CashScript: A high-level language for Bitcoin Cash Script

Rosco Kalis
rosco.kalis@student.uva.nl
rosco@bitcoin.com

July 31, 2019, 60 pages

Research supervisor: Adam Belloum, a.s.z.belloum@uva.nl
Industry supervisor: Gabriel Cardona, gabriel@bitcoin.com
Host organisation: Bitcoin.com, https://bitcoin.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

In the past years blockchain technology has seen a lot of interest, and especially *smart contracts* have risen in popularity. The biggest smart contract platform is Ethereum, but platforms like Bitcoin (BTC) and Bitcoin Cash (BCH) have support for simple smart contracts as well. Smart contracts in Bitcoin are written using a stack based assembly-like language called *Script*. Using Script is difficult and error-prone, like writing in Assembly. Some high-level languages exist for Script, most notably *Ivy* and *Spedn*. Ivy has better integration capabilities, while Spedn has more functionality. We design and implement CashScript to achieve Spedn's functionality while offering superior integration capabilities. We furthermore take inspiration from Ethereum's workflow to appeal to this body of developers. We evaluate CashScript by asking six participants to implement and integrate specific contracts. From this evaluation we conclude that CashScript is easier to integrate and less error-prone than the related work, although less efficient in terms of executable size. We also conclude that the CashScript language is syntactically similar to Solidity, but the workflow as a whole needs additional work to be familiar to Ethereum developers.

Contents

1 Introduction **6**
- 1.1 Research questions . . . 6
- 1.2 Contribution . . . 7
- 1.3 Outline . . . 7

2 Background **8**
- 2.1 Bitcoin . . . 8
  - 2.1.1 Blockchain . . . 8
  - 2.1.2 BTC & BCH . . . 9
  - 2.1.3 Bitcoin transactions . . . 10
  - 2.1.4 Bitcoin Script . . . 11
- 2.2 Compilers . . . 13

3 Related work **14**
- 3.1 Ivy . . . 14
- 3.2 BALZaC . . . 15
- 3.3 Spedn . . . 15
- 3.4 BitAuth Script . . . 16
- 3.5 Summary . . . 16

4 A high-level language for Bitcoin Cash Script **18**
- 4.1 Language design goals . . . 18
- 4.2 Bitcoin Script limitations . . . 19
- 4.3 Language specification . . . 19
  - 4.3.1 Control structures . . . 19
  - 4.3.2 Types . . . 19
  - 4.3.3 Type casting . . . 21
  - 4.3.4 Built-in functions . . . 21
  - 4.3.5 Global variables . . . 22
  - 4.3.6 Operators . . . 22
- 4.4 Artifacts . . . 22
  - 4.4.1 Artifact specification . . . 23
- 4.5 Extensions . . . 23

5 A compiler for CashScript **25**
- 5.1 Compiler implementation . . . 25
  - 5.1.1 Used tools . . . 25
  - 5.1.2 Lexical & syntax analysis . . . 26
  - 5.1.3 Semantic analysis . . . 26
  - 5.1.4 Code generation . . . 27
  - 5.1.5 Possible optimisations . . . 28
  - 5.1.6 Test suite . . . 29
- 5.2 Command line tool . . . 29
- 5.3 JavaScript SDK implementation . . . 30
  - 5.3.1 Contract . . . 30
  - 5.3.2 Instance . . . 30
  - 5.3.3 Transaction . . . 31

6 Evaluation **32**
6.1 Considered languages . . . 32
6.2 Metrics . . . 32
6.3 Setup & methods . . . 33
6.3.1 Contract implementation . . . 33
6.3.2 Contract integration . . . 33
6.3.3 Similarity to Ethereum's workflow . . . 34
6.4 Participants . . . 34

7 Results **35**
7.1 Overview of participants . . . 35
7.2 Contract implementation . . . 35
7.3 Contract integration . . . 37
7.4 Similarity to Ethereum's workflow . . . 38

8 Discussion **39**
8.1 Contract implementation . . . 39
8.2 Contract integration . . . 40
8.3 Similarity to Ethereum's workflow . . . 40
8.4 Threats to validity . . . 41
8.5 Conclusions . . . 41
8.6 Recommendations for further research . . . 41

References **43**

Acronyms **46**

Appendix A Code repository **47**

Appendix B CashScript grammar **48**

Appendix C Evaluation Assignment **51**
C.1 Contract implementation . . . 51
C.1.1 Specification . . . 51
C.1.2 Documentation links . . . 51
C.1.3 Setup . . . 52
C.1.4 Reference Implementations . . . 53
C.2 Contract Integration . . . 54
C.2.1 Participants & data . . . 54
C.2.2 Repository structure & setup . . . 54
C.2.3 Specification . . . 55
C.2.4 Documentation links . . . 55

Appendix D Literature study on programming language comparison **56**
D.1 Programming language comparison studies . . . 56
D.1.1 Henderson & Zorn (1994) [35] . . . 56
D.1.2 Prechelt (2000) [36] . . . 56
D.1.3 Mannila & De Raadt (2006) [37] . . . 56
D.1.4 Ebcioglu et al. (2006) [38] . . . 57
D.1.5 Fourment & Gillings (2008), [39] . . . 57
D.1.6 Bissiyandé et al. (2013) [40] . . . 57
D.1.7 Nanz et al. (2013) [41] . . . 57
D.1.8 Aruoba & Fernandez-Villaverde (2015) [42] . . . 57
D.1.9 Nanz & Furia (2015) [43] . . . 57
D.2 Applicability to Bitcoin Script languages . . . 58
D.2.1 Implementation-based metrics . . . 58
D.2.2 Static metrics . . . 58

List of Figures

2.1 Block contents and linking. 8
2.2 The BTC-BCH hard fork. 9
2.3 DAG of UTXOs. 10
2.4 Transaction with a change output. 11
2.5 Address generation from public key. 12
2.6 Pay-to-Script-Hash. 12

3.1 An example of an Ivy contract. 14
3.2 An example of a BALZaC transaction specification. 15
3.3 An example of a Spedn contract. 16
3.4 An example of a BitAuth Script template. 16

List of Tables

3.1 Overview of high-level Bitcoin Script languages. 17

4.1 Overview of typecasting. 21
4.2 Overview of operators with their precedence. 22

7.1 Overview of evaluation participants and their experience levels. 35
7.2 Results of the contract implementation in Bitcoin Script. 36
7.3 Results of the contract implementation in Spedn. 36
7.4 Results of the contract implementation in CashScript. 37
7.5 Results of the contract integration. 37
7.6 Results on similarity between the workflows of CashScript and Ethereum. 38

D.1 Overview of the quality and contents of the studies. 59
D.2 Overview of the metrics used in implementation evaluations. 60
D.3 Overview of the language features that were compared that are not based on implementations. Only features that occur in more than one paper are included. 60

Chapter 1

Introduction

In the past years blockchain technology has seen a lot of interest. The developer community in the blockchain space is steadily growing and has doubled in the past two years [1]. Developers are coming up with novel uses of blockchain using so-called *smart contracts*. The biggest smart contract platform on the market is Ethereum, but other platforms like Bitcoin (BTC) and Bitcoin Cash (BCH) have support for simple smart contracts as well.

