Solidity in Spoofax

Language Engineering Project 2018

Taico Aerts
Delft University of Technology
T.V.Aerts@student.tudelft.nl

Abstract

This paper provides an overview of a Solidity compiler written in Spoofax. It explains the main challenges and problems that were encountered in its development and how these were addressed. The compiler support nearly all of the syntax and has extensive type checking, simple compilation and a few optimizations.

Keywords Spoofax, Solidity, Ethereum, Compiler

ACM Reference Format:
Taico Aerts. 2018. Solidity in Spoofax: Language Engineering Project 2018. In *Proceedings of Language Engineering Project (LEP'18).* Delft University of Technology, Programming Languages Group, 8 pages.

1 Introduction

Solidity is a programming language for Ethereum smart contracts. With the increasing popularity of Ethereum, more and more programmers and hackers join in to try and make money with these smart contracts. As such, demand for more language functionalities, better security and more debugging functionality has increased. Solidity is a programming language that is still in its infancy, but is already widely used and rapidly changing. This report describes how I have built a compiler for Solidity. The compiler is written in Spoofax, a language designer's workbench [3] which aims to make the compiler pipeline simpler, more readable and more flexible for changes.

This compiler was made for the Language Engineering Project 2018, a course at Delft University of Technology. The aim of the course is to make a compiler that covers as much of a specific language as possible, while focusing on one or more specific aspects of the compilation. Initially, my objectives were the compilation itself and possible optimizations. However, over the course of the project, the focus shifted more towards type checking and name resolution.

The created compiler has a near complete syntax definition, an extensive type checking and name resolution algorithm, and can compile simple programs to Ethereum Virtual Machine bytecode. All the source code is available online. [1]

First, Section 2 explains the objectives and achievements of the project. Section 3 then shows interesting aspects of the Solidity programming language. Then, Section 5 explains the syntax definition with the main challenges and shortcomings. Section 6 follows by explaining the different transformations that are applied in a desugaring phase, followed by Section 7 which goes in depth about the type checking and name resolution. Then, the details of the compilation and its difficulties are explained in Section 8. Finally, Section 9 ends with a reflection and recommendations for future work.

2 Objectives and Achievements

Initially, the objectives were the compilation itself and possible optimizations. Since every instruction that is executed costs money, good optimizations are very important. For example, some design patterns that are commonly used in other programming languages result in very high execution costs in Solidity. This problem can be partially solved by creating a smarter compiler, which makes the language more accessible to newcomers.

However, over the course of the project, the focus shifted more towards type checking and name resolution. Solidity has a very extensive type system. A good type checker for Solidity is necessary as the compiler needs a lot of information in order to compile sequences to the correct forms. This is especially true when aiming for optimizations. As such, it was necessary to focus more on type checking first, before going too deep into compilation and optimization.

Another factor is that the documentation of the compiler itself and of the EVM are quite lacking. The best way to learn how certain items are compiled is to look at the source code of the compiler and at the bytecode that is generated by the compiler. While these combined give a lot of insight into how things work, it remains difficult to fully understand certain aspects of the compilation.

The compiler that I have created has a syntax definition that almost completely covers all the features of solidity. Only tuples and inline assembly are not fully supported. The type checking is also very extensive, with only certain high level features such as `using X for` unsupported. The actual compilation is rather basic, and supports only simple programs. In terms of optimization, the compiler performs constant folding and dead code elimination.

3 Ethereum and Solidity

This section provides a short overview of the basics of smart contracts and of the Solidity programming language.

[1] https://github.com/MetaborgCube/metaborg-solidity/

LEP'18, June 2018, Delft University of Technology
2018.

3.1 Smart contracts

Ethereum Smart contracts are programs that are deployed to the blockchain. A smart contract can be invoked by sending money to it in the form of a transaction. Every instruction that has to be executed costs a certain amount of money, which is all payed by the caller. In the end, all remaining money is transferred back to the caller with optionally some return values.

An example of a scenario in which someone can use smart contracts is for ticket sales of an event. Buying a ticket involves making a transaction of enough money to the contract. A small amount of this money is used for the execution of the program, which registers the user on the blockchain and returns them a ticket. The money for the ticket itself is transferred to the account of the organizer.

