Code Similarity Detection Tool
Design Documentation

Kevin Cao

April 3, 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 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.

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*. *atom* includes the basic primitive data types, such as integers, booleans, symbols, empty lists, and also includes variables.

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

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. *def*s detail function construction by first taking a variable name to describe the function, and then variable parameters, which are used in the *exp* body of the *def*. *exp*s can use these *def*s in functional applications, which are categorized under *exp*s.

$$
\begin{array}{lcl}
\mathit{atom} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{sym} \mid \texttt{`()} \mid \mathit{var} \\
\mathit{exp} & ::= & \mathit{atom} \mid \texttt{(if } \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(let ([}\mathit{var}\ \mathit{exp}\texttt{]) } \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(+ } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(- } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(}\cdot\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(- } \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(cons } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(eq? } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(and } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(or } \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(not } \mathit{exp}\texttt{)} \\
 & \mid & \texttt{...} \\
\mathit{def} & ::= & \texttt{(define (}\mathit{var}\ \mathit{var}\ \texttt{...) } \mathit{exp}\texttt{)} \\
R_1 & ::= & \mathit{def} \ldots\ \mathit{exp}
\end{array}
$$

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.