Code Similarity Detection Tool
Design Documentation

Kevin Cao

April 5, 2021

Contents

1 Introduction

With the rising popularity of cryptocurrencies, it has become crucial that crypto-transactions are not only trustworthy, but also efficient and reliable. While Bitcoin continues to control a majority of the crypto-market, alternatives like Ethereum have also been growing in popularity.

For some of these alternatives, transactions are controlled and automated using "smart contracts". These smart contracts are computer programs that explicitly detail the rules and regulations to an agreement between two or more parties. Once the terms of the agreement have been met, the contract will self-execute and complete the transaction as agreed upon.

1.1 Purpose

For many transactions, the corresponding smart contracts are derived from pre-existing popular and mature contracts. By detecting which parent contracts the newer contracts are derived from, the validation and verification process of the contracts can be expedited.

As such, a Code Similarity Detection Tool would serve the purpose of assisting in the process of determining the original source of a new contract.

1.2 Scope

This Software Design Document details a basic system that serves as a proof of concept for a Code Similarity Detection Tool. For the sake of simplicity, the tool will compare the similarity between code written in a simplified subset of the lisp-based programming language Racket. The system first translates the source code into an Abstract Syntax Tree (AST), and then converts the AST into a variation of De Bruijn notation before running the similarity algorithm. Lisp-based languages are best suited for this type of translation, but other languages can also be similarly converted, although the process is a bit more involved.

Furthermore, it should be clarified that the Code Similarity Detection Tool detailed in this document evaluates syntactical similarity, *not semantic similarity*. The reasoning behind this is detailed later.

2 Conversion of Source Code

2.1 Programming Language Scope

As mentioned in the previous section, the tool will act on a simplified subset of the Racket programming language, which will be referred to as R_1. As such, it is best that this subset is defined explicitly.

The most basic component in R_1 is an *atom*. Atoms include the basic primitive data values such as integers, booleans, and strings. It also includes variables and empty lists. Furthermore, as functions are first-class citizens in R_1, inbuilt functions can also be considered as atoms.

However, most programs in R_1 will not simply be made up of atoms —R_1 is largely comprised of *exp*s, or expressions. An expression can be a simple atoms, an if-statement, a let-statement, a function application, or a lambda function. Figure 1 shows some examples of expressions.

Of course, in a functional programming language like R_1, it is imperative that function definitions are explicated. Functional definitions are in a class of their own, *def*s. Definitions detail function construction by first taking a variable name to describe the function, and then variable parameters, which are used in the expression body of the definition. Expressions can use the function name from the definition in functional applications, which are categorized under expressions.

Finally, an R_1 program can be described as a list of function definitions followed by a list of expression statements.

```
atom  ::= int | bool | str | empty | var | + | - | · | cons | eq? | and | or | not | ...
exp   ::= atom | (if exp exp exp) | (let ([var exp]) exp)
        | (+ exp exp) | (- exp exp) | (· exp exp) | (- exp)
        | (cons exp exp) | (eq? exp exp) | (and exp exp) | (or exp exp) | (not exp)
        | (λ (var ...) exp) | ...
def   ::= (define (var var ...) exp)
R1    ::= def ... exp ...
```

Figure 1: The concrete syntax of R_1. "..." signifies a list of the previous token.

Compared to Racket, R_1 is a fairly barebones language—however, R_1 will be sufficient in demonstrating the core mechanic of the system.

2.2 Abstract Syntax Trees

Before we can begin the similarity algorithm, we first need to convert the source code into an Abstract Syntax Tree, which abstracts away the source code into the node objects of the tree. In short, we convert the concrete syntax detailed in Figure 1 into the abstract syntax detailed in Figure 2.

```
op    ::= + | - | · | cons | eq? | and | or | not | ...
atom  ::= (Int int) | (Bool bool) | (Str str) | (Empty) | (Var var) | (Op op)
exp   ::= atom | (If exp exp exp) | (Let (Var var) exp exp) | (Apply exp exp ...)
        | (Lambda ((Var var ) ...) exp) | ...
def   ::= (Def (Var var) ((Var var ) ...) exp)
R1    ::= (Program (def ...) (exp...))
```

Figure 2: The abstract syntax of R_1.

