6L-8

関数型プログラミング言語におけるコードクローン検出

石川　琢也 †　　太田　剛 ‡
静岡大学大学院情報学研究科 †　　静岡大学情報学部 ‡

１．導入

コードクローンとは、ソースコード中に存在する互いに一致または類似したコード片である。コードクローンは、ソースコードのコピー＆ペーストによる再利用や、頻繁に用いられる処理などにより生成され、保守を困難にする原因となる。

コードクローンは、その違いの度合いにより、次の三種類に分類される。

（ｉ）コーディングスタイルを除いた、完全に一致するコードクローン(Exact Clone)
（ii）変数名や関数名、一部の予約語のみが異なるコードクローン(Renamed Clone)
（iii）文の挿入や削除、変更が行われたコードクローン(Gapped Clone)

２．既存のツール

これまでに、手続き型プログラミング言語や、オブジェクト指向プログラミング言語のソースコードに対して、コードクローンを検出するさまざまな手法が提案されてきた。有名なものとして CCFinder[1]がある。このツールは、ソースコードを字句解析によりトークン列に変換してクローン検出を行うトークンベース検出手法を用いており、Exact Clone や Renamed Clone に対応することが可能である。

しかしながら、CCFinder のような手続き型言語に対応するツールを関数型言語に対して用いると、検出に必要な識別子に関する情報を落としてしまう上、Gapped Clone に対応するのも難しく、関数型プログラミング言語に頻繁に現れる入れ子構造に対応できない。そのため、関数型言語用のツールが別途必要となるが、現状では、関数型プログラミング言語 Erlang の対話形式リファクタリングツール Wrangler に組み込まれたコードクローン検出機能のみである。この機能もまた、(X+Y)と(X+(Y+1))のような完全一致ではないが類似するようなコード片を検出できない問題点があるため、反単一化という概念を用いて対応する手法が提案されている[2]。

A Code Clone Detection Algorithm for A Functional Programming Language
†Takuya Ishikawa, Graduate School of Informatics, Shizuoka University
‡Tsuyoshi Ota, Facility of Informatics, Shizuoka University

３．目的

上記の状況を受けて本研究では、関数型プログラミング言語のソースコードに対して、コードクローンを検出する手法を提案、実装を目的とする。対象言語は、Common Lisp を想定する。

４．本研究の手法

上記のように、関数型プログラミング言語では、識別子の扱いと入れ子構造がクローン検出を難しくしている。そこで本研究では、字句解析時のトークン変換ルールと、構文木の部分的な抽象化により、この問題に対応する手法を提案する。

４．１　検出アルゴリズム

クローン検出プロセスは、次の３つのステップで行われる。

字句解析

入力として与えられたソースコードを字句解析し、トークン列へと変換する。Common Lisp では、ループなどの構文構造も関数が担うため、組み込み関数は区別する必要がある。また、ユーザ定義の関数も個別のトークンに変換される。

接尾辞配列の構築と分析

トークン列から接尾辞配列を構築する。トークン列の接尾辞とは、トークン列のある開始位置から終端までの部分列を指す。接尾辞配列は、全ての開始位置の接尾辞を辞書順に並べたもので、これを利用して接尾辞の先頭の共通部分を調べることで、トークン列の類似部分を抽出することができる。

抽象構文木の構築と比較

抽出した類似部分トークン列から構文木を構築する。Common Lisp の場合、単一のトークンはそのままノードとして構文木に追加される。左括弧と右括弧で囲まれた部分（リスト）も一つのノードとして構文木に追加され、リスト中のトークンは、追加されたノードの子ノードとして追加される。例えば、(* x 2)と(* x (+ y z))は、図１のように構文木が構築される。

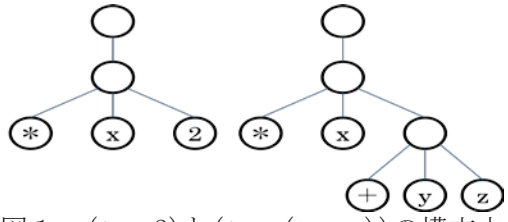

図１　(* x 2)と(* x (+ y z))の構文木

構築した構文木に対し、深さごとに抽象化を行う。対象となる深さより深い部分木（すなわち深く入れ子になったリスト）は、単一のトークンとして捉え、抽象化する。図１の２つの構文木を深さ２で抽象化した場合、図２のように抽象化される。このように入れ子構造を取り除くことで、Gapped Clone にも対応できる。深さの数だけ抽象構文木を構築する（図１右の場合、深さ３まであるので、３つの抽象構文木ができる）。

