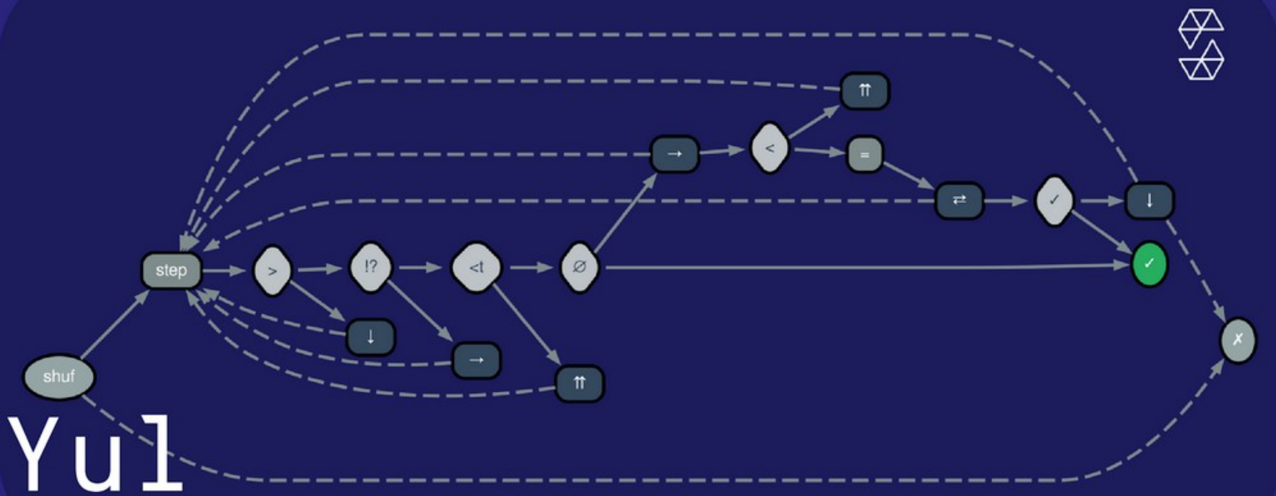
18 NOV 2025



SSA-CFG Yul

A new viaIR backend

clonker / Moritz Hoffmann

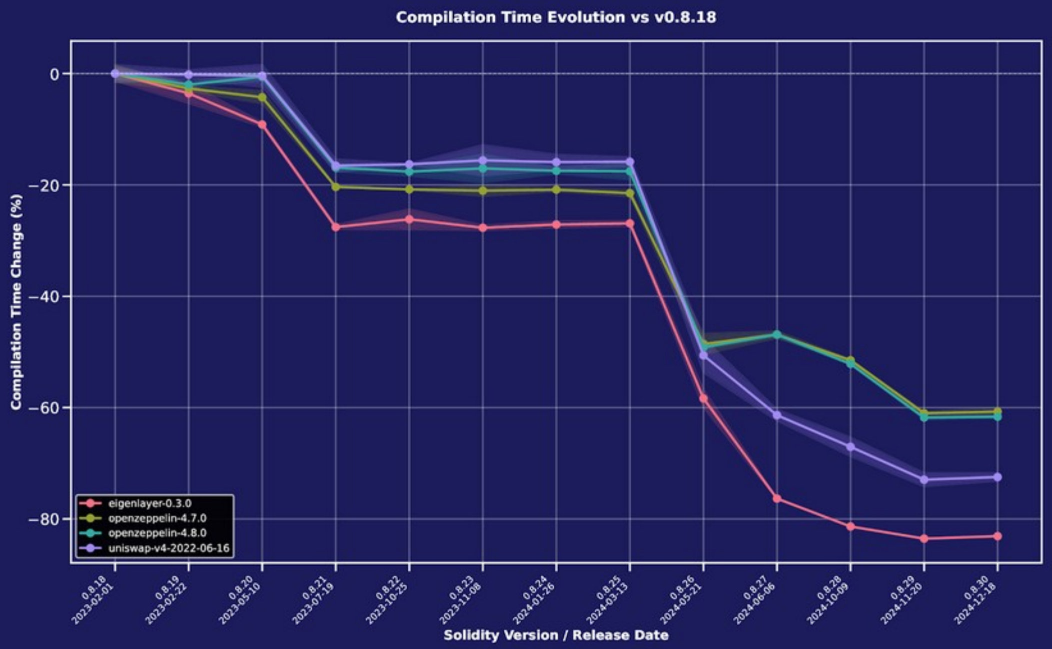
ToC

Performance of current viaIR

vialR over the versions and years

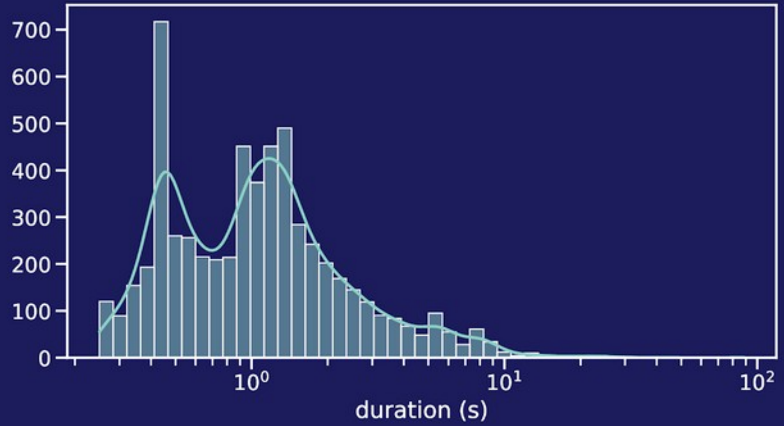
on v0.8.30:

- eigenlayer: $\sim 4.4\,\mathrm{m}$
- OZ v4.7.0: $\sim 24.7\,\mathrm{s}$
- OZ v4.8.0: $\sim 27.0\,\mathrm{s}$



Gathering some data

- compiled ~ 6k contracts with latest solc release (v0.8.30)
- 5.5% took longer than 5s