No infrastructure is required to set up this ticket sale. Everything happens automatically and securely, and the fact that your ticket is valid is stored forever on the blockchain.

3.2 Solidity

Solidity is the language that is used to program these smart contracts. Solidity looks a bit like a mix of Javascript, Python and C++. It is a high level language with static typing, inheritance and complex user-defined types. Listing 1 shows an example of greeter contract. When creating an instance, you specify a greeting. You can then call the greet method, which will send that greeting back to you. The instance can also be killed with the kill function to return all its money to its owner.

```
1  contract Greeter {
2    address owner;
3    string greeting;
4
5    constructor(string _greeting) public {
6      owner = msg.sender;
7      greeting = _greeting;
8    }
9
10   function greet() constant returns (string) {
11     return greeting;
12   }
13
14   // Function to recover the funds on the contract
15   function kill() {
16     if (msg.sender == owner) selfdestruct(owner);
17   }
18 }
```

Listing 1. Example of a smart contract

4 Architecture

The compiler is split into six different parts: a syntax definition, desugaring, type checking and name resolution, compilation and optimization (pre-compilation and post-compilation).

5 Syntax Definition

The syntax definition is written in SDF3 and resides in the syntax folder. It is the most complete component of the language definition. Almost all of the syntax that is valid in Solidity 4.0.23 is supported by the provided syntax definition. The syntax definition follows the rules as stated in the solidity documentation. [2]

At the time of writing, the provided documentation is not consistent with the behavior of the solidity compiler. In the case where the compiler disagrees with the documentation, my syntax definition follows the compiler.

In the following subsections, I will highlight interesting aspects as well as the main challenges of the syntax definition. I also explain why certain language features are not supported.

5.1 Unsupported syntax

While the syntax definition is very comprehensive, it does not cover all the language features that Solidity has to offer. The following section explains which features are not supported and the reason why this support was left out.

5.1.1 Inline Assembly

Solidity gives programmers the option to write low level code in a similar fashion to how C allows inline assembly. There is a special language for this inline assembly. Due to time constraints, I decided to leave out support for this inline assembly. However, in order to keep language coverage as large as possible, I decided to support assembly blocks. The content of an assembly block is not checked and can contain anything. The only requirement is that all curly brackets are matched, i.e. every { has a matching }. A note is added to inform the user that the inline assembly is ignored.

The elements in an assembly block are split into strings, split by spaces and newlines. For example, the code shown in Listing 2 is parsed as `Assembly(ASMBlock(["n", ":=", "byte(0x0)"]))`.

```
1 assembly {
2   n := byte(0x0)
3 }
```

Listing 2. Inline assembly

5.2 Challenges

There were also multiple challenges regarding the syntax definition. The following subsections explain these challenges and show how they have been solved.

5.2.1 Tuples

Solidity has support for tuples, which allows functions to return multiple values of potentially different types, as long as this information is available at compile time. An example of the tuple syntax is shown in Listing 3. Lines 1-3 show how variables can be declared as a tuple. Lines 4-6 show how

variables can be assigned from tuples. Line 7 shows an example of a function signature for returning a tuple. Whenever the name of a variable is omitted, the corresponding tuple element is dropped (lines 2, 3 and 5).

```
1 var (a, b) = (10, 20);
2 var (a, ) = f();
3 var (,) = f();
4 (a, b) = f();
5 (,) = f();
6 (a) = 10;
7 function f() returns (int, int) {}
```

Listing 3. Tuples in solidity.

Initially, almost all of the syntax in Listing 3 was supported. The only unsupported feature was the construction of an actual tuple, e.g. `(10, 20)`. The reason that this was not supported is because it conflicted with the so-called bracket rule for expressions. The bracket rule allows any expression to be wrapped in a pair of parentheses. It is used for disambiguation, for example in `(10 + 20) * 3`. Even when giving tuples priority over the bracket rule, the parser still attempts to interpret the fragment as `10, 20`, without the parentheses. It also interferes with other rules and causes a lot of syntax parsing problems. However, the solution for this problem eventually turned out to be quite simple. By defining tuples as having at least two expressions in them, it no longer conflicts with the bracket rule. Literal tuples with only a single expression are parsed as normal non-tuple expressions, and as such are still parsed correctly.

