Specification

Zero-Knowledge circuit

for continuation-passing interpreter

March 23, 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 Lurk

Next we provide a summary of Lurk elements.

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>` is 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

```
(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 1))
```

2.2 Store

3 Circuit

3.1 Overview

Each reduction step is mapped into a Frame.

We group Frames into a MultiFrame object.

We construct a CircuitFrame for each MultiFrame.

A Circuit is a sequence of CircuitFrames,

The circuit mimics the evaluation of Lurk expressions.

reduce_expression() implements reduce_with_witness() in the circuit.

global symbols contains Lurk symbols, such that we can easily compare with

reduce_sym()

We use Boolean logic and equality tests (against global symbols) to obtain calculate auxiliary variables that will allow us to decide how we are going to take decision about control flow of a Lurk program. Using it we also can update the environment and store accordingly.

reduce_cons()

Takes an expression and reduces to smaller pieces, allocating in the circuit for later utilization.

car_cdr() gadget is used to constrain the reduction.

For each possible expression type that can be reduced, we include a clause in a multicase gadget.

The multicase selection depends on the head, which is the car of the expression.

We return the result of the multicase.

make_thunk()

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

3.2.1 Macros

Boolean

Equality

Pick

3.2.2 Constraints

Arithmetic operations

Utils

3.2.3 Pointer

Tag
    Hash

3.2.4 Data

allocate
    reverse lookup

3.2.5 Multicase

case
    multicase
    optimization

4 Final remarks

5 References