Smart contracts in Bitcoin are written using a stack-based assembly-like language called *Script*, or *Bitcoin Script* to avoid ambiguity. At the moment, complex uses of Script are not very widespread, as it can be difficult and error-prone to write it by hand. In general purpose programming, there is a wide range of high-level languages that can be used in place of low-level bytecode. This kind of ecosystem is lacking in the case of Bitcoin Script.

A few implementations of high-level languages do exist for Bitcoin Script. The most feature-complete ones are Ivy [2] and Spedn [3]. Ivy is created for BTC, while Spedn is created for BCH. Ivy's functionality is limited to signature, time, and equality checks [2], whereas Spedn has support for all functionality that Bitcoin Script allows [3]. Ivy can be integrated into JavaScript projects through their Node Package Manager (NPM) package [2], while Spedn is only available as a command line tool that compiles into Bitcoin Script [3].

Although a high-level language makes it easier to write these scripts, it is still not trivial to integrate the compiled Bitcoin Script into applications. This is where Ethereum stands head and shoulders above Bitcoin, as smart contracts in Ethereum enjoy high levels of abstraction in both writing *and* usage. Several Ethereum Software Development Kits (SDKs) exist for different general purpose programming languages, such as web3js for JavaScript [4]. These SDKs make it possible to call smart contract functions as if they were regular functions native to the general purpose language.

This is why we create CashScript, a new high-level Bitcoin Script language and SDK that offers a similar workflow to Ethereum's Solidity language and web3js SDK. Ethereum's smart contract capabilities are far more advanced than Bitcoin's [5, 6], so this project is not attempting to replicate Ethereum-like functionality, but rather its workflow. This additionally allows developers from the Ethereum community to get involved with BCH more easily and vice-versa, which can improve collaboration.

CashScript sets itself apart from the existing Bitcoin Script languages by combining the full functionality that Spedn offers, with being easily integrated into JavaScript applications like Ivy. The language furthermore takes inspiration from Ethereum's Solidity and web3js. This creates more advanced integration capabilities than Ivy, and allows developers to cross over between Ethereum and BCH.

1.1 Research questions

To conduct our research we formulate the following research questions:

> *Can we create a high-level language and SDK for Bitcoin Script that has the same extent of functionality as the existing work but is easier to integrate into JavaScript applications?* (1)

We create a language and SDK that has the same extent of functionality that Spedn and the other related work offers. In addition we create an SDK that can be used to seamlessly integrate our language into JavaScript applications.

Can we create a high-level language and SDK for Bitcoin Script that offers a similar workflow to Ethereum? (2)

We research the syntax of the Solidity language, and we look at the limitations of Bitcoin Script. We keep these limitations in mind and replicate the concepts in Solidity that can be transferred. We then create an SDK that offers a similar user experience to Ethereum's biggest JavaScript libraries - web3js and Truffle. This is done by wrapping CashScript contracts in JavaScript objects, allowing users to call their functions as if they were regular JavaScript functions.

1.2 Contribution

The main contribution of this work is a high-level language and SDK for Bitcoin Script. This work sets itself apart by combining the extent of functionality of Spedn with significantly better integration capabilities, offering a similar workflow to Ethereum.

1.3 Outline

Chapter 2 provides the required background information on Bitcoin and compiler construction to be able to understand the thesis. Chapter 3 contains an overview of the related Bitcoin Script languages. Chapter 4 discusses the design and specification of the CashScript language. Chapter 5 discusses the implementation of the compiler of this high-level language and the related JavaScript SDK. Chapter 6 discusses the language's evaluation. Chapter 7 presents the results of this evaluation. Chapter 8 discusses the implications of the results, and extracts the main findings of the research.

Chapter 2

Background

2.1 Bitcoin

Bitcoin is a peer-to-peer electronic cash system first proposed in 2008 by the pseudonymous Satoshi Nakamoto. Bitcoin uses a blockchain to distribute its ledger over a network of independent nodes so that there is no single point of failure, and no central control that might be compromised. It uses a consensus algorithm called *Proof-of-Work* that allows these independent nodes to approve correct transactions and reject malicious ones [7].

2.1.1 Blockchain

The blockchain is a data structure that is distributed over a number of independent nodes. It derives its name from the chain of so-called *blocks* that it uses to store its data [8]. Block headers include the root of a *Merkle tree* – a special kind of tree that allows quick validation of data through hashes [9]. This Merkle tree is used to store the actual data inside these blocks. To make the chain resistant to manipulation, block headers also include a timestamp and a hash of the previous block [8]. The way these blocks are structured and linked together can be seen in figure 2.1.

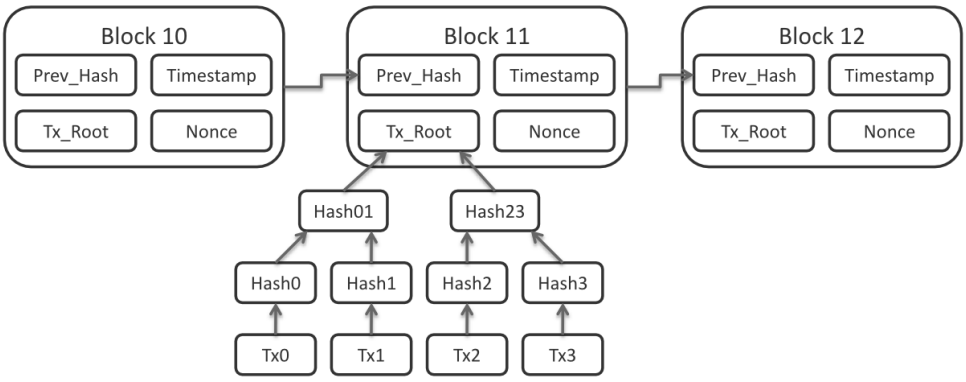

Figure 2.1: Block contents and linking.
Downloaded from [8]

The Bitcoin blockchain and many other public blockchains use a *consensus algorithm* called Proof-of-Work (PoW) [9]. This algorithm works by attaching a *nonce* to every block header, and changing this nonce until the hash of the block matches a certain prefix. This process is called *mining*, and is attempted by many nodes at the same time, until one of them has found a correct solution. This process of mining is very expensive, but other nodes can verify the solution very quickly [7].

