの大きさの比の上限を 2 つ目の閾値として設定する。関数化した際に引数の数が多すぎると関数にする意味が薄くなるからである。LCP の値が閾値を超えた場合に 2 つのソースコード断片の中で引数となるものの計算を行う。引数になるものの条件は変数であること、type2 のコードクローンにおける識別子や定数が異なる部分であること、3.2 節で述べるギャップ部分であることの 3 つである。

字句列の接尾辞配列を用いた検出手法は字句情報のみを利用することで高速な検出を可能としており、大規模なソースコードにも適用可能なコードクローン検出ツールである CCFiner[8] もこの手法をもとに実装されている。しかし字句列の接尾辞配列による検出は以下の問題点がある。まずギャップを含む type3 のコードクローンが検出できないことである。また、ソースコードの構文木のどの部分木と比較してもその部分木の途中から開始する、あるいはその部分木の途中までで終了するソースコード断片がコードクローンとして検出されてしまう問題もある。以降ではこのようなコードクローンを構文的に不完全なコードクローンと呼ぶ。今回提案する手法はこれらの問題を一部解決しており、それぞれについて以下の節で述べる。

3.2 ギャップを含んだコードクローン検出

前節で述べた字句単位の検出手法では type3 のギャップを含んだコードクローンの検出を行うことができない。字句の比較において、コピー元から変更を加えた部分、すなわちギャップが存在した

場合、そこで字句の不一致が発生し、以降の一致部分は無視されるか、あるいは別のコードクローンとして検出されてしまう。図 6 は図 3 と図 4 のソースコードにおけるギャップによる字句の不一致部分を赤字で示す。

```
fun oSum n = n +
if (n=1) then 1
else oSum(n-2)
```

```
fun oPro n = n *
if (n=m) then (n/10)
else oPro(n-2) m
```

図 **6.** 図 3 と図 4 のコードクローン対に存在するギャップ

本研究ではギャップを含んだ type3 のコードクローンの検出を行えるように、LCP の計算時に構文木を用いる。今回の提案手法では関数適用のみに注目し、関数適用で生じるギャップを含んだコードクローンの検出を行う。関数適用によるギャップを考慮した LCP の計算方法をアルゴリズム 1 に示す。アルゴリズム 1 においては、構文木の各ノードが対応するソースコード中の開始位置と終了位置を持つことを仮定する。アルゴリズム 1 において、接尾辞 S_n と S_{n-1} の LCP の計算を行う際、まずそれぞれの接尾辞のインデックスとして用いる i と j、計算結果を格納する lcp を 0 で初期化する。$S_n[i]$ と $S_{n-1}[j]$ の比較を行い、等しい場合は i, j, lcp に 1 を加える。$S_n[i]$ と $S_{n-1}[j]$ が一致しなかった場合は関数 *compareAPP*(アルゴリズム 2) を呼ぶ。

この関数 *compareAPP* ははじめに $S_n[i]$ と $S_{n-1}[j]$ が含まれるソースファイルの構文木情報を取得する。その際取得した構文木は *ASTSet* に保存しておき、再び同じソースファイルの構文木が必要になった際に構文解析を行わず、*ASTSet* から取得する。$S_n[i]$、$S_{n-1}[j]$ を含む一番小さい関数適用式を関数 *visit*(アルゴリズム 3) によりそれぞれ取得する。関数 *visit* の返り値から関数適用式の開始位置を取得し、それぞれが一致するかを検査する。一致した場合はそれぞれの終了位置および開始位置を返し、一致しなかった場合は -1 を返す。

関数 *compareAPP* からの返り値をもとに $S_n[i]$ と $S_{n-1}[j]$ が関数適用により生じたギャップであるかどうかを判定し、そうであった場合は得られた関数適用式全体を一つの字句としてみなして lcp の値を計算し、それぞれ関数適用の終了位置まで i および j を移動させる。そうでない場合はループを抜ける。$S_n[i]$ または $S_{n-1}[j]$ が EOF になるまで続け、最後に LCP$[n]$ に lcp を格納する。以上の処理を接尾辞配列全体に適用する。

