Code Similarity Detection Tool
Design Documentation

Kevin Cao

April 6, 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 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 semantical 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 atom, 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*`)`
R_1	::=	*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 standardize 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 (See Table 1). As each expression is parsed, we generate the corresponding `struct`, and recurse on its components.

`If`
con : condition
consq : consequent
alter : alternative
`Let`
v : variable name
val : value to bind
body : main body of let
`Lambda`
p-ls : list of parameters
body : main body of lambda
`Def`
fn-name : function name
p-ls : list of parameters
body : main body of function
`Apply`
fn : applied function
args : arguments for function
`Program`
def-ls : list of definitions
exp-ls : list of expressions
	`Int`
	n : integer value
	`Bool`
	b : boolean value
	`Str`
	s : string
	`Op`
	o : operation
	`Var`
	v : variable name
	`Empty`

Table 1: `struct` Representations of AST Nodes

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. With 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 Conversion

Once we have finished the conversion from concrete syntax to an AST, converting to De Bruijn notation is quite simple, especially since we are not evaluating the program's behavior, only changing the syntax form.

To calculate the number of bindings between a variable's occurrence and its initial binding, we maintain an environment of variables as we evaluate an expression. For this use case, the environment is quite simple—it is simply a list of variables, and each time we encounter a variable binding, we add the variable to the front of the list. When we see an occurrence of a variable, we look for its index in the environment, and that will be the `#` in our `v#`.

If the variable does not exist in the environment, we assume that the variable is a function that has already been defined in the source code, and leave it as is. As we are simply checking

the syntax of the source code and are not checking for errors, we assume that there are no occurrences of free (unbound) variables.

There was some consideration made toward the neccessity of standardizing function names as well, perhaps relabeling functions in the form `f#`, so that there could be a differentiation made between functions and variables. However, it was ultimately decided that doing so would not have a large effect on the similarity algorithm.

3 The Similarity Program

Up until this point, the program has been implemented in the Racket programming language. The algorithm is best implemented in an object oriented programming language, so at this point, we make a transition from Racket to Python.

3.1 Comparing AST Expressions

Comparing AST expressions can be done by comparing similar structures together and recursively comparing their components. We can use a point-based system to represent similarity—for every similar feature, we add a point. The total number of points would be their **similarity score**.

For *atom* nodes, if two atomic nodes are of the same structure, that adds 1 point of similarity. If they share the same structure and the same value, add another point of similarity. For `Empty` nodes, two `Empty` nodes will automatically have 2 points of similarity as they have no value to compare.

Comparing two expression nodes of the same structure is a bit more involved as the subcomponents will need to be recursed on. Before discussing the comparison methods for expression nodes, we need to explain what is called the **similarity cap**. The similarity cap is a value used to determine the relative similarity of two programs—otherwise, larger programs will naturally have a higher similarity score as they will have more AST nodes to process. To calculate the similarity cap of a program, count the number of AST nodes in the tree, counting *atom* structures twice as they can have a max similarity score of 2.

- `If`

  Compare the two condition expressions and take the similarity score as the first score.

  To compare the consequents and alternatives, calculate the combined similarity score of comparing consequent to consequent and alternative to alternative. Then calculate the combined similarity score of comparing consequents to alternatives. Choose the larger of the two scores as the second similarity score.

The sum of the two similarity scores will be the similarity score of the two `If` structures.

- `Let`

  The sum of the similarity scores from comparing the value expressions and comparing the body expressions will be the similarity score.

  As all variables have De Bruijnized, there is no need to consider the new variable created in the `Let`.

- `Apply`

  Calculate the similarity score from comparing the applied functions.

  To calculate the similarity score of the arguments, we will want to pair up each argument with the most similar argument from the other structure. Call the list of arguments from the first `Apply` structure `args1` and the second list of arguments from the other structure `args2`.

  Take the first argument from `args1`. Compare it to each argument from `args2` and pick the argument that has the highest similarity score. Remove that argument from `args2`.

