Generating EVM bytecode from Yul in the new via-IR pipeline

Daniel Kirchner

Ethereum Foundation

ekpyron

daniel.kirchner@ethereum.org

- Yul is a thin, low-level abstraction over EVM opcodes.
- But it retains more structure than a mere list of opcodes:
  - Control-flow Structures: `if`, `for`, `switch`
  - User-defined Functions: `function f(a) -> r { leave }`
  - Variables: `let x := 42`
- No explicit jumps and no explicit stack manipulation (swaps, dups).

There is a trivial transformation from Yul to EVM assembly:

```
{
    let x := calldataload(0)
    if gt(x, 0x15) {
      revert(0,0)
    }
}
```

```
  0x00
  calldataload

  0x15
  dup2
  gt

  iszero
  tag_1
  jumpi

  0x00
  0x00
  revert

tag_1:
  pop
```

But is this optimal?

solc --strict-assembly

```
  0x00
  calldataload

  0x15
  dup2
  gt

  iszero
  tag_1
  jumpi

  0x00
  0x00
  revert

tag_1:
  pop
```

solc --strict-assembly --optimize

```
  0x15

  0x00
  calldataload

  gt

  tag_1
  jumpi

  stop

tag_1:
  0x00
  dup1
  revert
```

Goals

- Produce optimal code:
  - Avoid unnecessary swaps and dups.
  - Efficient arrangement of basic blocks.
- Efficiently use the stack to avoid stack-too-deep errors.
- Simple and robust:

  Avoid introducing bugs due to complex transformations.

Goals

- Produce optimal code:
  - Avoid unnecessary swaps and dups.
  - Efficient arrangement of basic blocks.
- Efficiently use the stack to avoid stack-too-deep errors.
- "Simple" and <u>robust</u>:

  Avoid introducing bugs due to complex transformations.

Approach

- Step 1 [simple]:

  Generate a *Control Flow Graph*.

- Step 2 [complex]:

  Algorithmically generate the full *stack layouts*

  between blocks and between operations.

- Step 3 [simple]:

  Generate concrete opcodes following the stack layouts of Step 2.

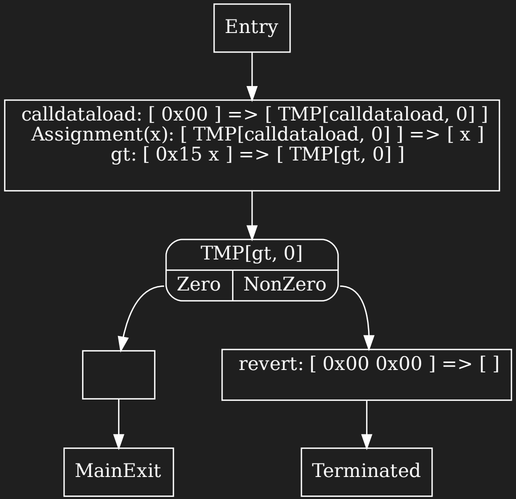
  Can validate correctness relative to Step 1!

Step 1: Generate a Control Flow Graph

- Similar to naive direct opcode generation.
- Split into basic control flow blocks.
- Split basic blocks into sequences of operations with inputs and outputs.
- Simple and bug-resistent.

Step 1: Control Flow Graph

```
{
    let x := calldataload(0)
    if gt(x, 0x15) {
      revert(0,0)
    }
}
```

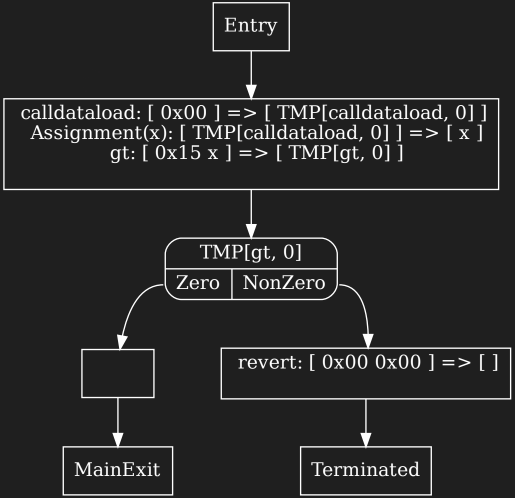


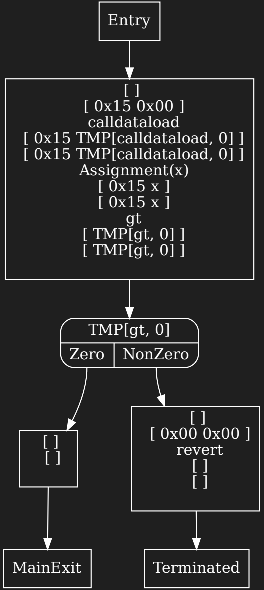
Step 2: Algorithmically generate full Stack Layouts.