関数 *visit*(アルゴリズム 3) は構文木の根とその構文木に含まれる字句を受け取り、その字句を含む最も小さい関数適用式を返す関数である。字句の位置情報をもとに子ノードへと探索していくが、その際にノードが関数適用式だった場合にリスト *visitedNode* にそのノードを格納する。*visitedNode* へ最後に格納されたノードを返すことで最も小さい関数適用式を取得する。このアルゴリズムでは関数適用により生じるギャップのみを扱っているが、他の構文への拡張は *visitedNode* へ他の構文のノードも加えることで実現できる。

表 2 はアルゴリズム 1 が図 6 のギャップを含んだコードクローン対の部分を処理している時の LCP 計算の一部を示したものである。LCP$_{base}$ はギャップを考慮しない計算結果であり、LCP$_{alg1}$ はギャップを考慮した計算結果である。

3.3 構文的に不完全なコードクローンの除外

前節までの手順で検出したコードクローンの一例を図 7 に示す。この例のように、検出したコードクローンには構文的に不完全な部分が含まれることがある。字句単位の検出で字句の情報のみを用いた場合はこの問題を解決できない。字句単位の検出手法を用いている CCFinder[8] では前処理としてトップレベルで定義されている関数の終わりに特別な字句を挿入している。この字句は他のあらゆる字句と異なり、トップレベルの異なる関数にまたがったコードクローンが検出されるのを

アルゴリズム **1** . ギャップを考慮した LCP の計算

for $n \in [1,\ tokenNumber\text{-}1]$ **do**
  $i = 0,\ j = 0,\ lcp = 0$
  while $S_n[i] \neq \mathrm{EOF}$ **and** $S_{n-1}[j] \neq \mathrm{EOF}$ **do**
    if $S_n[i] = S_{n-1}[j]$ **then**
      $i = i + 1,\ j = j + 1,\ lcp = lcp + 1$
    else
      $(APPRight_1,\ APPRight_2,\ APPLeft) = compareAPP(S_n[i],\ S_{n-1}[j])$
      if $APPRight_1 \geq 0$ **and** $APPRight_2 \geq 0$ **then**
        $lcp = lcp + APPLeft - i + 1$
        $i = APPRight_1$
        $j = APPRight_2$
      else
        break
      end if
    end if
  end while
  $\mathrm{LCP}[n] = lcp$
end for

アルゴリズム **2** . $compareAPP(token_1,\ token_2)$: 字句を 2 つ受け取り、それぞれを含む最も小さい関数適用式の開始位置が同じかどうかを比較する

/* *ASTSet* にはこれまで構文解析したファイルの先頭ノードがファイル名と関連付けられて格納されている */
$topNode_1 = get(token_1.fileName,\ ASTSet)$
if $topNode_1 = nil$ **then**
  $topNode_1 = parse(token_1.fileName)$
  $add\ (ASTSet, (topNode_1,\ token_1.fileName))$
end if
$topNode_2 = get(token_2.fileName,\ ASTSet)$
if $topNode_2 = nil$ **then**
  $topNode_2 = parse(token_2.fileName)$
  $add\ (ASTSet, (topNode_2,\ token_2.fileName))$
end if
$APP_1 = visit(topNode_1,\ token_1)$
$APP_2 = visit(topNode_2,\ token_2)$
if $APP_1.left = APP_2.left$ **then**
  return $(APP_1.right,\ APP_2.right,\ APP_1.left)$
else
  return $(-1, -1, -1)$
end if

アルゴリズム **3** . *visit* (*topNode*, *token*): 構文木の根と字句を受け取り、その字句を含む最も小さい関数適用式を返す

```
visitedNode = [ ], candidateNode = [topNode]
while node.left ≠ token.left or node.right ≠ token.right do
  node = get (candidateNode)
  if checkRange (node, token) then
    /* node の範囲に token が存在するかを確認 */
    if node.type = APPEXP then
      add (visitedNode, node)
    end if
    add (candidateNode, node.childNode)
  end if
end while
return getLast (visitedNode)
```