  Take the next argument from `args1` and repeat the previous step again.

  Do so until all pairings have been made. Sum up the similarity scores from the resulting pairs and the similarity score of the applied functions to achieve a tentative similarity score.

  If there are excess unpaired arguments, sum up the similarity caps of the excess arguments and subtract that from the tentative score above (with a minimum total score value of 0). The resulting value is the similarity score of the `Apply` structures.

- `Lambda`

  Calculate the similarity score of the bodies of the lambda expressions and subtract from that the difference in function arity to determine the overall similarity score.

- `Def`

  This is the same as `Lambda`—calculate the similarity score of the bodies of the definitions and subtract from that the difference in function arity.

- `Program`

  First, pair up function definitions in the same way as described for arguments in `Apply`. Sum up the similarity scores from the resulting pairs.

  Do the same for the list of expressions in both `Program` structures. Sum up the similarity scores from the resulting pairs.

  Sum up both similarities scores from the definitions and expressions to calculate the similarity score of the two `Program`s.

To calculate the relative similarity between two source files, calculate the similarity cap of the two files separately. Divide the similarity score by the smaller of the two similarity caps to calculate the relative similarity.

3.2 Implementation

3.2.1 Python OOP Design

Fortunately, the `structs` described in Table 1 serve as a decent starting point for the OOP design of the AST implementation in Python. For better scalability, we just need to create some abstract classes to represent the tree structure. See Figure 6 for more details.