- individual contracts take minutes

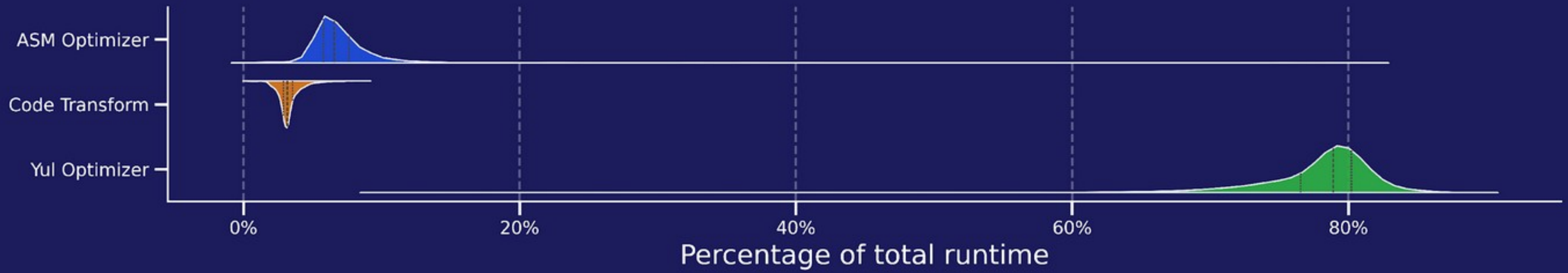



```
select * from compiled_contracts where compiler == 'solc' and
↪ version like "0.8.27%"
```

https://docs.sourcify.dev/docs/repository/sourcify-database/

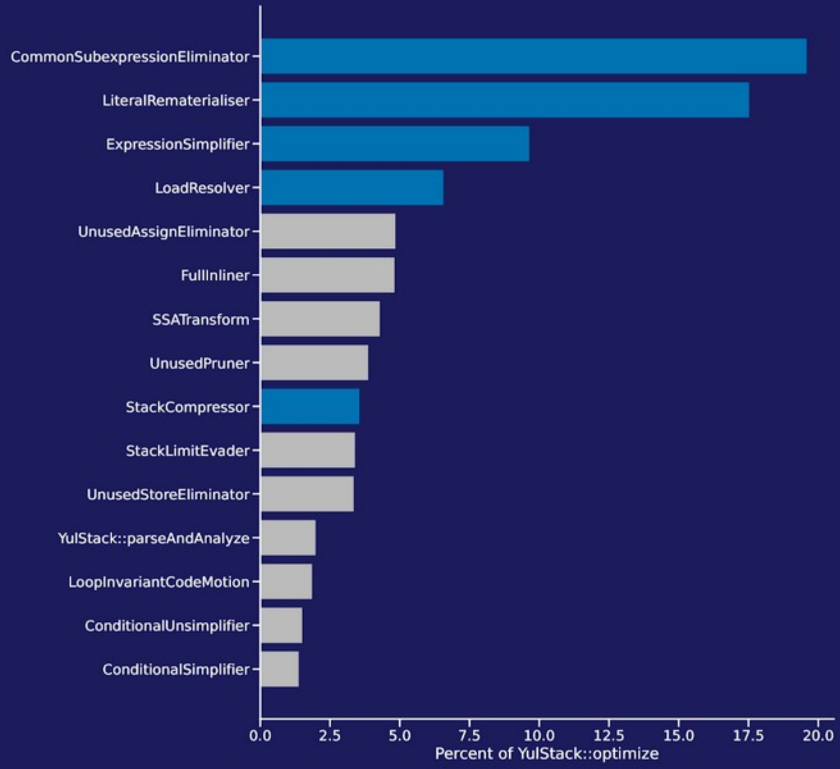
viaIR pipeline performance

```
solc --via-ir --optimize *args
CompilerStack::compile
YulStack::parseAndAnalyze
YulStack::optimize
YulStack::assembleEVMWithDeployed
OptimizedEVMCodeTransform::run
Assembly::optimize
bytecode
```