表 **2.** 図 3 と図 4 のコード断片を含むソースコードに対する LCP 計算の一部

n	start$[n]$	LCP$_{base}[n]$	LCP$_{alg1}[n]$	s$[n]$
31	8	-	-	`n = 1 ) then ...`
32	26	2	2	`n * if ( n = ...`
33	4	9	12	`n + if ( n = ...`
34	23	1	1	`n m = n + ...`

防いでいる。本研究ではそのような字句の挿入は行わず、構文木情報を用いることでことで構文的に不完全なコードクローンを除外した。

```
n = n + if (n=1) then 1 else oSum(n-2)
n = n * if (n=m) then (n/10) else oPro m (n-2)
```

図 **7.** 構文的に不完全なコードクローン対

コード断片と構文木のノードが与えられた時、それらの位置関係を次の 3 つに分類する。ノードの開始位置および終了位置がコード断片に含まれる時、それらの位置関係を R1 とする。ノードの開始位置および終了位置がコード断片に含まれない時、それらの位置関係を R2 とする。コード断片の開始位置および終了位置がノードの開始位置から終了位置の範囲に存在する時、またはノードの開始位置および終了位置のいずれか一方のみがコード断片に含まれる時それらの関係を R3 とする。これらの位置関係 R1、R2、R3 を図 8 に示す。

これらのコード断片と構文木の関係を用いて、構文的に不完全なコードクローンを除去する手順をアルゴリズム 4 に示す。このアルゴリズムではソースコード断片 C の位置情報とそれが存在するソースコードファイル全体の構文木 T を受け取る。コードクローンを分解した結果を格納するリスト (以降では *retList* と表す) を空に、探索するノードを格納するリスト (以降では *targetList* と表す) に構文木の根を入れて初期化する。関数 *get* により *targetList* からノードを 1 つ取り出し、コードクローンとそのノードの位置関係を関数 *calcRelation* により計算する。関数 *calcRelation* はソースコード断片とそのソースコード断片が含まれる構文木を受け取り、図 8 で示した位置関係を返す関

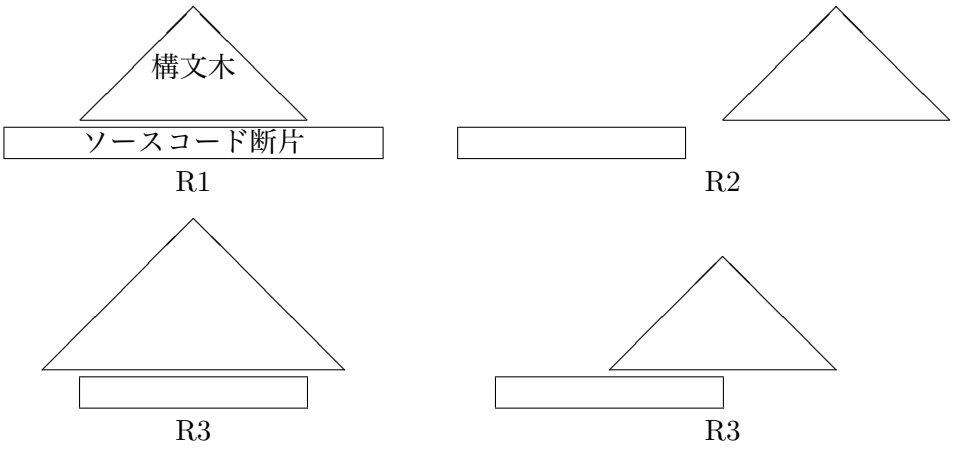

図 **8.** コード断片と構文木との位置関係

数である。関係がR1だった場合ノードを *retList* に追加する。R2だった場合、そのノードはコードクローンに一切含まれないため何もしない。R3だった場合そのノードの子ノードを *targetList* に追加する。これらの操作を *targetList* が空になるまで続け、*targetList* が空の場合は *retList* を返して終了する。このアルゴリズムをコードクローン対を含む構文木それぞれに対し適用する。

このアルゴリズムを適用することで検出したコードクローンが複数のコードクローンに分割される。分割されたコードクローンの大きさが検出するコードクローンの大きさの閾値を下回ったものを取り除くため、これらの処理によりコードクローン総数の増減がある。

