Spedn Documentation

{o} Software

Apr 07, 2020

Getting Started:

Spedn is a high level smart contracts language for Bitcoin Cash. It is designed for explicitness and safety:

- It is statically typed - detects many errors at compile time
- It is explicitly typed - no guessing what the expression is supposed to return
- It is purely-functional - free of side effects, the common source of bugs
- It has a familiar C-like syntax

Warning: Spedn is an experimental tool. It is not recommended to be used on mainnet yet.

CHAPTER 1

Quick start guide

1.1 Build from sources

1. Intsall Haskell Tool Stack.
2. Download Spedn sources.

```
$ git clone https://bitbucket.org/o-studio/spedn.git
```

3. Build and install Spedn.

```
$ cd spedn/spedn
$ stack install
```

1.2 Installation from npm

Alternatively, you can install a JavaScript version from npmjs repository:

```
$ npm i -g spedn-cli
```

1.3 Your first contract

Create a file mycontract.spedn with a following content:

```
contract ExpiringTip(Ripemd160 alice, Ripemd160 bob) {

    challenge receive(Sig sig, PubKey pubKey) {
        verify hash160(pubKey) == bob;
        verify checkSig(sig, pubKey);
```

(continues on next page)

(continued from previous page)

```
    }

    challenge revoke(Sig sig, PubKey pubKey) {
        verify checkSequence(7d);
        verify hash160(pubKey) == alice;
        verify checkSig(sig, pubKey);
    }
}
```

Compile with command:

```
$ spedn compile -c mycontract.spedn
```

You should get a compiled contract template similar to this:

```
<alice> <bob> 2 PICK TRUE EQUAL IF 3 PICK HASH160 OVER EQUALVERIFY (...)
```

1.4 Coming soon

Instantiating the template. Address generation. Redeeming.

CHAPTER 2

Understanding Script

Before developing contracts with Spedn it is worth understanding what they are compiled to and how Bitcoin Cash transactions internally work.

2.1 There is no spoon…

From a user perspective it's convenient to perceive a Bitcoin Cash address as a kind of account with a balance. But this is just a nice abstraction over a mechanism that works in slightly more complicated way.

There is no account. Every transaction contains inputs and outputs. An output consists of an amount of bitcoins and a script (often called `scriptPubKey`) specifying some spending conditions for that amount. An input is a reference to some output of a previous transaction and some script (called `scriptSig`) satisfying the spending condition from `scriptPubKey`. In a typical transaction, `scriptPubKey` contains a public key of a coin owner and `scriptSig` contains a signature matching that public key - hence the names. An output that is not yet referenced by any other transaction is called *Unspent Transaction Output* (UTXO).

A UTXO can be perceived as a lockbox containing a single coin.

An address is a user readable representation of a standard scriptPubKey.

2.2 Kinds of boxes

You can spot two kinds of addresses in Bitcoin Cash:

2.2.1 Pay To Public Key Hash (P2PKH)

This is an "ordinary" address representing a very simple script that checks two condidtions:

- If the public key provided in scriptSig matches the hash in scriptPubKey when hashed with SHA-256 and then RIPEMD-160.

- If the signature provided in scriptSig is valid for that key.

2.2.2 Pay To Script Hash (P2SH)

This is a "smart contract" address. Instead of public key hash it cointains a hash of an entire script that is called a `redeem script`. The scriptSig is supposed to provide the actual script that matches this hash and arguments to it.

2.3 Making fancy boxes

All those scripts are bytecode that run in a stack machine. A human readable representation (assembly language) of this bytecode is called… Script. Script is a FORTH-like, stack oriented language containing numerous opcodes, some generic, some very Bitcoin-specific. It intentionally lacks support for recursion what guarantees that all scripts finish (and even do so in deterministic time).

Writing scripts in Script is quite hard. This is why Spedn was created. It's a high level language that compiles to Script. Contracts written in Spedn represent redeem scripts for P2SH addresses.