5.2.2 Modifiers

Modifiers were an interesting problem. In Solidity, as well as in many other languages, modifier positions and ordering are quite flexible. An example is shown in Listing 4.

```
1 function f() public constant {}
2 function f() constant public {}
3 function f() private {}
```

Listing 4. An example of different modifier positions and ordering.

While flexible in position and ordering, there are still rules that limit which modifiers can be used where. For example, defining a function to be both `public` as `private` or using the same modifier multiple times is not allowed.

In a syntax definition, it is difficult to express such a constraint. At each position, any modifier is technically allowed, as long as there are not two modifiers of the same category.

I considered multiple different options for enforcing these constraints. The first option I considered was to enforce a set ordering. For example, first visibility modifiers, then mutability modifiers. This does enforce that there can be only one modifier from each category, but it does make the compiler stricter than the official language specification. The second option I considered was to specify each possible combination of modifiers, e.g. `public constant`, `constant public`, `constant`, `public` or the empty sequence. While this would enforce the constraints, it makes the syntax definition unnecessarily complex and large as the number of possible modifiers grows. Adding support for another modifier would mean an exponential growth in the amount of definitions. Finally, it is also possible not to enforce these constraints in the syntax definition, but rather in the desugaring or type checking.

I decided to do the latter and enforce the constraints during desugaring. By utilizing this method, the definition retains all the flexibility as defined by the language specification, but still enforces the logical constraints as a compiler should. The desugarer groups the modifiers by their categories. If there are multiple modifiers in one group, it replaces all modifiers with a single `InvalidMod` with an appropriate error message. This error message is then reported to the user by the type checker. The reason for this is that the desugarer only applies AST to AST transformations, and does not have the capability to add error messages in a flexible way. The type checker is able to do so easily, so creating an AST element with a warning seemed like a good solution.

5.2.3 Calls and Casts

In solidity, calls and casts are very similar syntax wise, and is not always distinguishable from each other by syntax alone. Listing 5 shows the syntax for a cast (line 1) and the syntax for a call to a function with a single argument (line 2). For primitive types like int8, it is not that difficult to disambiguate these cases, but for user defined types, this is not possible by syntax alone.

```
1 int8(10)
2 myFunction(10)
```

Listing 5. A cast (line 1) and a function call (line 2)

In fact, it is possible to define a type and a function in such a way that the function shadows casts. An example of this is shown in Listing 6. It is impossible to distinguish between a typecast to the contract type *a* and a call to the function *a* without using name resolution.

```
1 contract a {
2   function a(int x) {
3     return a(x);
4   }
5 }
```

Listing 6. A function shadowing casts.

In fact, there are cases in which it is ambiguous regardless of name resolution, as shown in Listing 7.

There is no way to determine if this is a cast from *b* to *a* (which is valid) or a call to the function *a* with argument *x* of type *b* (which is also valid). The solidity language specification does not provide any information on how to handle

```
contract a {}
contract b is a {
  function a(b x) {
    return a(x);
  }
}
```

Listing 7. Example of an ambiguous expression.

this problem. However, after investigating this case with the official solidity compiler, it seems that functions actually shadow types, so in the above example `a(x)` would be a function call to the function *a* and not a cast.

For the syntax definition, I decided to give preference to casts, but to avoid any user defined types. So `int8(10)` will become `Cast(IntType(8), 10)`, but `myType(10)` will become `Call("myType", 10)`. This behaviour reflects the shadowing rules properly in most cases.

6 Desugaring

In the desugaring phase, multiple transformations are applied to the AST. These are to simplify the AST, to add information required for type checking or compilation and/or to optimize the program. This section only goes over the more interesting transformations that are applied. A full overview of all transformations can be found in the documentation on GitHub.

6.1 Transformations

This section explains the more interesting transformations that are applied in the desugarer.