アルゴリズム **4** . ソースコード断片 C に含まれる構文的に不完全なコードクローンを取り除いたコードクローンを返す

```
retList = [ ], targetList = [T]
L :
tT = get (targetList)
switch (calcRelation (C, tT))
case R1:
  add (retList, tT)
case R2:
  continue
case R3:
  add (targetList, (subtree tT))
end switch
if targetList = [ ] then
  return retList
else
  goto L
end if
```

4 コードクローンの除去

この節では3節の手法で検出したコードクローンをソースコード上から除去する手法を提案する。コードクローンの検出は自動で行うが、コードクローンの除去においてはツールのユーザが除去す

るかどうか選択できるようにする。検出したコードクローンの中にはユーザが除去したくないものが含まれる可能性がある。

ツールは検出したコードクローンが存在するファイル名および行と列、クローンの大きさ、その部分のソースコードをユーザに提示する。ユーザはそれらの情報を確認し、実際にそのコード断片をコードクローンと判断し、ソースコード上から取り除くかを判断する。ユーザが削除を選択した場合のみコードクローンの除去を行う。

コードクローンの除去は、コードクローン対のギャップ部分を引数とする関数として抽出することにより行う。関数の生成については、3.1 節で述べた引数となる部分は新しい変数で、それ以外の部分はそのまま書いて生成する。関数名および引数名は削除時にユーザに問い合わせ、ユーザが入力した名前を用いる。生成した関数宣言を挿入するべき位置はファイルの先頭、コードクローン対を有効範囲として含む最もコードクローンに近い位置、あるいは新しいファイルを生成するなど様々に考えられる。ユーザは生成した関数宣言のソースファイルへの挿入を行い、その関数を用いてコードクローン部分を関数適用式で書き換える。

本研究のツールが提示する関数を用いてコードクローンを除去すると、プログラムの意味が変わったり、コンパイルエラーが生じる場合がある。例として、let 式を含むコードクローン対を図 9 に示す。これらのコードクローンから関数宣言を生成し、コードクローンを除去したソースコードが図 10 である。この時、除去に用いられる関数の実引数に含まれる x はどこにも宣言されておらず、コンパイル時にエラーとなる。もし他の場所で x が宣言されていた場合は型エラーになったり、プログラムの意味が変わってしまう。このような場合には、コードクローンの提示のみを行い、除去するための関数宣言の提示はしないことが望ましい。

```
let
  val x = 1
in
  add x 3
end
```

```
let
  val x = 2
in
  add (x+1) 3
end
```

図 **9.** let 式を含むコードクローン対

```
fun R a b = let
              val x = a
            in
              add b 3
            end
...
R 1 x
...
R 2 (x+1)
```

図 **10.** 図 9 のコードクローンを取り除いたソースコード

5　実装

提案手法に基づき、Standard ML のプログラムからコードクローンを検出するツールを SML# version 2.0.0 のコンパイラの構文解析プログラムを用いて実装した。そのため、他の Standard ML 処理系の独自の拡張を含むソースコードに対しては対応していない。実装したツールは `http://www.cs.ise.shibaura-it.ac.jp/ppl2016-cc/` で公開している。

このツールは 3 つ以上のコード断片がコードクローンであった場合、複数のコードクローン対として検出される。コード断片 A、B、C がこの順で現れており、かつコードクローンであった場合、A と B がコードクローン対、B と C がコードクローン対であると検出される。A と C の比較は行われないため、A と C はコードクローン対として検出されない。

ツールはユーザからは検出するソースコード群を含むディレクトリのパス、コードクローンの大きさの下限を定める閾値 (以降では th_size と表記) と、コードクローンを除去する際にクローンの大きさと抽出する関数の引数の数の割合に関する閾値 (以降では th_para と表記) を 2 つのパラメータとして受け取る。コードクローンの大きさは字句数である。与えられたソースコードファイルすべての字句解析を行い、字句の種類による接尾辞配列を構成する。アルゴリズム 1 に従い LCP の計算を行い、LCP が th_size より大きく、かつ th_para を満たすものをコードクローンとして保存する。すべての LCP の計算を終えたのち、アルゴリズム 4 に従い構文的に不完全なコードクローンを除外し、処理後の結果をユーザに提示する。ユーザに提示する情報は、コードクローンの存在するファイル名、行番号、列番号、コードクローンの大きさである。検出されたコードクローン対全てに対して 4 節の手順に従い、ユーザが選択したコードクローンのみ除去を行う。