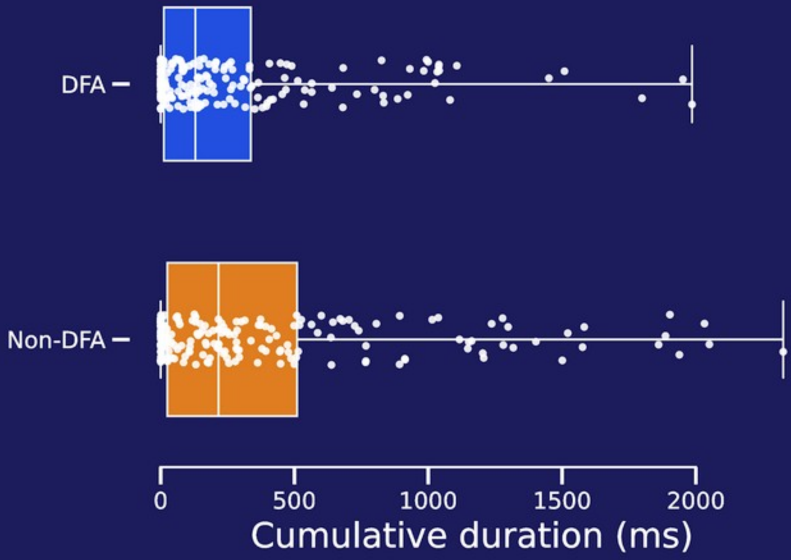
Yul optimizer performance: Steps depending on data flow

- Strongly depends on structure of input
- For eigenlayer v0.3.0--on my machine 5min--up to 60% of YulStack::optimize in data-flow dependent optimizer passes



Yul optimizer performance: Steps depending on data flow

- Strongly depends on structure of input
- For eigenlayer v0.3.0--on my machine 5min--up to 60% of `YulStack::optimize` in data-flow dependent optimizer steps
- But also: Cumulatively 30% more time spent in non data-flow dependent steps



What we learn from the performance data

- Eigenlayer v0.3.0: takes 5 minutes to compile.
- Bottleneck depends on specific AST, different steps of the optimizer can dominate runtime $\rightarrow$ need improvements across various steps.
- SSA form enables $\mathcal{O}(1)$ def-use queries and more efficient data-flow analyses; proven effective in LLVM/GCC.
- Gives opportunity to design and benchmark data structures and algorithms to finally enable fast viaIR compilation.
- Enables easier to maintain codegen and better stack-too-deep handling, too!

What is SSA-CFG?

SSA form

- Each variable is assigned exactly once.

non SSA

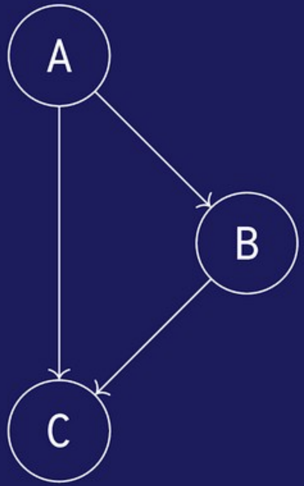
```
let y := 1
y := 2
let x := y
```

SSA

```
let v_1 := 1
let v_2 := 2
let v_3 := v_2
```

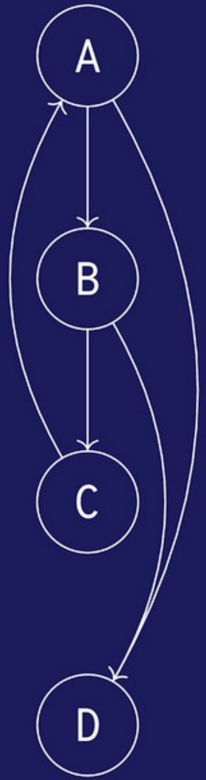
- Used in many SOTA compilers (LLVM, GCC, glslang, ...)
- Makes data-flow based optimizations more efficient