Mining is also the process by which new Bitcoins are introduced to the total monetary supply. Miners validate transactions and secure the network for which they are paid new coins – the *block reward* – in a special transaction called a *Coinbase* transaction. This attaches a financial risk on incorrectly validating transactions through the high costs of the mining process. At the same time it attaches a financial reward on correctly validating transactions through the block reward. This process ensures that the mutually distrusting network can work together to validate transactions [7].

2.1.2 BTC & BCH

In 2017, Bitcoin faced big scalability issues, as the network was unable to process the amount of transactions that were coming in. This caused very long confirmation times and very high transaction fees [1]. By that time, a debate on tackling these expected issues had been going on for several years. In those years it had not been possible to reach a consensus on this, but the urgency demanded that a solution had to be found, lest the high fees and transaction times would prevent users to adopt the technology.

One of the proposed solutions was to increase the maximum blocksize to accommodate more transactions per second and reduce the transaction fees. The blocksize limit at the time was set to 1 MB maximum as it had been since the early days of Bitcoin [2]. However, computers have become faster and more efficient, so they should be able to handle the higher block sizes [10].

Not everyone agreed with this solution, as a number of people argued this would increase node operating costs, which would decrease decentralisation. They valued node operating costs over the increased transaction throughput and decreased costs that raising the blocksize limit would bring. Instead they wanted to implement an algorithm called *SegWit*, which separates a transaction's unlocking signature, or *witness* from the rest of the transaction data. One of the effects of SegWit is that it allows blocks to hold more transactions without technically raising the blocksize limit, although its main goal is to counter transaction malleability [3].

Ultimately, the different camps were unable to reach a compromise on the solution, which caused the network to execute a so-called *hard fork*, or split, which resulted in two separate chains [11]. Of the two resulting chains, one has kept the original Bitcoin name (BTC), while the other was branded Bitcoin Cash (BCH). This hard fork can be seen in figure 2.2.

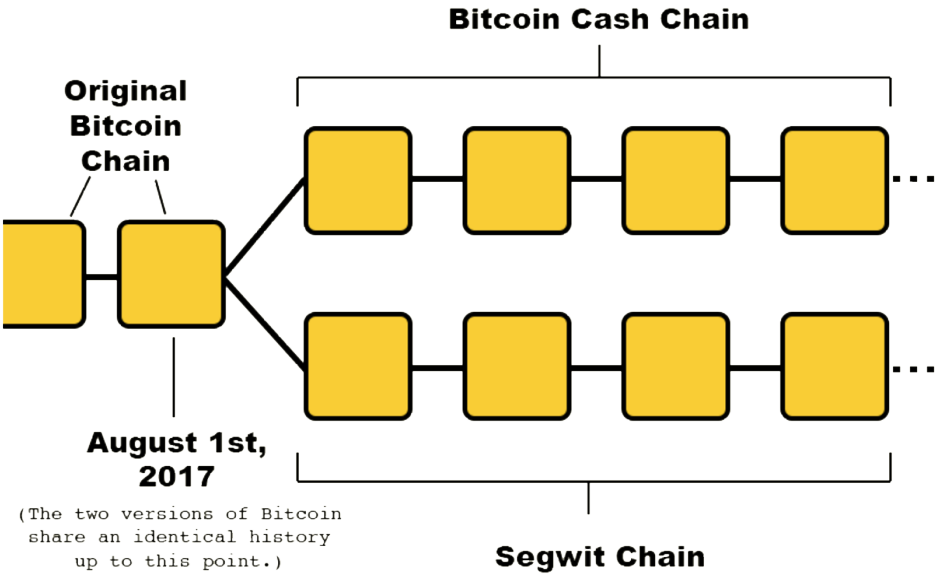

Figure 2.2: The BTC-BCH hard fork.
Downloaded from Bitcoin.com

BTC has since then partially adopted SegWit and plans to use a second layer solution called the *Lightning Network* as its primary means for scaling. This solution is still in its early stages, and can not be reliably used yet. BCH has since then raised the blocksize limit to 32 MB, and has implemented several other features to improve scalability and enhance functionality such as Canonical Transaction Ordering (CTOR) and Schnorr signatures.

BTC has so far done better in terms of price and has seen more usage [4] [5], although its scaling has not improved by much since the hard fork [6]. On the other hand, BCH's price is a lot lower and it gets used less, but it has the possibility to scale to many more transactions while keeping lower fees and achieving

[1] https://bitinfocharts.com/comparison/transactionfees-btc-bch.html#log
[2] https://github.com/bitcoin/bitcoin/commit/a30b56ebe76ffff9f9cc8a6667186179413c6349
[3] https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki
[4] https://bitinfocharts.com/comparison/transactions-btc-bch.html#log
[5] https://bitinfocharts.com/comparison/price-btc-bch.html#log
[6] https://bitinfocharts.com/comparison/size-btc-bch.html#log

higher throughput. This has been demonstrated in several *stress tests* that have been executed on the BCH network in the past year [7] [8].

For brevity, the term Bitcoin is used to refer to both BTC and BCH in the remainder of this work. Whenever it is necessary to make the distinction, we refer to the two different chains as BTC and BCH.

2.1.3 Bitcoin transactions

While Bitcoin's transaction model is quite complex, it is easy to use Bitcoin without knowing how transactions work technically. Users can use their Bitcoin wallet to transact Bitcoin with other Bitcoin users, while their wallet application handles all underlying abstractions. In order to explore Bitcoin's Script language, however, it is necessary to understand how Bitcoin transactions work under the hood.

Bitcoin transactions use so called *transaction outputs*, which are chunks of Bitcoin currency. Whenever such an output is available and spendable, it is called an Unspent Transaction Output (UTXO). Every Bitcoin full node keeps track of all existing UTXOs which is called the *UTXO set*. A user's Bitcoin balance is simply the sum of all UTXOs in the UTXO set that can be spent by the user's wallet. There are blockchain indexing services that make this UTXO set easily accessible, so that wallets don't have to crawl the blockchain for UTXOs themselves [5].

