University of Amsterdam

Individual Written Assessment

Designing `legalease`, a Smart Contracts programming language

March 8, 2021

Lecturer:
dr. Ana Oprescu

Student:
René Kok
13671146

Course:
Programmeertalen

Course code:
5062PROG6Y

1 Introduction

Smart Contracts [**what-are-smart-contracts**] are key to simplify complex blockchain transactions. Smart contracts can streamline this complex process that involves several intermediaries because of a lack of trust among participants in the transaction.

The ultimate goal of this essay is to design a programming language that best fits the use cases of smart contracts. The language is designed on the basis of the design choices: the type system, state management, compilation/interpretation strategy, evaluation strategy (lazy/eager), parameter evaluation strategy (call by value/reference), communication semantics (synchronous/asynchronous), higher-order functions and anonymous functions.

1.1 Relevant characteristics of application domain

1.2 Use case description

2 Analysis

2.1 The type system

Because smart contracts are designed for transactions, a smart contract needs to be accurate and secure. No hidden surprises at runtime. That is why I chose to have the language static typed. Static type checking has the advantages that the smart contract is guaranteed to meet a number of type safety features for all possible inputs. In addition, a static typing language is better optimized as opposed to dynamically typed language, because the compiler knows if a program is correctly typed. This results in a smaller and faster binary because no dynamic safety checks need to be performed.

While type inference does not affect runtime of a program, because types are inference occurs at compile-time, an explicit type inference seems to be the best choice for smart contracts because explicit type annotations serve as documentation for the code.

I also made the choice for nominative typing. While structural typing is more flexible, nominative typing is less prone to errors. It's common for objects to be structural equivalent, but semanticily different. polymorphism can be implemented through a shared interface such as in the Java programming language.

2.2 State management

2.3 Compilation/interpretation strategy

I've chosen for an ahead of time (AOT) compilation. In contrast to the JIT compiler, the original source code does not have to be supplied, but a compiled smart contract is uploaded. Normally, a compiled version must be written for each type of system. This is not the case for our smart contracts, as the code only runs on a specialized virtual machine designed for our blockchain. AOT compilers can also perform more complex and advanced code optimizations, which in most cases of JIT compiling is considered to be too costly. With AOT compilation, the programs runs faster because the program is already compiled before running.

2.4 Evaluation strategy

I made the choice for an eager evaluation strategy to avoid unexpected behavior. In addition, lazy evaluation is difficult to combine with imperative functions such as input/output, because the order of operations becomes indeterminate.

2.5 Parameter evaluation strategy

I choose for call by value as parameter evaluation strategy. I have made this choice so that the scope of a variable is clear. So the chance of unwanted side effects is reduced by modifying a copy of a value as opposed to call via reference.

2.6 Communication semantics (synchronous/asynchronous)

As far as I know, there is no language that is completely asynchronous. I want my programming language to be synchronous, but it needs to support asynchronous communication semantics.

For example, information can be retrieved from multiple sources at the same time by means of asynchronous requests, which means that the processing time of a smart contract is shorter than when this information is retrieved synchronously.

2.7 Higher-order functions

I want my programming language to support higher-order functions. A good use-case for higher-order functions is validating an object. The basic idea is a function that takes an object as an argument and then any number of functions that must evaluate to true for the object to be considered valid. See "Appendix A: Higher-order functions example" For a short example of applying Higher order functions to validate an object.

2.8 Anonymous functions

A few minor disadvantages have emerged during my research into anonymous functions. These points are: Anonymous functions are harder to debug and anonymous functions are not reusable. Because anonymous functions are an addition to the language and not a trade-off, I have made the choice to offer support for anonymous functions. So programmers are free to make use of anonymous functions.

3 Discussion

4 Conclusions

5 Appendix A: Higher-order functions example

```
bool enoughBalance(User, Item) {
    return User.balance >= Item.price;
}

bool oldEnough(User, Item) {
    return User.age >= Item.min_age;
}

function validate(User, Item, Validators[]) {
  for (int i = 0; i < tests.length; i++) {
    if (!tests[i](obj)) {
      return false;
    }
  }

  return true;
}
```