JAVADITY'S DOCUMENTATION

VINCENT REBISCOUL

CONTENTS

ABSTRACT

Javadity [1] is a Solidity to Java translator. It takes a so-called smart contract as input and outputs a Java program which should be equivalent to the Solidity program. The goal is then to run tests on this Java program and use tools for Formal Verification in Java. Here, we aimed to produce a Java program compatible with the KeY project. The goal of this documentation is to explain what the translator can do, how to use it and how it is implemented to allow people to modify it.

1 JAVADITY

For now, Javadity takes as input a solidity program and outputs a Java file called NoName.java with the translation in it (this should change soon).

2 IMPLEMENTATION

2.1 Design of Javadity

Javadity works by taking a Solidity program as input, it parses to produce an AST. Then, using a visitor pattern, a Java AST is produced. At last, small corrections are made to the code, using in particular a symbol solver. Therefore, the translation works in three parts:

- Parsing: create an AST of the Solidity program
- Translating the Solidity AST to a Java AST: for each node of the Solidity AST recursively create an equivalent Java node

- Corrections and Symbol Solving: small corrections are made to the AST to make slight changes, yet necessary

The implementation is using two Java libraries, Javaparser [2] and Antlr [3]. Antlr is a parser generator, using the grammar of Solidity [4], it produces a parser which is then used to make a Solidity AST from a Solidity program. Then, a visitor is used to create a Java AST using Javaparser. Javaparser is a Java parser and it can be used to create an AST from scratch (which is the feature we are interested in here). Javaparser also embeds a symbol solver, which is a program that can determine the type of an expression. Indeed, in the translation, unsigned integers are represented with a class Uint256, but it can be problematic, for example when it comes to array accesses. To access to value within an array, the index must be an integer. The symbol solver allows us to find array accesses, check that the index is an integer, if not, the object is converted to an integer.

2.2 Build

How to build Javadity is detailed on the README of the github repository [1]. The program is composed of 5 files:

- `Solidity.g4`, it is the grammar used here for Solidity. With this, we can generate a parser and a visitor with Antlr.
- `TranslateVisitor.java`, it defined the visitor that will generate the Java AST from the Solidity AST
- `Helper.java`, contains a set of convenient static methods to help with the translation
- `SymbolSolver.java`, contains static methods that are designed to apply the small corrections and the that use the symbol solving module
- `App.java`, contains the main function

To understand the following, basic knowledge of Javaparser, Antlr and Solidity is expected

2.3 Parsing

A parser was generated by Antlr using the grammar of Frederico Bond [5] with slight changes. Antlr also generates a basic class for a Visitor [6], which is extended to create the visitor that translates the Solidity AST to a Java AST (implemented in the file TranslateVisitor.java).

2.4 Translating the Solidity AST to a Java AST

2.4.1 *Example*

Antlr gives us the Solidity AST, a visitor will traverse it and output a Java AST. Let us look at an example with the logical AND:

```java
public Node visitAndExpression(SolidityParser.AndExpressionContext ctx) {
  Expression expr1 = (Expression) this.visit(ctx.expression(0));
  Expression expr2 = (Expression) this.visit(ctx.expression(1));

  return new BinaryExpr(expr1, expr2, BinaryExpr.Operator.AND);
}
```

Here, `ctx` has type `SolidityParser.AndExpressionContext`, `ctx` represents the AST with the root node being the logical AND: this node has two children: `ctx.expression(0)` (which is the left-hand side expression) and `ctx.expression(1)` (the right-hand side expression). Recursively, the visitor will translate the ASTs `ctx.expression(0)` and `ctx.expression(1)` into two Java ASTs (`expr1` and `expr2`). Finally, the function returns the Java AST `BinaryExpr(expr1, expr2, BinaryExpr.Operator.AND)`. This was a really simple example, because the Solidity AST and the Java AST are really close when it comes to expressions. However, there are major differences between the two languages. Many cases are more complicated but still simple enough, so they will not be detailed here. We will focus more on the parts of the translation that are more involved, where we try to simulate some specific behaviour of Solidity into Java.

2.4.2 *The basics of the translation*

The goal of the Java translation is to have a Java program with a behaviour close to the Solidity program. In Solidity, there are some types that do not have an equivalent in Java. Thus classes were created to simulate these types. There are mainly three types that are widely used in Solidity, and thus we focused on them to have an accurate translation: `uint256`, `address` and `mapping(address => t)` with `t` an elementary Solidity type. The mapping translation will be detailed later, let us give a quick glance at the `uint256` type and the `address` type.

