elle Documentation

Mario M. Alvarez

Jan 24, 2020

Contents:

CHAPTER 1

Overview

Elle is a compiler focused on generating code for the Ethereum Virutal Machine (EVM). What sets is apart from other Ethereum compiler technologies is that it is *foundationally verified*: that is, it is implemented inside of Isabelle, a *proof assistant* that enables programmers to state and prove *mathematical theorems* about their code. In the case of Elle, a theorem exists stating that the behavior of the code output by the Elle compiler matches the behavior of its input (see Correctness).

Assuming we trust the model of Elle's source language (an LLL/Yul/WASM-like structured programming layer on top of EVM) and the semantics of EVM (drawn from the Eth-Isabelle project, we can have complete confidence that Elle generates EVM programs that match the programmer's intent: that is, that behave the same way that input programs are supposed to behave.

In the rest of this documentation, we'll cover *:ref:how to install Elle <installation>*, *how to use Elle's FourL frontend <usage_>* as an end user to compile smart contracts written in the LLL language into EVM bytecode. Next, we'll dive into the details of Elle's source-level representation, covering its *syntax <syntax_>* and its *formal semantics <semantics_>*. Next, we'll talk about *the internals of the implementation <implementation_>* of the Elle compiler (along with the FourL frontend) as well as its *correctness proof <correctness_>*.

Elle is intended to be supplanted by Gazelle, a . As such, Elle itself is unlikely to see significant changes at this point. Nonetheless, Elle as it exists is a useful system: it can be used to compile real-world LLL smart contracts; namely Dan Ellison's Echo smart contract, Ben Edgington's LLL-ERC20 token contract, and the LLL implementation of the ENS registry. For these examples, see the tests directory.

CHAPTER 2

Installation

Contents

Elle's installation is quite straightforward if all you want to do is use the FourL frontend as described in *Usage of the LLL Frontend*, and somewhat more involved if you want to be able to generate, build, check, or modify the implementation and proofs of Elle.

2.1 Getting Elle

In the rest of this file, let > stand for a bash-compatible shell prompt (that is, lines starting with > are to be understood as input)

Clone Elle from the Github repo as follows:

```
> git clone https://github.com/mmalvarez/eth-isabelle
```

Later in this file, we assume that Elle is checked out into a directory called `eth-isabelle`. (This repository has this name because it is a fork of *Yoichi Hirai's eth-isabelle <https://github.com/pirapira/eth-isabelle>* project specifying EVM in the Isabelle proof assistant.)

Since Elle is no longer under active development, the `master` branch should be stable enough to build reliably. For a version of Elle that is more guaranteed to build successfully, try the `ITP2019` branch, which contains an artifact submission for the 2019 edition of the Interactive Theorem Proving conference:

```
> cd eth-isabelle
> git checkout ITP2019
```

2.2 Installation as an End-User of Elle

The following dependencies are required to build `llllc`, the FourL command-line interface to the FourL compiler frontend that makes use of Elle's verified compiler.

- Ocaml (tested with v4.05.0)
- OcamlBuild
- OCaml Zarith

On an Ubuntu-like system, these can be installed as follows:

```
> apt-get install ocaml ocamlbuild libzarith-ocaml libzarith-ocaml-dev
```

Once these dependencies are installed, navigate to `eth-isabelle/elle/generated` and run `make`. This should succeed and generate a file called `llllc`. When run on an lll file, it will print (to standard output) a hexadecimal representation of the bytecode produced by the compiler for that smart contract (similar to Solidity LLL's command-line tool, but with fewer options and error messages). Files to run `llllc` on can be found in the `eth-isabelle/elle/tests` directory.

For more details on using `llllc`, see *Basic Usage*.

2.3 Installation for Modifying and Examining Elle

The Elle git repository includes the file *eth-isabelle/elle/generated/FourL.ml <https://github.com/mmalvarez/eth-isabelle/blob/master/elle/generated/FourL.ml>*. This file is generated from the formal Isabelle model contained in the rest of the Elle repository, and is all that is needed to build a working executable version of Elle/FourL as described in *end-user-installation*.

In order to work with the formal model directly, Isabelle itself is needed as a dependency. Elle requires Isabelle 2018, which can be downloaded *here<https://isabelle.in.tum.de/website-Isabelle2018/index.html>* (binaries for Linux, MacOS, and Windows are provided).

Once Isabelle is installed, the user will need to set up Lem, a framework used to generate the some of the Isabelle specifications used by Elle. In order to do this, first run the following, to update the Lem submodule contained in Elle's git repository:

```
> cd eth-isabelle
> git submodule init
# output snipped
> git submodule update lem-installation
# output snipped
```

we have tested with the version of Lem having the Git hash of `0927743c1bd31d7bba20a54260ba4c4dd3ce49e9`. Newer versions should also work. Older versions may not support generating code compatible with Isabelle2018.

In order to build Lem, run the following:

```
> cd lem-installation
> make
# output snipped
```

If this succeeds, it will generate an executable called `lem`. Add it to your path, and ensure the it will look for its libraries in the correct place, by running the following:

```
> `export PATH=$PWD/lem-installation/bin:$PATH`
> `export LEMLIB=$PWD/lem-installation/library`
```

Finally, navigate back to the root of the repository (`eth-isabelle`), and run the following to build the `.thy` files that Elle depends on of from their Lem sources:

```
> make lem-thy
```

2.3.1 Examining Elle Sources

Isabelle allows `.thy` files representing formal models and proofs to be grouped together into *sessions*. Sessions make it easier to automate the process of compiling Isabelle developments, as well as allowing for caching the results of compilation and proof-checking so that work does not need to be repeated each time Isabelle is re-opened. Elle contains a session called `ElleCorrect`, which packs together all the files containing Elle's correctness proofs into a single session file.