6.1.1 Return statements

In order to support return statements at arbitrary positions, the type of the function to which a return statement belongs must be available for the type checking. Due to limitations in NaBL2, the language used to write the type checker, this is non-trivial during type checking itself. Instead, the desugarer adds the name of the enclosing function to return statements. During type checking, this name can then be resolved to the closest function definition and its signature, which allows checking the return expression with the function return type.

6.1.2 Constant folding

Literal numbers are constant folded to allow proper typechecking. This is necessary since an expression like `1.2 * 5` would be regarded as the integer number 6 by the solidity compiler, even though there are no floating point numbers in the language (partial support was added recently). The desugarer also removes units, e.g. `2 minutes` is converted into `120`. Finally, the desugarer calculates the number of bits that would be required to fit the number (rounded up to multiples of 8) into a signed integer and the number of bits required to fit it into an unsigned integer. This information is used by the type checking to determine the smallest type (int8 vs uint8 vs int16) of integer literals.

Literal strings are treated in a similar way, and their length is added to the AST.

6.1.3 Big numbers

Solidity supports 256 bit integer numbers (and uses these by default). Since these numbers are much larger than those supported by most other programming languages (usually 64 bit integers are the largest), I've had to implement specialized stratego strategies for working with these large integers. These strategies use Java's BigInteger and BigDecimal to handle numbers of arbitrary size.

6.1.4 Literals

In order to determine the smallest amount of bytes required for strings, the typechecker needs to know the length of a string. This information is added in the desugarer for this reason.

6.1.5 Builtin Functions

Solidity has a few built in functions. These functions are global and can be called at any point, but the names are not keywords, so it is possible for a user to specify a function of the same name.

There are 2 functions in particular that are of interest, `require` and `revert`. The first function is used as an assertion with an optional error message if that assertion fails. If the assertion fails, contract execution is stopped in a special way, returning left over money to the caller along with the error message. This feature was added after popular request, since the normal behaviour in case of an error is to consume all money and halt execution immediately, providing no feedback to the caller. Revert is the unconditional version of require, returning money to the caller with an optional message.

The problem with these functions is that they are overloaded; the message is optional. Both forms of these functions are commonly used in contracts, so supporting both is preferable. However, the language used for typechecking, NaBL2, does not offer a good way to implement support for overloaded functions. As such, I decided to work around this restriction with the desugarer.

First, the desugarer checks if there is a function declaration with the name *require* or *revert*. If there is, it does nothing. Otherwise, it changes all function calls to these functions by prefixing the name with a "!" and suffixing the name with the amount of arguments being passed. For example, a call `require(x, ''Error'')` is effectively changed to a call to `!require2(x, ''Error'')`. Now, calls the different overloaded functions can be disambiguated by the typechecker. The exclamation mark is used to prevent conflicts with any user defined function, as it is not a valid character for a function name.

6.2 Simplicity

This section describes the transformations that are applied to simplify the AST. Since most of these are not as interesting, they will not be discussed in depth.

6.2.1 Functions and Constructors

Solidity has a few different types of functions in the syntax, but in terms of how they are used in the type checking, they are pretty much the same. As such, I decided to desugar functions into a generic format: `Function(name, parameters, modifiers, returnParameters, statement)`. Void functions get assigned returnParameters of `VoidType()` and the so-called fallback function is considered as a function with the name `*fallback` (`*` is not a valid character in function names, so no conflicts can occur).

For constructors, the name of the enclosing contract is added to the constructor element. This way, the constructor can be seen as a function which returns an instance of the contract type, which simplifies type checking.

6.2.2 Tuples

Any tuple declaration or assignment that contains only one or no items is converted into a normal declaration or assignment, or just the expression respectively. In short, any unnecessary use of tuples is removed.

6.2.3 For loops

For statements are converted into while statements where this is trivially possible, e.g. `for (;;) ...` and `for (; i < 10; ) {...}`. For statements that specify the third parameter are not transformed, as continue statements make the conversion to a while loop non-trivial.