CHAPTER 3

Syntax overview

3.1 Contract Templates

A contract template in Spedn represents a template for generating a P2SH address and corresponding redeem script. It can be parametrized. Contract parameters have to be specified to instantiate it, that is - to generate a particular contract with an address.

You can perceive a contract template as a specification of a pin tumbler lock mechanism while a contract is a particular lock and parameters are pin lengths in it.

Syntax:

> **contract** *ContractName* **(** [*type paramName* [**,** . . .]] **) { }**

Example:

```
contract SomeContract(Ripemd160 pubKeyHash, int x) {
    // challenges
}
```

3.2 Challenges

A challenge is a set of conditions that have to be met to redeem a coin locked in a contract. Challenges specify arguments that will be expected to be provided in `scriptSig` when redeeming the coin. A contract must contain at least one challenge and a challenge must define at least one argument. Challenges must have unique names.

A challenge introduces a lexical scope so two different challenges can define an argument with the same name.

When redeeming a coin, a redeemer must choose one of the challenges and satisfy its conditions.

You can perceive a challenge as a keyhole in a lock and arguments as keys.

Syntax:

> **challenge** *name* **(** *type argName* [**,** . . .] **) *statement* **

Example:

```
challenge someChallenge(PubKey pubKey, Sig signature) {
    // statements...
}
```

3.3 Statements

A challenge can contain any number of statements. To be precise - it contains a single statement but this can be a block statement which can contain any number of statements.

There are the following kinds of statements:

3.3.1 Verification

The most important statement and often the only one needed. It evaluates an expression and fails the script if the result is false.

Syntax:

> **verify** *expr* **;**

Example:

```
verify hash1 == hash2;
```

3.3.2 Variable binding

You can define a local variable that will be accessible down in the same lexical scope and nested scopes.

Syntax:

> *type* *name* **=** *expr* **;**

Example:

```
int a = b + c;
```

There is also a possibility to define 2 variables in case of using the split operator. If one of the results is unnecessary, you can ignore it with a low dash operator.

Syntax:

> *type* **[** *leftName* , *rightName* **]** **=** *expr1* @ *expr2* **;**
>
> *type* **[_,** *rightName*] = *expr1* @ *expr2* **;**
>
> *type* [*leftName* **, _]** = *expr1* @ *expr2* **;**

Example:

```
bin [prefix, _] = secret @ 4;
```

3.3.3 Conditional

You can conditionaly execute a branch of code. A branch introduces a new lexical scope and it can be a verification, block or another conditional.

Syntax:

if **(** *condition* **)** statement [**else** *statement*]

Example:

```
if (num % 2 == 1)
    verify checkSig(sig, alice);
else
    verify checkSig(sig, bob);
```

3.3.4 Block

A block is a statement that groups several statements for sequential execution. A block introduces a lexical scope. The last statement must be a verification or conditional.

Syntax:

{ [*statements...*] **}**

Example:

```
if (num % 2 = 1) {
    verify checkSig(sig, alice);
} else {
    verify checkSig(sig, bob);
    verify checkSequence(5d);
}
```

3.3.5 Loop

There are no loops, it's Bitcoin.

3.4 Lexical scopes

Spedn creates common, nested lexical scopes for parameters, arguments, variables and functions. There can be no 2 identical names within the same scope. Also - name shadowing is prohibited so a nested scope cannot redefine a name present in its parent scope.

There are following scopes in the nesting order:

- **Global scope** - contains predefined functions and type constructors
- **Contract scope** - introduced by the contract, contains contract parameters
- **Challenge scope** - introduced by the challenge, contains challenge arguments and local variables
- **Local scope** - introduced by *if/else/block* statements, contains local variables

