Simplicity
A New Language for Blockchains

Russell O’Connor
roconnor@blockstream.com

BPASE
January 25, 2018
• Crypto-currencies use a blockchain as a distributed ledger.
• Appending transactions to this ledger transfers funds between participants.
• Funds are locked by small programs.
• These programs use cryptography to authorize transactions.
• More complex programs allow for escrow, covenants, digital swaps, et cetera.
Why not use JavaScript?
Why not use JavaScript?

- Ethereum’s primary smart-contract language, Solidity, is syntactically similar to JavaScript.
- Solidity is compiled to machine code for an abstract machine called EVM.
Language Design for Blockchains

Why not use JavaScript?

- Ethereum’s primary smart-contract language, Solidity, is syntactically similar to JavaScript.
- Solidity is compiled to machine code for an abstract machine called EVM.

What could go wrong?
The Ethereum network is currently undergoing a DoS attack

Posted by Jeffrey Wilcke on September 22nd, 2016.

URGENT ALL MINERS: The network is under attack. The attack is a computational DDoS, ie. miners and nodes need to spend a very long time processing some blocks. This is due to the EXTCODESIZE opcode, which has a fairly low gas price but which requires nodes to read state information from disk; the attack transactions are [...]
GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas

Submitted 1 year ago by ethererik

As the operator of http://ethereumpyramid.com I am of course watching the "competition" closely. ;-) One of the more popular contracts (by transaction count) is GovernMental (Website: http://governmental.github.io/GovernMental/ Etherscan: http://etherscan.io/address/0xf45717552f12ef7cb65e95476f217ea008167ae3 ). Probably in part of the large jackpot of 1100 ETH.