Generate the full stack layouts (relative to the base of the current Yul function) for entering and exiting basic control flow blocks and between operations.

- Generate stack requirements "backwards" and "bottom-up".
- Maintain optimal locations for variables on stack.
- Consolidate stack layouts on control flow joins.
- Avoid the necessity for access to deep stack slots.
- Complex transformation!

Step 2: Stack Layouts





Step 2: Stack Layouts

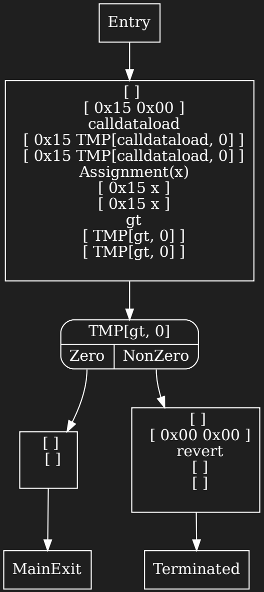
```
calldataload:
  [ 0x00 ] => [ TMP[calldataload, 0] ]
Assignment(x):
  [ TMP[calldataload, 0] ] => [ x ]
gt:
  [ 0x15 x ] => [ TMP[gt, 0] ]
```

```
[ 0x15 0x00 ]
calldataload
[ 0x15 TMP[calldataload, 0] ]
[ 0x15 TMP[calldataload, 0] ]
Assignment(x)
[ 0x15 x ]
[ 0x15 x ]
gt
[ TMP[gt, 0] ]
[ TMP[gt, 0] ]
```

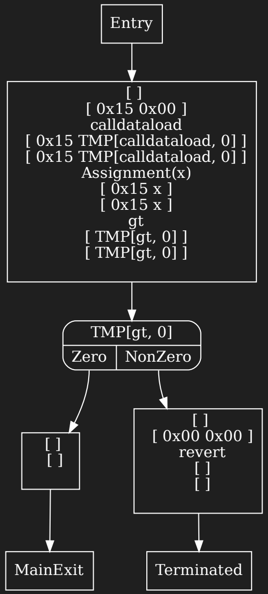
Step 3: Generate concrete opcodes following the stack layouts.

- Maintain a symbolic copy of the current stack.
- Traverse the Control Flow Graph.
- Generate the stack layouts at each block transition and between all operations.
- Validate the sanity of the current stack at each operation and each control flow transition.

Step 3: Generate EVM Assembly.



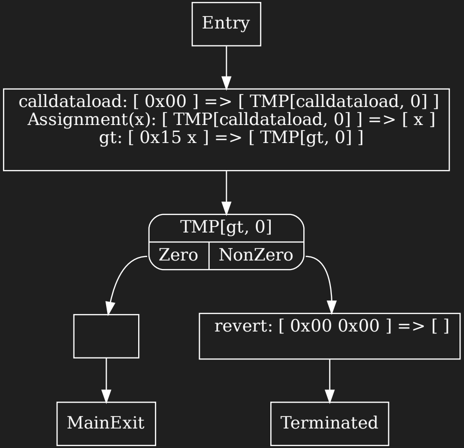

```
/* stack = [] */
0x15
0x00
/* stack = [ 0x15, 0x00 ]; Validate stack top */
calldataload
/* stack = [ 0x15, TMP[calldataload, 0] ] */
/* Assignment to x => stack = [ 0x15, x ] */
/* Create stack layout required for gt operation [no-op] */
gt
/* stack = [ TMP[gt, 0] ] */
/* Create block exit layout [no-op] */
/* Block exit is conditional jump; validate condition */
tag_1
jumpi
 ...
```



Step 3: Generate EVM Assembly.

```
  ...
  jumpi
  /* stack = [] */
  stop
tag_1:
  /* stack = [] */
  /* Create stack layout for revert operation. */
  0x00
  dup1
  /* stack = [ 0x00, 0x00];
   * Validate stack top for operation */
  revert
```

Step 3: Generate EVM Assembly.