Exhaustive example:

```
// a global scope, names like checkSig, min, max are reserved.

// contract scope begins
contract X(int a, int b) { // names a, b are defined

    // challenge scope begins
    challenge a( // it's OK for the challenge to be named a because challenge names␣
↪don't occupy the name table.
        int c // name c is defined
        /* int a      // BAD - already defined in contract scope */)
    {
        verify a >= b;
        /* verify a == d // BAD - d is not yet defined */
        int d = a + b;    // name d is defined
        if (d > 0)
        // if scope begins
        {
            int e = d % c;
            verify e == 0;
        }
        // if scope ends; e is gone.
        else
        // else scope begins
            verify a == b;
        // else scope ends
        /* verify e == 1 // BAD - e is gone */
    }
    // challenge scope ends; c, d are gone

    // challenge scope begins
    challenge b(int c, int d) // names c, d are defined
    {
        verify c == d;
    }
    // challenge scope ends; c, d are gone

}
// contract scope ends; a, b are gone
```

CHAPTER 4

Types

4.1 Basic Types

Basic types reflect types Script operates on.

- **bool** - a boolean value. Can be either `true` or `false`. `verify` and `if` statements expect an expression returning this type.
- **int** - a 32-bit signed integer. Literals of this type can be specified in dec or hex.

```
int a = -1234;
int b = 0xff00i; // notice `i` suffix
```

- **bin** - an array of bytes. Literals of this type are specified in hex.

```
bin arr = 0x112233445566778899 00aabbccddeeff;
```

4.2 Domain-Spcecific Types

To increase safety, Spedn introduces meaningful types that help with catching semantic errors at compile time.

4.2.1 Numeric types

These types add meaning to a raw `int`. They must be explicitly casted from `int` with a type constructor. They cannot be casted back to `int`.

- **Time - represents an absolute time.** Can be expressed as a Unix Timestamp or a Block Height and variously defined.

```
Time x = `2018-10-13 21:37:00`; // defined with a time literal
Time y = TimeStamp(1539466620); // conversion from `int` interpreted as Unix␣
↪Timestamp
Time z = TimeStamp(584834);     // conversion from `int` interpreted as Block␣
↪Height
```

- **TimeSpan** - represents a relative time period. Can be expressed as a number of blocks or 512-seconds periods.

```
TimeSpan x = 1d 2h 3m 4s; // Time units literal. Be awre that the number␣
↪will be rounded down to full 512s periods
TimeSpan y = 10b;         // Blocks literal.
TimeSpan z = Blocks(10);  // Conversion from `int`
```

4.2.2 Binary types

These types add meaning to a raw `bin`. They can be implicitly casted to `bin`. They must be explicitly casted from `bin` with a type constructor.

- **PubKey** - represents a public key.

```
PubKey alice = PubKey(0x11223344556677889900aabbccddeeff);
```

- **Sig** - represents a tx signature (which can be checked with `checkSig`).

```
Sig alice = Sig(0x11223344556677889900aabbccddeeff);
verify checkSig(alice, alicePubKey);
```

- **DataSig** - represents a data signature (which can be checked with `checkDataSig`).

```
DataSig alice = DataSig(0x11223344556677889900aabbccddee);
verify checkDataSig(alice, preimageHash, alicePubKey);
```

- **Ripemd160** - represents a result of RIPEMD-160 hash.

```
Ripemd160 h = hash160(pubKey);
```

- **Sha1** - represents a result of SHA-1 hash.

```
Sha1 x = sha1(secret);
```

- **Sha256** - represents a result of SHA-256 hash.

```
Sha256 x = hash256(secret);
```

4.2.3 Special types

These are types that can appear in expressions but you cannot define variables of them.