5.1　SML#コンパイラの再利用について

SML#コンパイラの構文解析器が生成する構文木の各中間ノードおよび葉ノードは、ファイル名、行番号、列番号および構文の種類の情報を保持している。本ツールではこれらの情報をアルゴリズム 3 の関数適用の取得およびアルゴリズム 4 のソースコード断片と構文木の位置関係の計算で利用する。

構文木の各ノードが持つ開始位置、終了位置とソースコードとの関係を図 11 に示す。LCP の計算およびコードクローンの分解において、構文木情報が必要になったとき、SML#コンパイラの `Parser` モジュール内の関数 `parse` を呼ぶことで構文木情報を得る。また一度取得した構文木は保存しておき、再び同じファイルの構文木が必要になった際は関数 `parse` を呼び出さずに構文木を取得する。関数 `parse` の結果の抽象構文木において、変数の親ノードは必ず関数適用式であり、3.2 節のアルゴリズム 3 の関数適用式の取得において、変数を含む最も小さい関数適用式はその変数のみからなる関数適用式となる。

字句解析にも SML#コンパイラの字句解析プログラムを利用した。本ツールではソースコードを字句列に変換後、structure が入れ子になっている場合に内側の structure へアクセスするための structure 名がドットでつながれた長い識別子 (qualified identifier[13]) を連結し、1 つの字句として扱っている。コードクローン検出ツールである CCFinder[8] でも類似した処理が C++や Java のソースコードに対して行われる。

6　実験

前節で述べたツールを用いて SML#コンパイラ [1] の Standard ML で書かれた部分を対象としてコードクローン検出実験を行った。表 3 はソフトウェアの大きさ、および実行時間である。実行時間は検出したコードクローンの総数により増減するため、いくつかの閾値で実行した結果の平均を

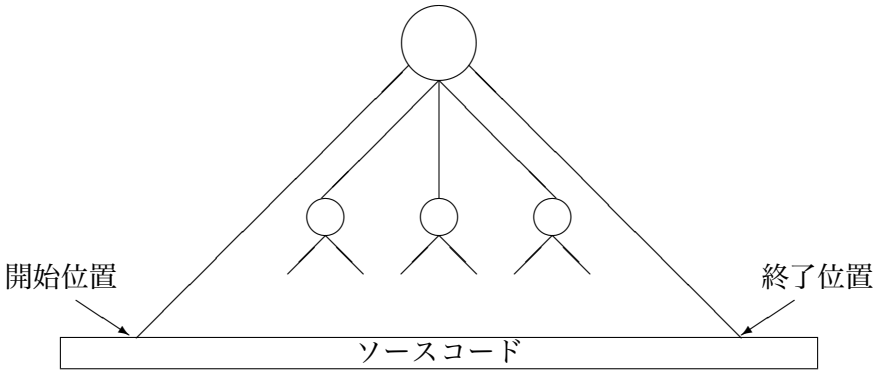

図 **11.** 構文木中の中間ノードが保持しているソースコードの範囲情報

載せた。実験環境は CPU が 3.40GHz Intel Core i7-3770、OS は Windows7 の VMware Player 上の Ubuntu 14 32bit であり、メモリを 4GB 割り当てて実行した。

ツールに与える閾値ごとに、検出されたクローン対の数を表 4 にまとめる。表 4 は、th_size のみの条件で検出されたクローン対の数 ($allClones(th\_size)$) と、その中で閾値 th_para を満たし、さらに構文的に不完全なものを除いたクローン対の数 ($clones(th\_size, th\_para)$) を示す。表 4 におけるクローン数は、type1、type2、type3 のクローン対すべてを含む。これらのクローン対において、type1、type2 を除いた (ギャップを含む) クローン対の数を、閾値が th_size=10、th_para=0.3 の場合について示す。クローン総数 48886 のうち、関数適用によるギャップを含んだクローン対は 468 組であり、その中で th_para を満たし、かつ構文的に中途半端なクローンを除いたクローン対は 92 組であった。

