Consider an account-based ledger. It might be organized by time and contain a bunch of transactions in order. When we add a new transaction and want to check if it's valid, we'd have to scan back all the way until the genesis block to verify that someone has the correct amount of money in their account. As a result, verification is slow. Miners would have to maintain some parallel data structure to speed up these checks, but that's a big requirement.

Bitcoin, instead, is transaction-based, rather than account-based. There's no notion of a "balance" for an individual address. Hash pointers tell you exactly where to look.

Scripts

Output "addresses" are really *scripts*. Scripts let you specify arbitrary conditions necessary to claim money. Input "addresses" are *also* scripts. The two get concatenated and run to verify that the transaction is legitimate. Usually called *scriptSig* ("input", or "unlock") and *scriptPubKey* ("output", or "lock").

Script is not Turing-complete, but that's by design. It makes the scripts very easy to run, computationally inexpensive, and safe. It's stack-based, has some cryptographic support, limits time and memory, and does not allow looping or recursion. There are 256 opcodes, which include arithmetic operations and some cryptography primitives, like a variety of hash functions.

99.9% of the time, the Bitcoin script follows a predictable format: "¡sig¿ ¡pubKey¿ OP_DUP OP_HASH160 ¡pubKeyHash¿ OP_EQUALVERIFY OP_CHECKSIG". This is just Pay-to-Public-Key. If anything goes wrong during Script execution, there'll be some error or you'll have something left on the stack at the end, in which case you know that it failed.

Roughly, $\tilde{0}.01\%$ are MULTISIG and $\tilde{0}.01\%$ are Pay-to-Script-Hash. The remaining scripts are errors or proof-of-burn.

Multisig

The "OP_CHECKMULTISIG" operator is particularly interesting. It provides built-in support for join signatures, where you can specify n public keys and say that you require t signatures for verification. Supports at most 15 keys.

Proof-of-burn

These scripts are of the form "OP_RETURN ¡arbitrary data¿". They burn money, because OP_RETURN always fails.

Should senders specify scripts?

Usually, the person who's sending the money has to execute the script. It's a little odd. Instead, with Pay-to-Script-Hash, you only send the hash of the script that's going to be executed.

Script Applications

Escrow transactions

Alice wants to buy online from Bob. Alice doesn't want to pay until after Bob ships. Bob doesn't want to ship until after Alice pays.

Pick an escrow agent, Judy, and pay to a 2-of-3 MULTISIG requiring two of Alice, Bob, and Judy. Judy can resolve disputes, etc.: for example, if Bob sends bad goods, she can rule with Alice to invalidate the transactions; or if Alice doesn't pay, Judy can still help Bob gets the funds. In addition, there's no way for Judy to steal the money, which is a nice property–she needs to cooperate with either Alice or Bob to move it. Note that if Judy dies or loses her private key, the money can be lost forever.

Green addresses

Alice wants to pay Bob. Bob doesn't want to wait for six verifications of confirmation (i.e., to guard against double-spends), or is offline completely.

You can introduce a trusted third-party, like a bank. Alice shows the bank her credit card and asks for the bank to make a "green payment" to Bob of the form *Pay x to Bob, y to Bank* (where y is the leftover of the coin the bank is spending). The bank should be trusted by Bob and can assume that there's no double-spend.

If the bank ever did double-spend, there'd be a huge credibility loss.

Efficient micro-payments

Alice is a phone customer, and Bob is the phone company. Alice wants to pay Bob for each minute of phone service, but she doesn't want to incur a transaction fee for each small payment.

We start with a MULTISIG transaction that requires Alice and Bob to verify. Alice sends Bob 100 Bitcoin–more than she'd ever want to spend on phone service. Alice uses her phone for a minute and produces a transaction that sends 1 Bitcoin to Bob and 99 to Alice, which is signed only by Alice (awaiting Bob's signature). She continues to send transactions for each minute. In the end, Bob signs the final transaction. He can't sign multiple transactions because it would be a double-spend.

If Bob never signs, the coins are burned. The fix: demand a timed refund transaction that returns her money at time t.

Forks

What happens if we try to upgrade Bitcoin (i.e., the Bitcoin software)? Why would we even want to do that in the first place?

Well, there are a lot of hard-coded limits (e.g., 10-minute average creation time per block, 1M bytes per block, 100M Satoshis per Bitcoin), and over time these may not be the best choices. In particular, there are some significant throughput limits in Bitcoin (turns out to support roughly 7 transactions per second), compared to 2,000-10,000 transactions per second for VISA. Bitcoin clearly cannot handle the volume of transactions that the world wants.

A "hard-forking" change to Bitcoin would be problematic, because nodes with the new software might accept certain blocks that the old nodes dislike. There are two types of incompatibility: the old nodes reject the new blocks, or the new nodes reject the old blocks (the latter results in an orphan block every time an old node mines a new block).

Soft forks are doable: these limit the set of valid transactions or the set of valid blocks. You need the majority of nodes to enforce these new rules, but the old nodes will still approve them. Pay-to-Script-Hash, for example, was a soft-fork.