However, in order to be able to step through the proofs contained in the `ElleCorrect` session, it's better not to run the `ElleCorrect` session, since. Therefore, to examine Elle's proofs, run Isabelle-Jedit, with the `HOL` session

```
isabelle jedit -d ./lem -l HOL
```

For some proofs (particularly the more complex ones in `elle/ElleCorrect`) you will need to increase the editor's limit on the number of allowed tracing messages (or else the proofs will pause and appear to get stuck). To do this, navigate through the Isabelle/JEdit menus as follows

```
Plugins > Plugin Options > Isabelle > General > Editor Tracing Messages
```

Increase this value to 30000.

The most interesting proofs are in `eth-isabelle/elle/ElleCorrect`. The final correctness theorems for the compiler are `elle_alt_correct*` in `eth-isabelle/elle/ElleCorrect/ElleAltSemantics.thy`

For more details on the structure of Elle, see *Implementation*.

2.3.2 Recreating FourL.ml

The command-line binary version of the Elle-based FourL compiler depends on `FourL.ml`, an Ocaml file that is produced from a formal Isabelle model via Isabelle's built-in extraction mechanism. As such, FourL.ml can be regenerated from Elle's sources, provided Isabelle is installed. This can be done as follows:

```
> isabelle jedit -d ./lem -d ./elle -l ElleCorrect
```

This will open the `ElleCorrect` session (building this session for the first time can take some time - as much as a couple of hours on a 16Gb machine). Once this session is done being processed, open the file `eth-isabelle/elle/ElleCorrect/FourLExtract.thy`. If that file is processed to the end (which can be forced by moving the cursor to the end of the file) it will create a new version of `eth-isabelle/elle/generated/FourL.ml`, which can then be built as described in *end-user-installation*.

CHAPTER 3

Usage of the LLL Frontend

Contents

In order to provide an convenient interface for programmers to work with Elle's verified compilation system, Elle provides a frontend called *FourL* that allows users to translate programs written in LLL to EVM bytecode. Unlike other implementations of LLL, FourL uses Elle's verified translation algorithm as a core layer, ensuring that address resolution and label scoping are handled properly. (For more details about what exactly Elle handles, see *Implementation* and *Correctness*).

3.1 Basic Usage

After building the `llllc` frontend as described in *Installation as an End-User of Elle*,

```
> cd eth-isabelle/elle/generated
> ./llllc ../tests/if.lll
6001600a576003600d565b60025b
```

This hex-code is output in the same format as Solidity LLL, and can be used with other existing tooling for deployment, testing, and static analysis. (For instance, ganache and web3.js have been used for testing `eth-isabelle/elle/tests/echo.lll`)

3.2 Supported LLL Constructs

FourL supports a large subset of LLL, but does not support the entire language. The supported constructs are listed below. For more information about the meaning these and other LLL commands, see the lll documentation

Table 1: FourL supported commands

Command	Notes/Caveats
`seq`	Unlike Solidity LLL, sequencing does not clean up the stack after push instructions
`if`	Expands to Elle control-flow
`when`	Expands to Elle control-flow
`unless`	Expands to Elle control-flow
`for`	Expands to Elle control-flow
`returnlll`	Supported only in a special case: when a single returnlll instruction occurs at the end of the constructor to retur
`lit`	Implemented using push rather than codecopy. As such, only supports up to 32-bit constants.
`+/add`	
`-`	
`*`	
`div`	
`exp`	
`/`	
`%`	
`sha3`	
`keccak256`	
`&`	
`|`	
`^`	
`~`	
`shr`	
`&&`	
`||`	
`!`	
`=`	
`!=`	
`>`	
`<`	
`<=`	
`>=`	
`mstore`	
`mload`	
`return`	
`stop`	
`calldataload`	
`calldatacopy`	
`calldatasize`	
`callvalue`	
`caller`	
`sstore`	
`sload`	
`log0-log4`	
`event0-event4`	

Table 1 – continued from previous page

Command	Notes/Caveats
`revert`	

Support for new constructs can be added by modifying the list of FourL macros (`default_lll_funs`) in eth-isabelle/elle/FourL.thy. This will require regenerating `FourL.ml` as described in *Installation for Modifying and Examining Elle*.

3.3 Debugging Failed Compilation

Unfortunately, the current version of Elle lacks detailed error reporting. Compilation either succeeds, in which case bytecode is output, or it fails, in which case a failure cause is not reported. This is one aspect of Elle that needs to be corrected in its next incarnation, a generalized compiler called Gazelle.

One option is simply to try to try to identify minimal error cases by writing smaller lll programs and trying to understand the cause of the failure.

Another, more advanced option for understanding failures in the Elle/FourL compiler involves running the compiler inside of the Isabelle proof assistant as described in running-compiler-in-isabelle. In this way, one can run different phases of the compiler separately to identify where exactly the error is happening. This requires setting up Isabelle and Lem as described in *Installation for Modifying and Examining Elle*.

3.4 Inspecting Bytecode

To help inspect the output of `llllc`, you may find it useful to use the EVM bytecode parser contained in eth-isabelle: eth-isabelle/parser/hexparser.rb

You'll need an installation of Ruby (tested with 2.5.1p57) to use this tool. It takes hex bytecodes like those output by `llllc` (or other compilers for Ethereum, such as Solidity LLL) on standard input and outputs (on standard output) a series of mnemonics describing the opcodes in the input.

CHAPTER 4

Elle Syntax

Contents

(Note: this documentation page is adapted from an entry in the Elle Github wiki)

4.1 Goal: Compiling Structured Code to EVM

The Elle source language, also known as *Elle-Core*, captures *structured programming* abstractions and enables their translation to Ethereum EVM bytecode through a verified compiler (whose details are described in *Implementation*).