The timer on the jackpot ran out and the lucky winner can now claim it. However, as part of paying out the jackpot, the contract clears internal storage with these instructions:

```javascript
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

This compiles to code which iterates over the storage locations and deletes them one by one. The list of creditors is stored as an array of 65 addresses, this would require a gas amount of 5057945, but the current maximum gas amount for a transaction is only 4712388.

So: PonziGovernmental is stuck for the moment and 1100 ETH are in limbo.
A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency

By NATHANIEL POPPER  JUNE 17, 2016
$30 million below Parity: Ethereum wallet bug fingered in mass heist

Crypto-cash leak made possible by software stuff-up

By Richard Chirgwin 20 Jul 2017 at 00:55
Parity’s multisig wallet deleted

2017-11-07: TODO
Problems with EVM

- Difficult to assign costs to operations.
- Difficult to bound the costs of programs.
- Complex and informal language semantics makes reasoning difficult.
What about Bitcoin?

- Bitcoin’s programming language is too inexpressive.
  - e.g. multiplication has been disabled since 2010.
There are two methods in software design. One is to make the program so simple, there are obviously no errors. The other is to make it so complicated, there are no obvious errors.
What we need is Simplicity
to create a solid foundation for building smart contracts upon.
What is Simplicity?

Simplicity is low-level language designed for specifying user-defined programs to be evaluated within an adversarial environment.
Simplicity’s Features

- Typed combinator language
- Finitely-complete instead of Turing-complete
- Simple, formal denotational semantics
- Formal operational semantics
- Easy static analysis of computational costs
Type System

- $\mathbb{1}$ - Unit type
- $A + B$ - Sum types
- $A \times B$ - Product types
Every Simplicity expression, $t$, is a function from an input type, $A$, to an output type, $B$.

$$t : A \vdash B$$
Simplicity Language

Typing rules for the terms of core Simplicity.
Denotational Semantics

Formally, the language and semantics are defined in the Coq proof assistant. Informally the semantics are as follows:

\[
\begin{align*}
\llbracket \text{iden} \rrbracket (a) & := a \\
\llbracket \text{comp } s \hspace{1pt} t \rrbracket (a) & := \llbracket t \rrbracket (\llbracket s \rrbracket (a)) \\
\llbracket \text{unit} \rrbracket (a) & := \langle \rangle \\
\llbracket \text{injl } t \rrbracket (a) & := \sigma^L (\llbracket t \rrbracket (a)) \\
\llbracket \text{injr } t \rrbracket (a) & := \sigma^R (\llbracket t \rrbracket (a)) \\
\llbracket \text{case } s \hspace{1pt} t \rrbracket \langle \sigma^L (a), c \rangle & := \llbracket s \rrbracket \langle a, c \rangle \\
\llbracket \text{case } s \hspace{1pt} t \rrbracket \langle \sigma^R (b), c \rangle & := \llbracket t \rrbracket \langle b, c \rangle \\
\llbracket \text{pair } s \hspace{1pt} t \rrbracket (a) & := \langle \llbracket s \rrbracket (a), \llbracket t \rrbracket (a) \rangle \\
\llbracket \text{take } t \rrbracket \langle a, b \rangle & := \llbracket t \rrbracket (a) \\
\llbracket \text{drop } t \rrbracket \langle a, b \rangle & := \llbracket t \rrbracket (b)
\end{align*}
\]
Denotational Semantics
So simple, it fits on a T-shirt.

\[
\begin{align*}
\text{[iden]}(a) & := a \\
\text{[comp }s\text{ }t]\(a\) & := [t]\([s]\(a\)) \\
\text{[unit]}(a) & := () \\
\text{[injl }t\text{]}(a) & := \sigma^L([t](a)) \\
\text{[injr }t\text{]}(a) & := \sigma^R([t](a)) \\
\text{[case }s\text{ }t\text{]}(\sigma^L(a), c) & := [s](a, c) \\
\text{[case }s\text{ }t\text{]}(\sigma^R(b), c) & := [t](b, c) \\
\text{[pair }s\text{ }t\text{]}(a) & := ([s](a), [t](a)) \\
\text{[take }t\text{]}(a, b) & := [t](a) \\
\text{[drop }t\text{]}(a, b) & := [t](b)
\end{align*}
\]
Example Expressions
\[ 2 := 1 + 1 \]
\[ 2^2 := 2 \times 2 \]
\[ 2^4 := 2^2 \times 2^2 \]
\[ \vdots \]
Half-Adder

half-adder : $2 \times 2 \vdash 2^2$

half-adder := case (drop (pair (injl unit) iden))

(drop (pair iden (comp (pair iden unit)

(case (injr unit) (injl unit))))))
SHA-256 Block Compression

\[ \text{sha-256-block} : 2^{256} \times 2^{512} \vdash 2^{256} \]

We use the formal semantics of Simplicity to prove in Coq that our implementation of sha-256-block is correct.
Expression DAGs

![DAG diagram of a half-adder](image)
Expression DAGs
Simplicity in a Blockchain
By recursively hashing the DAG of a Simplicity program, we compute a Merkle root that commits to the program.
Merkle Roots

#(iden) := SHA256Block⟨tag_{iden}, \lfloor 0 \rfloor_{2^{512}}⟩

#(comp s t) := SHA256Block⟨tag_{comp}, \langle #(s), #(t) \rangle⟩

#(unit) := SHA256Block⟨tag_{unit}, \lfloor 0 \rfloor_{2^{512}}⟩

#(injl t) := SHA256Block⟨tag_{injl}, \langle #(t), \lfloor 0 \rfloor_{2^{256}} \rangle⟩

#(injr t) := SHA256Block⟨tag_{injr}, \langle #(t), \lfloor 0 \rfloor_{2^{256}} \rangle⟩

#(case s t) := SHA256Block⟨tag_{case}, \langle #(s), #(t) \rangle⟩

#(pair s t) := SHA256Block⟨tag_{pair}, \langle #(s), #(t) \rangle⟩

#(take t) := SHA256Block⟨tag_{take}, \langle #(t), \lfloor 0 \rfloor_{2^{256}} \rangle⟩

#(drop t) := SHA256Block⟨tag_{drop}, \langle #(t), \lfloor 0 \rfloor_{2^{256}} \rangle⟩
Witness Values

Simplicity adds a special witness combinator that directly provides input values such as digital signatures.

\[ b : B \]

\[ \text{witness } b : A \vdash B \]

These witness values are not committed as part of the commitment Merkle root.

\[ \#(\text{witness } b) := \text{SHA256Block}(\text{tag}_{\text{witness}}, [0]^{2512}) \]
At redemption time, one provides a full Simplicity DAG, including witness values.

- The system checks that the Merkle root matches the commitment.
- The system evaluates the Simplicity program and ensures that it succeeds.
Redemption

At redemption time, one provides a full Simplicity DAG, including witness values.

- The system checks that the Merkle root matches the commitment.
- The system evaluates the Simplicity program and ensures that it succeeds.

During redemption unused branches of the Simplicity program can be pruned.
More Features

Full Simplicity is intended to support:

- Signature aggregation
- Covenants
- Delegation
Operational Semantics
An abstract machine, called the Bit Machine, evaluates Simplicity expressions.

<table>
<thead>
<tr>
<th>read frame stack</th>
<th>write frame stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10011??110101000]</td>
<td>[11??1101]</td>
</tr>
<tr>
<td>[0000]</td>
<td>[111??]</td>
</tr>
<tr>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td></td>
</tr>
</tbody>
</table>

Example state for the Bit Machine

Simple instructions for the Bit Machine manipulate the two stacks.
Simplicity Costs

The cost (or weight) of a Simplicity program is determined by

• the size of the programs DAG.
• the number of steps the Bit Machine model takes.
• the amount of memory needed by the Bit Machine model.

This costs can be quickly bounded using static analysis.
Evaluating a Simplicity subexpression can only read from a fragment of the active read frame and can only write to a fragment of the active write frame.
Jets allow the Simplicity interpreter to recognize common subexpressions and preform the computation with C code instead.

Discounted jets allow reduced costs for Simplicity programs that use these jets.
Advantages of Jets:

- Jets provide a formal specification of their behaviour.
- Jets themselves cannot introduce new effects.
- Jets are transparent when reasoning about behaviour.
Formal Verification
Formal Verification

We want to be able to build smart contracts that

- manage millions of dollars worth of funds.
- are custom tailored.
- may be single purpose.
- run on a public blockchain.

Formal verification is the only conceivable way of achieving the safety required.
Formal Verification

- Simplicity (language)
- C impl.
- x86 Assembly

VST -> CompCert
Formal Verification

Formal Security Properties  FCP  Smart Contract Protocol  Front-end (language)  Compiler  Simplicity (language)  VST  C impl.  CompCert  x86 Assembly
Solutions provided by Simplicity

• Avoids denial of service:
  • The Bit Machine provides a simple model for computational resource costs.

• Avoids running out of gas:
  • Simple static analysis algorithms can bound resource costs of arbitrary programs.

• Avoids hacks:
  • Practical proofs of program correctness made possible by formal semantics in Coq.

• Add privacy:
  • Unused code is pruned before appearing on the Blockchain.
Read more about Simplicity

https://arxiv.org/abs/1711.03028