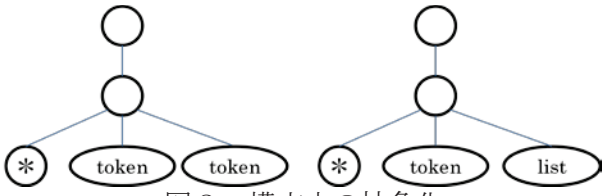

図２　構文木の抽象化

最後に、構築した抽象構文木を比較して、最終的なクローンコード片を検出する。

５．評価実験

クローン検出ができているかを調査する。その方法として、意図的に Gapped Clone を埋め込んだ Common Lisp プログラムを用意し、本研究の手法でクローン検出を行い、元のコードとクローンとなるコードがどの程度検出できているか、その正答率を調査する。ある一つのクローンペアの正答率は、LEN(c)をコードcのトークン数として、次の式で計算する。

$$\text{正答率} = \frac{\frac{\mathrm{LEN}(\mathrm{dc}_{\mathrm{original}})}{\mathrm{LEN}(\mathrm{c}_{\mathrm{original}})} + \frac{\mathrm{LEN}(\mathrm{dc}_{\mathrm{intentional}})}{\mathrm{LEN}(\mathrm{c}_{\mathrm{intentional}})}}{2}$$

ただし、$\mathrm{c}_{\mathrm{original}}$は元のコード、$\mathrm{c}_{\mathrm{intentional}}$は意図的クローンコード、$\mathrm{dc}_{\mathrm{original}}$は検出できた元のコードの一部、$\mathrm{dc}_{\mathrm{intentional}}$は検出できた意図的クローンコードの一部である。

元のプログラムは、Common Lisp で書かれたオープンソースソフトウェアのランダムに選んだソースコード（1000 行程度）で、一部をコピー＆ペーストし、式の挿入・削除・変更を行った。意図的なクローンは 4 組作成し、それぞれのクローンがどの程度検出できたかを計算した。

また比較のため、手続き型プログラミング言語に使われる CCFinder に代表されるトークンベースのクローン検出法についても、同じ Common Lisp コードを適用して実験を行った。なお、CCFinder は Common Lisp に対応していないため、字句解析時の変換ルールと接尾辞分析部を自作して模したものを使用して実験を行った。

実験結果を表１に示す。クローン１は単一の式の追加・削除・変更がされたクローン、クローン２は式中のシンボルをリストに変換、またはリストをシンボルに変換したクローン、クローン３は元とは異なる複数の式の追加・削除・変更がされたクローン、クローン４は関数の変更と式の順序の変更がされたクローンである。

表１　実験結果

	クローン1	クローン2	クローン3	クローン4
手続き型トークンベース	12.39%	47.28%	32.52%	25.68%
本研究	70.03%	90.02%	45.55%	57.21%

クローン１は７割程度、クローン２は９割程度、クローン部分を検出できている。これは、単一の式に対する変更やシンボルとリストの変更であるため、本アプローチの部分木の抽象化がうまく働いたためと考えられる。逆にクローン３と４の正答率が低いのは、抽象構文木の構造が大きく変わることが原因と考えられる。

６．まとめ

本研究では、関数型プログラミング言語のクローン検出が行えるように、関数名の扱いと抽象構文木の部分木の抽象化を行った。その結果、既存の手続き型ではうまく検出できなかったクローンを検出することができた。

今後の課題としては、複数の式の変更にも耐えられる抽象化方法の考案が挙げられる。また、抽象構文木の構造の変化の度合いにより、クローンとして検出していいかを判断する手段も必要になると考えられる。

参考文献

[1] T. Kamiya, S.Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol.28, no.7, pp.654-670, July 2002.

[2] Li, H. and Thompson, S.: Similar Code Detection and Elimination for Erlang Programs, in Practical Aspects of Declarative languages 2010 (PADL2010), LNCS 5937, pp.104-118, Springer-Verlag, 2010.