What exactly is meant by structured programming? It corresponds to some features of languages that we usually take for granted: the ability to perform sequencing, if-statements, and loops in a predictable way that enables us to reason about different sub-components of a program separately, and then combine the results together soundly.

More concretely, suppose we have the following rather contrived program **P1**, that pushes two values onto the stack (the following is pseudocode for EVM bytecode):

```
; program P1
push 0x00
JUMPDEST
```

(continues on next page)

(continued from previous page)

```
pop
push 0x01
```

If run from the beginning, this code is innocuous: the JUMPDEST instruction will have no effect, the first PUSHed value will be POP'd back off, and we will end up with 0x01 at the top of the stack. However, if execution starts from the JUMPDEST, we have a problem: if the stack is empty beforehand, running just the latter 3 lines of P1 will cause a stack underflow and halt execution. This can happen, for instance, if P1 resides at (code-buffer address) 0xA0, and we have a possible path through the program that goes through the following code-snippet, sub-program P2:

```
; program P2
push 0xA1 ; address of JUMPDEST of p1
jump
```

We might want to prove that P1 will never cause a stack underflow. However, if P2 begins running with an empty stack, it will call into P1 in an unintended way that will cause the machine to crash. The fundamental problem here is that **jumps in EVM are always based on absolute addresses**, meaning there is no way to protect your code from another part of the program that happens to know the address of an internal JUMPDEST to which jumping would violate your code's invariants.

Instead, we can enforce a *structured* programming discipline - eschewing explicit jumps (i.e., "considering goto harmful") and instead using conditionals and loops to describe branching control flow. By denying users of the source-language (*Elle-Core*) the ability to do arbitrary jumps, we are able to compose sub-programs in predictable ways.

4.2 De Bruijn Indices and Structured Programming

Elle-Core provides a very general kind of structured programming, able to express the usual structured constructs such as *if* and *while*, but also more intricate control-flow structures, while maintaining the guarantee that "internal" labels within a sub-program cannot be improperly accessed through a notion of scope.

Elle's approach to representing scope is inspired by a practice from the programming-languages community called *de Bruijn indices* that provide a convenient way to describe scoped variables (it is also similar to the approach taken by WebAssembly). For an example of how this looks in a traditional context, consider a function that takes two parameters, discards the first, and returns the second. In the lambda calculus, we write this as

```
(\ x . (\ y . y))
```

However, we have chosen arbitrary variable names, which brings inconveniences. Nothing stops us from writing

```
(\ y . (\ x . x))
```

which expresses the same function. Perhaps more concerning, we could have equally written

```
(\ x . (\ x . x))
```

which also corresponds to the same function, because of the scoping convention of the lambda calculus: *if we bind a variable name twice, the innermost binding takes precedence*.

De Bruijn indices provide a clever trick to eliminate this ambiguity of naming and get a more canonical representation of functions that does not depend on specific variable names. The idea is that, because inner bindings take precedence, we can always describe variables in relative terms: each variable is uniquely distinguished by *how many levels up in the syntax tree that variable was bound*. So our example above becomes

```
(\ . (\ . #0))
```

#0 returns the second (innermost) parameter (we are zero-indexing). Had we wanted to return the first parameter instead, we would have written

```
(\ . (\ . #1))
```

With Elle, we do something similar, applying this same notion of scoping discipline to our labels. Each sequencing node (sequencing together Elle subprograms) creates a new context in which a new jump-target (label) can be described. Specifically, sequence nodes in Elle can have exactly zero or one label node poiting up to them, which corresponds (if there is one) to the JUMPDEST instruction that will be the target of this scope's jump. Jumps work similarly, specifying their targets based on which scope they will jump to ("jump n" means "jump to the label bound in the scope n levels up in the syntax tree") .

For instance, here is Elle pseudocode for an IF statement:

```
seq [
  seq [
    push 0x01
    jumpI #0
    push 0x02
    jump #1
    label #0
    push 0x03
    label #1
    ]]
```

Note that this approach provides the locality that we need: two disjoint Seq nodes will have no way of referencing each other's corresponding bound label.

4.3 Elle-Core Syntax

To cut to the chase, here is the syntax definition for the Elle-Core language, as implemented in Isabelle:

```
type_synonym idx = nat
datatype ll1 =
  L "inst"
  (* de-Bruijn style approach to local binders *)
  | LLab "idx"
  | LJmp "idx"
  | LJmpI "idx"
  (* sequencing nodes also serve as local binders *)
  | LSeq "ll1 list"
```

4.4 Label Resolution in Elle

Hopefully I've convinced you that de Bruijn indices are a convenient way to represent the binding structures Elle needs to handle. Next I'm going to describe how we translate this code (that is, syntax trees of type ll1) into EVM bytecode.

Our first step is to calculate *locations* (referred to in the codebase as *quantitative annotations*, or *qan*) for each instruction in our program. The idea is as follows. EVM instructions take up a certain number of bytes as specified in the EVM specification. *Seq* constructs do not take up any space other than the space taken up by their members. *Label* constructs take up one byte (the size of a JUMPDEST instruction). *Jump* instructions take a variable number of bytes, depending on the length of the address to jump to - this number of bytes starts at 2 (one for the JUMP itself, one for the PUSH instruction that puts its address onto the stack) but increases to accommodate the size of the address (the PUSH payload) that is actually calculated.

Once we have locations computed for all of our syntax-tree nodes, we begin examining the binding structure. For each sequence node, we examine all LLab nodes descended from it. If an LLab node is descended from an LSeq node at a distance of **n**, and that LLab's parameter (a natural number representing the index) is equal to n-1 (remember that we are using zero-indexing), the LLab's location within the tree rooted at that LSeq node is recorded.