6.2.4 LValue Assignments

LValue assignments are converted to a simpler form, e.g. `a += 10` is converted into `a = a + 10`. In contrast to some languages, these expressions are guaranteed to be equivalent to each other in Solidity (Solidity does not have threads). Removing the LValue assignments makes the type checking and compilation simpler, and the transformation itself is trivial.

6.3 Optimizations

This section describes the transformations applied by the desugarerer that are aimed at optimization.

6.3.1 Dead code

The desugarer is able to perform simple dead code analysis. Any statement following a return, break, continue or throw statement is wrapped in a `DeadCode` node. The user is presented with a warning that the code is unreachable, and the compiler can ignore the DeadCode statements entirely.

7 Type Checking and Name Resolution

The type checking is after the syntax definition the most extensive component of the compiler. It supports all possible types and almost all of the syntax. The only unsupported functionality are imports and the `using X for ...` syntax.

7.1 Interesting

This section will explain the most interesting aspects of the type checking and name resolution.

7.1.1 Number of types

Solidity has a large number of types in the form of Int(8 * N) where N is a number between 1 and 32. However, NaBl2 does not support defining relations like `IntT(N) <sub! IntT(N+8)`. Instead, all the possible subtyping relations had to be defined. There are approximately 800 lines of NaBL2 code simply for defining each of the subtyping relations.

This also caused problems, as NaBL2 turns out to have a time limit on parsing its own files. If the timeout is hit, the parsing crashes and the file is not compiled. I was able to combat this problem by splitting things up into more files, and by reducing the number of relations by using functions in combination.

I have defined a function *simplifyType*, which transforms a type into its basic form, e.g. `IntT(8)` to `IntTS()`. Operations that can be applied on integers are the same for all sizes of integers, as long as the types are both subtypes. So, in these cases it suffices to use the simplified type for the relation. With this technique, the number of required relations is reduced significantly.

7.1.2 Functions and fields on simple types

Solidity has different functions and fields that can be called and accessed on simple types. For example, length on arrays, push and pop on dynamic sized arrays, `.call(...)` on addresses. I wanted to implement this in a sort of generic way, such that I can still easily distinguish between an address and a user-defined type. After trying different things, I found that it was possible to put scopes in these types. With a getScope function, it was then trivial to get the scope from a type, or to fail since the function could not be executed on the specifc type (unscoped). The downside of this method is that it is means that in order to construct an address type, the scope must first be resolved.

I have used a special namespace for these built-in types, defined in the global scope. Then, they can be resolved from anywhere, giving the type definition. This is still a bit verbose, but I did not find a better way. For address, I was able to create a named rule that will return the address type given any AST term. However, for arrays, there is extra information in the type such as the length (fixed size arrays) and the type of elements in the array. Since this information is not present in the AST at all locations, it cannot be passed to a

rule. As such, these still need to be resolved in every rule where they are used.

7.1.3 Literals

In order to correctly determine the type of literals, additional information is added by the desugarer (see Section 6.1.4). Numeric literals get the type `NrT(i, u)`, where `i` is the minimum number of bits required to represent the number as a signed integer and `u` is the minimum number of bits required to represent the number as an unsigned integer. Through the subtyping relations, the exact type is then determined based on where and how the literal is used. An example is shown in Listing 8.

```
function f() {
  int16 x = 1;    //NrT(8, 8) -> IntT(16)
  uint8 y = 200; //NrT(16, 8) -> UIntT(8)
  int8 z = 200;   //Error
}
```

Listing 8. Example of type assignment to literal numbers

String literals are treated in a similar fashion. String literals can be assigned to fixed size byte arrays if the byte array is at least the length of the string. This behavior is implemented by adding the length of the string to the AST during desugaring. The length is then used in the type. For example, the string "hello" is assigned the type `StringLitT(5)`. The possible target types are again determined through subtyping relations.

8 Compilation

This section describes the compilation of Solidity to bytecode for the Ethereum Virtual Machine

8.1 The Ethereum Virtual Machine

Solidity code is compiled to bytecode for the Ethereum Virtual Machine (EVM) [1, 4]. The EVM is a pure stack-based machine, which means that there are no local variables or registers for temporary assignments. This in turn makes compilation more difficult as local variables must be stored on the stack.

