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×B, contains pairs of elements with the first one from the type A and the second one from the type B.

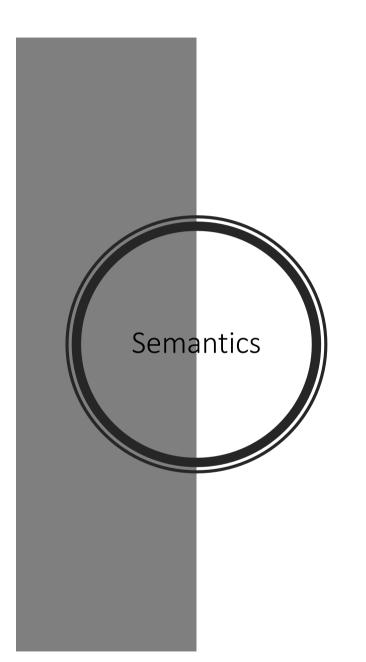
Types

- 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.

Typing Rules For The Terms.



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

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

$$2 := 1 + 1$$

 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.

$$\lceil \sigma^{\mathbf{L}} \langle \rangle \rceil_2 \coloneqq 0$$
$$\lceil \sigma^{\mathbf{R}} \langle \rangle \rceil_2 \coloneqq 1$$

Example

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

not :
$$2 \vdash 2$$

not := comp (pair iden unit) (case (injr unit) (injl unit))

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

$$2^{1} := 2$$
$$2^{2n} := 2^{n} \times 2^{n}$$

Example

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

```
\begin{aligned} \mathsf{half-adder} &: 2 \times 2 \vdash 2^2 \\ \mathsf{half-adder} &\coloneqq \mathsf{case} \ (\mathsf{drop} \, (\mathsf{pair} \, (\mathsf{injl} \, \mathsf{unit}) \, \, \mathsf{iden})) \\ &\qquad \qquad (\mathsf{drop} \, (\mathsf{pair} \, \mathsf{iden} \, \, \mathsf{not})) \end{aligned}
```