If more than one such LLab is found for any one LSeq node, the compiler fails, as the user has given an invalid program. After all, it would not do to have the following (this is real Elle code this time, not pseudocode):

```
LSeq [
LJmp 0,
LSeq [
 LLab 1,
 L (PUSH_N [0])
],
LLab 0
]
```

The root LSeq node in this example has two labels "pointing upward" to the same sequence node. This creates an unacceptable ambiguity: to which label should the jump dispatch control flow? Depending on which we pick, we would have have added either 1 or 0 elements to the stack, so clearly they have different behavior. Elle will fail to compile code if it detects this condition.

4.5 Resolving Jump Addresses

Of course, we're not quite done: we still have to compute the addresses that each of our LJmp nodes will jump to (i.e., what value will be pushed onto the stack before the jump). At this point things get tricky. To save space, we want to minimize the number of bytes we push for each jump. Thus, we begin with the optimistic assumption that each jump target's address will be represented with one byte. With this assumption, we begin looking up the addresses of the labels corresponding to each jump and attempting to fit them into the number of bytes we have allocated.

If we ever fail to fit an address in the space we have allotted, we increase the number of bytes allocated to that jump by 1. Then we recalculate the addresses of all the Elle syntax nodes that must now be shifted, forget all the addresses of jumps we have so far resolved, and then begin the process again. Forgetting the previously resolved jumps is necessary, as their targets' addresses may have changed as a result of the 1-byte adjustment we just made.

Once we have resolved all jump addresses successfully, we have reached a form where we can quite easily write out our program as a sequence of EVM instructions. This forms the bytecode output by the Elle-Core compiler.

4.6 Conclusion

In this post, I have described the syntax of Elle-Core, the intermediate representation of the Elle system enabling structured programming, and its translation to EVM. The goal of the Elle project is to formally verify this translation. The translation itself is described in *Implementation*, and additional details about the verification can be found in *Correctness*.

CHAPTER 5

Elle Semantics

Contents

(Note: this documentation page is adapted from an entry in the Elle Github wiki

5.1 Goal: A Formal Meaning for Elle Programs

This document is intended to describe the semantics of the Elle-Core intermediate representation, the code that is translated to EVM via the Elle system using a verified procedure. What it means for this procedure to be "verified" is precisely that the behaviors of the source programs according to their semantics (described here) match the behaviors of the compiled EVM code, according to the EVM semantics described in Eth-Isabelle.

Before continuing, it is worth noting that this semantics uses the original EVM semantics to describe the behavior of individual instructions. However, all higher-level control flow is described by the Elle semantics, capturing the key abstraction Elle is intended to provide (structured control flow).

In the formal Isabelle development implementing Elle, this is actually described as a *big-step operational semantics*. For clarity of exposition here (i.e., to make the presentation more comprehensible for people not already familiar with inductive semantics) I will describe it here as a "non-deterministic" interpreter. The big-step version of the semantics can be found in `elle/ElleCorrect/ElleAltSemantics.thy`, here

Elle's semantics is nondeterministic "in theory" but not "in practice". What I mean by this is that while source programs can describe nondeterministic behaviors, the Elle compiler will refuse to compile them into EVM programs (this is important, as the EVM is by necessity a deterministic virtual machine). Additionally, it is relatively straightforward to characterize which Elle programs are deterministic (a predicate called "valid3" captures precisely this condition, discussed in more detail in *:ref: valid3<valid3>*).

Elle's interpreter uses a *logical program counter* which consists of an index into a syntax tree. In particular, this is expressed as a list of natural numbers, describing the path taken through the tree at each syntax node (that is, "[0,1,2]" means "the root's first child's second child's third child"; these paths are zero-indexed). In the codebase these lists are referred to as *childpaths* or *cp* for short. Most of the correctness proof revolves around showing that this logical program counter advances in such a way that the Elle program's behavior matches that of the EVM program with its standard (integer) program counter.

In the code below, "@" is list concatenation.

5.2 Informal Description of Semantics

5.2.1 Instructions

- Instruction nodes carry an EVM instruction that is guaranteed not to be a JUMP (i.e. no arbitrary modifications of the original program counter)
- If an instruction node is at the end of the tree (there is no next node), we return the result of running that instruction in the EVM semantics
- If the instruction is not at the end, we run the EVM instruction to get a new state, advance the logical program counter to the next instruction, and continue executing the Elle semantics from there

5.2.2 Labels

- Elle labels have the same semantics as an EVM JUMPDEST instruction

5.2.3 Jumps

- Jump nodes in Elle carry a "depth" parameter, pointing to a particular Sequence node a certain number of levels "up" in the tree.
- Label nodes in Elle also carry a "depth" parameter, which also points "up" the tree a certain number of levels to a Sequence node for which that label is a corresponding jump target
- For an Elle program to be valid, each Sequence node must have exactly one (or zero) label nodes "pointing up" to it
  - For valid Elle programs, the semantics of a jump involves moving the logical program counter to the location of the single label corresponding to the target of the jump
- Elle's semantics also describes execution of invalid programs, with more than one jump target
  - In these cases, Elle's execution *nondeterministically splits*, creating "parallel" executions that separately move the program counter to *each* of the jump targets, and continue executing from there.

- This behavior cannot be realized by the EVM, and is prevented by checks in Elle's compilation process that assure that programs with multiple labels will not compile successfully to EVM

5.2.4 Conditional Jumps

- If the head of the EVM stack in the current state is zero (i.e. the conditional jump will not execute in EVM) and there is no next node in the tree, we are done and can return the current state (after subtracting gas for the jump instruction)
- If the head of the EVM stack is zero and there is a next node, we run the unsuccessful jump (subtracting the gas), increment the logical program counter to point to the next node in the tree, and then continue executing
- If the head of the EVM stack is nonzero (the conditional jump will execute) the semantics are the same as that of a jump (other than the different gas cost) - see above.

