bll, symbll, bllsh

bll – Basic Bitcoin Lisp Language

- What is it?
  - Little language like script
  - Also has ***Loops***
  - Also has ***Structured data***
  - Simplest possible thing that has both of those

- What does bll add to script?

- Loops
  - Single “op_eval”-like opcode
- Structured data
  - pairs of objects

symbll – What is it?

- Variation on bll that's easier to program
- It's own language that is interpreted directly
- Very close to bll
  - Easy to compile to bll
  - Minimal surprises in what the generated bll looks like

- Main features are:
  - Named symbols
  - Named functions with named parameters
  - Short-circuiting "if" macro
  - "report" macro for printf-style debugging

bllsh – What is it?

- REPL for bll and symbll
- Step-through debugger for bll and symbll
- Compiler for symbll

- Commands:
  - eval / blleval
  - def / undef
  - tx / tx_in_idx / tx_script / utxos
  - debug / blldebug
  - step / next / cont / trace
  - compile / program

What’s bll give you?

- Less hassle working around script’s limitations:
  - WOTS+ in script: 22kB+2kB
  - WOTS+ in bll: 3.6kB+2kB
- Directly implement new features (eg, ANYPREVOUT, graftroot) without a soft fork
- *“Permissionless innovation”*

Details: Generalised opcodes

- Bignum support
- Opcodes operate on lists of arguments
- Re-enable opcodes
- Add new general functions
- Allow for future upgrades

- Calculate 100! Or implement your own ECC curve math.
- (+ 1 2 3) vs “1 2 ADD 3 ADD”
- CAT, MUL, etc
- bip340_verify, bip342_txmsg, tx, secp256k1_muladd
- (softfork ...)

Details: Explicit bounds on computation

- Each opcode has a computation “time” cost, which may depend on its arguments/result
- Total object allocation pool is limited
- Txs should have a way of adding virtual weight, allowing more computation, but still subject to block limit

- Tapscript current limits:
  - 1 SHA256D calculation over 520 bytes of data per tx weight unit
  - ~520kB memory usage (1000 stack items of 520 bytes each)
- (Note that memory usage limit affects ability to verify scripts in parallel)

Details: Computation model

- Goals:
  - Well-defined (it’s consensus!)
  - Efficient
- Currently:
  - Continuation passing style
  - Tail recursion elimination
  - Reference counting, with no self-referential structures
  - Small number of opcodes
  - Very small number of “macros”

- 37 normal opcodes
  - Normal opcodes take each argument in turn, evaluate it, do something with it, and return a result at the end.
- Only 4 macros, that behave “specially” (namely “a”, “q”, “partial” and “softfork”)

Details: ~~stack manipulation~~

- Bitcoin script has 19 opcodes for stack manipulation
- bll has 5 opcodes, but ~infinity if you count environment access codes

- bll expressions are always evaluated against an environment
- The environment is a bll object, which may be a pair of bll objects, each of which… you get the idea.
- “1” is the environment as a whole, “2” is the left item, “3” is the right item, “4” is the left/left item, etc...

Philosophical considerations

- Thing to think about
  - Computation vs verification
  - Turing completeness
  - People can do bad things
  - Special case opcodes vs general opcodes

Computation vs verification

- Programming on the blockchain is for verification, not computation.
- The result you get is either “1” – this transaction is valid, or “0” – it’s not. If the result is “0”, it doesn’t go in a valid chain.

- “The solution was script, which generalizes the problem so transacting parties can describe their transaction as a predicate that the node network evaluates. The nodes only need to understand the transaction to the extent of evaluating whether the sender's conditions are met.
- “The script is actually a predicate. It's just an equation that evaluates to true or false. Predicate is a long and unfamiliar word so I called it script.”
  - Satoshi, June 17, 2010

Turing completeness

- Turing complete means “cannot be sure this terminates”
  - Not turing complete because computation limits ensure termination
  - Also not turing complete because script sizes are bounded by the block size

- Simplicity proposes “finitary completeness”, which (AIUI) gives you a strict bound on execution time after doing type checking, which itself is linear.

People can do bad things

- New types of spam
  - Same limits as current chain (limits on data/computation per block)
- Construct covenants
  - You define your own scriptPubKey; let others burn their funds if they want
- Unsafe wallet software
  - Don’t trust things just because they’re “Bitcoin”

People can do bad things

- Put other assets on Bitcoin’s blockchain
  - Threat is that it may mean other assets’ txs are more valuable than BTC payments, pricing out BTC from the Bitcoin blockchain
  - Already true thanks to ordinals/inscriptions/runes/etc
  - But also already true of payments: considering sending someone BTC in order to exercise an in-the-money option, just prior to expiry
  - Possibly not desirable: Bitcoin is expensive and slow; why not put your assets on something cheap and fast?

People can do bad things

- MEV
  - Authorise a transaction with something other than a SIGHASH_ALL sig
  - Your authorisation may be able to be pulled out and put together on some other transaction in a way that loses you money
  - Ultimately miners have the most flexibility here, so are most likely to win, hence MEV
- “Don’t do that”
  - Have your authorisation set an explicit fee (ie, the difference between the value of the inputs you’re authorising spending and the outputs you’re requiring to exist)
  - UTXO-model vs account-model makes it much easier for changed conditions to invalidate previous authorisations

People can do bad things

- ...is just another way of saying

  “permissionless innovation”

Special case vs general opcodes

- Special case opcodes
  - Easier to use correctly
  - Harder to misuse
  - Shorter to encode on-chain
  - Less flexible
  - Can be hard to work out the optimal specification
  - Still possible to misuse
  - Providing new features require consensus changes

- Special case opcode:
  - 2 <P1> <P2> <P3> 3 CHECKMULTISIG
- General opcode:

  -
    ```
    for (pk : pks) {
        i += checksig(pk)
    }
    assert(i >= 2)
    ```

Special case vs general opcodes

- General opcodes
  - Easier to experiment with
  - Covers more use cases with less code
  - Opens up all sorts of behaviour, even bad ones

Future work / TODO

- Finish coding “flexible earmark” example
- Define a success condition
  - (evaluates to non-nil? to nil? to “1”? to anything that’s not an error?)
- Rewrite from python to C++
- Extra opcodes?
- Merge to inquisition / deploy on signet
- More use-cases / demos
- Formal specification

- C++ implementation
  - Easier to measure what opcode computation costs should be
  - Allows apples-to-apples comparison against simplicity implementation
- Formal specification
  - Allows oranges-to-oranges comparison with simplicity
  - Prove symbll code executes the same as the bll code it compiles too

Links

- github.com/ajtowns/bllsh
- bitcoinops.org/en/topics/basic-bitcoin-lisp-language/