Smart Contracts on Concordium

CONCORDIUM

WebAssembly low-level language

Concordium's core on-chain language is WebAssembly (Wasm), a portable, well-defined assembly-like language. Wasm is an internet standard that is gaining a lot of traction in recent years and is already supported in major web browsers.

Many programming languages can already be compiled into Wasm, which potentially enables support of a large range of smart contract languages. Wasm allows for low-level control of the on-chain code, which helps with optimizations when adding support for cryptography in smart contracts.

Among permissionless blockchain platforms, there are very few common standards for smart contracts, however, Wasm is one of the few that is seeing adoption by multiple platforms.

Rust high-level language

Concordium uses Rust as its first high-level smart contract language. Compared to WebAssembly, Rust focuses on providing better ergonomics and making smart contract development more accessible. It incorporates many attractive features based on years of research in programming languages: an expressive type system, type inference, and pattern-matching. Combined with safe low-level resource control through special features of the type system, this gives a safe language allowing for zero-cost abstractions. The low-level resource control helps to carefully design smart contracts saving on execution costs.

The number of companies that use Rust to write safe and performant code is growing. The Rust ecosystem is quite friendly, with good documentation and good support for WebAssembly. Many high-quality libraries exist that can be used off-the-shelf and compiled into WebAssembly.

Ultimately, any language that is able to compile to WebAssembly will be able to target the Concordium chain.

Libraries, Tools, and Documentation

Concordium has a strategy to use existing programming languages with a strong community of developers to lower the threshold for many developers to start working with smart contracts. Many existing Rust development tools can already be used to write smart contract code for Concordium. Concordium provides the `cargo-concordium` tool that allows developers to extend the standard Rust package manager `cargo` with Concordium-specific commands for building and testing smart contracts. Moreover, the tool also allows for generating contracts from pre-defined templates. For example, it is possible to generate an NFT contract and tailor it to the user's needs.

The Concordium standard library `concordium-std` provides support for smart contract development in Rust and exposes a high-level API that smart contract writers can use, alleviating them from the need to deal with low-level details of how the interaction with the chain works. Developing token smart contracts is supported by `concordium-cis2`, which provides the standard interface for tokens compliant with the Concordium CIS2 token standard. Smart contracts use *schemas* that give a unified structured description of binary data. Schemas are used for integrating with off-chain tools and displaying data in a human-readable format.

The Concordium libraries and tools provide a testing framework for extensive off-chain testing of smart contracts. The framework features unit testing and randomized property-based testing.

The smart contract development framework is extensively documented. The documentation features tutorials, best practices, technical details, onboarding guides for developers coming from other blockchains, and other aspects of smart contract development.

The development is open source and available on GitHub.

Formal Verification

Smart contract development involves many risks that do not show up in, for example, web programming: the cost of mistakes is very high, and possibilities for fixing bugs are limited. This requires paying close attention to smart contract quality.

Concordium works in collaboration with Aarhus University, Denmark to develop theories and tools for providing high correctness guarantees. The formal verification process involves developing mathematical models of smart contract behavior and providing mathematical proof of correctness [NS19]. One of the outcomes of the collaboration is the *ConCert smart contract verification framework* [ANS20]. As part of this development, several examples of smart contracts were formally verified. Randomized testing techniques were employed in ConCert for discovering vulnerabilities in smart contracts [MNAS22]. Further research was directed on how to formally connect the mathematical model to Rust implementation [AMNS22]. The framework also helped to shape Concordium token standards. Currently, our internal science team is working on integrating the verification framework with the Concordium infrastructure. Such integration will enable the use of formal verification techniques by smart contract developers.

References

[NS19] Jakob Botsch Nielsen, Bas Spitters. Smart Contract Interactions in Coq. 1st Workshop on Formal Methods for Blockchains, 3rd Formal Methods World Congress, 2019. https://doi.org/10.1007/978-3-030-54994-7_29

[ANS20] Danil Annenkov, Jakob Botsch Nielsen, Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2020. https://dx.doi.org/10.1145/3372885.3373829

[MNAS22] Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding Smart Contract Vulnerabilities with ConCert's Property-Based Testing Framework. 4th International Workshop on Formal Methods for Blockchains, 2022. https://doi.org/10.4230/OASIcs.FMBC.2022.2

[AMNS22] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, Bas Spitters. Extracting functional programs from Coq, in Coq. Journal of Functional Programming (JFP), Volume 32, 2022, e11. https://doi.org/10.1017/S0956796822000077