- The `uint256` type is translated as a class called `Uint256`. The operator have been redefined, and the class `Uint256` has the methods `sum`, `sub`, `mul`, `div`... For example, if `x` and `y` are `uint256` in Solidity, `x + y` will be translated as `x.sum(y)` in Java where `x` and `y` are instances of the class `Uint256`. Note that `x.sum(y)` does not modify `x` but it outputs a new instance of `Uint256` containing the right value.

- The `address` type is translated as a class called `Address`. In Solidity, an address is a constant unsigned integer where you can only use the operators of comparison. But you can also perform actions like a transfer: `anAddress.transfer(50 ether)` that will transfer 50 ethers to the account at address `anAddress`. So in the class `Address`, in Java, there are all the functions that can be used in Solidity on an address, like `transfer`.

2.4.3 *The Visitor*

The visitor which does the translation is called TranslateVisitor. It contains a few fields to help with the translation. First, there is a mapping called `typesMap` from names to typenames. Indeed, by looking at the grammar 1, we learn that a contract is a list of contract parts. A contract part can be a struct definition, an enum definition, a function definition etc... All these objects have an identifier (except for the constructor), so when one visits the contract parts list, one wants to remember what is the type of each identifier. It can be helpful, for example in Solidity, when one defines an enum, it creates a new type. Thus, in the translation, if there is the definition of an enum, then the translator has to remember the name of the enum so that when it comes across this type, the translator will know that the associated value should be translated as an unsigned integer (this is how, in the translation, we represent a variable that has the type of an enum 2).

The Java translation needs to import some files to work properly. Indeed, to represent the specific types of Solidity like `uint256` and `address`. These imports are put in the `imports` array.

There is one last important field, it is the list `structConstructors`. A struct in Solidity is translated as a class in Java. The `structConstructors` list stores the constructors for the struct. This will be detailed later on when we will discuss the translation of the structs.

Figure 1: Extract from the Solidity grammar (Antlr)

```
contractDefinition
: ( 'contract' | 'interface' | 'library' ) identifier
( 'is' inheritanceSpecifier (',' inheritanceSpecifier )∗ )?
'{' contractPart∗ '}' ;

contractPart
: stateVariableDeclaration
| usingForDeclaration
| structDefinition
| constructorDefinition
| modifierDefinition
| functionDefinition
| eventDefinition
| enumDefinition
;
```

Figure 2: Example of the translation of an enum

Listing 1: Enum in Solidity

```
enum Coin { HEADS, TAILS }

function aFunction() public {
  uint x = 5;
  Coin c = HEADS;
  if (c == TAILS)
  x = 4;
}
```

Listing 2: Corresponding translation in Java

```
  public static class Coin {

    public static Uint256Int HEADS = new Uint256Int(0), TAILS = new
          Uint256Int(1);
  }

  private void aFunction() throws Exception {
    Uint256Int x = new Uint256Int(5);
    Uint256Int c = HEADS;
    if (c.eq(TAILS))
    x = new Uint256Int(4);
  }
```

Listing 3: A mapping in Solidity

```
mapping(address => uint256) balances;
```

Listing 4: The corresponding Java translation

```
Uint256Int[] balances = new Uint256Int[MAPPING_SIZE];
```

Figure 3: Translation of a mapping

2.4.4 *The Helper class*

If you look at the `translation_details.md` file, you will see that there are some functions and some variables that have to be present in the translation (for example, the "magic variables" `msg`, `block` and `tx` or the function `require`). The functions in the class Helper give us the AST representing those objects (for example, the function `Helper.getMagicVariables()` outputs the declarations of the magic variables).

The Helper class also gives us easy access to the class representing specific types in Solidity. For example, the `getUintType()` method returns the type `Uint256` in Java that models the type `uint256` in Solidity.

2.4.5 *Struct*

In Solidity, there are structs, which are similar to structs in C. To translate a struct, in Java we need to create a new class containing all the variables. Thus, what the visitor does is when it visits the definition of a struct, it will create a constructor for this struct and store it in the field of the visitor `structConstructor`. Note that what we call a constructor is a function that outputs an object, not a "Java" constructor. Hence, because the constructors are stored, the visitor can add them to the class (during the visit of the node `ContractDefinition` (where the class representing the contract is created).

2.4.6 *Arrays*

The translation of arrays is straightforward, an array in Solidity is translated as an array in Java. Note that we only handle the arrays with fixed-length for now. Java 7 does not support dynamic arrays, so the translation of dynamic arrays is not trivial.

