Lurk Report

Artem Gureev[a]

[a]Heliax AG

* *E-Mail: artem@heliax.dev*

Abstract

This paper presents the report on the Lurk language, it's main functionality and ability to be usable as a Juvix backend.

Keywords: Lurk ; Juvix ; Circuits ;

Contents

1. Introduction

Lurk is a Lisp written in Rust for zkSNARKs supporting basic cryptographic functionality using Poseidon hashing as well as an Incrementally Verifiable Compulation (IVC) friendly evaluator.

The report is aimed at a quick overview of Lurk's basic functionality as well as its possible value for the Juvix project.

There are several things to note before starting the report:

1. The report is based on Lurk v0.2.0 and as presented in the source material listed below
2. The report will make comparative claims based on the assumption that Lurk is working with the Nova backend, the only officially supported backend.
3. Another currently working backend is Groth16. A backend in possible development is Halo2.
4. There are Rust and Lisp versions of Lurk availiable. The Lisp version is currently unsupported.

The main materials one is encouraged to consult and read alongside the report are:

1. For a high-level description, the Lurk paper Amin et al. (2023)
2. For a low-level description, the official Lurk spec Lurk (2022a)
3. For a quick intro, the Programmer's Intro of the Lurk team Lurk (2022b)

2. Spec

2.1. Frontend. The section is devoted to the description of Lurk, pointing to its central advantages and mechanics.

Lurk is set up as an interpreter. It performs computations given an expression. The REPL takes a Lurk expression, then evaluates it recursively by spreading it out in **frames**. A **frame** is given by a 3-tuple consisting of

1. Lurk expression
2. Environment
3. Continuation

Expressions are recursively given. There are strings, unsigned 64-bit integers, basic numbers (represented as elements of field of Pallas numbers[1] suitable for a Nova backend), equality predicate, cons , let , if expression, and letrec , lambda , basic arithmetic operations on field elements etc. expressions following the evaluating strategies using usual semantics. Note that letrec allows for definitions of recursive functions.

Each expression can be assigned to a DAG. E.g. an expression (let ((a (* 1 2))) (+ a (- 3 4))) can be presented as

$$\begin{tikzcd}
& & & \text{let} \arrow[dll] \arrow[dr] & & & \\
& * \arrow[dl] \arrow[dr] & & & + \arrow[dl] \arrow[dr] & & \\
1 & & 2 & a & & - \arrow[dl] \arrow[dr] & \\
& & & & 3 & & 4
\end{tikzcd}$$

The expressions which make Lurk interesting with respect to cryptography are built-in commit expressions, which - given a Lurk expression - make a cryptographic commitment using a Poseidon hash to any other Lurk expression. In particular, Lurk has in-built functional commitments. Committing an expression produces a field element with secret 0. A commitment can be opened using expression open if and only if the REPL has a history of that field element being a commitment[2]. An explicit secret for a commitment may be made with secret command.

Each expression is given by a pointer, consisting of a tag and a hash managed by a store.

As a Lisp, everything is an expression evaluated left-to-right strictly and is either self-evaluating, reduces to something self-evaluated, or produces an error.

An **environment** is given by a consing of expressions, which were introduced by let and letrec expressions. Once expressions are evaluated, they are taken out of the environment.

A **continuation** presents the next step for the computation to continue. Theoretically, it presents itself as a poset pointing to the type of the operation to be evaluated, as well as the predecessors to the operation. E.g.

```
> (+ 2 40)
INFO lurk::eval > Frame: 0
Expr: (+ 2 40)
Env: NIL
Cont: Outermost

INFO lurk::eval > Frame: 1
Expr: 2
Env: NIL
Cont: Binop{ operator: Sum, unevaled_args: (40), saved_env: NIL, continuation: Outermost }

INFO lurk::eval > Frame: 2
Expr: 40
Env: NIL
Cont: Binop2{ operator: Sum, evaled_arg: 2, continuation: Outermost }

INFO lurk::eval > Frame: 3
Expr: 42
Env: NIL
Cont: Terminal

[3 iterations] => 42
```

Here outerbox is a state referring to the entire exression. Once the reducer returns to outermost it then moves to terminal to finish the computation and print results. binop stands for identifying that the right side of a binary operation will be evaluated, while binop2 is a continuation signaling that we, having evaluated both sides of a binary operation, are going to a frame actually reducing the operation to a field value.

Hence, the reducer works by taking an expression, making it into a default frame[3] applies the reducer per appropriate semantics moving from frame to frame until it reaches a frame with terminal evaluation

The approach for using the frames is: "*Do the minimum amount of work in the current step [...] and give the rest of the computation to (sic) [...] continuation...*" [4]

[1] By default. Other primes are also availiable

[2] Being a Poseidon hash result, a commitment is binding

[3] with nil environment and outermost continuation

[4] https://www.youtube.com/watch?v=wKqiloOeogo

2.2. Backend. Now that we have demonstrated the reduction mechanism, let us discuss how this fits with the cryptographic backend.

Arguably the one thing that makes the interpeter frontend really special is the built-in cryptographic commitment functionality[5].

However, what is important is how those computations get checked and specifically how the frames approach to evaluation allows the Nova backend to be so (arguably) quick.

Nova uses the approach of IVC or recursive ZKPs. It splits a statement into components and uses the fact that it provided proofs to said components to create a proof of an aggregate statement. This approach has multiple advantages for certain types of proofs, including the ability to prove things about recursive function without unrolling as well as parallel proof computation. In particular, Nova implements a prover which can prove statements of the from $y = F^l(x)$ for F a (potentially non-deterministic) computation Kothapalli et al. (2021).