5.2.5 Sequences

- If a sequence is empty (has no sub-nodes), and there is no next node *after* the current sequence node, we are already done executing and can return the current state
- If a sequence is empty and there is a next node, we advance the logical program counter to that node and continue executing
- If a sequence is non-empty, we begin executing from that sequence's first child

5.3 Interpreter (Pseudo)Code

```
Function get (root, cp) {
  if (cp == []) return root;
    else {
      if (node_type(root) != Seq) return null;
      else {
        if(nth (root, head(cp)) = null) return null;
        else return get(nth (root, head(cp)), tail(cp));
      }
  }
}

Function getnext (root, cp) {
   if (cp == []) return null;
   cp' = butlast (cp);
   cpl = last (cp);
   if (getnext (root, (cp'@(cpl+1))) = null) {
     return getnext (root, cp');
   }
   else {
       return (cp'@(cpl + 1));
   }
}

// Function get_label_cp(root)
// returns locations of all labels pointing up to root

Function ellesem (root, cp, state) {
```

(continues on next page)

(continued from previous page)

```
  switch (get (root, cp)) {
    case null: return emptyset;

    // instructions carry which EVM instruction to execute
    case Inst(i):
      if(getnext (root, cp) = null) return evm_sem i state;
      else return ellesem (root, getnext (root, cp), evm_sem i state);

    // labels are jumpdests
    case Label(d):
      if(getnext (root, cp) = null) return evm_sem JUMPDEST state;
      else return ellesem (root, getnext(root, cp), evm_sem JUMPDEST state);

    // jumps carry a depth - how many scopes up to jump to
    case Jump(d):
      ctxpath = take((length cp - d), cp); //take all but last d elements
      context = get(root, ctxpath);
      s = get_label_cp context;
      return set{cp' | ellesem(root, cps, state)};

    case JumpI(d):
      // if we should jump
      if(hd (evm_stack (st)) != 0) {
        ctxpath = take((length cp - d), cp); //take all but last d elements
        context = get(root, ctxpath);
        s = get_label_cp context;
        return set{cp' | ellesem(root, cps, state)};
      }
      else {
        if(getnext (root, cp) = null) return state;
        else return ellesem(root, getnext(root, cp), state);
      }

    // sequences carry a list of sub-nodes
    // we jump to all labels pointing to the Seq node "d" levels up
    case Seq(l):
      if(l == []) {
        if(getnext (root, cp) = null) return state;
        else return (ellesem(root, getnext(root, cp), state));
       }
       // if the list has children, run its first child
       else return ellesem(root, cp@[0], state);
  }

}
```

5.3.1 Notes on Interpreter

The “jump” and “jumpI” cases in the above code explicitly return sets of states, which captures the nondeterminism of the semantics. All other (deterministic) cases can be considered to be implicitly returning singleton sets containing the single next state (for clarity I have left these implicit).

Again, the actual semantics of Elle programs are phrased somewhat differently, using an inductive relation rather than an explicit interpreter. While it would be possible to encode this interpreter directly in Isabelle and explicitly prove that it matches the inductive semantics given in `elle/ElleCorrect/ElleAltSemantics.thy`, this has not been done for Elle, although it is planned for Elle's successor, Gazelle. Nonetheless, the intepreter is likely easier to read and understand for most programmers not used to seeing formal semantics.

CHAPTER 6

Implementation

This page describes some salient aspects of how the Elle compiler is implemented. Details about the implementation of the correctess proof are deferred until *Correctness*.

In *Elle Syntax*, we described the syntax of the Elle language from the user's perspective. However, internally, Elle uses a series of annotations to describe Elle programs at various stages of compilation. These can be found in `elle/ElleSyntax.thy`. This more elaborated representation takes the form of general datatype with extension points (type parameters) into which we can insert annotations later, along with the specific syntax extensions used at various stages of the compiler.

The Elle compiler proceeds in several phases, outlined in the remainder of this document. Note that links in this file are to the (frozen) `ITP2019` branch of the repository, to ensure consistency of line numbers as master evolves. Though the line numbers may change, the general ideas should not.

6.1 Phase 1 - Generating Size Annotations

As a first step, the Elle compiler generates a pair of integer annotations for each node in the syntax tree given to the compiler as input. These annotations correspond to the range of bytes taken up by the code that will be generated from the syntax tree in the program buffer. These are calculated as one would expect: instructions are simply the length of the encoding of the instruction as bytecode, labels correspond to the lengths of EVM JUMPDEST instructions, and sequence nodes have lengths equal to the sum of the lengths of all their children.

The implementation of this compiler phase can be found here, in `elle/ElleCompiler.thy`.

After this phase (and throughout the Elle compiler thereafter), syntax trees will be proven to conform to the following predicates `ll_valid_q` and `ll_validl_q` (on elaborated Elle syntax trees and lists of elaborated Elle syntax trees, respectively) (the size-annotations are referred to as "quantitative annotations", and abbreviated by "q" or "(x, x')", throughout). These predicates can be found here, in `elle/ElleCorrect/Qvalid.thy`.

Note that, other than for the `s` annotations on JUMP and JUMPI, none of the syntax tree node annotations have any impact on the size of the generated code corresponding to a node (thus, no effect on the inference rules for `ll_validl_q`). This makes sense, as they correspond purely to compile-time artifacts that are not present in the generated code.

6.2 Phase 2 - Finding Labels

In the second phase of compilation, we enforce the invariant that *each Seq node has exactly 0 or 1 descended labels*, as defined according to the `ll3'_descend` predicate, which can be found in `elle/ElleCorrect/Valid3.thy`, here.