CFG



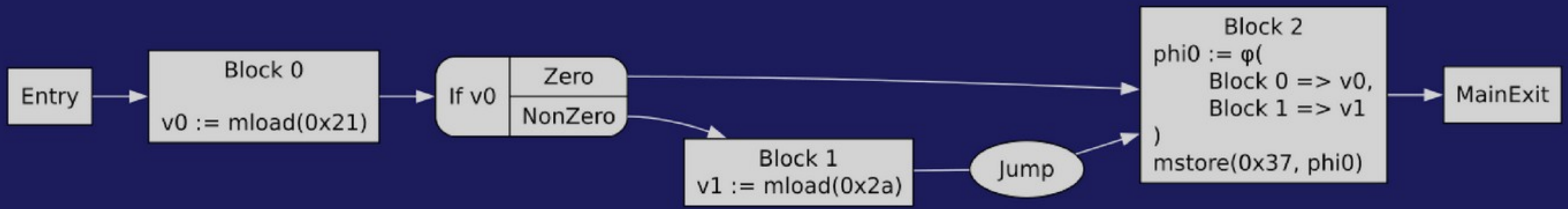

```
if (c) { ; } ;
```


```
for (;;) { ; } ;
```



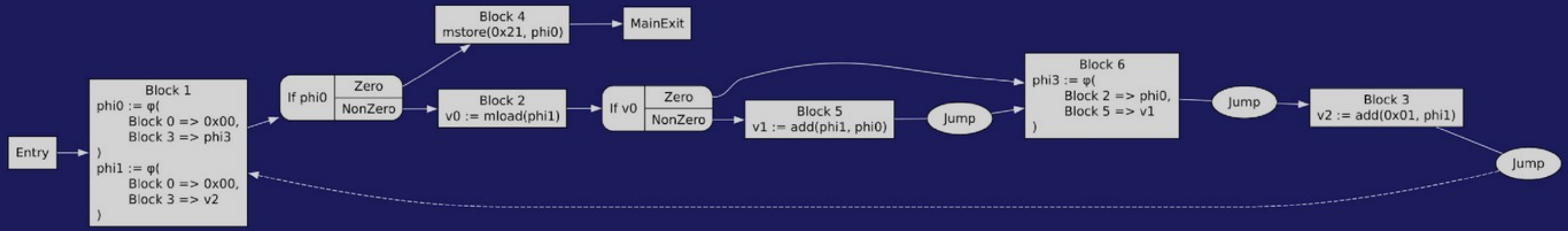
The Yul SSA-CFG: If

```
{
  let x := mload(33)
  if x {
    x := mload(42)
  }
  mstore(x, 55)
}
```



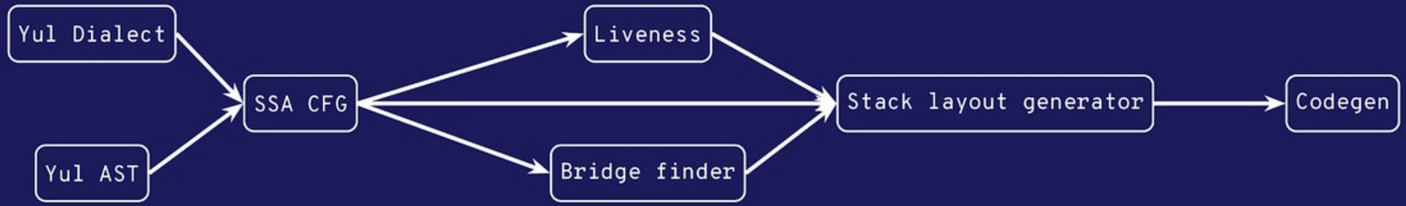
The Yul SSA-CFG: For

```
{
  let x
  for {let i := 0} x {i := add(i, 1)} {
    if mload(i) {
      x := add(x, i)
    }
  }
  mstore(x, 33)
}
```