In order to make the compilation easier, I have created a simple language definition for describing the bytecode, called EBC. It has a rather simple syntax for the different instructions. An example code fragment for EBC is shown in Listing 9.

```
PUSH2 0x20
DUP2
PUSH1 0x1
ADD
STOP
```

Listing 9. Example of EBC code

The corresponding AST is very similar, as shown in Listing 10.

```
Code([
  PUSH("2", "20"),
  DUP("2"),
  PUSH("1", "1"),
  ADD(),
  STOP()
])
```

Listing 10. Example of EBC code

8.2 Code generation

The code generation is performed in two steps. First, the desugared Solidity AST is transformed into an EVM bytecode AST via a set of transformation rules. To get the actual compiled code, the EVM bytecode AST is converted into a hexadecimal string by translating each instruction into its corresponding opcode. This hexadecimal string can then be executed by an Ethereum VM.

The following sections explain the main challenges of the code generation, and how these challenges were addressed.

8.2.1 Local variables

Since local variables need to be stored on the stack, the position of each variable on the stack must be remembered. The main problem is when there are too many local variables. The `SWAP` and `DUP` instructions are only able to access down to the 16th element on the stack. Any element below that is not reachable, until items are popped. With simple, fixed local variable assignments, this means that certain syntactically and semantically correct programs cannot be compiled.

An algorithm for determining the best position for each variable would be very complex. In fact, the official solidity compiler does not support functions with more than 16 local variables. Since this limit includes the function parameters as well, this can be frustrating.

For my compiler, I decided to simply generate instructions such as `SWAP(17)`. While these are invalid, it is possible that the post-compilation optimization removes certain items from the stack, making these instructions possible. If not, then the final compilation step, the conversion to the hexadecimal opcodes, will be the one that fails. It also makes an algorithm for determining better stack positions easier, as it can use these invalid instructions to determine what items need to be moved, and how far they need to be moved.

Another problem with the stack is that we must keep track of the state of the stack during the compilation. In order to generate the correct sequence of bytecode to refer to a local variable, we need to know exactly how many other things are on the stack.

I have written a stack implementation in Java, which is used from Stratego. Instead of returning a new immutable stack from each call, the stack is instead stateful, making it much easier to use. At the start of a function, all local variables are assigned a location on the stack. A sequence

of bytecode is generated to push either 0 or the variables initial value, if it is constant, to the stack. Then, the rest of the statements are compiled. Each time an instruction would push something to the stack, it is also pushed to the compiler stack. The same happens when items are popped from the stack. With this information, the state of the stack is known at all points, as well as the position of all local variables.

Compiling a reference to a variable then becomes `DUP(i)`, where *i* is the index of the variable from the top of the stack. Compiling return statements is more complex. For example, to compile `return a`, we need to remove everything from the stack except for *a*. The algorithm that does so works as follows. First, elements are popped until *a* is at the top of the stack. Then the element is swapped down the stack as far as possible. This process is repeated until only *a* is left on the stack. The corresponding set of instructions is generated in exactly the same way, using `POP` and `SWAP`.

8.2.2 Continue and break

In for loops and while loops, it is possible to use continue and break statements. At compilation, these statements correspond to jumping to just before the condition is checked and after the entire loop respectively. As such, the compiler needs to keep track of what these locations are at the current position. The compiler keeps

8.2.3 Function calls

Each public function of a contract is callable. However, since the caller does not have the source code of the contract, they do not know where in the contract that specific function starts. Besides that, it would be problematic security-wise to allow the caller to specify where they want to start execution from. Instead, the contract must specify a so called ABI specification. This ABI specification mentions the signatures of the available functions. Calls are actually made by hashing the function signature with sha3, and passing that as the first argument to the contract. The contract does a lookup in a switch-case like matter, and if the hash matches with any hash as generated by the compiler, then the contract will jump to that function.