UTXOs are used as inputs for Bitcoin transactions, and produce new UTXOs as outputs, which can be consumed by future transactions. Multiple UTXOs can be used as inputs for a single transaction, and multiple UTXOs can be produced by a single transaction. Effectively, chunks of Bitcoin value move in a Directed Acyclic Graph (DAG) of transactions consuming and producing UTXOs. An exception to this is the *coinbase* transaction, which is the first transaction in every block and has no UTXOs as inputs. Instead this coinbase transaction creates a new UTXO that is spendable by the miner as a reward for mining the block [5]. An excerpt of such a DAG can be seen in figure 2.3

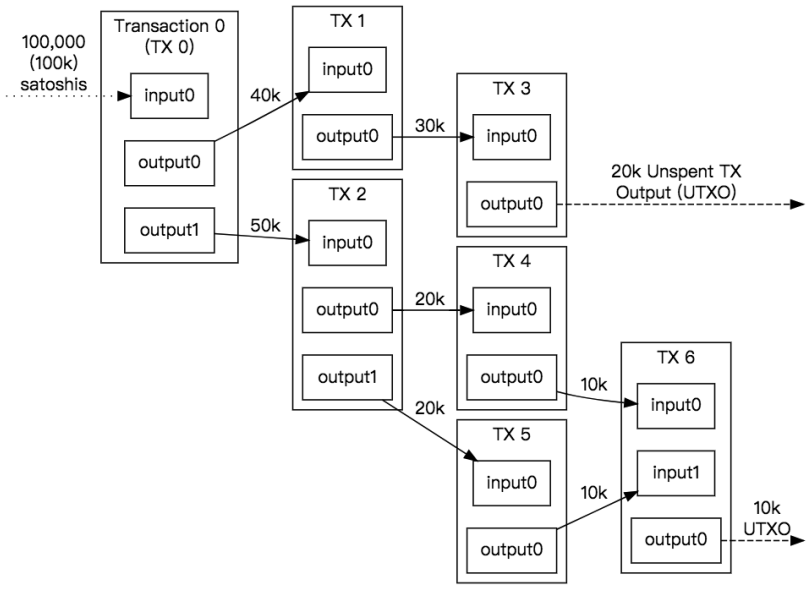

Figure 2.3: DAG of UTXOs.
Downloaded from [12]

Transaction outputs can have an arbitrary integer value denominated in the unit *satoshi*, which represents $1/10^8$ of a Bitcoin. The output is indivisible after it is created, which means that any UTXO needs to be spent in its entirety inside a transaction. So whenever a user wishes to use a UTXO worth 1000 satoshis to send 100 satoshis to someone, their transaction needs to pay 100 satoshis to the other party, and pay 900 in change back to their own wallet. Realistically, part of the input funds would be reserved for transaction fees as well [5]. An example of this kind of *change transaction* can be seen in figure 2.4.

[7] `https://cointelegraph.com/news/bitcoin-cash-stress-test-results-21-million-transactions-cause-no-surge-in-fees`
[8] `https://news.bitcoin.com/new-bitcoin-cash-stress-test-sees-700000-transactions-in-one-day/`

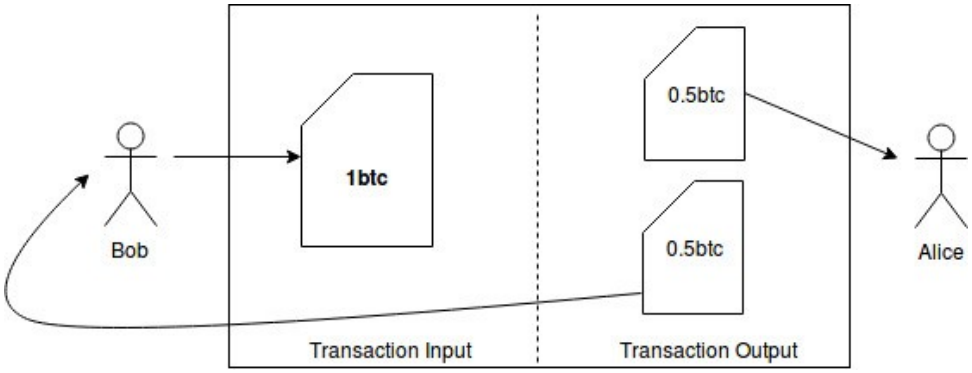

Figure 2.4: Transaction with a change output.
Downloaded from `https://medium.com/cybermiles/diving-into-ethereums-world-state-c893102030ed`

Now that we understand the building blocks and basic structure of Bitcoin transactions, we can study how Bitcoin makes sure these transactions can only be spent by the "owner" of the UTXO. When a transaction is created, a so-called *locking script* is included with the output. This script specifies the conditions that must be met to spend the output [5].

To spend this output, an *unlocking script* is provided, which satisfies the conditions specified in the locking script, and allows the output to be spent. The spend authorisation is considered valid if and only if the unlocking and locking scripts are able to execute completely, and the resulting value is TRUE (non-empty byte sequence). All Bitcoin validating nodes execute the unlocking and locking scripts to validate that the unlocking script actually satisfies the locking script [5].

2.1.4 Bitcoin Script

Locking and unlocking scripts are written using Bitcoin's transaction scripting language, creatively named *Script*. To avoid ambiguity, it can also be referred to as *Bitcoin Script*. Bitcoin Script is a stack based assembly-like language that is executed from left to right. It is designed to be limited in scope, and is intentionally not Turing complete, as its main use is the validation of programmable money, not general purpose computing [5].

Bitcoin Script is stateless, meaning it only uses the information contained within the script itself. This statelessness means that a Script can be deterministically validated on any machine [5]. This gives increased performance and predictability, although it does limit the usefulness of the scripting language [13]. In contrast, smart contracts on Ethereum have a persistent state that can be changed with smart contract interactions. A simple example is the ERC20 token standard [9], which allows anyone to encode a token inside a smart contract. The contract keeps track of the balances of all token holders, and token holders can use the smart contract to transact with other users.

Such functionality is impossible using only Bitcoin Script, as Bitcoin Script can not keep track of these balances and act accordingly. But there are solutions in development that work around this by creating hybrid solutions, such as the Simple Ledger Protocol (SLP) token standard on BCH.

Theoretically, any kind of locking / unlocking script can be used in Bitcoin transactions. However, only *standard* scripts are accepted by mainnet miners for security reasons [12]. The types of scripts that are considered standard by the Bitcoin mining software are Pay-to-Public-Key (P2PK), Pay-to-Public-Key-Hash (P2PKH), Pay-to-Script-Hash (P2SH), Pay-to-Multi-Signature (P2MS) and NULLDATA. Of these, only P2PKH, NULLDATA and P2SH are actively used, so these are the ones we elaborate on.

The most simple script type is P2PK. In this script type, the locking script contains a public key with a *CHECKSIG* opcode. The unlocking script provides a signature corresponding to the public key. The signature and public key are checked with this CHECKSIG opcode, and the UTXO is spendable if the signature and public key match [14].

