SPL

A New Language for Blockchains

Introduction To BTC Script

- Bitcoin Script is a stack-based language similar to Forth. A program in Bitcoin Script is a sequence of operations for its stack machine
- Bitcoin Script has conditionals but no loops, thus all programs halt and the language is not Turing complete
- All Bitcoin Script operations are pure functions of the machine state except for the signature-verification operations.

Introduction to EVM

- The EVM is a Turing-complete programming language with a stack, random access memory, and persistent storage.

- These ad-hoc programs are regularly broken owing to the complex semantics of both Solidity and the EVM; the most famous of these failures were the DAO and Parity's multiple signature validation program

Introduction to SPL

- Create an expressive language that provides users with the tools needed to build novel programs and smart contracts.
- Enable static analysis that provides useful upper bounds on the amount of computation required.
- Minimize bandwidth and storage requirements and enhance privacy by removing unused code at redemption time.
- Maintain Bitcoin’s design of self-contained transactions whereby programs do not have access to any information outside the transaction.
- Provide formal semantics that facilitate easy reasoning about programs using existing off-the-shelf proof-assistant software.

Types

- The unit type, written as 1, is the type with one element.

- A sum type, written as $A + B$, contains the tagged union of values from either the left type A or the right type B

- A product type, written as $A \times B$, contains pairs of elements with the first one from the type A and the second one from the type B.

Types

- There are no recursive types in SPL
- Every type in SPL only contains a finite number of values

Terms

- **Gentzen's Sequent Calculus**
- The **unit** term returns the singular value of the unit type and ignores its argument.
- The **injl** and **injr** combinators create tagged values, while the **case** combinator, Simplicity's branching operation, evaluates one of its two subexpressions based on the tag of the first component of its input.
- The **pair** combinator creates pairs, while the **take** and **drop** combinators access first and second components of a pair respectively.
- The **iden** and comp combinators are not specific to any flavor of type. The iden term represents the identity function for any type and the comp combinator provides function composition.

$$\frac{}{\mathsf{iden} : A \vdash A} \qquad \frac{s : A \vdash B \qquad t : B \vdash C}{\mathsf{comp}\, s\, t : A \vdash C}$$

$$\frac{}{\mathsf{unit} : A \vdash \mathbb{1}}$$

$$\frac{t : A \vdash B}{\mathsf{injl}\, t : A \vdash B + C} \qquad \frac{t : A \vdash C}{\mathsf{injr}\, t : A \vdash B + C}$$

$$\frac{s : A \times C \vdash D \qquad t : B \times C \vdash D}{\mathsf{case}\, s\, t : (A + B) \times C \vdash D} \qquad \frac{s : A \vdash B \qquad t : A \vdash C}{\mathsf{pair}\, s\, t : A \vdash B \times C}$$

$$\frac{t : A \vdash C}{\mathsf{take}\, t : A \times B \vdash C} \qquad \frac{t : B \vdash C}{\mathsf{drop}\, t : A \times B \vdash C}$$

Typing Rules For The Terms.

Semantics

$$
\begin{aligned}
[\![\mathsf{iden}]\!](a) &:= a \\
[\![\mathsf{comp}\, s\, t]\!](a) &:= [\![t]\!]([\![s]\!](a)) \\
[\![\mathsf{unit}]\!](a) &:= \langle\rangle \\
[\![\mathsf{injl}\, t]\!](a) &:= \sigma^{\mathbf{L}}([\![t]\!](a)) \\
[\![\mathsf{injr}\, t]\!](a) &:= \sigma^{\mathbf{R}}([\![t]\!](a)) \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{L}}(a), c\rangle &:= [\![s]\!]\langle a, c\rangle \\
[\![\mathsf{case}\, s\, t]\!]\langle\sigma^{\mathbf{R}}(b), c\rangle &:= [\![t]\!]\langle b, c\rangle \\
[\![\mathsf{pair}\, s\, t]\!](a) &:= \langle [\![s]\!](a), [\![t]\!](a)\rangle \\
[\![\mathsf{take}\, t]\!]\langle a, b\rangle &:= [\![t]\!](a) \\
[\![\mathsf{drop}\, t]\!]\langle a, b\rangle &:= [\![t]\!](b)
\end{aligned}
$$

Completeness

- It is not completed but enough for blockchain

- SPL cannot express general computation. It can only express finitary functions, because each SPL type contains only finitely many values. However, within this domain, SPL’s set of combinators is complete: any function between SPL’s types can be expressed.

Example

- We begin by defining a type for a bit, 2, as the sum of two unit types

$$\mathbb{2} := \mathbb{1} + \mathbb{1}$$

- We choose an interpretation of bits as numbers where we define the left-tagged value as denoting zero and the right tagged value as denoting one.

$$\lceil \sigma^{\mathbf{L}} \langle \rangle \rceil_{\mathbb{2}} := 0$$
$$\lceil \sigma^{\mathbf{R}} \langle \rangle \rceil_{\mathbb{2}} := 1$$

Example

- We can write SPL programs to manipulate bits. For example, we can define the not function to flip a bit

$$\mathsf{not} : \mathbb{2} \vdash \mathbb{2}$$
$$\mathsf{not} \coloneqq \mathsf{comp}\,(\mathsf{pair}\ \mathsf{iden}\ \mathsf{unit})\,(\mathsf{case}\,(\mathsf{injr}\ \mathsf{unit})\,(\mathsf{injl}\ \mathsf{unit}))$$

- By recursively taking products, we can define types for multi-bit words

$$\mathbb{2}^1 \coloneqq \mathbb{2}$$
$$\mathbb{2}^{2n} \coloneqq \mathbb{2}^n \times \mathbb{2}^n$$

Example

- We can write a half-adder of two bits in SPL.

$$\mathsf{half\text{-}adder} : 2 \times 2 \vdash 2^2$$

$$\begin{aligned}\mathsf{half\text{-}adder} \coloneqq \mathsf{case}\ & (\mathsf{drop}\,(\mathsf{pair}\,(\mathsf{injl}\,\mathsf{unit})\ \mathsf{iden}))\\ & (\mathsf{drop}\,(\mathsf{pair}\,\mathsf{iden}\ \mathsf{not}))\end{aligned}$$