University of Amsterdam

Individual Written Assessment

Designing `legalease`, a Smart Contracts programming language

March 23, 2021

Student:
René Kok
13671146

Lecturer:
dr. Ana Oprescu

Course:
Programmeertalen

Course code:
5062PROG6Y

1 Introduction

Smart Contracts (Nigel Gopie, 2018) 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

Smart contracts are automated contracts with specific instructions programmed. They help people exchange anything of value in a transparent way while avoiding intermediaries. Smart contracts can be applied in any industry where transactions take place.

Smart contracts are run in a virtual machine because it needs to be deterministic, terminable and isolated. A program can give different outputs on different types of machines. For a smart contract you want the output to always yield the same results.

1.2 Use case description

2 Analysis

2.1 The type system

A type system defines how a programming language groups data into different data types. The major categories of the type system are: static or dynamically typed, manifest or inferred type inference and nominal or structural type system.

The purpose of a type checker is to prevent an incorrect operation from being executed. A static type checker does this by stopping the compiler from generating a program with type errors, a dynamic type checker halts the program as it is about to make a type error (Matthews, 1987). 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.

The purpose of type inference is to retrieve the type of an declared variable. Manifest typing is an explicit identification and requires that the the type of each variable is declared. Inferred typing determines the type of an variable by the type of the assigned initialization expression. 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 semantically different. polymorphism can be implemented through a shared interface such as in the Java programming language.

2.2 State management

A program can be stateful or stateless. A stateful program stores it's state in variables. These variables can be read or mutated during execution of the program. A stateless program does not store it's state in variables. The program's data is passed through chained functions.

A stateless program is according to research more reliable (Merelo and García-Valdez, 2018) because it is not depending on it's state. Because it doesn't rely on it's state, multi-threading is easier to program then a stateful program. Because the synchronization of the states no longer has to be taken into account (Merelo and García-Valdez, 2018). Because of these reasons my language will be stateless.

2.3 Compilation/interpretation strategy

Just-in-time (JIT) compilation during program execution and ahead-of-time (AOT) compilation during software installation are alternate techniques used by managed language virtual machines (VM) to generate optimized native code while simultaneously achieving binary code portability and high execution performance (Wade et al., 2017).

I've chosen for an ahead-of-time 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

In a function call, arguments expressions may be evaluated eagerly to obtain the arguments value before evaluating the function body. Alternatively, one might evaluate arguments lazily, that is, postpone evaluation of the arguments until we have seen that the value of the argument is really needed. If the argument is not needed, the argument is never evaluated. If it is, then the argument is evaluated and cached in case it will be needed again (Sestoft, 2012a).

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

The most common way of parameter evaluation strategy are call by value and call by reference. With call by value, a copy is made of the data structure. When the copy is adjusted within the function, the original remains the same. With call by reference, the original value is adjusted.

Since legalease's state management is stateless, it is not possible to provide a value per reference, as it does not have a state to 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

A higher-order functional language is one in which a function may be used as a value, just like an integer or a boolean. That is, the value of a variable may be a function, and a

function may take a function as argument and may return a function as a result (Sestoft, 2012b).

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.

In addition, anonymous functions are often used in combination with higher-order functions.

3 Discussion

4 Conclusions

References

Matthews, D. C. (1987). Static and dynamic type checking., In *Dbpl*. Citeseer.

Merelo, J. J., & García-Valdez, J.-M. (2018). Going stateless in concurrent evolutionary algorithms, In *Applied computer sciences in engineering*, Cham, Springer International Publishing.

Nigel Gopie, P. (2018). *What are smart contracts on blockchain?* https://www.ibm.com/blogs/blockchain/2018/07/what-are-smart-contracts-on-blockchain/

Sestoft, P. (2012a). Higher-order functions. In *Programming language concepts*. London, Springer London. https://doi.org/10.1007/978-1-4471-4156-3

Sestoft, P. (2012b). Higher-order functions. In *Programming language concepts* (pp. 77–91). London, Springer London. https://doi.org/10.1007/978-1-4471-4156-3_5

Wade, A. W., Kulkarni, P. A., & Jantz, M. R. (2017). Aot vs. jit: Impact of profile data on code quality, In *Proceedings of the 18th acm sigplan/sigbed conference on languages, compilers, and tools for embedded systems*. https://doi.org/https://doi.org/10.1145/3078633.3081037

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;
}
```