[9] `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`

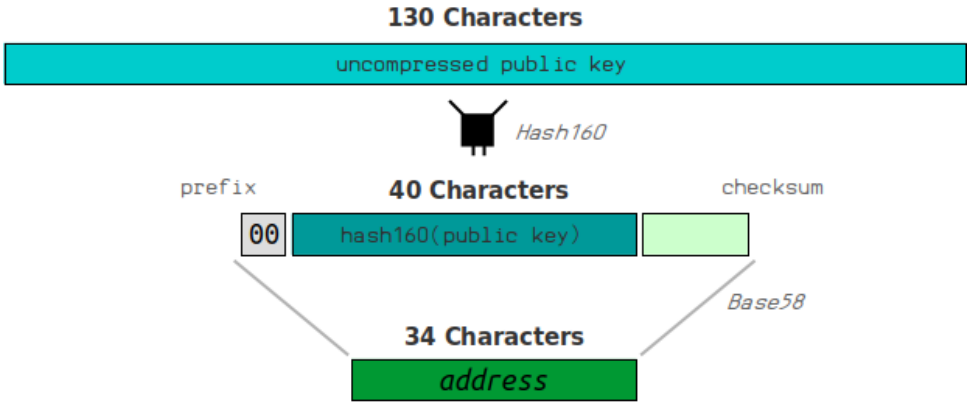

Figure 2.5: Address generation from public key.
Downloaded from [14]

To make it more convenient for users to transact Bitcoin, the P2PKH type was developed, where the locking script contains a public key hash instead of the public key itself. The unlocking script then provides a corresponding public key and signature. The UTXO is spendable if a hash of the provided public key matches the hash in the locking script and the provided public key and signature match each other. A public key hash is translatable to a *Bitcoin address* (figure 2.5), which is a lot shorter than a full public key. This allows users to share their addresses with each other instead of the much longer public key. Almost all regular Bitcoin transactions use P2PKH [5, 14].

The NULLDATA script type is used to embed data on the blockchain. Using the OP_RETURN opcode inside a locking script results in a *provably unspendable* transaction output, which does not need to be stored in the UTXO set [5]. As a result, any Bitcoin amount that is locked inside this output is lost forever. OP_RETURN can be used to store any data on the blockchain and enables simple use-cases such as Proof of Existence [10] or Memo.cash [11], but can also be used for more sophisticated on-chain protocols, such as SLP [12], a protocol that allows sub-currencies, or *tokens*, to be created and transacted on the BCH chain [15].

To enable the creation of other, more complex locking scripts, P2SH can be used. The locking script of a P2SH output contains the hash of a custom, more complex script, called the *redeem script*. To spend this output, the user has to provide the original redeem script, as well as the unlocking script to this redeem script. The execution of a P2SH script is a bit different than P2PKH, in that it is executed in two parts. First the redeem script is hashed and checked against the hash in the locking script. If they match, the unlocking script is used to unlock the redeem script as if the redeem script was the initial locking script (figure 2.6) [5, 14].

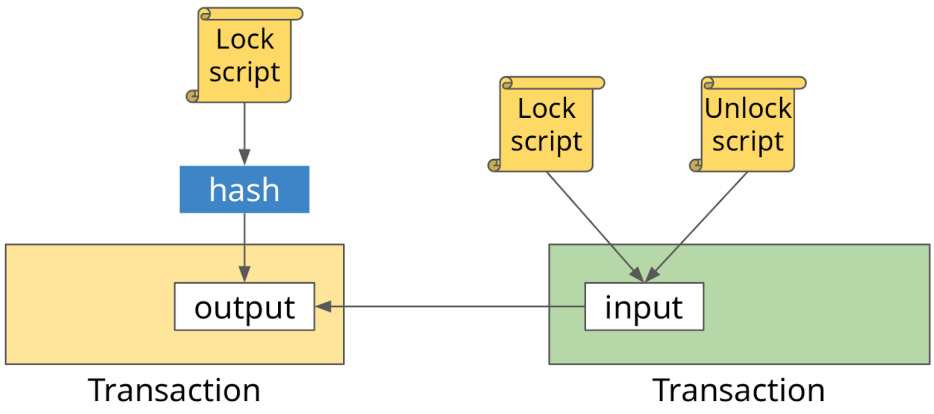

Figure 2.6: Pay-to-Script-Hash.
Downloaded from `https://medium.com/codechain/renewed-pay-to-script-hash-60df881e1613`

[10] `https://proofofexistence.com/`
[11] `https://memo.cash/`
[12] `https://simpleledger.cash/`

At first it may seem unnecessarily complicated to use P2SH compared to using the redeem script as a regular locking script. However, P2SH offers several benefits. It shifts the burden of constructing the locking script from the sender to the recipient. Without P2SH, every sender would have to know the redeem script in order to send Bitcoin to this script. With P2SH the sender only needs to know the hash of this script, while the recipient needs to hold the full script. The script hash can be transformed to a regular Bitcoin address through the process outlined in figure 2.5. This means a sender can send Bitcoin to this script like sending Bitcoin to a regular address. It also shifts the burden of script storage from the output – which is stored on the blockchain and in the UTXO set – to the input – which is only stored on the blockchain. This increases overall performance, as it keeps the UTXO set smaller [5].

While Script is not Turing complete and quite limited in scope, there are interesting "smart contracts" that can be encoded into these P2SH redeem scripts. One of the common examples is an m-of-n multi-signature wallet, where there are n owners to a wallet, and at least m signatures are needed before an output can be spent [5]. Another example is a hash collision contract, where an output can be spent by providing a hash collision for some hashing algorithm [14]. BCH is looking to innovate in the Script area, and it has recently enabled the new OP_CHECKDATASIG opcode that enables the use of *oracles* [16] and covenants [17–19].

2.2 Compilers

A compiler is a program that can read a program in one language - the *source* language - and translate it into an equivalent program in another language - the *target* language. In the process a compiler also reports any errors in the source program. A compiler typically goes through several phases phases: lexical analysis, syntax analysis, semantic analysis, optimisation, and code generation [20]. These compiler phases are revisited in chapter 5 when the compiler implementation is discussed.

The *lexical analyser* – or *lexer* – reads the stream of characters inside the source program, and groups these characters into meaningful sequences, or *tokens*. These tokens commonly have the form (*token-name*, *attribute-value*). The token name is an abstract symbol that can be used during the syntax analysis. The attribute value usually points to an entry in the symbol table for the token, which is used in the later stages of semantic analysis and code generation [20].

After the lexical analysis, the resulting token stream is passed on to the *syntax analyser* – or *parser*. These tokens are used to create a parse tree, a tree-like intermediate representation that depicts the grammatical structure of the token stream [20]. This parse tree is usually quite close to the actual structure of the source code, so it is transformed into an Abstract Syntax Tree (AST), which is a more abstract representation. During the construction of the trees the parser makes sure that the order of operations is correctly encoded.

