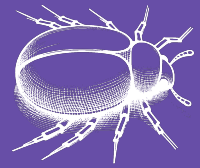
debugging data for ethereum

ethdebug format :: overview and status



devcon7 🇹🇭 กรุงเทพฯ

debugging ethereum today requires guesswork

more compiler output
would help tremendously
… but *what* output?

project intro

ethdebug.github.io/format

the ethdebug project is designing a debugging data format for compilers to output and for debuggers to read

currently funded by EF

part of `Argot Collective` spin-out

not building a debugger (yet?)

about me

hi, i am g. nick! gnidan.eth

ethdebug project lead

argot collective member

i once built a solidity debugger
(and was responsible for its development for 5+ years)

chocolatey past

other dev tools too sometimes idk

project goals

create a universal format

not just for Solidity and not just source maps, we aim to support all languages and use cases

enable real-life debugging

debugging in dev is great, but don't you also want to know why that money disappeared?

optimize for adoption

we know this stuff is complicated, so we take docs and reference impls. seriously!

lower the cost of blockchain insight

barely-decent debuggers cost millions of dollars to build, and that's just one first step towards human comprehension

making debugging tools today

this has happened [or is happening] dozens of times:

1. spend weeks of engineering time bashing solc in weird ways
2. think you understand how something works
3. implement the same behavior and hope it's right
4. fix it later when solc changes or you find out you were wrong
5. repeat for lifetime of project

now, what about Vyper? other future important languages?
software tools for languages supporting billions of dollars of assets are a **whole-ecosystem concern**.

string smaller than wordsize

storing `"hello world"`:

```
  ...
[...0000]: 68656c6c6f20776f726c640000000000000000000000000000000000000016
  ...
```

string larger than wordsize

storing `"solidity storage is a fun lesson in endianness"`:

```
  ...
[...0001]: 000000000000000000000000000000000000000000000000000000000000005d
  ...
[...0cf6]: 736f6c6964697479207374726167652069732012066756e206c6573736f6e
[...0cf7]: 20696e20656e6469616e6e6573730000000000000000000000000000000000
  ...
```

full gist with other examples

my favorite example: Solidity has two *entirely different* ways to store a string!

format deep dive

(i.e., the technicals)

compiler-debugger interaction model

compiler behavior

while executing the translation pipeline from high-level source materials into bytecode instructions, compilers keep track of translation details and emit these **per-instruction**, e.g., *"this instruction came from this source range"* or *"after this instruction, it is safe to read the* `balances` *variable allocation"*.

debugger operation

debuggers observe each instruction as it executes in a running EVM, then find corresponding translation details from compiler output, effectively reducing each pair *(machine instruction, compiler annotations)* as a state transition inside a coherent high-level language model.

some challenges

the compiler

debuggers do dynamic analysis: they must concern themselves with runtime.

unfortunately, compilers are very limited here: debuggers will have to track stack-height among other things.

the optimizer

a good optimizer will use many techniques to reduce the cost to deploy and interact with a smart contract.

techniques such as bytecode deduplication means the potential for ambiguity.

software artifacts so far

- formal JSON schemas for ~60% of our currently-understood data model
- examples for every schema, validated by automated tests
- top-down “explainer” documentation for main schemas
- reference implementation for reading variables from the machine, with accompanying end-to-end integration tests and thorough documentation

current schema drafts

(an incomplete list)

ethdebug/format/program

annotate a contract's bytecode instruction by instruction

[30% complete]

ethdebug/format/type

represent built-in & user-defined types for debuggers to display

[initial draft with known outstanding changes]

ethdebug/format/pointer

describe data allocation in terms of arbitrary runtime state

[draft + implemented for debugger reference]

program schema

high-level state info for every bytecode instruction

ethdebug format

Explore | View source | Playground

ethdebug/format/program

type : object

- **compilation** Compilation reference by ID
- **contract** object **required**
- **environment** Bytecode execution environment **required**
- **context** ethdebug/format/program/context
- **instructions** (ethdebug/format/program/instruction)[] **required**

Example values :

program schema key concepts

- one program, one bytecode
- programs contain a list of instruction annotations
- instruction annotations describe high-level context
- debuggers step the machine and lookup each instruction
- instruction annotation data informs the high-level state

```
{
  "offset": 0,
  "operation": { "mnemonic": "PUSH1", "arguments": ["0x60"] },
  "context": {
    "code": {
      "source": { "id": 5 },
      "range": { "offset": 10, "length": 30 }
    },
    "variables": [{
      "identifier": "x",
      "declaration": {
        "source": { "id": 5 },
        "range": { "offset": 10, "length": 56 }
      },
      "type": {
        "kind": "string"
      },
      "pointer": {
        "location": "storage",
        "slot": 0
      }
    }]
  }
}
```