- **List - can be only created as literals passed to functions that expect them,** which is currently `checkMultiSig` only.

```
verify checkMultiSig([sig1, sig2], [key1, key1]);
```

- **Verification - almost like `bool` but the only thing you can do with it is to pass it to `verify`.** This is a return type of `checkLockTime` and `checkSequence` functions.:

```
verify checkSequence(8b);
```

CHAPTER 5

Operators

Precedence	Operator	Description	Associativity
1	`-a`	Unary minus	right to left
1	`!a`	Logical NOT	right to left
2	`a / b`	Integer division	left to right
2	`a % b`	Modulo	left to right
3	`a + b`	Integer addition	left to right
3	`a - b`	Integer subtraction	left to right
4	`a . b`	bytes arrays concatenation	left to right
5	`a < b`	Less than	left to right
5	`a <= b`	Less than or equal	left to right
5	`a > b`	Greater than	left to right
5	`a >= b`	Greater than or equal	left to right
6	`a == b`	Equal	left to right
6	`a != b`	Not equal	left to right
6	`a === b`	Numeric and equal	left to right
6	`a !== b`	Numeric and not equal	left to right
7	`a & b`	Bitwise AND	left to right
8	`a ^ b`	Bitwise XOR	left to right
9	`a | b`	Bitwise OR	left to right
10	`a && b`	Bolean AND *Note: Both a and b are always evaluated.*	left to right
11	`a || b`	Boolean OR *Note: Both a and b are always evaluated.*	left to right
12	`a @ b`	Split bytes array `a` at position `b`.	none

CHAPTER 6

Functions

6.1 Math Functions

- `int abs(int a)`

  Returns an absolute value of the argument.
- `int min(int a, int b)`

  Returns the smaller argument.
- `int max(int a, int b)`

  Returns the larger argument.
- `bool within(int x, int min, int max)`

  Returns `true` if `x >= min && x < max`.

6.2 Hashing Functions

- `Ripemd160 ripemd160(bin bytes)`

  Returns a RIPEMD-160 hash of the argument.
- `Sha1 sha1(bin bytes)`

  Returns a SHA-1 hash of the argument.
- `Sha256 sha256(bin bytes)`

  Returns a SHA-256 hash of the argument.
- `Ripemd160 hash160(bin bytes)`

  Returns RIPEMD-160 hash of SHA-256 hash of the argument.

- `Sha256 hash256(bin bytes)`

  Returns double SHA-256 hash of the argument.

6.3 Cryptographic Checks

- `bool checkSig(Sig sig, PubKey pk)`

  Validates a transaction signature `sig` againnst a public key `pk`.

- `bool checkMultiSig(List<Sig> sigs, List<PubKey> pks)`

  Validates the set of signatures against the set of public keys.

- `bool checkDataSig(DataSig sig, bin msg, PubKey pk)`

  Validates a signature `sig` of an arbitrary message `msg` against a public key `pk`.

6.4 Timelock Checks

- `Verification checkLockTime(Time t)`

  Validates whether the spending transaction occurs after time `t`, expressed as a block height or a timestamp.

- `Verification checkSequence(TimeSpan duration)`

  Validates whether the spending transaction happens after `duration` relative to the locking transaction, expressed as a number of blocks or number of 512 seconds-long periods.

6.5 Array Operations

- `bin num2bin(int num, int size)`

  Converts a number `num` into a bytes array of size `size`.

- `int bin2num(bin data)`

  Converts a bytes array `data` to an integer. The array is treated as little-endian.

- `int size(bin data)`

  Returns the length of `data`.

- `bin fst([bin, bin] data)`

  Returns the first element of a tuple (result of `@` operator).

  ```
  bin left = fst(0xaabbccdd @ 2);
  // left == 0xaabb
  ```

- `bin snd([bin, bin] data)`

  Returns the second element of a tuple (result of `@` operator).

  ```
  bin right = snd(0xaabbccdd @ 2);
  // right == 0xccdd
  ```

- `DataSig toDataSig(Sig data)`

  Converts a signature suitable for `checkSig` function (with a sighash flag) to a signature suitable for `checkDataSig` function (without a sighash flag).

  ```
  verify checkSig(sig, pubKey);
  verify checkDataSig(toDataSig(sig), preimageHash, pubKey);
  ```

6.6 Type Constructors

- `PubKey PubKey(bin data)`
- `Ripemd160 Ripemd160(bin data)`
- `Sha1 Sha1(bin data)`
- `Sha256 Sha256(bin data)`
- `Sig Sig(bin data)`
- `DataSig DataSig(bin data)`
- `Time TimeStamp(int timestamp)`
- `Time TimeStamp(int blockHeight)`
- `TimeSpan Blocks(int number)`