I have created an ABI specification, but the ABI is not yet generated from the code. The EBC code cannot be executed until the ABI support is added. It is however possible to have the official solidity compiler generate the ABI. It is then possible to test the generated bytecode.

8.3 Optimization

Optimization is performed at two stages. First, optimization is applied to the Solidity AST before the transformation into an EVM bytecode AST. Then, the transformation from the Solidity AST to the EVM bytecode AST happens as usual. The second phase of optimization happens on the EVM bytecode AST.

8.3.1 Solidity optimizations

There are multiple optimizations which are applied to the solidity AST. These are as follows.

8.3.2 Unreachable code

Unreachable code is removed as much as possible. For example, one branch of if-else statements where the condition is constant can be removed. For example, the statement in Listing 11 is converted into `x = 5;`. The same technique is applied to for loops, while loops and do-while loops, if the condition is determined to be always 'false'.

```
1 if (true) {
2   x = 5;
3 } else {
4   x = 10;
5 }
```

Listing 11. Simple if statement that can be optimized.

However, variable declarations in the unreachable code are still kept. An example is shown in Listing 12 and Listing 13. This is because solidity has Javascript-like scoping rules. In contrast to programming languages like C and Java, variables are scoped per function, and not per block. But more importantly, in Solidity, referring to a variable that is not declared is an error, but referring to a variable before it is declared is valid. The variable initially holds the default value corresponding to its type. As such, all the variable declarations need to be kept, even those in unreachable code.

```
1 if (false) {
2   int y;
3   int z = 20;
4 } else {
5   x = 10;
6 }
```

Listing 12. Simple if statement that can be optimized.

```
1 int y;
2 int z;
3 x = 10;
```

Listing 13. Optimized version of Listing 12

8.3.3 EBC Optimizations

Post compilation optimizations were planned, but due to time constraints these could not be added.

9 Reflection and Future work

I learned a lot about how it is possible to write things as transformation functions for NaBL, which allows the very extensive type checking. I also learned more about the limitations of NaBL and why these are there. For another project, I would try to write the type system in Statix instead. I hope Statix has some form of dependent typing, as this would solve many of the shortcomings that NaBL has.

I also learned that making a compiler for a language where you don't know exactly how it should work is very difficult. Especially since the EVM has no local variables, there are a lot of limitations that you run into quickly when writing simple transformations.

9.1 Future work

There are various possibilities for future work. Especially the code generation needs to be enhanced to make the compiler as a whole usable. In terms of optimizations, there are a few different ideas which might be good optimization techniques. If these are implemented, the compiler would provide some benefits over the official compiler.

9.1.1 Post-compilation optimization

Some optimizations can be applied post-compilation. If an option is added to convert compiled code back into an EBC AST, then these optimizations could even be applied on sequences compiled with other compilers. It would make the optimizations usable even if some functionality is not yet implemented.

9.1.2 Removal of unused variables

Unused variables in functions can be removed. However, care should be taken if the variable is assigned the result of a function call. If the function call has side effects, it still needs to be executed. The optimizer could make use of the `pure` and `view` function modifiers that Solidity offers. These modifiers indicate that the function has no side effects, and thus that even the function call can be removed.

9.1.3 Cost optimization

As the user pays for each instruction that is executed, it makes sense to optimize for the cost. There are several techniques that could be employed, such as converting memory variables to local variables where possible, removing superfluous push/pop instructions and optimizing variable positions on the stack.

9.2 Constant propagation

Constant propagation could be implemented as an optimization technique. This means that variables that have constant values are eliminated as much as possible.

Acknowledgments

The author would like to thank Jesse Busman for providing his knowledge about the workings of the Ethereum Virtual Machine and the compilation process.

References

[1] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. *Ethereum Project White Paper* (2014).

[2] Ethereum. 2018. Solidity - Solidity 0.4.25 documentation. Retrieved June 27, 2018 from https://solidity.readthedocs.io/en/develop/index.html

[3] MetaBorg. 2018. The Spoofax Language Workbench - Spoofax 2.4.0 Documentation. Retrieved June 27, 2018 from http://www.metaborg.org/en/latest/

[4] Gavin Wood. 2018. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2018).