SML#コンパイラから検出されたコードクローンの例を図 12 に示す。図 12 において、$clone_1$、$clone_2$、$clone_3$、$clone_4$ がコードクローンであると検出された。通常の字句単位でのコードクローン検出手法では $clone_1$ と $clone_4$ はコードクローンとして検出されるが、$clone_2$ と $clone_3$ はギャップがあるため検出されない。$clone_2$ と $clone_3$ のギャップ部分は関数適用の形となっているため、提案手法を用いることでコードクローンとして検出される。

提案手法では比較する関数適用を最も小さいものに限定した。字句の不一致が発生した際、取得する関数適用式を最も小さいものに限定しないことでより多くのコードクローンが検出されると思われるが、計算量は増加する。

表 **3.** コードクローン検出実験結果

ソフトウェア名	ファイル数	トークン数	LOC	実行時間
SML#	171	693936	112442	4 時間 1 分 31 秒

7 関連研究

本研究では字句および構文木の情報をもとに検出を行ったが、コードクローンの検出にはその他にも様々な手法が存在する [15]。7.1 節では 3 節で述べた字句単位での検出手法以外のコードクローンの検出手法やそれを実装したツールについて述べる。7.2 節ではギャップを含んだコードクローン

表 **4.** SML#コンパイラ中の Standard ML プログラムから検出されたコードクローン数

th_size	th_para	$allClones(th\_size)$	$clones(th\_size, th\_para)$
10	0.3	48886	7052
	0.1		5259
20	0.3	28093	1925
	0.1		1054
30	0.3	17007	730
	0.1		601

```
...
| (REs as ((End _, _) ::  _)) => REs
...
| ((VarPat x ++ rule, env) ::  REs) =>
(WildPat (#ty x) ++ rule, VarInfoEnv.insert (env, x, path)) ::
removeOtherPat path REs  clone1
...
| ((LayerPat (VarPat x, pat) ++ rule, env) ::  REs) =>
removeOtherPat
path
((pat ++ rule, VarInfoEnv.insert (env, x, path)) ::  REs)  clone2
...
| ((OrPat (pat1, pat2) ++ rule, env) ::  REs) =>
removeOtherPat
path
((pat1 ++ rule, env) ::  (pat2 ++ rule, env) ::  REs)  clone3
...
| (RE ::  REs) =>
RE ::  removeOtherPat path REs  clone4
...
```

図 **12.** SML#コンパイラから検出されたコードクローン例

の検出を行うことができる他の手法の概要および本手法との比較を述べる。7.3 節では関数型言語を対象としたコードクローンに関する既存の研究について述べる。

7.1 コードクローンの検出手法の分類

コードクローンの検出手法は比較の粒度によって以下のような分類ができる。ソースコードを行ごとに分解し、同じ行が一定以上続いたものをコードクローンとして検出する手法がコードクローン研究の初期の段階で考案された [2]。非常に高速な検出が可能であるが、改行や空白の有無などのコーディングスタイルの影響を受ける。また、コーディングスタイルの影響を受けないようにするため、あるいは構文を考慮した検出ができるようにするため、ソースコードを AST へ変換し、類似する部分木をコードクローンとして検出する手法が考案された [3, 5]。これらの手法ではソースコードの構文を考慮に入れた検出が可能である。Koschke ら [9] はソースコード全体を構文解析し、得られた AST のノード (字句) を preorder で並び替えたのち、接尾辞配列による字句単位での検出を行う手法を提案した。この手法は本研究で提案した手法と同様、字句および AST の情報を用いている。この手法で対象するコードクローンは type1 と type2 であり、本研究で対象としたギャップを含んだ type3 のコードクローンは検出しない。またこの手法では構文的に不完全なコードクローンを除外するために、本研究と同様に検出されたコード断片と AST 内のノードの開始位置および

終了位置の比較を行っている。また、ソースコードをプログラム依存グラフ (以降では PDG) へ変換し、類似した部分グラフをコードクローンとして検出する手法が考案された [7]。この手法では、文の挿入や削除が行われても依存関係が損なわれない限りコードクローンとして検出することが可能である。