The *semantic analyser* uses the AST to build up a symbol table that includes all defined symbols inside the source code. It uses this symbol table to check for semantic consistency, which includes checking that all variables have been declared before they are used, and that there's no redefinitions of variables. Another important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. The compiler should throw an error, for example, if a floating point number is used to index an integer-indexed array. Depending on the language, it might be possible to perform *type coercion*, where a type is implicitly casted to a different one, to prevent these errors [20].

At some point in the process, the compiler executes several rounds of optimisation, unless disabled by the user. Depending on the objective, optimisation can mean different things. Usually the objective of the optimisation is execution speed, but it can also be the size of the target code or energy consumption [20]. In the case of Ethereum's Solidity language for example, compiler optimisation targets a lower *gas usage*, meaning that it's cheaper to execute [21].

The final step of a compiler is always the generation of the target program. The code generator takes the intermediate representation of the source program and maps it into the target [20]. Depending on the source language and target language this could have different levels of complexity. The compilation of a high-level language such as Lua into bytecode that can be executed on the machine is more complicated than the compilation of a language like TypeScript which compiles into JavaScript, a language on a similar level of abstraction.

Chapter 3

Related work

3.1 Ivy

Ivy is the oldest of the reviewed languages, as it was released in December of 2017 during the 2017 cryptocurrency bull-run [22]. It was created for BTC, without explicit support for BCH. Ivy is written in TypeScript and provides a JavaScript SDK for easy integration with other projects in the JavaScript ecosystem. The SDK makes it possible to compile, instantiate, and use the contracts written with Ivy [2]. The syntax of the language is similar to TypeScript, making it familiar to many web developers. An example of the language can be seen in figure 3.1.

Ivy's functionality as a language is very limited, as there is a lack of variable definitions and assignments. There is also a lack of most expressions, such as arithmetic and comparison operators other than equality / inequality. This makes Ivy very safe and very simple since many error-prone operations are not allowed. At the same time, it limits the usefulness of the language, as it only supports signature, time checks, and equality checks [2].

The authors of the language deem it too immature to be used in production, and the JavaScript SDK doesn't even allow users to create real transactions – either mainnet or testnet [2]. At the same time the project has seen no significant improvements since February of 2018 [1], so it appears that these immaturities will not be resolved.

```
contract TransferWithTimeout(
  sender: PublicKey,
  recipient: PublicKey,
  timeout: Time,
  val: Value
) {
  clause transfer(senderSig: Signature, recipientSig: Signature) {
    verify checkSig(sender, senderSig)
    verify checkSig(recipient, recipientSig)
    unlock val
  }
  clause timeout(senderSig: Signature) {
    verify checkSig(sender, senderSig)
    verify after(timeout)
    unlock val
  }
}
```

Figure 3.1: An example of an Ivy contract.
Extracted from [2]

[1] https://github.com/ivy-lang/ivy-bitcoin/commits

3.2 BALZaC

The second high-level interface that is focused on BTC is BALZaC, which is not necessarily a high-level language that compiles to Bitcoin Script, but instead a language to specify Bitcoin transactions in general [23]. This can be valuable as it removes the step of integrating the compiler output into a Bitcoin transaction, since the output already is a Bitcoin transaction. However, this integration step could also be done with an SDK as is the case with Ivy.

BALZaC is the only project that is academic rather than industry, and the language is based on the authors' paper *A formal model of Bitcoin transactions* [24]. This gives the project scientific backing, but as a result the look and feel of the language are also very formal and academic. An example of the language can be seen in figure 3.2.

The language allows you to specify *inputs*, which are unlock arguments, and *outputs*, which is the script that locks the UTXO. The scripting language used for this is a functional language that uses a simple expression syntax allowing for arithmetic and comparison operations, as well as time-based operations and signature checks [23]. This is more functionality than Ivy supports, but still lacks features such as variable assignment and straightforward conditional execution.

The only way to use the language is through its Integrated Development Environment (IDE) [2], which means that integration possibilities into Bitcoin applications and libraries are impossible without manual work. Because of this the language is mainly usable as a theoretical exercise and a way of formally specifying Bitcoin transactions, but not to actually be used in applications.

```
transaction T1 {
    input = T: 42
    output = 50 BTC: fun(x). x != 0  // any constraint chosen by the user
}
```

Figure 3.2: An example of a BALZaC transaction specification.
Extracted from [23]

3.3 Spedn

Spedn is a newer smart contract language that has been created specifically for BCH and it was first released in October of 2018 [3]. Since then it has been kept up to date, and it still experiences regular commits from its author [3]. This active development and constant improvement signal a higher level of maintenance than Ivy.

The documentation states that the language has a *familiar C-like syntax*, and it stays quite close to the underlying target code – the Bitcoin Script [3]. This suggests that the language is accessible to developers familiar with C, which is definitely a sizeable part of the global developer community. However, many modern applications are written with JavaScript and web technologies [4]. An example of the language can be seen in figure 3.3.

Spedn is written in Haskell, which lends itself very well for the construction of compilers as it has excellent pattern matching support. However, Haskell is not a very popular language, so there are not many developers familiar with Haskell tooling. At the start of this project, the Spedn Command Line Interface (CLI) could only be installed by downloading the source and building it using Haskell tools. Since then the author has updated the language, and it is currently available as a global NPM package, which largely mitigates the drawbacks of using Haskell.

The output of Spedn compilation is Bitcoin Script. To send money to this script, an address needs to be generated from it. And to spend money, there are many manual steps due to a lack of SDK support. While the language does makes Bitcoin Script more accessible, intricate knowledge of Bitcoin and Bitcoin Script is still required to use these Spedn contracts.

[2] `https://blockchain.unica.it/balzac/`
[3] https://bitbucket.org/o-studio/spedn
[4] https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/

```
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);
    }
}
```

Figure 3.3: An example of a Spedn contract.
Extracted from [3]

3.4 BitAuth Script

BitAuth Script is the newest kid on the block – released early 2019 – and also the one with the lowest level of abstraction. Rather than defining a new syntax, BitAuth Script can be seen as a templating engine for Bitcoin Script. By surrounding a statement with brackets, this statement can be used to push data into the script, which allows the users to add variables to their Script that can be filled in at a later time. An example of the language can be seen in figure 3.4.

BitAuth Script was created as part of the larger BitAuth IDE, which in turn is a part of a larger BitAuth project. This means that there are currently no ways to integrate this language into any web or other applications. However, this project is still under very active development, so this is likely to change in the near future.

```
OP_DUP OP_HASH160 <pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG
```

Figure 3.4: An example of a BitAuth Script template.
Downloaded from [25]

3.5 Summary

Spedn has all functionality we expect from a high-level Bitcoin Script language, but it lacks the integration capabilities that Ivy offers. Ivy lacks the scope of Spedn's functionality, and is targeted at BTC rather than BCH. Besides this, the project has been abandoned for over a year, and is unlikely to be usable at any point in the future. BALZaC is an interesting project, but is too academic to be practically used in industry, and lacks integration possibilities. BitAuth Script looks like a great project, but it lacks maturity, and it is still too close to Bitcoin Script to be classified as a high-level language.