CHAPTER 7

Command-line Interface

The general syntax is:

```
$ spedn COMMAND args
```

7.1 Compiling

To compile a contract to opcodes, use:

```
$ spedn compile -c MyContract.spedn
```

If the contract contains parameters, a template with placeholders will be generated. To instantiate the contract with particular parameter values, provide them as `key=value` pairs after `--`. For example, assuming MyContract has `alicePHK` parameter of type `Ripemd160` and `delay` parameter of type `TimeSpan`, you can use the following:

```
$ spedn compile -c MyContract.spedn --␣
↪alicePKH=0xb08f0f859f53873e8f02f6c0a8290a53e76a2e0a delay=1d1h
```

To compile a contract to a hex representation, use:

```
$ spedn compile -h -c MyContract.spedn --␣
↪alicePKH=0xb08f0f859f53873e8f02f6c0a8290a53e76a2e0a delay=1d1h
```

Note that in this case, the contract must be fully instantiated (all parameters values must be provided).

CHAPTER 8

BITBOX Integration

Spedn is available for NodeJS developers as an SDK extending capabilities of BITBOX SDK. TypeScript type definitions are provided out of the box.

8.1 Installation

NodeJS **v11** or newer is required. You can also use **v10** but then Worker Threads feature has to be explicitly enabled by `--experimental-worker` flag.

To install Spedn SDK in your JS project, type:

```
npm i spedn
# or
yarn add spedn
```

8.2 Compiler service

Spedn compiler runs as a service in a worker thread that you can start, use and dispose with `Spedn` class.

```
import { Spedn } from "spedn";

async function main() {

   const compiler = new Spedn();
   /* use compiler */
   compiler.dispose();

}
main();
```

Instead of manually disposing the service you can also use `using` function inspired by some languages, which guarantees automatic disposal of a resource also in case of exceptions.

```
import { Spedn, using } from "spedn";

async main() {

   await using(new Spedn(), async compiler => {
      /* use compiler */
   });

}
main();
```

8.3 Compiling contracts

To compile a source file use `compileFile` method. To compile source code in a string, use `compileCode`.

```
const BlindEscrow = await compiler.compileFile("./BlindEscrow.spedn");

const ExpiringTip = await compiler.compileCode(::`
   contract ExpiringTip(Ripemd160 alice, Ripemd160 bob) {
      challenge receive(Sig sig, PubKey pubKey) {
         verify hash160(pubKey) == bob;
         verify checkSig(sig, pubKey);
      }
      challenge revoke(Sig sig, PubKey pubKey) {
         verify checkSequence(7d);
         verify hash160(pubKey) == alice;
         verify checkSig(sig, pubKey);
      }
   }
`);
```

The output of those methods is a JavaScript class representing a contract template. Static field `params` describes what parameters are required to instantiate it.

```
console.log(ExpiringTip.params);
// Object {alice: "Ripemd160", bob: "Ripemd160"}
```

8.4 Instantiating contracts

To instantiate the template, just create an object of the contract class, providing parameters values. Parameters are passed as an object literal explicitly assigning values by names. Values of `bool` and `int` *Spedn* type can be passed as ordinary *JS* booleans and numbers. `Time` and `TimeSpan` are also passed as numbers (see BIP65 and BIP112 for value interpretation details). All the other types should be passed as *JS* `Buffer`.