In Figure 2, the components of R_1 are converted into objects. You may notice a new classification, *op*. This simply acts as an abstraction for inbuilt functions in the R_1 language so that we can create the `(Op op)` object in *atom*.

The rest of the Figure 2 is fairly self-explanatory, so we will not go into further detail here.

2.3 De Bruijn Notation

While certain structures of code, such as if-statements or function definitions, must follow a set syntax, variable names are up to the discretion of the developer. In order to properly perform the code similarity algorithm, we must ignore the variable names and normalize the syntax.

One method of doing so is to use De Bruijn indices—variables are instead replaced with numbers that indicate how many variable bindings exist between the variable occurrence and its corresponding binding. To avoid confusion with actual integers, we will rename variables as `v#`.

```
; Original Syntax
(λ (x)
  (λ (y)
    (λ (z)
      (+ x (+ y z)))))

; De Bruijnized Syntax
(λ
  (λ
    (λ
      (+ v2 (+ v1 v0)))))
```

Figure 3: Demonstrating conversion to De Bruijn indices.

However, this solution is not perfect—multi-variable functions would require currying in order to properly be De Bruijnized. Unfortunately, currying functions would make it difficult to perform similarity checks on function applications. In a later section, we will discuss the methodology for comparing two separate functional applications, but in short, we will need to collect all function arguments to properly compare. If functions are curried, then collecting the function arguments will require traversing down the AST and determining which expressions belong to the function application, which is messier than would be preferred (shown in Figure 4).

```
; Original Syntax
(define (func x y z)
  (+ x (+ y z)))

; Curried Syntax
(define (func x)
  (λ (y)
    (λ (z)
      (+ x (+ y z)))))

; Curried Function Application 1
(((func (+ 4 1)) (+ 2 3)) 6)

; Curried Function Application 2
(((func 6) (+ 4 1)) (+ 2 3))

; AST 1
(Apply
  (Apply
    (Apply
      (Var func) (Apply (Op +) (Int 4) (Int 1)))
    (Apply (Op +) (Int 2) (Int 3)))
  (Int 6))

; AST 2
(Apply
  (Apply
    (Apply
      (Var func) (Int 6))
    (Apply (Op +) (Int 4) (Int 1)))
  (Apply (Op +) (Int 2) (Int 3)))
```

Figure 4: The AST representation of two similar applications of `func` hide away the arguments of `func`, making it difficult to parse out the arguments for comparison.

It would be best to keep functions as is so that all of the arguments are kept within the same `Apply` object at the same level. As such, we use a variation of De Bruijn notation such that for multi-variable functions, variables are considered bound in the order in which they appear in the list of parameters. In essence, we simulate the effect of currying without actually expanding the function and currying it (See Figure 5).

```
; Original Syntax
(define (func x y z)
  (+ x (+ y z)))

; De Bruijn Variation Notation
(define (func)
  (+ v2 v1 v0))
```

Figure 5: Example of De Bruijn Variation Notation

2.4 Implementation

2.4.1 AST Conversion

Fortunately, the Racket programming language does make parsing R_1 rather simple as its `read` function will read individual expressions from the file. Racket's `match` function allows for recursive pattern matching, so a parser function can be applied recursively on the sub-expressions within expressions.

To facilitate in the process of converting the concrete syntax of the R_1 language into an AST, we can take advantage of Racket `structs`, lending to a somewhat OOP-style implementation. As each expression is parsed, we generate the corresponding `struct`, and recurse on its components.

After repeatedly calling the `read` function on the input file and parsing it, we are left with a list of various expression/definition AST nodes. After a quick partition of the list, we can then construct the `Program` struct with the disjoint list of definitions and expressions.

2.4.2 De Bruijn Notation