For this phase of the compiler, we traverse the Elle syntax tree. At each `Seq` node, we scan the sub-tree for all descended `Lab` nodes with an index "pointing back up" at the `Seq` node we are currently considering. If we find no such nodes, we mark the `Seq` node with an annotation indicating there is no label (an empty list). Otherwise, we take the first `Lab` node we find (in a preorder traversal), annotate it as having been "consumed" by a `Seq` node, and store the path to that label at the `Seq` node. In order to guard against multiple labels "pointing up" at the same `Seq` scope, this compiler pass fails if it ever encounters a `Lab` node that has not been consumed in its top-level traversal of the tree (since such a node corresponds either to a nonexistent scope, or to a scope which already has a label corresponding to it).

The compiler pass is implemented in the function `ll3_assign_label` in `elle/ElleCorrect/ElleCompiler.thy`, here

For reasoning about this phase and subsequent phases, we use the aforementioned``ll_descend`` predicate, which relates two Elle syntax trees and one list of natural numbers. This predicate captures situations in which the syntax tree `l2` can be found as a descendant of `l1` by treating `k` as a path through the tree, selecting which child-node to choose at each step.

After the second phase of compilation (if successful), syntax trees will be proven to conform to the inductive predicate `ll_valid3'`, which can be found here, also in `elle/ElleCorrect/Valid3.thy`. This predicate essentially corresponds to the intuition that each Sequence node has exactly zero or one labels referencing it, and that the locations of these labels are annotated on the sequence nodes.

6.3 Phase 3 - Resolving Jumps

Once we have located the unique corresponding label (or determined the nonexistence of such a label) for each sequence node in the second phase, we need to calculate target addresses for each jump node based on the locations of those labels.

This process involves, essentially, a fixed-point calculation over the Elle syntax tree, in order to ensure that sufficient space has been allocated to store the needed address at each `Jump` and `JumpI` node. This process is captured by the function `process_jumps_loop`, which can be found in `elle/ElleCorrect/ElleCompiler.thy`, here.

`process_jumps_loop` makes use of two auxiliary functions. The first is `process_jumps`, which captures one iteration of the size checks involved in the jump-resolution process. `process_jumps` returns one of three cases of result: either `Success` if all jumps have sufficient size to store their corresponding addresses, `Fail` if there is an un-recoverable error (such as invalid input) and `Bump` if a node in the tree has a jump-target size that needs to be incremented.

At each `Seq` node, `process_jumps` checks the node's annotation to see if there is a corresponding label. If there isn't one, `process_jumps` scans all descended `Jump` nodes to make sure there are no descended jumps that point to that `Seq` node (as such jumps would have no target), failing if it finds any. If there is a label according to the annotation, `process_jumps` looks up that label to find its address (failing if there isn't one to be found), and then runs on that sequence node's descendants (doing an in-order traversal) to find all jumps that point to the scope corresponding to this sequence node. If any are found, `process_jumps` checks the space allocated to that jump node against the space required to encode the address from the label that was looked up previously for the `Seq` node.

If there is enough space, `process_jumps` continues scanning the tree for other jumps corresponding to the same `Seq` node and performing the same check, ultimately returning `Success` if all of them have enough space. Otherwise, it returns the absolute location (as a `childpath`) of the first `Jump` node without enough space in the form of a `Bump` result.

(For `Seq` nodes descended from the root, `process_jumps` first performs these checks for jumps pointing up to the outer `Seq` node, then recursively performs the same checks on the descended `Seq` node.)

The second auxiliary function used by `process_jumps_loop` is `inc_jump`, which takes a path (corresponding to a `Jump` node) returned by `process_jumps` and increments its size, adjusting the size annotations of the rest of the tree in the process as appropriate.

To avoid a complicated termination argument for `process_jumps_loop` (functions in Isabelle need to be proven to terminate or they become very inconvenient to reason about), the execution of this function is "fuelled" (termination is justified by a decreasing natural-number argument, which is decremented once each time the loop is run - thus, once per time a `Jump` node's size needs to be incremented). If this fuel parameter is `0`, `process_jumps_loop` returns a failure (`None`) Otherwise, runs `process_jumps` on the root of the Elle syntax tree given as an argument. If `process_jumps` returns `Success`, `process_jumps_loop` returns the input syntax tree as nothing needs to be done. If `process_jumps` returns `Failure`, `process_jumps_loop` also fails (returns `None`). Otherwise, if `process_jumps` returns `Bump`, `process_jumps_loop` calls `inc_jump` on the child-path returned by `process_jumps`, and then calls `process_jumps_loop` on the same arguments (with fuel parameter decremented).

The correctness of `process_jumps_loop` is established by a series of validation passes that happen after it runs. However, `process_jumps_loop` is proven directly to produce `valid_q` results from `valid_q` inputs. Additionally, we define a function, `get_process_jumps_fuel`, which calculates a sufficient amount of fuel to ensure that `process_jumps_loop` terminates on its input (although this is not formally established with a proof).

By the end of running `process_jumps_loop`, we have a syntax tree that should obey the predicate `ll4_validate_jump_targets`. This predicate essentially makes sure that the indices of jump nodes (which point to the sequence node corresponding to the jump; i.e., to the scope the jump's target is in) correspond to a scope whose label has an address matching the address stored at the jump node (which is the address that will ultimately be written out to bytecode).

The definition of `ll4_validate_jump_targets` can be found in `elle/ElleCorrect/Valid4.thy`, here.

6.4 The Big Picture

At this point, we have produced a syntax tree that is valid as an `ll4` syntax tree, yet meets all of the predicates described above. In its final form, `ll4` contains all the information needed to generate concrete EVM machine code, including concrete addresses. At this point, `codegen'` is used to emit a list of bytes corresponding to the output bytecode. (`codegen` can be found in `elle/ElleCompiler.thy`, here)

An additional validation step is used after this point to ensure that all jumps are encodable in EVM (that is, their addresses are at least 1 byte and not more than 32 bytes). Code for these extra validators can be found in `elle/ElleCorrect/ElleAltSemantics.thy`, here. Examples of how all these pieces may be put together into a single verified compilation pipeline can be found in `elle/ElleCompilerVerified.thy` (here)

In the next section, *Correctness*, we will sketch the process by which the generated EVM instructions are proven correct with respect to the input program, making use of the information contained in these intermediate predicates.