In case of `ExpiringTip` you'll need 2 public keys which you can generate with BITBOX.

```
import { BITBOX } from "bitbox-sdk";

const bitbox = new BITBOX();
const mnemonic = "draw parade crater busy book swim soldier tragic exit feel top civil
↪";
```

(continues on next page)

(continued from previous page)

```
const wallet = bitbox.HDNode.fromSeed(bitbox.Mnemonic.toSeed(mnemonic));
const alice = bitbox.HDNode.derivePath(wallet, "m/44'/145'/0'/0/0");
const bob = bitbox.HDNode.derivePath(wallet, "m/44'/145'/1'/0/0");

const tip = new ExpiringTip({
   alice: alice.getIdentifier(), // Ripemd160 hash of Alice's public key
   bob:   bob.getIdentifier()    // Ripemd160 hash of Bob's public key
});
```

Once created, you can read the contract funding address and lookup for UTXOs (coins) that are locked in it. Also, a field `challengeSpecs` contains definitions of challenges and their parameters.

```
console.log(tip.getAddress("mainnet"));
// bitcoincash:pppvx30pcylxzhewr6puknpuvz7gjjtl4sdw4ezcnp

const coins = await tip.findCoins("mainnet");
// Array(2) [.....]

console.log(tip.challengeSpecs);
// Object {receive: Object, revoke: Object}
console.log(tip.challengeSpecs.receive);
// Object {sig: "Sig", pubKey: "PubKey"}
```

8.5 Spending coins

To spend coins, use `TxBuilder`. Provide tx inputs with `from` method and outputs with `to` method. Optionally, set a timelock with `withTimelock`. To send the transaction to the network use `broadcast` method. If you just want to build the transaction without broadcasting it, use `build` method.

`from` method accept a single coin or an array of coins as a first parameter. Because you can't (in most cases) sign the input without defining all the inputs and outputs first, `from` method does not simply accept scriptSig parameter. Instead, it accepts a `SigningCallback` function and the actual signing is deferred to the moment of calling `build`/`broadcast`.

`SigningCallback` accepts 2 parameters. The first one is an object containing contract challenges. The second one is a `SigningContext` which provides methods necessary for signing:

- `sign(keyPair, hashType)` - generates a siggnature valid for `OP_CHECKSIG`.
- `signData(keyPair, data)` - generates a signature valid for `OP_CHECKDATASIG`.
- `preimage(hashType)` - generates the same preimage as one used by `sign(keyPair, hashType)` (useful for `OP_CHECKDATASIG` covenants).

Note that methods accepting `hashType` always add `SIGHASH_FORKID` flag so you don't need to specify it explicitly.

`to` method accepts an address or a scriptPubKey buffer as its first argument and an amount (in satoshis) as the second one. You can also omit the amount at a single output - in this case, `TxBuilder` will treat this output as a change address and automatically calculate its amount choosing optimal transaction fee.

In the following example, all the previously found coins are spent using `receive` challenge but 5mBCH goes to Bob's new address and the rest goes back to Alice.

```
import { TxBuilder, SigHash } from "spedn";

```

(continues on next page)

(continued from previous page)

```
const txid = await new TxBuilder("mainnet")
    .from(coins, (input, context) =>
        input.receive({
            sig: context.sign(bob.keyPair, SigHash.SIGHASH_ALL),
            pubKey: bob.getPublicKeyBuffer()
        })
    )
    .to("bitcoincash:qrc2jhalczuka8q3dvk0g8mnkqx79wxp9gvvqvg7qt", 500000)
    .to(alice.getAddress())
    .withTimelock(567654)
    .broadcast();
```

8.5.1 Spending ordinary P2PKH

Spedn SDK provides also a class `P2PKH` which is a representation of an ordinary Pay to Public Key Hash address. You can instantiate it with a public key hash buffer or several factory methods:

```
import { P2PKH } from "spedn";

let addr = new P2PKH(bob.getIdentifier());
addr = P2PKH.fromKeyPair(bob.keyPair);
addr = P2PKH.fromPubKey(bob.getPublicKeyBuffer());
addr = P2PKH.fromAddress(bob.getAddress());
// all the above are equivalent
```

