18 NOVEMBER 2025

From Classic to Core Solidity

convergence and interoperability prospects

ABOUT ME

Kamil Śliwak

Solidity Compiler Team,
Argot Collective

- On the team since 2020.
- Formerly Solidity co-lead, now acting as project's facilitator in Argot's flat structure.
- Current focus: driving Classic Solidity design and preparing for the Core Solidity transition.

Context: read our recent blog posts

https://blog.solidity.org

- The Road to Core Solidity
- Core Solidity Deep Dive
- SSA-CFG Yul (TBA)
- From Classic to Core Solidity (TBA)

Motivation: avoid the infamous Python 2/Python 3 situation

- Full backwards-compatibility means keeping legacy cruft in the language, but breaking changes lead to upgrade friction.
- Did we strike the right balance?
- What tools do we have at our disposal to avoid spending 10 years in transition?

Talk agenda

1. Syntactic sugar
2. Syntax convergence
3. Automatic translation
4. Interoperability

Syntactic sugar

SAIL

Expressive type system

- generics
- type classes
- algebraic data types
- first-class functions
- type inference

Minimal features

- no contracts as objects
- no events or errors
- no arithmetic operators
- not even loops
- only one built-in value type: *word*
- heavy reliance on assembly

Core Solidity

Core Solidity = SAIL + stdlib + sugar

- All SAIL syntax is valid Core syntax.
- Standard library written in SAIL (Yul utility code that used to be generated).
- High-level abstractions known from Classic Solidity are now just syntactic sugar.

Core syntax is **very** close to Classic Solidity

- Most of the features from Classic Solidity are replicated exactly.
- Deliberate omissions: inheritance, try/catch, libraries.
- Superficial syntax differences: postfix types, ternary operator, function types.
- Explicit use of standard library.

Sugaring is powerful

Designed specifically to reproduce Classic Solidity syntax as-is. We could rebuild Classic Solidity on top of SAIL if we wanted to. But should we?

Too much sugar is not healthy

Core Solidity is just a means to an end. There are long-standing issues in the language that we must fix.

We would be breaking the syntax sooner or later. The transition is the best moment to lose some old baggage. Rebuilding flawed features just to drop them is counterproductive.

Convergence

Significant breaking changes are not a new thing in Solidity

Past examples:

- 0.5.0
  - Removal of the *var* keyword.
  - Strict scoping of variables.
  - Switch from loose assembly to strict assembly.
- 0.8.0
  - Checked arithmetic by default.
  - Strict conversions.

Language convergence through breaking releases

- Backporting the tedious but superficial Core Solidity changes to Classic Solidity:
  - postfix types
  - ternary operator and function type syntax
  - stub of the standard library
- Is a more gradual transition a good thing though?
  - Less pain, but longer.
  - We lean towards getting it over with quickly.

Prefix vs postfix types

prefix notation:

address payable a;

postfix notation:

a: address payable;

Ambiguity: *S[3];*

array of 3 elements of type *S*?

4th element of an array called *S*?

Parsing ambiguity

Cannot always determine if an identifier is a type without extra information.

In Solidity this affects type conversions, type args of builtins, in Core also optional type annotations.

Current solution: don’t accept complex types there.

Other solutions

Lexer hack: analysis info fed back to the parser. Complex and frowned upon in modern languages.

Postfix types: clean separation of type and non-type context.

Backporting postfix types would allow lifting some restrictions.

Automatic translation

AUTOMATIC TRANSLATION

Good at tedious superficial changes

- renamed builtins and keywords
- postfix types
- try/catch (maybe)

Bad at semantic and complex changes

- strict conversions (0.8.0)
- associativity of exponentiation (0.8.0)
- removal of inheritance

solidity-upgrade

Dropped in 0.8.18 (2023).

Could not completely automate migration between our breaking releases. Not useless, but only takes you halfway there and maintenance cost is high.

The nature of the transition (postfix types) may make it worth it.

Interoperability

Level 0: ABI-level interoperability

- Calling external functions across languages.
- Composability is a baseline expectation of the platform.

Level 1: Common compilation interface

- Arbitrary mix of Classic and Core Solidity files as input.
  - *pragma core solidity*
  - No cross-language imports.
- Single Standard JSON output - transparent to dev tooling.
- Internally: two independent pipelines.

Level 2: Function-level interoperability

- Cross-language imports
  - free functions
  - interfaces
  - events and errors
  - (file-level) constants
  - subset of types
- Internal function calls
- Deploying contracts with *new*.

Prerequisites:

- Same calldata, memory and storage layout for the shared types.
- Same cleanup rules for function parameters.
- Same mutability guarantees in pure and view functions.
- Same memory layout and memory management.

Challenges:

- Code bloat (independent utility code).
- Called function is a black box.
- Linking at Yul level.

Conclusions

Key takeaways

- Sugaring will take us most of the way there.
- Interoperability up to function level will help projects make a gradual transition.
- There are ways to go deeper if need be, with various trade-offs.

Q&A

Thank you!