その他にもソースコードから独自の特徴メトリックスを計算しその情報をもとに検出する手法 [12] や、ソースコードの構文が異なっても関数 (メソッド) の入出力関係やヒープの状態変化を見ることで機能が似ているコードクローンを検出する手法 [6] なども考案された。

これらの手法によって実装されたツールを表 5 に示す [2, 8, 10, 3, 5, 9, 12, 6, 7, 16, 17]。これらの検出手法のうち、AST を用いた手法、PDG を用いた手法はギャップを含んだコードクローンの検出を行うことができる。しかし、これらの手法は多くの計算時間を必要とするため大規模なソースコードへ適用すると長い時間がかかる。大規模なソースコードへ適用可能な手法としては村上らの手法 [16] と肥後らの手法 [17] がある。本研究とこれらの手法の比較を次節で行う。

表 **5.** 既存のツールの検出手法および対象言語

開発者	検出手法	対象言語
Baker[2]	行単位での比較	C
神谷 [8]	接尾辞木による字句単位での比較	C/C++, Java, COBOL など
Li[10]	接尾辞木による字句単位での比較	Erlang
Baxter[3]	AST の比較	C/C++,COBOL, Java など
Brown[5]	AST の比較	Haskell
Koschke[9]	AST の変換で得られる字句列での比較	Ada
肥後 [7]	PDG の比較	Java
Mayrand[12]	特徴メトリックスの計算	C/C++
Elva[6]	メソッドの入出力およびヒープの変化	Java
村上 [16]	Smith-Waterman アルゴリズム	Java, C
肥後 [17]	AGM アルゴリズム	(7.2 節に記載)

7.2 既存のギャップを含むコードクローン検出手法と本論文の提案手法の比較

肥後ら [17] はグラフのマイニングアルゴリズムである AGM を利用することでギャップを含んだコードクローンの検出を行った。この手法の優れている点として他の手法と組み合わせられる点が挙げられる。このツールは他のコードクローン検出ツールの後処理として実装されており、他のツールでは検出できなかったギャップを検出できる。したがってこのツール自体は対象言語を定めておらず、他のツールが対象とする言語ならば適用可能である。しかしこの手法ではすでに何らかの方法で検出した type1 および type2 のコードクローンから type3 を含むコードクローンを検出するため、ギャップの前後をコードクローンとして検出している必要がある。それに対し、本論文の提案手法はギャップによりコードクローンとして検出できないほど小さく分断されていたとしても検出が可能である。

村上ら [16] は Smith-Waterman アルゴリズムを利用することでギャップを含んだコードクローンの検出を行った。Smith-Waterman アルゴリズムは 2 つの配列の中から類似する部分配列のペアを検出するアルゴリズムであり、字句情報のみからギャップを含むコードクローンを検出することができる。このアルゴリズムではギャップの大きさの上限を定めるパラメータを必要とするためギャップが大きい場合、コードクローンとして検出ができない。一方でそれらを検出すために閾値を大きくするとユーザが望んでいないコードクローンも検出してしまう可能性がある。本論文の提案手法

ではギャップが特定の構文、今回は関数適用の形をしている限りギャップの大きさに関わらずコードクローンとして検出が可能である。

7.3 関数型言語を対象としたコードクローンの研究

ここでは既存の関数型言語を対象とした研究との比較を行う。筆者が調査した限り、Standard ML のコードクローンの検出を行う研究は存在しない。

Li ら [10] は字句単位の手法により Erlang を対象としてコードクローンの検出を行った。検出するコードクローンは type2 までであり、本研究で対象とした type3 のコードクローンは検出の対象としていない。

Brown ら [5] はソースコードから AST を構築し、式の構成要素の比較を行うことで Haskell を対象としたクローン検出手法を示した。この手法においては式の構成要素がギャップ部分を除いて等しい type3 のクローンの検出が可能であり、ギャップ部分が関数適用式となっているクローンの検出も可能である。しかし、Brown らの手法 [5] によるクローン検出は最悪の場合、ソースコードの大きさに対し $O(n^2)$ の計算時間を必要とし、大規模なソースコードに対して適用すると多くの実行時間がかかる。また、比較する式の識別子が異なるとき、それがトップレベルで宣言された識別子の場合はクローンとして検出されない。