These languages all clearly have different goals and different audiences in mind. Spedn aims to be familiar for C developers, BALZaC is geared towards academics, Ivy clearly targets web developers, and BitAuth Script mainly provides a better way to write Bitcoin Script for those that are already familiar with it. One thing these projects neglect is that there is a large community of blockchain developers that already work with Ethereum. Allowing these developers to cross over to BCH can lead to improved collaboration between the communities.

With CashScript we combine Spedn's functionality with Ivy's integration capabilities. We furthermore focus on creating a syntax that is familiar to Ethereum developers and creating an SDK with a similar workflow to Ethereum's. A comparison between the related work and our proposed language CashScript can be seen in figure 3.1.

Table 3.1: Overview of high-level Bitcoin Script languages.

	Bitcoin fork	Opcode support	Implementation language	Tooling	Syntax similarity	Maintenance	License
Ivy [5]	BTC	Limited	TypeScript	JavaScript SDK	TypeScript	Abandoned	MIT
BALZaC [6]	BTC	Limited	Java	Online IDE	N/A	Active	N/A
Spedn [7]	BCH	Full	Haskell	CLI	C	Active	MIT
BitAuth [8]	BCH	Full	TypeScript	Online IDE	Bitcoin Script	Active	MIT
CashScript [9]	BCH	Full	TypeScript	CLI & JavaScript SDK	Solidity	Active	MIT

[5] https://github.com/ivy-lang/ivy-bitcoin
[6] https://github.com/balzac-lang/balzac
[7] https://bitbucket.org/o-studio/spedn/
[8] https://github.com/bitauth/bitauth-ide
[9] https://github.com/Bitcoin-com/cashscript

Chapter 4

A high-level language for Bitcoin Cash Script

4.1 Language design goals

Something that should be avoided in language design is the violation of expectations. What this means is that we should not take familiar constructs and have them behave in a different way than the norm. An example of this is the usage of the '%' sign as a comment in languages such as *Turing* or *LaTeX*, or the automatic list sorting of the *ABC* language. This also means it's detrimental to use unexpected keywords for familiar tasks, such as using the keyword **cond** instead of **if**, or using **x** as the multiplication operator rather than ***** [26].

This is reinforced by the designer of the D programming language who advocates for using "tried and true" patterns and constructs as this enables faster learning and easier usage of the language. He also states that people tend to optimise for minimising keystrokes, which is detrimental, as the time spent actually typing is never the bottleneck of software development. Additionally, the written code needs to be readable for the author and for the people they work with, and minimising keystrokes can affect the readability negatively [27].

According to GitHub, the most popular programming language is JavaScript [1]. This reinforces our belief that it is important to have a Bitcoin Script language that is easily integrated with JavaScript projects. In the smart contract development ecosystem, Solidity is the most used language [28], and is used to write smart contracts for the Ethereum blockchain and compatible chains. We want to focus on making CashScript familiar for Solidity developers to leverage these existing developers.

Because Bitcoin Script potentially controls large amounts of actual funds, safety is a very important feature of the language. One way of defining programming language safety is measuring how early errors can be detected [29]. This is doubly important for Bitcoin Script as it is not possible to change any errors in these contracts as soon as they are being used. Toby Ho lists several important classes of errors in the context of safety, namely null pointers, array index out of bounds, type errors, or incorrect function call arguments [29]. These safety issues need to be addressed in the language design.

The existing languages share this focus on safety. Ivy adds safety by disabling a large chunk of functionality that could be expected from a programming language [2]. Spedn opts to keep most of this functionality, but has a very elaborate type system with many different types and very strict type checking [3]. With CashScript, we do not want to limit functionality in the way Ivy does, but at the same time we believe that the amount of types that Spedn uses ends up being detrimental to productivity. We want to hold on to the safety features that this type system provides, without being a burden on the user.

This leads to the following concrete design goals.

- CashScript should be familiar to Solidity developers
- CashScript should not require Bitcoin Script knowledge
- CashScript should have strong safety without compromising on usability

[1] `https://octoverse.github.com/projects#languages`

4.2 Bitcoin Script limitations

Bitcoin Script is intentionally not Turing complete and has several functional limitations that need to be taken into account when designing a high-level language on top of it. In this section we discuss the way these limitations affect the design and implementation of CashScript.

The most important limitation is the lack of loop support. This means that most loops are also impossible to implement in a high-level language, although simple bounded loops could be implemented and unrolled by the compiler. We do not think this will provide meaningful functionality, and do not implement these kinds of loops in CashScript. The lack of loops also diminishes the usefulness of arrays or lists. If lists can not be iterated over, they are basically a convoluted way of storing variables that might as well be stored in a named variable. This is why we exclude arrays or lists from our language for now.

Another limitation is in the script size. Bitcoin Scripts have a maximum size of 10000 bytes [2], which means that transactions above that limit can not be created at all. There is an additional rule that states the max size of a script element is 520 bytes [3]. With P2SH, the redeem script needs to be hashed, for which it needs to be pushed to the stack in its entirety. This means that the redeem script is effectively a single script element, and has to conform to this 520 byte limit. Another limitation is on the number of operations a script may contain, which is set to 201 operations [4]. These limitations don't influence CashScript's functionality, but they do place an increased importance on the size of the compiled output.

A final important limitation is the lack of some arithmetic operations. Bitcoin Script has a number of arithmetic opcodes that were disabled because they caused bugs at the time. These opcodes are still disabled on the BTC chain, but BCH has made steps to address the issues in these opcodes and re-enable them [5]. However, the OP_MUL opcode for multiplication remains disabled for the time being. Simple multiplications could be unrolled into several additions, but this is not productive, especially considering the Script size limits. Eventually, it is planned to re-enable the OP_MUL opcode, which is why we do not attempt to implement unrolled multiplication.

4.3 Language specification

In this section we give an overview of the structure of a CashScript file, the available control structures, types, operators, and built-in functions. In addition to this specification, the full language grammar of CashScript is included in appendix B in the form of the Another Tool For Language Recognition (ANTLR) grammar specification that is used to parse CashScript source files.

A CashScript file contains exactly one contract definition. A contract in CashScript is a collection of functions that can be used to spend the funds that are locked in this contract. These contracts can be instantiated using the contract's parameters, and their functions can be called by specifying the correct function parameters.