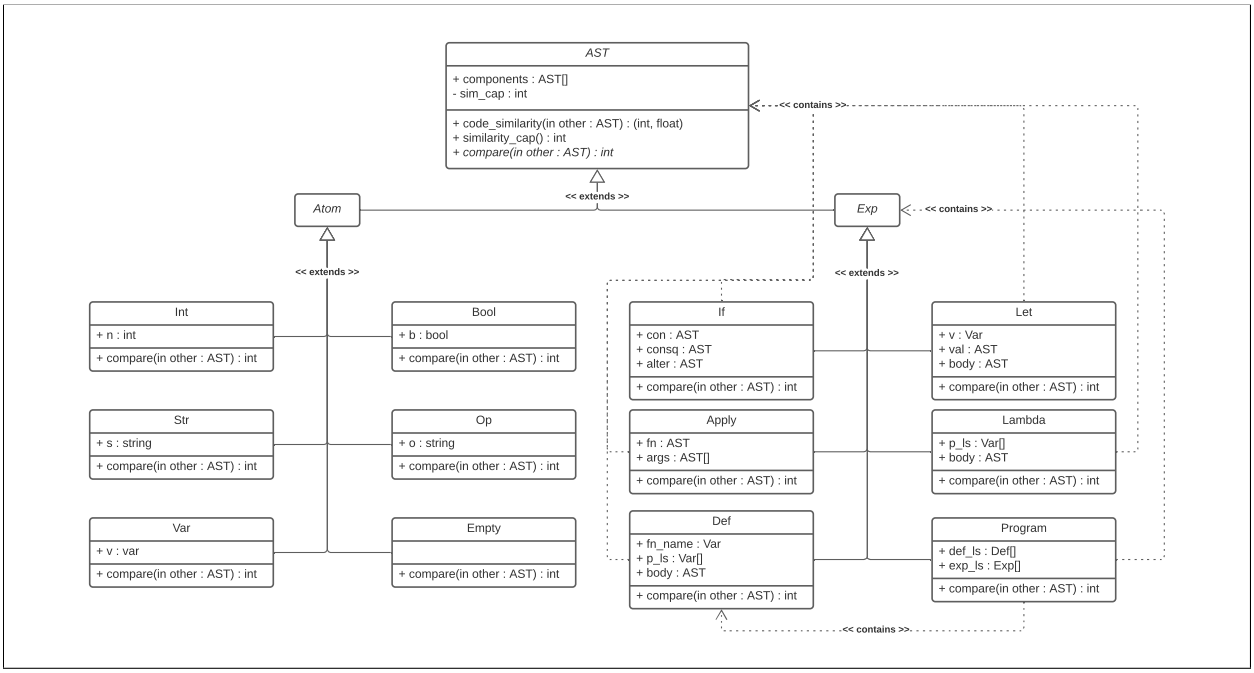

Figure 6: UML Diagram for OOP Design of AST

In addition to the `struct` fields introduced in Table 1, we have added two additional fields and three new methods to `AST` objects. The `compare` method calculates the similarity score as detailed in the previous section. Likewise, `similarity_cap` calculates the similarity cap. The field `sim_cap` serves to allow for lazy evaluation of the similarity cap—once `similarity_cap` is called once, then `sim_cap` is updated with the calculated value. In all future calls to `similarity_cap`, `sim_cap` is returned instead of reperforming calculations. The field `components` is used in calculating the similarity cap; it contains all components of the node required to calculate the similarity cap so that they can be recursed on. This allows for better abstraction, as all `Exp` objects calculate their similarity cap in the same manner.

While the `Atom` and `Exp` abstract classes are largely unneeded, they were included for the sake of scalability in the future. It is possible that such abstract classes would be needed, so they were included in the design process. The implementations of the `compare` methods are fairly straightforward and follow the algorithms described in the previous section.

3.2.2 Converting Racket AST to Python AST

Converting Racket AST `structs` to Python objects is also fairly painless. We simply match on the AST nodes, create strings representing Python object construction, and recurse on the components to create the other constructors.

From there, we can pipe the constructed strings through `stdout` and into a Python script that runs Python's `eval` function to create the described objects.

3.2.3 Terminal Interface and Behavior

To interact with the similarity program, we need some sort of terminal interface that we can work with to process files for similarity. The purpose of the Code Similarity Detection Tool is to detect how similar a provided smart contract is to pre-existing popular and mature contracts. As such, it follows that we maintain a database of the aforementioned pre-existing contract. In this project, those files would be stored in the `archetypes/` directory.

The program needs to first read all of the `archetypes/` files and create individual ASTs for each file. Then terminal program should ask for a single file to run the Code Similarity Detection Tool on and generate the AST to check from the provided file. Then simply run the comparison function between that AST and all ASTs generated from the `archetypes/` directory and output the pairing with the highest relative similarity.

4 Weaknesses and Improvements

4.1 Hiding Similarities in Differing Structures

Unfortunately, this design is not without flaws. The code similarity algorithm simply returns 0 if two AST nodes are not of the same structure. This means the following example would return 0 as a code similarity score, despite being fairly similar:

```
(let ([x 5])
  (+
    (if (< x 10)
        12
        x)
    5))

; vs

(+
  (let ([x 5])
    (if (< x 10)
        12
        x))
   5)
```

Figure 7: An edge-case example that shows the weakness of the algorithm.

It is certainly possible to ameliorate this issue—perhaps discarding the variable binding in `Let` structures when performing comparisons between a `Let` structure and a different structure could work. However, further testing would be required to determine if this causes any other issues.

4.2 Syntactical vs Semantical Analysis

As mentioned in the beginning of this document, this tool evaluates syntactical similarities. In other words, it is unable to determine if two programs behave similarly, only if they look similar. The following two programs would score highly syntactically, despite being different semantically:

```
(define (fib n)
  (if (and (eq? 1) (eq? 0))
    1
    (+ (fib (- n 1)) (fib (- n 2)))))

; vs

(define (fact n)
  (if (and (eq? 1) (eq? 0))
    1
    (* n (fact (- n 1)))))
```

Figure 8: Syntactical vs Semantical Analysis

Syntactical analysis, unsurprisingly, is a much simpler task than semantical analysis. Take for example, a case where two programs apply the same 2+-ary function, but with the arguments in different orders. Determining similarity in syntax would be simple, but determining behavioral similarity would require discerning whether the function is deterministic, commutative, etc. Furthermore, according to Rice's Theorem, fully determining behavioral equivalence of two programs is undecidable.