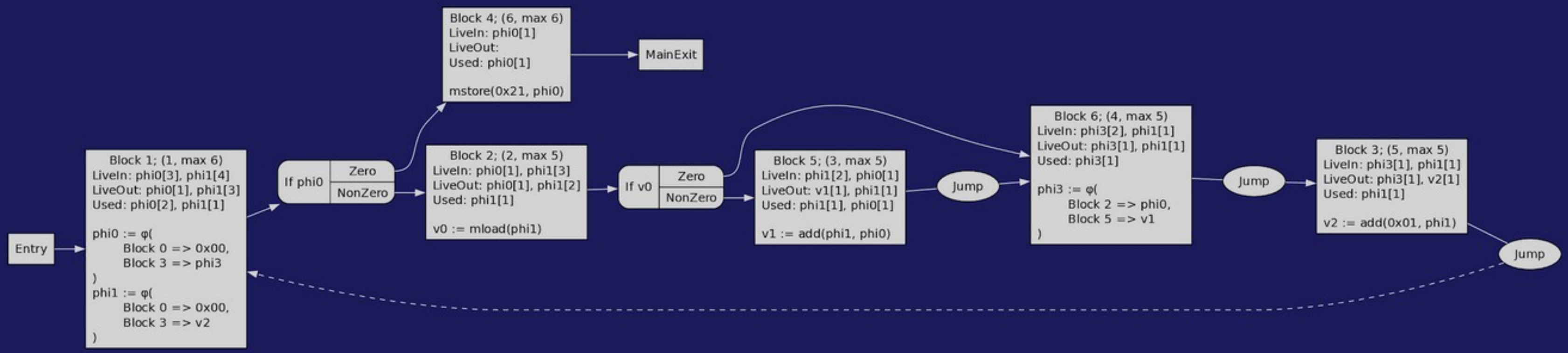
Codegen on SSA-CFG Yul

From Yul to SSA-CFG to bytecode



- Constructing the SSA-CFG based on
  Braun, M. et al. Simple and efficient construction of static single assignment form. In Compiler Construction: 22nd International Conference, CC 2013, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22 (pp. 102-122). Springer Berlin Heidelberg.
- Liveness analysis based on
  Rastello, F., & Tichadou, F. B. (Eds.). (2022). SSA-based Compiler Design. Springer.

Liveness example



Info on

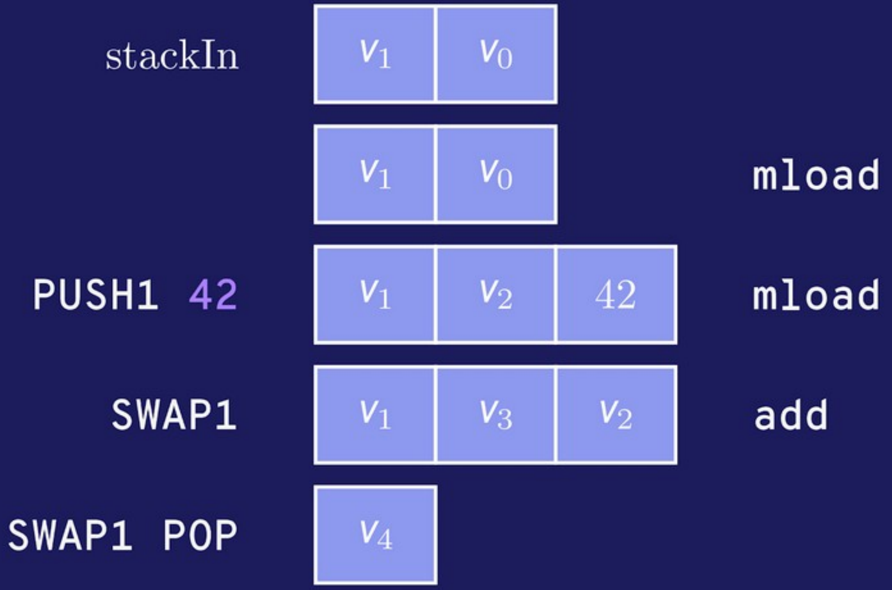
... what is live at which point in time,

... what is used in a block,

... how often something is used downstream.

Stack layout generation

Inside a block, for each operation, assign input layouts.