8 まとめと今後の課題

本論文では関数適用に着目し、字句単位でのコードクローン検出に構文木情報を利用することで関数適用により発生したギャップを含むコードクローンを検出する手法を提案した。また字句単位での検出手法の課題の一つであった構文的に不完全なコードクローンを検出してしまう問題も、構文木情報を用いてそれらを複数のコードクローンに分解することで不完全な部分をコードクローン候補から除外した。提案手法に基づき、Standard ML で書かれたコードクローンの検出ツールを実装し、コードクローン検出実験を行った。

今後本ツールをより実用的にするためには以下のような課題が考えられる。

- 本手法では不一致が発生した場合、ギャップ部分を関数適用に限定した。しかし、本論文で提案したアルゴリズムは関数適用に限らず他の構文にも 3.2 節で述べた通り、容易に応用可能である。様々な構文に拡張し、それぞれに対して考察を行うことが望まれる。
- 本実装ではギャップを探索するためにギャップを含むソースコード全体に対して構文解析を行い、AST を取得した。しかしギャップを探索する上で必要としない部分の AST の作成も行っている。ギャップを含むコード断片部分のみに対して構文解析を行うことで実行速度の改善が見込まれる。しかし、通常の構文解析器では構文エラーとなるので、特別な構文解析処理が必要となる。
- 本論文では字句単位での検出手法に一部構文木の情報を利用することで、11 万行程度の Standard ML のソースコードからギャップを含んだコードクローンの検出を行った。既存の AST を用いる手法と全く異なる手法であり、今後、計算量を字句数や関数適用式の数をパラメータとして算出し、既存の検出手法との計算量の比較を行う。
- Standard ML は型推論を備えた言語であり、プログラム中に変数の型情報を書く必要がない。しかし本研究のコードクローン検出手法においては型を考慮しておらず、検出したコードクローンを除去した際に生成した関数をそのまま利用すると型エラーになる可能性がある。現在は検出したコードクローンをユーザに提示し、ユーザの判断により型エラーになる場合は除去しないようにできるが、将来的には型エラーを含むコードクローンは除外する、あるいは型エラーを含む旨も同時にユーザに提示することなどが考えられる。

謝辞

本論文について大変有益な意見を下さった査読者の方々に感謝します。

参考文献

[1] SML#プロジェクト. `http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/`.

[2] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pp. 49–49, 1993.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceeding of the International Conference on Software Maintenance 1998*, pp. 368–377. IEEE, 1998.

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[5] Christopher Brown and Simon Thompson. Clone detection and elimination for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 111–120, 2010.

[6] Rochelle Elva and Gary T. Leavens. Semantic clone detection using method IOE-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, pp. 80–81. IEEE, 2012.

[7] Yoshiki Higo and Shinji Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *Proceeding of 16th Working Conference on Reverse Engineering*, pp. 315–316. IEEE, 2009.

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[9] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceeding of 13th Working Conference on Reverse Engineering*, pp. 253–262. IEEE, 2006.

[10] Huiqing Li and Simon Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 169–178. ACM, 2009.

[11] Udi Manber and Gene Myers. Suffix Arrays: A new method for on-line string searches. *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948, 1993.

[12] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance 1996*, pp. 244–253. IEEE, 1996.

[13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[14] Shinji Uchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Kenichi Matsumoto, and Hideo Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. 45, No. 3, pp. 1–11, 2005.

[15] 神谷年洋, 肥後芳樹, 吉田則裕. コードクローン検出技術の展開. コンピュータソフトウェア, Vol. 28, No. 3, pp. 29–42, 2011.

[16] 村上寛明, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二. Smith-Waterman アルゴリズムを利用したギャップを含むコードクローン検出. 情報処理学会論文誌, Vol. 55, No. 2, pp. 981–993, 2014.

[17] 肥後芳樹, 植田泰士, 楠本真二, 井上克郎. AGM アルゴリズムを用いたギャップを含むコードクローン情報の生成. 信学技報. SS, ソフトウェアサイエンス, Vol. 107, No. 392, pp. 61–66, 2007.