Specification

Zero-Knowledge circuit

for continuation-passing interpreter

March 24, 2022

Abstract

This document describes how Lurk circuits are constructed. It is a **work in progress**. Right now it contain only a short overview of the specification. The reader can expect a complete description will be provided in the near future.

Contents

1 Introduction

Lurk is a functional programming language based on Lisp. It is based on Continuation Passing Style (CPS), which is an idea described in the book "Essentials of Programming Languages" [2]. We will summarize the main concepts involved in the design of the language, such that reader can understand how zero-knowledge proofs [4, 5, 1] are constructed to provide privacy to Lurk's programs.

2 Lurk

In this section we provide a summary of Lurk main elements. Shortly, an *expression* represents a computation involving literals, variables, operations and procedures. Variables are handled by an *environment*, which is responsible for binding variables and values. Also, we use *continuations* to indicate what must done to finish the computation.

We have that the system's IO is formed by an expression, an environment and a continuation. These 3 elements are essential to comprehend Lurk. Each element is represented as a pointers, which is implemented using hash functions. In particular, we use Poseidon [3] to instantiate our pointers.

The environment has a list of bindings, which correspond to a mapping between variables and values. It has local bindings, meaning that the mapping is valid only for a specific evaluation of an expression. On the other hand, the global state is represented by the *store*, which behaves as a memory of the system. We have that the store is global, while the environment is local.

2.1 Overview

- **t, nil:** are self-evaluating and represent true and false, respectively.

- **if:** it has the format `(if <test> <consequent> <alternate>)` and represents a conditional expression. It **must** receive all 3 parameters, where the `<test>` is an expression used to select the result; the `<consequent>` is selected if the `<test>` evaluates to non-`nil`; and `<alternate>` is selected otherwise. We remark that differently from other programming languages, the `<alternate>` expression is mandatory.

- **lambda:** it has the format `(lambda <formals> <body>)` and represents a procedure. The environment when the lambda expression is evaluated is used as a **closure**, which is extended with `<formals>`, a list of variables. The unique expression in the `<body>` is evaluated and returned as result of the lambda expression.

- **let:** it has the format `(let <bindings> <body>)` and represents an assignment expression, where `<bindings>` represents a list of pairs in the form `(<variable>, <init>)`; and `<body>` is a unique expression.

- **letrec:** it has the same format as `<let>` expressions, following the same rules, but also allowing recursion.

- **quote:** it has the format `(quote <datum>)` or `(' <datum>)` and evaluates to `<datum>`.

- **atom:** it has the format `(atom <e>)`, and it evaluates to `t` if `<e>` is not a list, and evaluates to `nil` otherwise.

- **cons, car, cdr:** The expression `(cons <a> <d>)` produces a pair whose `car` is `<a>` and `cdr` if `<d>`.

- **arithmetic operations:** it has the format `(<op> <e1> <e2>)`, where `<op>` corresponds to an arithmetic operation $(+, -, \star, /)$. We have that `<e1>` is evaluated before `<e2>` and the operation is carried out in the finite field that is used in the subjacent zero-knowledge backend.

- **equality:** it has the format `(<op> <e1> <e2>)`, where `<op>` can be either `=` or `eq`. The equality symbol `=` is used to compare expressions whose result is a number (finite field elements), while the symbol `eq` is used to compare pointers.

- **emit:** it has the format `(emit <e>)` and is used to return the result of the evaluation of `<e>` as a public value, which can be used to define the instance of the zero-knowledge statement.

- **begin:** is has the format `(begin <e> ...)`. The sequence of expressions is evaluated from left to right and the last result is returned.

- **current env:** it returns the current environment represented as an association list.

- **eval:** it has format `(eval <exp>)` or `(eval <exp> <env>)`. The evaluation of `<exp>` is used as an expression in the environment obtained from the evaluation of `<env>`. If no `<env>` is provided, an empty environment is used.

2.1.1 Fibonacci example

See next the code snippet that implements the Fibonacci's sequence:

```
> (letrec ((next (lambda (a b n target)
        (if (eq n target)
            a
                (next b
                (+ a b)
                (+ 1 n)
                    target))))
            (fib (next 0 1 0)))
```

```
       (fib 10))
[521 iterations] => 55
```

3 Circuit

Here we describe the construction of Lurk's circuit.

3.1 Overview

A Lurk program is split into a sequence of *reduction steps*, also called iterations. In the Fibonnaci example above we have 521 iterations, and each one is mapped into a *frame*. We group a set of frames into a MultiFrame object.

We construct a CircuitFrame for each frame, and a Circuit is a sequence of CircuitFrames, where the output of a previous one is linked to the input of the next. The circuit mimics the evaluation of Lurk expressions.

In `eval.rs` it is possible to see the implementation of evaluation of Lurk's expressions. An important function is called reduce_with_witness(), which is responsible for the computation of reduction steps with its witnesses. Since a reduction step is exactly what we want prove in zero-knowledge, we provide an implementation that carries out the same computation, but in the circuit. This implementation is accomplished by reduce_expression(). Therefore we are going to describe how this implementation works.

We call *global symbols* the set of Lurk symbols that are pre-computed, such that we can easily compare with symbols found during the evaluation of expressions.

To reduce an expression, we require a substancial amount of conditions that depend upon comparisons among symbols. Those conditions involver mostly allocation of variables and pointers in the circuit, Boolean logic using those elements, and conditionals. This functionality is implemented in method reduce_sym(). The Boolean logic and equality tests are executed against global symbols and other auxiliary variables created along the process in order to decide how we are going to take decisions about control flow of Lurk programs. Using it we also can update the environment and the store accordingly.

One of the most important building blocks in a functional language like Lisp and Lurk is `cons`. It is used to concatenate `car` and `cdr`. Hence, whenever we want to break big expressions into smallers ones, we can use a combination of those elements. In function reduce_cons() we deal with each different situation where those elements are necessary. Namely, to reduce an expression into smaller pieces, we allocate in the circuit some auxiliary variables for later utilization. In particular, car_cdr() gadget is used as a building block. Since this computation is required for different situations, depending on the type fo expression we are handling, for each one we include a clause in a multicase gadget. Thus we can easily select desired result using this approach. The multicase selection is based on the `head` of an expression, which is given by the `car` of the expression. We return the result of the multicase.

apply_continuation()

For each continuation tag we need compute the next expression, environment, continuation and thunk. Therefore we have to allocate the appropriate pointers. This task is executed in 2 stages:

- Some continuations require the calculation of a new pointer, while others don't. For those which indeed need a pointer, since implementation of pointer is based on a hash computation, and because hashes are expensive in the circuit, we use a multicase to select the appropriate hash preimage. Then we can compute the hash just once. This allows us to avoid computation of unnecessary hashes.
- Selection of the continuation results.

3.2 Gadgets

In order to construct the circuit we use the following building blocks:

- Macros:
  - Boolean.
  - Equality.
  - Conditionals.
  - Pick.
- Constraints:
  - Arithmetic operations.
  - Utils.
- Pointers: tag and hash.
- Data: allocate and reverse lookup.
- Multicase.

4 Final remarks

In this document we presented the construction of Lurk's circuit, such that Lurk's programs can be proved using zero-knowledge.

5 References

References

[1] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. `https://ia.cr/2019/1021`.

[2] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

[3] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. `https://ia.cr/2019/458`.

[4] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[5] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021. `https://ia.cr/2021/370`.