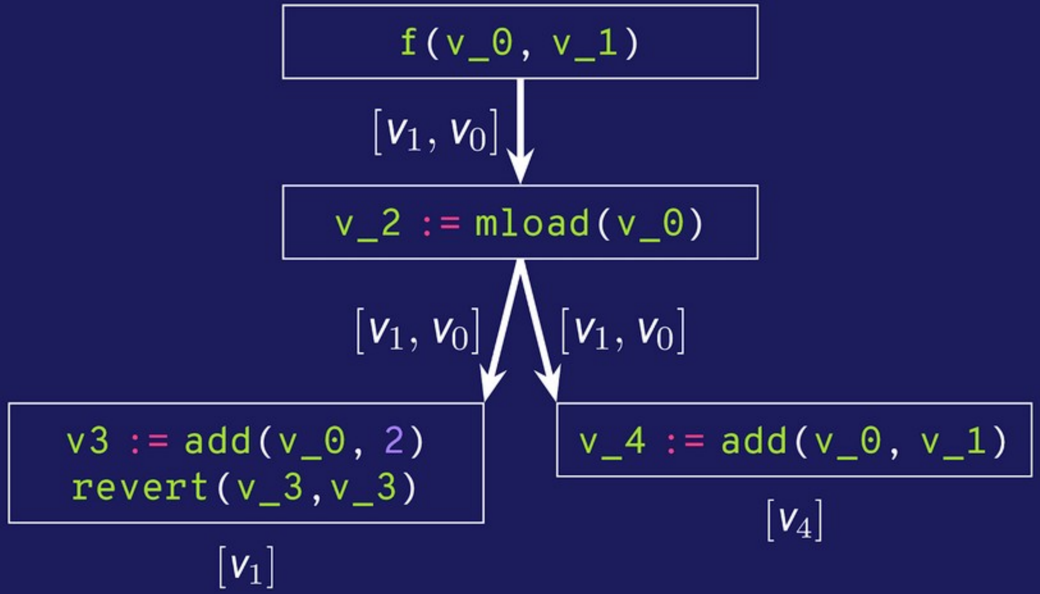



```
function f(v_1, v_0) -> v_4
    v_2 := mload(v_0)
    v_3 := mload(42)
    v_4 := add(v2, v3)
}
```

Stack layout generation

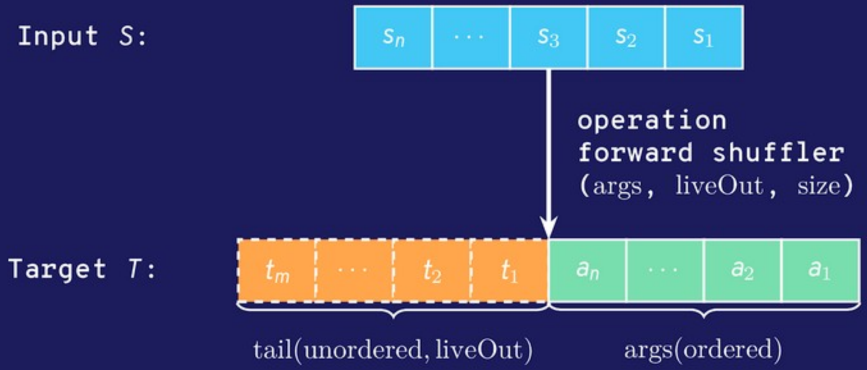
For each block, assign input/output stack layouts.

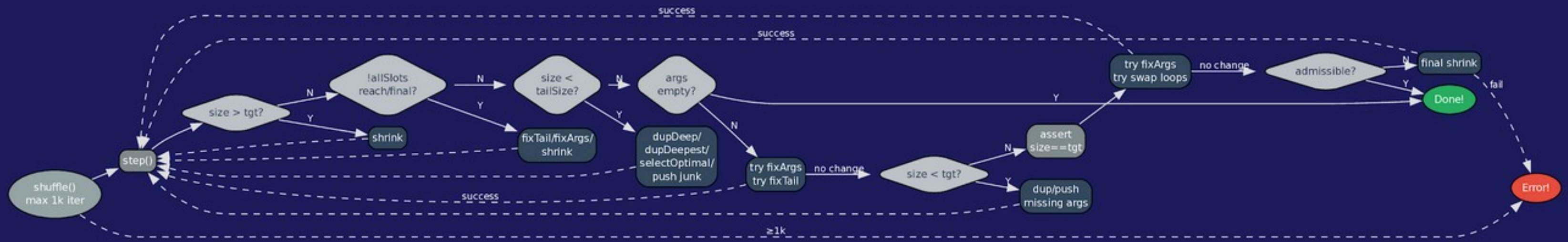
```
function f(a, b) -> c {
  if mload(a) {
    a := add(a, 2)
    revert(a, a)
  }
  c := add(a, b)
}
```



The greedy operation forward shuffler

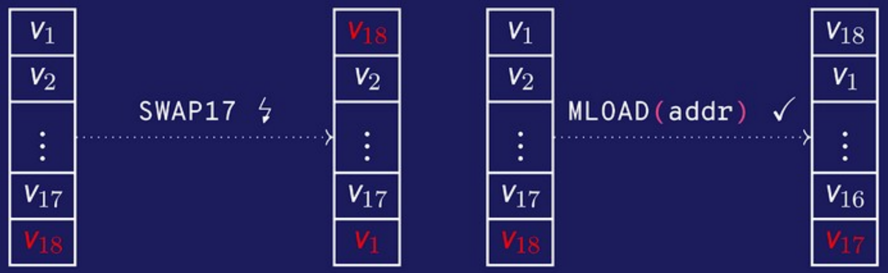
- For each operation, prepare an admissible stack
- Each step in the shuffler has to strictly improve the situation