CHAPTER 7

Correctness

Here we describe the theorem establishing the correctness of the Elle compiler, and sketch its proof.

Note that links in this file are to the (frozen) `ITP2019` branch of the repository, to ensure consistency of line numbers as master evolves. Though the line numbers may change, the general ideas should not.

In *Implementation*, we discussed the steps taken by the Elle compiler when translating code from the Elle language to EVM, and the specifications proven about the code generated after each step. In this section we will discuss the proofs that formally establish these invariants for each step, as well as the final theorem about the end-to-end process of compilation that ties them all together.

- For each phase of the compiler (other than jump resolution), we prove that only annotations with no impact on the semantics of the program when translated to EVM are modified. This enables us to lift results about the outputs of compilation passes to results about their input programs.
- Syntax trees generated by `ll_phase1` satisfy the predicate `ll_valid_q`, so long as they only contain valid instructions according to the `inst_valid` predicate (a predicate that rules out use of instructions that would violate Elle's guarantees, such as arbitrary jumps not governed by Elle's scoping mechanism). This is proved by a relatively straightforward induction on the structure of the tree.
- We prove that subsequent passes of the compiler preserve the `ll_valid_q` predicate. Because only annotations without effect on the generated code are changed in most passes, this is relatively straightforward. For jump resolution, we additionally prove that expanding jump nodes to allow for larger addresses preserves `ll_valid_q` via a lemma about the effect of increasing the size of a jump by 1 (`ll_bump`).
- We prove that the output of the second pass of the compiler meets the predicate `ll_valid3`. This is done by means of a verified validator pass that runs after the second pass of the compiler. This pass essentially supplies an executable version of `ll_valid3`: it iterates over the structure of the tree, gathering the set of label nodes that point up to each sequence node. Each sequence node annotation is then checked against the gathered label nodes to ensure the annotations match (that is, either there are 0 nodes pointing up at the sequence node in question and the annotation is `[]`, or the annotation is a nonempty list corresponding to the child-path of the single node that points up to the sequence node that holds the annotation). Otherwise, the validation pass fails and the compiler produces no output. Verification of this pass involves a relatively straightforward induction on the structure of the input tree (strictly speaking, on the structure of the proof that the input tree is valid according to `ll_valid_q`, which mirrors the syntactic structure of the tree.)

- For the third phase of the compiler (resolving and resizing jumps) we make use of another validation pass, which runs after the body of the compiler. This validator is proven to only accept input code which satisfies the predicate given in (TODO: link to code).
- After this final compilation phase, we run a validator to ensure that the lengths of encoded jump addresses match the length annotations on the `Jump` and `JumpI` nodes; that is, that all addresses we are going to encode fit exactly into the space we have allocated for them.
- Finally, we run one last validator to sanity-check the jumps contained in the final code output by Elle. This validator checks to ensure that jumps are encodable in EVM (that is, that no jumps to addresses less than 1 byte or more than 32 bytes in length are present in the code).
- At this point, we can easily dump the final, fully annotated version of the Elle syntax tree to EVM instructions. It is these instructions that the final proof of Elle's correctness will reason over.

For details of the theorem statement, see elle_alt_correct in `elle/ElleCorrect/ElleAltSemantics.thy`.

7.1 The Corrcetness Theorem

This correctness theorem can be paraphrased as follows: suppose we have an Elle program `t` that ends in a state `st'` when started in state `st` at node `cp` (under the Elle semantics). If the `t` is valid under the `valid3` predicate as well as passing jump-targets validation),

the result of dumping this Elle program to EVM bytecode yields a program that steps from `ir` (with program counter set to the program counter corresponding to the start of the instruction pointed to by `st`), it will end in a state that will differ from `ir'` only in the value of the program counter (unless insufficient interpreter fuel has been given to the EVM interpreter, in which case a larger value of fuel would yield such a final state - this is what is captured by the predicate `program_sem stopper prog fuel net (setpc_ir st targstart) = InstructionToEnvironment act vc venv`). `program_sem` comes from the original eth-isabelle project on which Elle is based.

7.2 Setting Up the Proof

Programs produced by the Elle compiler meet the validity predicates `valid3'` and pass the jump-target validator, so this means that when the output of the Elle compiler is run in EVM semantics, the result will always be the same as the result given by running the source Elle program under Elle's semantics. This establishes one direction of simulation: that Elle programs are simulated by the EVM programs output by the Elle compiler under the conditions given above.

Because the state spaces for the input and output program semantics are virtually the same, and both languages are deterministic (assuming the Elle program in question is valid, but the compiler will produce `None` as an output in the case that it is not), proving this one direction is sufficient to establish the stronger *bisimulation* results that are common in the compiler-correctness literature - informally the argument is as follows: suppose we have that the EVM semantics steps from `st` to `st'` for some program `prog`, that `prog` was produced from an Elle program `eprog`, and that `st` has a program counter value of `pc`. Unless `pc` is the address-value of the middle of a multi-word instruction (see below), `pc` will be the address of the start of an instruction, meaning there will be at least one node in `eprog` with a left-side annotation equal to this address.