2.4.7 *Mappings*

For now, mappings work only from `uint256` to any elementary type and from `address` to any elementary type. Mappings are implemented as arrays 3, meaning that the number of addresses or unsigned integers that can be mapped are a lot smaller than in Solidity. This make the translation a lot easier but it is restrictive.

One can define the size of a mapping in the class Helper. Then, all mappings will have this determined size, meaning that if you try to access to the value mapped by the address `addr` and that `addr > mappingSize`, you will get a runtime error.

If `balances` is a mapping from `address` to `uint256` and `addr` is an `address`, in Solidity to access to the value with the key `addr`, you would do `balances[addr]`, it is translated in Java as `balances[addr.ID]` where the field `ID` is the actual number representing the address (in Java `addr` is an instance of the class `Address`).

2.4.8 *Modifiers*

The modifiers are specific to Solidity and are not supported by Javadity. There is a mechanism in Javadity that tries to mimic the behaviour of modifiers but it should

Listing 5: An array access in Solidity

```
anArray[4]
```

Listing 6: The Java translation before symbol solving

```
anArray[new Uint256Int(4)]
```

Listing 7: The final Java translation

```
anArray[new Uint256Int(4).asInt()]
```

Figure 4: Translation of an array access

not be used. Nevertheless, I will detail how it works in case someone wants to use it anyway or who knows how to improve it.

When the visitor meets the definition of a modifier `modifier(type x)...` it stores the Solidity AST in a mapping. Then, when it reaches a function definition using the modifier `function name(type a, type b)modifier(a)...`, first it translates the body of the function into a Java AST. Then it traverses the Solidity AST of the modifier with a special modifier that will 3 things

- it will translate the Solidity AST into a Java AST like basic visitor
- it will replace all occurences of variable `x` (the parameter of the modifier) by `a` (the parameter given to the modifier in the function definition).
- it will replace the placeholder _ of Solidity by the body of the function that was translated before

It may do this several times, depending on the number of modifiers. Obviously, this is not a proper support of modifiers. For example, in the variable `a` is defined in `modifier`, then renaming all occurences of `x` by `a` will lead to conflicts.

3 SYMBOL SOLVING AND CORRECTIONS

Once the visitor has produced a new Java AST, some changes need to be done before having a "usable" program. Indeed, within an AST, there are no notions of types, this can lead to errors in the translation. This is why we need a symbol solver (here we use the symbol solver of the library Javaparser). A symbol solver will allow us to determine the type of an expression and will help us making slight changes to the current Java AST.

3.1 The array accesses

For example, we need to fix the array accesses. Look at the figure 4, one can see that the translation before symbol solving is not correct. Indeed, all integer literals are translated as `Uint256`, but the index of an array has to be a integer, thus the array access `anArray[new Uint256Int(4)]` does not make sense here. The symbol solver allows us to find all array accesses where the index is a `Uint256` or an `Address` (remember that mappings from `Address` to any type are represented as arrays) and to access to the integer stored in this class. This is what the methods `correctArrayAccess` does in `SymbolSolver.java`

x

Listing 8: Solidity

```
uint256 x;
```

Listing 9: Java

```
Uint256 x = new Uint256(); // an initializer has been added
```

Figure 5: Add initializer

3.2 Initialization of variables

In Solidity, even when the user does not explicitly initialize a variable, it is given a default value (whose byte representation is all zeros). Thus in the Java translation, all user-defined variables have to be initialized, except for the "magic variables". Thus, we traverse the AST and when we find a declaration of a variable that is not initialized (and if this variable is not a magic variable), we add an initializer to it (if it is not a primitive type, we just create a object using the constructor without parameters), see fig 5. Note that we do not use the symbol solver here because we do not care about the type of the variable. This is implemented in the method `setDefaultValue` in `SymbolSolver.java`

4 WHAT JAVADITY CANNOT DO

Javadity is a simple translator and it cannot do a lot of things. The use of `nameOfAFunction.call()` is unsupported (same thing for delegatecall). We only aim to capture the high level features of Solidity.

REFERENCES

[1] https://github.com/rebiscov/Javadity/

[2] http://javaparser.org/

[3] http://www.antlr.org/

[4] https://solidity.readthedocs.io/en/v0.4.24/miscellaneous.html#language-grammar/

[5] https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

[6] https://en.wikipedia.org/wiki/Visitor_pattern/