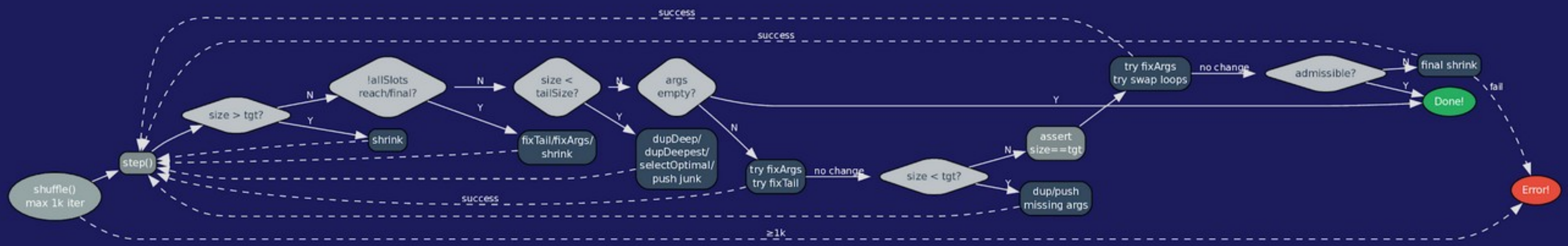




The greedy operation forward shuffler

- For each operation, prepare an admissible stack
- Fix stack-too-deep for unreachable operation arguments

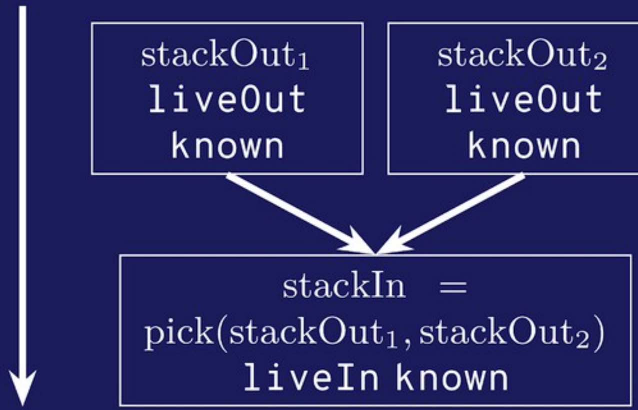




Comparison to current viaIR stack layout generation

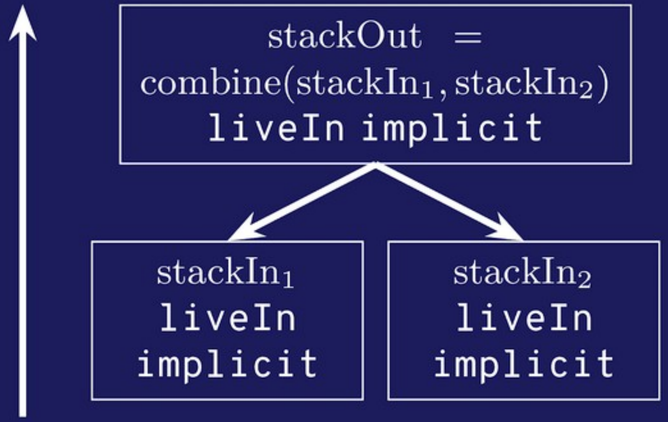
SSA-CFG viaIR

- top-down stack layout gen
- explicit liveness
- constructive control-flow joins