P2PKH contracts can be spent just like any other contract - they have `spend({sig, pubKey})` challenge, but you can also replace the whole signing callback with a convenient helper `signWith(keyPair)`. Let's modify the previous example to spend additional input.

```
import { signWith } from "spedn";

const bobsCoins = await addr.findCoins("mainnet");

const txid = await new TxBuilder("mainnet")
    .from(coins, (input, context) =>
        input.receive({
            sig: context.sign(bob.keyPair, SigHash.SIGHASH_ALL),
            pubKey: bob.getPublicKeyBuffer()
        })
    )
    .from(bobsCoins[14], signWith(bob.keyPair))
    .to("bitcoincash:qrc2jhalczuka8q3dvk0g8mnkqx79wxp9gvvqvg7qt", 500000)
    .to(alice.getAddress())
    .withTimelock(567654)
    .broadcast();
```

8.5.2 Spending generic P2SH

Spedn SDK provides also a class `GenericP2SH` for interoperability with any Pay to Script Hash contract created without Spedn. To work with that kind of contract, you just need to know its redeemScript and what arguments it expects. The generated class will have a single challenge `spend` with parameter requirements as specified in the constructor.

```
import { GenericP2SH } from "spedn";

const contract = new GenericP2SH(redeemScriptBuffer, { sig: "Sig", someNumber: "int" }
↪);
```

CHAPTER 9

Zero Conf Forfeits

This example is based on /u/awemany's proposal for securing 0-conf transactions. In addition to a regular payment output and a change output we create also a forfeit output. The forfeit can be ordinarily spent by the customer which would be nonsensical if he also wanted to doublespend. If the doublespend is actually attempted then the miner can spend the forfeit by presenting a proof of that.

Read the details here or watch a presentation.

```
contract Forfeit(
    Ripemd160 inputPKH,     // a public key hash used to redeem the input in the
↪payment tx
    Ripemd160 customerPKH   // a public key hash to be used to redeem the forfeit
    ) {

    // This challenge is used by the customer to reclaim the forfeit.
    // Basically, a typical P2PKH.
    challenge ok(PubKey pubKey, Sig sig) {
        verify hash160(pubKey) == customerPKH;
        verify checkSig(sig, pubKey);
    }

    // This challenge can be used by a miner to claim the forfeit
    // if he can prove there was a doublespend attempt.
    challenge fraud(
        Sig paymentSig,          // A signature used in payment transaction
        bin paymentPayload,      // Signed data from the transaction
        Sig doublespendSig,      // Another signature taken from the doublespend
↪attempt
        bin doublespendPayload, // Signed data from the doublespend
        PubKey pubKey            // Public Key matching both signatures
    ) {
        // If the provided PK matches the one from the payment input...
        if (hash160(pubKey) == inputPKH) {
            // verify the signature provided in that payment...
            verify checkDataSig(paymentSig, paymentPayload, pubKey);
```

(continues on next page)

(continued from previous page)

```
            // and that there was seen some other transaction which also validly␣
↪signed that input...
            verify checkDataSig(doublespendSig, doublespendPayload, pubKey);
        } else {
            // otherwise don't allow to spend it
            verify false;
        }
    }
}
```

CHAPTER 10

ChainBet Protocol

ChainBet is a proposed Bitcoin Cash protocol to enable on-chain betting. You can read the details here.

The flow of the bet consists of several steps that can be expressed in Spedn.

10.1 Escrow Preparation

10.1.1 Alice Escrow Address

The main purpose of Alice's escrow address is to reveal Alice's Secret A when spent. It will require both Alice and Bob's signature plus the secret. By requiring the secret, it reveals it to Bob, thus fulfilling that part of the commitment scheme.

Alternatively, Alice can retrieve the funds unilaterally after 8 confirmations in the situation when Bob abandonds the betting process.

```
contract ChainBetAliceEscrow(PubKey alicePK, PubKey bobPK, Ripemd160 commitment) {

    challenge cancel(Sig aliceSig) {
        verify checkSequence(8b);
        verify checkSig(aliceSig, alicePK);
    }

    challenge proceed(Sig aliceSig, Sig bobSig, bin secret) {
        verify hash160(secret) == commitment;
        verify checkMultiSig([aliceSig, bobSig], [alicePK, bobPK]);
    }
}
```

10.1.2 Bob Escrow Address

The main purpose of Bob's escrow address is to prevent Bob from double spending. Once the funding transaction is created, Alice's secret will be revealed. If Bob sees that he has a loss, he could theoretically attempt to double spend his input to the funding transaction, thereby invalidating it.