The small-step evaluation via frames that Lurk provides allows for a logical splitting of a Lurk expression into blocks. The expression part of a frame is straightforwardly evaluable[6] while there is an explicit protocol for how to apply the continuation[7].

This allows for the circuits built to be minimal: "Instead of building (sic) the whole Merkle tree in a single step..."[8] we allow for a proofs to be incrementally given and minimize circuit size.

The compilation of individual expressions to circuits is not much different from the compilation proposed by Gureev and Prieto-Cubides (2023) or Czajka (2023) via VampIR. The core difference lies in the IVC approach and its components for which Lurk (seemingly) was built.

3. Juvix interactions

Given the description above, we can consider how Lurk fits with the Juvix project in the ZKP language scene in two aspects:

1. As a backend
2. As a competitor

As Lurk is Turing-complete, it can evidently be used as a Juvix backend. However, there seems to be an evident way to use teh Lurk backend through Yatima[9]. Yatima is a compiler providing a Lurk backend to Lean4. While the project is not in active development, discussions with the Lurk team seems to indicate that it is still usable and is capable of compiling any Lean4 code which does not read/write to/from another file.

Viewing JuvixCore as a subcomponent of (Extensional) MLTT, we would be able to straightforwardly translate JuvixCore programs to Lean4 format. Implementing this will hence provide a direct way of using Lurk:

$$\begin{tikzcd}\text{JuvixCore} \arrow[r, hook] & \text{Lean4} \arrow[r, "\text{Yatima}"] & \text{Lurk}\end{tikzcd}$$

There are several caveats here, however: Yatima targets the Lisp version of Lurk and has quite an overhead. With respect to targeting the Lisp Lurk rather than the Rust version, the question is minor as syntax seems more or less interchangable.

In terms of competitors, while Lurk itself is not high-level enough to be a e.g. contract language, there are projects in development which might interest Anoma and Juvix team in particular.

Glow[10] is a fronend to Lurk offering an environment for smart contract creation. In this sense, it can be considered a Juvix alternative. However, the project seems to be in active development rather than in usage. The whitepaper is dated 2020, while there is also a grant proposal[11] dated 2021.

FVM[12] is a WASM based virtual machine in the Filecoin architecture that has proposed an integration with Lurk[13]. The documentation indicates that the integration is a work in progress and is aimed at verifying computational claims through Lurk.

4. Concluding remarks

There are three main reasons for paying attention to Lurk

1. It is a high-level interactive Turing-complete language for ZKP
2. In-built cryptographic commitments including (higher-order) functional once

[5]Specifically the (higher-order) function ones
[6]This is due to the fact that - as stated in the previous section - the current frame does minimal work
[7]For explicit compilation consult Lurk (2022a)
[8]https://www.youtube.com/watch?v=wKqiloOeogo
[9]https://github.com/lurk-lab/yatima
[10]https://github.com/Glow-Lang/glow
[11]https://github.com/filecoin-project/devgrants/issues/405
[12]https://github.com/filecoin-project/ref-fvm
[13]https://github.com/filecoin-project/devgrants/issues/808

3. The small-step evaluation paradigm allows for easy integration for IVS approach to ZKP, reducing circuit size and getting rid of recursion unrolling on compilation

While the language itself is quite useful, interactive, computationally fast as well as offering cryptographicly important build-ins, the core importance of Lurk seems to be that it is a Nova (and similar languages) front-end. That is, other than having functional commitments, it's core design is aimed at IVC evaluation.

Juvix can take steps to target Lurk as a back-end, however, that step seems justified if we are actively interested in using IVC methods, i.e. if we want to compile e.g. to Nova. In particular, instead of thinking of a frame-evaluation-like method for Juvix we can port a Juvix command to Lurk in a straightforward way and let it do the block-splitting work for succesfull IVC usage. However, note that the current backend integration for Lurk is also in early development. Hence, it may make sense to think of a direct compiler from Juvix to Noma instead.

Moreover, something to note is that there seem to be ongoing project for implementing ProtoStar[14] for Halo2 which seems to be integrating an IVC approach similar to one in Nova (taking inspiration from Sangria project, currently deprecated). Since VampIR already supports a Halo2 backend we can also consider upgrading VampIR syntax in the future to include recursive functions that are solely compilable with Halo2 backend.

References

Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. Lurk: Lambda, the ultimate recursive knowledge. Cryptology ePrint Archive, Paper 2023/369, 2023. URL https://eprint.iacr.org/2023/369. https://eprint.iacr.org/2023/369. (cit. on p. 1.)

Lurk. Lurk spec, 2022a. URL https://blog.lurk-lang.org/posts/circuit-spec/. (cit. on pp. 1 and 3.)

Lurk. A programmer's introduction to lurk, 2022b. URL https://blog.lurk-lang.org/posts/prog-intro/. (cit. on p. 1.)

Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021. URL https://eprint.iacr.org/2021/370. https://eprint.iacr.org/2021/370. (cit. on p. 3.)

Artem Gureev and Jonathan Prieto-Cubides. Geb Pipeline. *Anoma Research Topics*, August 2023. doi:10.5281/zenodo.8262815. URL https://doi.org/10.5281/zenodo.8262815. (cit. on p. 3.)

Lukasz Czajka. Juvix to VampIR Pipeline. *Anoma Research Topics*, August 2023. doi:10.5281/zenodo.8246535. URL https://doi.org/10.5281/zenodo.8246535. This document is based on Juvix v0.4.1, Geb v0.4.0, and VampIR v0.1.3. (cit. on p. 3.)

[14] https://eprint.iacr.org/2023/620.pdf