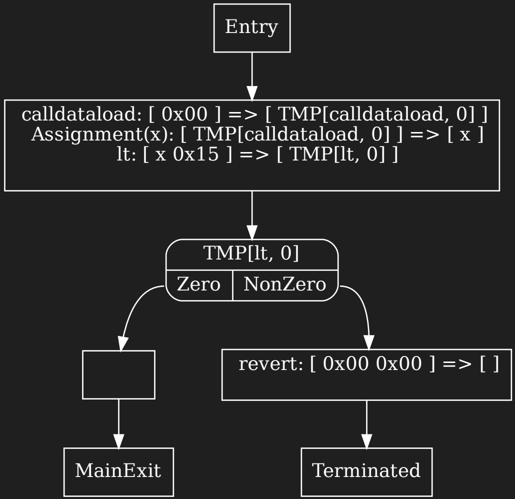



```
/* stack = [] */
0x15
0x00
/* stack = [ 0x15, 0x00 ] */ √
calldataload
/* stack = [ 0x15, TMP[calldataload, 0] ] */ √
/* Assignment to x */
/* stack = [ 0x15, x ] */ √
gt
/* stack = [ TMP[gt, 0] ] */ √
tag_1
jumpi
 ...
```

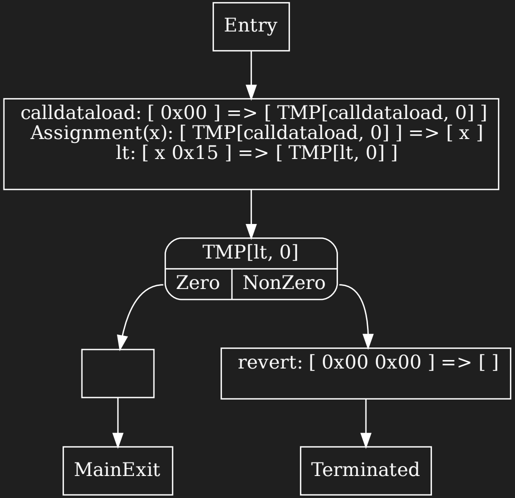
Minor changes can have non-local effects!

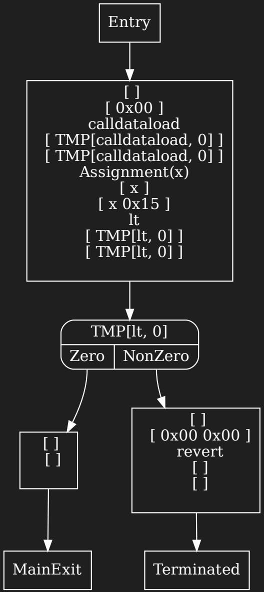
Step 1: Control Flow Graph

```
{
    let x := calldataload(0)
    // if gt(x, 0x15) {
    if lt(0x15, x) {
      revert(0,0)
    }
}
```

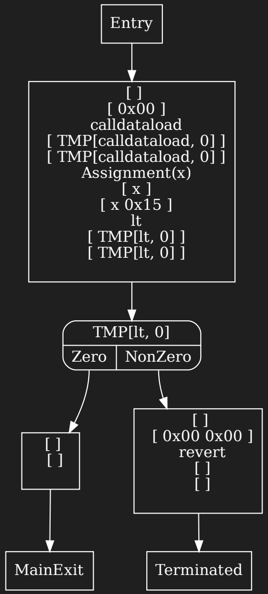


Step 2: Stack Layouts





Step 3: Generate EVM Assembly.




```
/* stack = [] */
0x00
/* stack = [ 0x00 ]; Validate stack top for operation */
calldataload
/* stack = [ TMP[calldataload, 0] ] */
/* Assignment to x => stack = [ x ] */
/* Create stack layout required for lt operation */
0x15
/* stack = [x, 0x15] */
lt
/* stack = [ TMP[lt, 0] ] */
/* Create block exit layout [no-op] */
/* Block exit is conditional jump;
     validate condition is on stack top */
tag_1
jumpi
...
```

Stack Shuffling

- The approach requires a general stack shuffling algorithm.
- Any stack layout may contain duplicates and arbitrary slots (JUNK).
- Greedy algorithm that iteratively chooses the best operation.
- Performs the shuffling on a symbolic representation of the stack.
- Can verify correctness.
- Can precisely report unreachable slots.

Advantages

- Validity of the Stack Layouts can be verified.
- Stack Layout Generation has full freedom for complex transformations.
- The Stack Layout Generator can react to Stack-Too-Deep situations and try to resolve them.
- If unsuccessful, the Stack Layout Generator can report a precise set of candidate variables for moving to memory.

Advantages

- Gas reduction by several percentage points.
- Reduction in code size.
- `solc --via-ir --optimize`

  should (almost) never suffer from stack-too-deep errors anymore.

Disadvantages

- Less predictable byte code.
- But we can output the full stack layout at each proper operation as debug data!

  We just need to coordinate defining a format for that.

@Tooling and @Debuggers

- DO NOT try to reproduce this by hand.
- DO NOT try to employ heuristics on this.
- Whatever you do, it WILL break.
- Let's really define a proper debugging format instead.

Thank you!