By first moving the funds into escrow and requiring Alice's signature in addition to Bob's to spend, Bob cannot on his own attempt a doublespend.

Of course, it is necessary for the transaction that funds the escrow account to have at least 1 confirmation before the funding transaction is attempted, because otherwise Bob could doublespend that, invalidating both itself and the child transaction (the funding transaction).

Alternatively, Bob can also retrieve his own funds unilaterally after 8 confirmations in the situation when Alice abandonds the betting process.

```
contract ChainBetBobEscrow(PubKey alicePK, PubKey bobPK) {

    challenge cancel(Sig bobSig) {
        verify checkSequence(8b);
        verify checkSig(bobSig, bobPK);
    }

    challenge proceed(Sig aliceSig, Sig bobSig) {
        verify checkMultiSig([aliceSig, bobSig], [alicePK, bobPK]);
    }
}
```

10.2 Phase 5: Funding Transaction

Alice should now have both of Bob's signatures, so she can spend from both escrow addresses to create the (main) funding transaction. lice should wait until both escrow transactions have at least one confirmation before broadcasting the funding transaction. Otherwise, she risks a double spend attack where Bob learns her secret, discovers he has lost the bet, and then tries to double spend the input to the Bob escrow account.

Using a shorthand notation where Alice's Secret is "A" and the hash is "HASH_A", and Bob's Secret is "B" and its hash is "HASH_B", then we can say that the main P2SH address is a script that allows the funds to be spent if:

Alice can sign for her public key AND Hash(A)= HASH_A AND Hash(B)=HASH_B AND A+B is an odd number.

. . . or if Bob can sign for his public key AND Hash(A)= HASH_A AND Hash(B)=HASH_B AND A+B is an even number.

. . . or if Alice can sign for her public key and the transaction is more than 4 blocks old.

```
contract Bet(
    Ripemd160 aliceCommitment,
    Ripemd160 bobCommitment,
    PubKey alicePK,
    PubKey bobPK) {

    challenge odd(bin aliceSecret, bin bobSecret, Sig aliceSig, bool cancel) {
        if (!cancel) {
            verify hash160(aliceSecret) == aliceCommitment;
            verify hash160(bobSecret) == bobCommitment;

            bin [a, _] = aliceSecret @ 4;
```

(continues on next page)

(continued from previous page)

```
            bin [b, _] = bobSecret @ 4;
            verify (bin2num(a) + bin2num(b)) % 2 == 1;
        }
        else verify checkSequence(8b);

        verify checkSig(aliceSig, alicePK);
    }

    challenge even(bin aliceSecret, bin bobSecret, Sig bobSig) {
        verify hash160(aliceSecret) == aliceCommitment;
        verify hash160(bobSecret) == bobCommitment;

        bin [a, _] = aliceSecret @ 4;
        bin [b, _] = bobSecret @ 4;
        verify (bin2num(a) + bin2num(b)) % 2 == 0;

        verify checkSig(bobSig, bobPK);
    }
}
```

CHAPTER 11

Roadmap

Spedn is an early, experimental tool with a lot of plans:

- Macros
- Extended support for covenants and tx preimage introspection
- Compiled code optimizations
- IDE with a debugger
- . . . and more

Check out the Trello board to see what's currently going on.

CHAPTER 12

Contributing

Every kind of contribution is appreciated, especially:

- Syntax ideas and other features propositions
- Code review
- Unit tests
- Bug reports
- Usage examples and docs improvement

CHAPTER 13

Contract

- Telegram Channel
- Issue tracker
- #spedn-lang channel on Electron Cash Slack
- Twitter