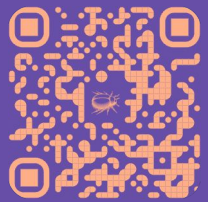
example instruction object

this warning is probably worth an entire slide:

although we believe our approach to be viable for optimized code, the program schema is still quite incomplete and many assumptions remain untested

type schema

describe the different kinds of things



ethdebug format

Root schema

Explore | View source | Playground

▸ ethdebug/format/type

Elementary type schema

Explore | View source | Playground

▾ ethdebug/format/type/elementary

type : object

▸ **kind** Known elementary kind **required**

Canonical representation of an elementary type

polymorphic discriminator The value of the **kind** field determines which sub-schema applies:

uint | **int** | **bool** | **bytes** | **string** | **ufixed** | **fixed** | **address** | **contract** | **e**

type schema key concepts

- types are known or unknown
- types are elementary or complex
- complex types contain other types
- types can be referenced by ID or represented wholly
- user-defined types include reference to source definition

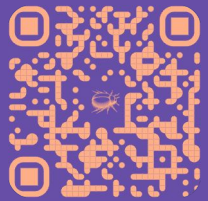
```
{
  "kind": "fixed",
  "bits": 256,
  "places": 10
}
```

```
{
  "kind": "array",
  "contains": {
    "type": {
      "kind": "uint",
      "bits": 256
    }
  }
}
```

example elementary and complex types

pointer schema

compile-time representations for finding variables at runtime



ethdebug format Documentation Specification ⚠ INCOMPLETE DRAF

ethdebug/format/pointer › Schema

Schema

Explore View source Playground

ethdebug/format/pointer

type : object

Example values :

Example 0 Example 1 Example 2 Example 3 Example 4

```
{
  "location": "storage",
  "slot": 2
}
```

A schema for representing a pointer to a data position or a range of data positions in the EVM.

An **ethdebug/format/pointer** is either a single region or a structured collection of other pointers.

If Then Else

type : object

location any **required**

oh no, a warning already

Greenspun's tenth rule [revised for ethdebug]

3 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Greenspun's tenth rule of programming is an aphorism in computer programming and especially programming language circles that states:[1][2]

JSON Schema

> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

... we're almost done, please don't leave

lambda calculus in JSON, really?

unfortunately, there is a big compile-time constraint

compilers know *how* they allocate data but not always precisely *where* (e.g., there's no "free memory pointer" until runtime). this quickly becomes categorically insufficient for finding dynamic allocations or complex structures.

we must describe data allocations as expressions with unbound terms whose values are resolved only upon observing a running EVM.

```
{
  "location": "storage",
  "slot": 2
}
```


example pointer

```
{
  "define": { "string-storage-contract-variable-slot": 0 },
  "in": {
    "group": [{
      "name": "length-flag",
      "location": "storage",
      "slot": "string-storage-contract-variable-slot",
      "offset": { "$difference": ["$wordsize", 1] },
      "length": 1
    }, {
      "if": {
        "$remainder": [
          { "$sum": [{ "$read": "length-flag" }, 1] },
          2
        ]
      },
      "then": {
        "define": {
          "string-length": {
            "$quotient": [{ "$read": "length-flag" }, 2]
          }
        },
        "in": {
          "name": "string",
          "location": "storage",
          "slot": "string-storage-contract-variable-slot",
          "offset": 0,
          "length": "string-length"
        }
      },
      "else": {
        "group": [{
          "name": "long-string-length-data",
          "location": "storage",
          "slot": "string-storage-contract-variable-slot",
          "offset": 0,
          "length": "$wordsize"
        }, {
          "define": {
            "string-length": {
              "$quotient": [
                { "$difference": [{ "$read": "long-string-length-data" }, 1]
                2
              ]
            },
            "start-slot": {
              "$keccak256": [{
                "$wordsized": "string-storage-contract-variable-slot"
              }]
            },
            "total-slots": {
              "$quotient": [
                {
                  "$sum": [
                    "string-length", { "$difference": [ "$wordsize", 1 ] }
                  ]
                },
                "$wordsize"
              ]
            }
          },
          "in": {
            "list": {
              "count": "total-slots",
              "each": "i",
              "is": {
                "define": {
                  "current-slot": {
                    "$sum": ["sta
                  },
                  "previous-length": {
```

see this example with documentation and comments

just kidding. here's the example you were afraid of.

it's less bad than you think.

all `string storage` (e.g.) values just reuse this same static template, and compilers don't need to output unused pointers.

pointer implementation guide for debuggers.

it's written for humans!

we were apprehensive too, so we tested it.

because this area of the format is so complex, we made sure to author a "literate programming" style reference implementation and accompany this implementation with thorough integration tests.

we hope that this kind of implementation guide will enable debugger authors/maintainers to adopt this format with minimal difficulties!

closing points

there's lots of work still to be done, but we're quite proud to present the current state of the effort.

interested in learning more or contributing? join our Matrix.chat and watch the repository!

Matrix.chat

Github

thank you for your time 🙏

ขอบคุณครับ

khob khun khrap