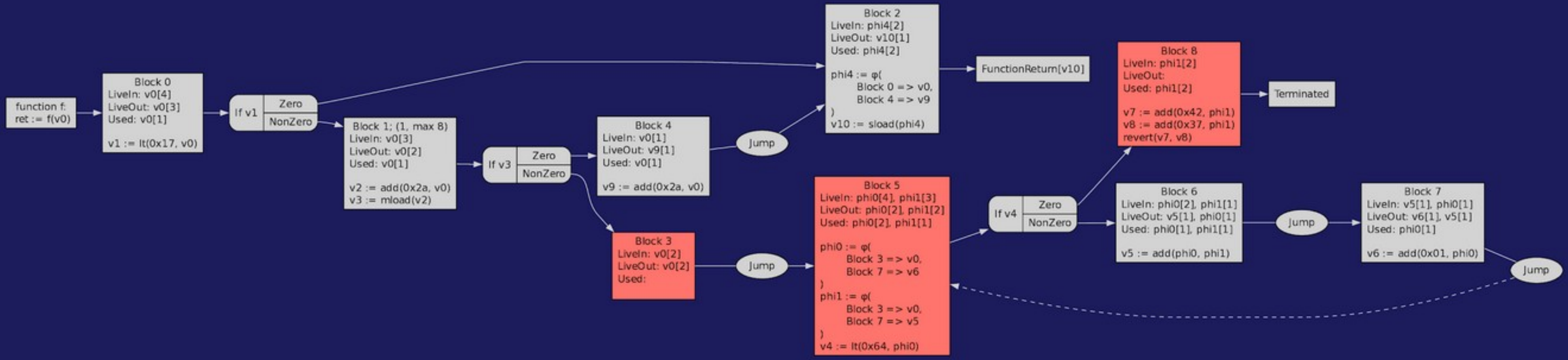
current viaIR

- bottom-up stack layout gen
- implicit liveness
- heap algorithm for control-flow joins

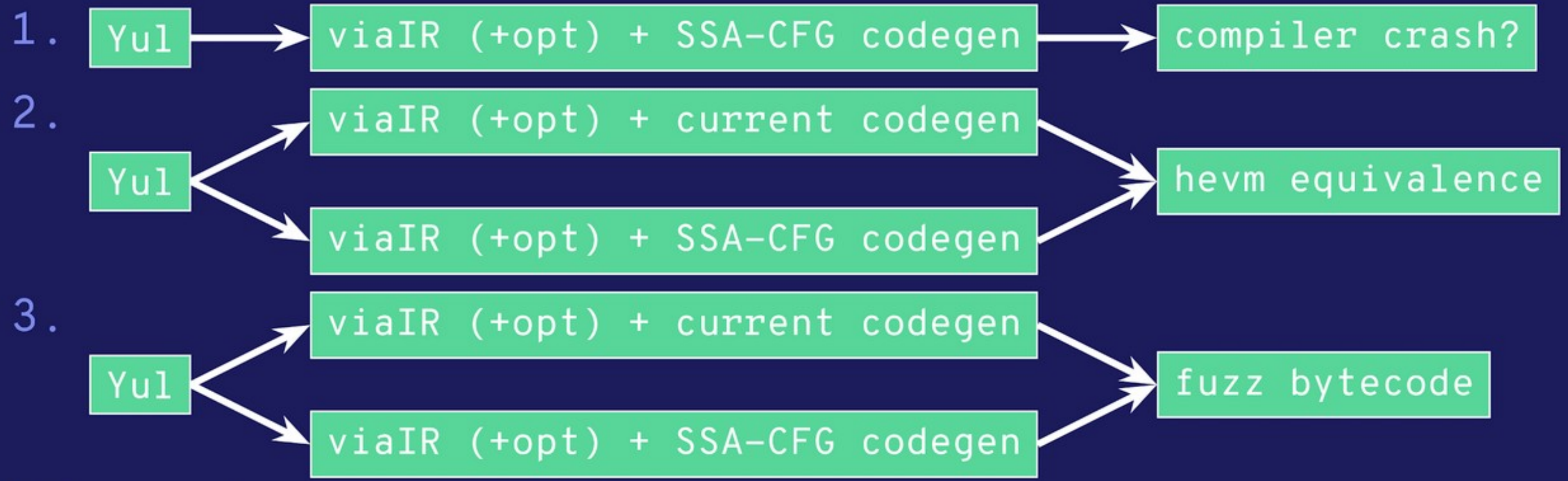


Terminating paths

- Opposing to user function returns, EVM termination cleans up whatever is left on stack. Saves POPs.
- When do we want to retain superfluous things on stack?
- Explicit in top-down SSA-CFG viaIR!



(Fuzz-)testing the implementation



Yul may be generated by

- producing random Solidity code,
- producing random Yul code directly,
- or by drawing it from, e.g., Sourcify.

Outlook

- Initial prototypical implementation without stack-too-deep handling almost ready to be released under experimental flag in solc.
- Bytecode performance on par and often even better than current viaIR.
- Time needed to compile comparable as the heavy optimizer passes are still part of the non-SSA bit of the pipeline.
- Next up:
  - Stack to memory moving
  - Stabilizing the current implementation through (fuzz) testing and extensive reviews
  - Implement data-flow heavy optimizer steps in the SSA-CFG representation to speed up compile time.

Q&A