Since the Elle semantics is deterministic for valid programs, and `eprog` must be valid (or the compiler would not have produced any output program), we know that there must exist some `st''` such that `eprog` steps from `st` to `st''` under the Elle semantics with a childpath pointing to any node with an annotation matching the value of `pc`. (There may be more than one such node, if the node is the first element of a sequence, but the sequence semantics implies that the semantics of running the Elle program from any of these nodes will produce the same result). Now, we have two cases. If `st'' = st'` (with the possible exception of the final program-counter values being different), we have the converse result that we wanted to prove and are done. Otherwise, we can apply the correctness theorem of Elle to get that `prog` steps from `st` to `st''` under the EVM semantics, a contradiction since `st' != st''` (beyond just differences in program counter values) and the EVM semantics is deterministic.

As alluded to above, there are some EVM states which do not have a corresponding state in the Elle semantics for a particular program, but these states correspond to instruction-pointer values that are invalid in the sense of not being the address of the beginning of any instruction. An EVM program executing from the beginning should never reach such a program-counter value, so it is safe not to consider such states if what we care about ultimately is the behavior of whole EVM programs run from the beginning (which it is).

7.3 Correctness of Elle: Proof Sketch

In order to establish the correctness theorem of Elle, we use the induction principle for the `elle_alt_sem` predicate. This requires us to prove 11 goals (corresponding to the 11 cases in the semantics).

1. For the first case, we need to prove that instructions at the end of an Elle program behave the same way as running the same instruction at the end of the corresponding EVM program. This amounts to a straightforward case-analysis on possible EVM instructions, in order to demonstrate that running the instructions (followed by `elle_halt`) yields the same result as running the same instruction at the end of the corresponding EVM program.
2. For the second case, we need to prove that program executions beginning with instructions not at the end of the program behave the same way as their corresponding EVM programs. This proof is similar to the first case, except that instead of running `elle_halt` at the end and comparing the final results, we need to appeal to our inductive hypothesis (which says that running the Elle program and EVM programs after the given instruction yield the same result.) In order to apply our inductive hypothesis, we need to prove that the states match after executing a single instruction, which is similar to the beginning to the proof of the first case.
3. The third case is for label nodes at the end of Elle programs, its proof is almost identical to the first case (minus the exhaustive case-analysis of EVM instructions).
4. The fourth case is similar to the second case in the same way that the third case is to the first: again we need to appeal to an inductive hypothesis about running the remainder of the program, and doing so involves reasoning about the execution of a single JUMPDEST instruction (as in the third case).
5. The fifth case involves reasoning about the effects of the `Jump` Elle instruction. The core of this proof is a case-analysis on the three disjunctive cases of the specification for jump-target validity. Since we are talking about a whole tree (rather than a tree in context) we only have two cases: one where the jump in question points up to the context corresponding to the root of the tree (a `Seq` node) and one where the jump points up to an intermediate node which is, in turn, a descendant of the root.

   In both cases, the `ll_valid3` predicate is used to show that, because the label node corresponding to the jump node's `Seq` node is unique, the Elle semantics selects a single jump target which corresponds to the jump target given by the EVM semantics of the compiled code.

   Additionally, some (rather tedious) reasoning is needed to prove that when the target address of the jump is serialized in to an EVM stack value and then deserialized again (to calculate the new program-counter value in the EVM semantics) the value is preserved. This mostly boils down to proving that the address in question does not overflow a 256-bit EVM integer, which is guaranteed by the fact that the address is not more than 32 bytes (shown by an additional validation pass run at the end of the compiler).

   The second case is largely similar to the first, except that some extra reasoning needs to be done to show that the descended `Seq` node also satisfies the `ll_valid3` predicate, and then to translate the results of the reasoning on the subtree in which the jump is taking place back up to a statement about the meaning of the jump in the context of the overall syntax tree descended from the root node. For details, the reader can refer to our Isabelle formalization.

6. The sixth case involves reasoning about the effects of the `JumpI` Elle instruction when the conditional jump is taken. This case is similar to the fifth case, except that a bit of additional reasoning is needed to prove that the jump is indeed taken.
7. The seventh case involves reasoning about `JumpI` in the case where the conditional jump is not taken and the `JumpI` instruction in question is at the end of the code. This case is similar overall to cases 1 and 3, since the semantics of `JumpI` where the jump is not taken are not dissimilar to that of the label case (decrement gas by the correct amount, increment the program counter)
8. The eighth case involves reasoning about `JumpI` in the case where the conditional jump is not taken and the `JumpI` instruction in question is not at the end of the code. This case is similar overall to cases 2 and 4.
9. The ninth case is the case of executing an empty sequence node at the end of an Elle program. Because the empty sequence has no effect on the machine state (nor does running an empty series of EVM instructions), this corresponds to showing that the effects of running `elle_halt` matches the effect of running at the end of an EVM program. Essentially this is an easier version of cases 1, 3, and 7.
10. The tenth case corresponds to executing an empty sequence node somewhere other than the end of an Elle program. As with cases 2, 4, and 8, this involves applying an inductive hypothesis stating that the execution behaviors of the remainder of the program are the same between the Elle an EVM versions when started in the same state. Because an empty sequence of instructions leaves the state unchanged in both the Elle and EVM versions of the program, the hypothesis almost immediately applies.
11. The eleventh case corresponds to running a nonempty sequence in Elle. In this case, we get an inductive hypothesis about the effect of running the given Elle program starting at the first element of that sequence. We need to show that the address in the output code corresponding to the start of the sequence in Elle is the same as that of the start of its first element, which is straightforwardly proved using auxiliary lemmas about the behavior of the `ll_valid_q` predicate on lists. Once we have this, we can apply our inductive hypothesis to complete the proof.

It should be noted that we prove two different versions of this lemma. The first, `elle_alt_correct` talks about correctness in cases where the execution of the Elle and EVM programs terminate successfully; the second, `elle_alt_correct_fail`, deals with the case where the execution ends in a “crashed” state (either because the EVM stack limit was exceeded, the EVM was not supplied enough gas to complete the execution, or an invalid instruction was reached).

CHAPTER 8

Indices and tables