4.3.1 Control structures

The only control structures in CashScript are `if` and `else`, with loops and return statements left out due to their incompatibility with the underlying Bitcoin Script. If-else statements follow the usual semantics known from C or JavaScript. Parentheses can not be omitted for conditionals, but curly braces can be omitted around single-statement bodies. There is no implicit type conversion from non-Boolean to Boolean types as there is in C and JavaScript, so `if (1) { ... }` is not valid CashScript.

4.3.2 Types

We include several types that can be used inside CashScript smart contracts. In the compiled Bitcoin Script, all these types are represented as generic byte sequences. An exception to this is the integer type, which is encoded as little-endian minimally encoded byte sequence. The additional types are added for convenience and type safety, as this allows the compiler to catch potential type errors at compile time or in the JavaScript SDK.

[2] https://github.com/Bitcoin-ABC/bitcoin-abc/blob/master/src/script/script.h#L31-L32
[3] https://github.com/Bitcoin-ABC/bitcoin-abc/blob/master/src/script/script.h#L23
[4] https://github.com/Bitcoin-ABC/bitcoin-abc/blob/master/src/script/script.h#L26
[5] https://www.bitcoincash.org/spec/may-2018-reenabled-opcodes.html

Boolean

`bool`: The possible values are constants true and false.
Operators:

- `!` (logical negation)
- `&&` (logical conjunction, "and")
- `||` (logical disjunction, "or")
- `==` (equality)
- `!=` (inequality)

The operators `||` and `&&` don't apply common short-circuiting rules. This means that in the expression `f(x) || g(y)`, `g(y)` will still be executed even if `f(x)` evaluates to true.

Integer

`int`: Signed integer of 32 bit size.
Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Arithmetic operators: `+`, `-`, unary `-`, `/`, `\%` (modulo).

Note the clear lack of the `*` and `**` (exponentation) operators as well as any bitwise operators. While integer sizes are limited to 32 bits, the output of arithmetic operations can exceed this size. This will not result in an overflow, but instead the script will fail when using this value in another integer operation. Division and modulo operations will fail if the right hand side of the expression is zero.

To make integer values more meaningful, we added several numeric units that allow users to represent either value – `satoshis/sats`, `finney`, `bits`, `bitcoin` – or time – `seconds`, `minutes`, `hours`, `days`, `weeks` – using the integer type. These units automatically convert the integer to the lowest possible unit (e.g. `14 days == 2 weeks`. These numeric units are used to add semantic meaning to the values they represent. This makes it easier to create timelocks and other time-based contracts.

String

`string`: ASCII-encoded byte sequence.
Operators:

- `+` (concatenation)
- `==` (equality)
- `!=` (inequality)

Members:

- `length`: Number of characters that represent the string.
- `split(int)`: Splits the string at the specified character and returns a tuple with the two resulting strings.

Bytes

`bytes`, `bytes20`, `bytes32`: Byte sequence, optionally bound to 20 or 32 bytes.
Operators: Operators:

- `+` (concatenation)
- `==` (equality)
- `!=` (inequality)

Members:

- `length`: Number of bytes in the byte sequence.
- `split(int)`: Splits the byte sequence at the specified byte and returns a tuple with the two resulting byte sequences.

Pubkey

`pubkey`: Byte sequence representing a public key.

Operators:

- `==` (equality)
- `!=` (inequality)

Sig

`sig` : Byte sequence representing a transaction signature.
Operators:

- `==` (equality)
- `!=` (inequality)

Datasig

`datasig` : Byte sequence representing a data signature.
Operators:

- `==` (equality)
- `!=` (inequality)

Array & Tuple

These types are not assignable, and only have very specific uses within CashScript.

Arrays are only able to be passed into `checkMultisig` functions using the following syntax:

```
checkMultisig([sig1, sig2], [pk1, pk2, pk3]);
```

Tuples can only arise by using the `split` member function on a `string` or a `bytes` type. Their first or second element can be accessed through a familiar array indexing syntax:

```
string question = "What is Bitcoin Cash?";
string answer = question.split(15)[0].split(8)[1];
```

4.3.3 Type casting

Some types can be implicitly or explicitly casted to each other, as shown in table 4.1.

Table 4.1: Overview of typecasting.

Type	Implicitly castable to	Explicitly castable to
bool		int
int		bool, bytes, bytes20, bytes32
string		bytes
bytes		bytes20, bytes32, int, sig, pubkey
bytes20	bytes32, bytes	bytes32, bytes
bytes32	bytes	bytes20, bytes
pubkey	bytes	bytes
sig	bytes	bytes, datasig
datasig	bytes	bytes

4.3.4 Built-in functions

- **require** takes a Boolean value and throws an error if it is false.
- **abs** takes an integer value and returns its absolute value.
- **min** takes two integer values and returns the smallest of the two.
- **max** takes two integer values and returns the largest of the two.
- **within** takes an integer and checks if it is within a passed range.
- **ripemd160** takes a value and returns its ripemd160 hash.

- **sha1** takes a value and returns its sha1 hash.
- **sha256** takes a value and returns its sha256 hash.
- **hash160** takes a value and returns the ripemd160 hash of its sha256 hash.
- **hash256** takes a value and returns its double sha256 hash.
- **checkSig** takes a transaction signature and a public key, and performs a signature check on them.
- **checkMultiSig** takes an array of transaction signatures and an array of public keys, and performs a multi-sig check on them.
- **checkDataSig** takes a data signature, a data value and a public key, and performs a data signature check on them.

4.3.5 Global variables

There are two global variables that give information about the transaction in the context of the contract. **tx.time** represents the block number in which the transaction is included, while **tx.age** represents age of the transaction output in blocks. These variables can be used in the creation of Hash Time Locked Contracts (HTLCs) or similar contracts. Due to limitations within the underlying Bitcoin Script, these global variables can only be used in the following ways:

```
require(tx.time >= <expression>)
require(tx.age >= <expression>)
```

4.3.6 Operators

Table 4.2: Overview of operators with their precedence.

Precedence	Description	Operator
1	Parentheses	(<expression>)
2	Type cast	<type>(<expression>)
3	Function call	<function>(<args...>)
4	Tuple index	<tuple>[<index>]
5	Member access	<object>.<member>
6	Postfix increment / decrement	++ , −−
7	Unary minus	−
7	logical NOT	!
8	Division and modulo	/ , \%
9	Addition and subtraction	+ , −
9	String / bytes concatenation	+
10	Numeric comparison	< , > , <= , >=
11	Equality and inequality	== , !=
12	Logical AND	&&
13	Logical OR	||
14	Assignment	=

4.4 Artifacts

Ethereum has the concept of the Application Blockchain Interface (ABI) [21]. These ABIs contain all the information needed to interface with the smart contract as it is deployed on the blockchain [21], hence the name Application Blockchain Interface. These ABIs are used by libraries that integrate with Ethereum, such as Truffle [30]. Truffle integrates this ABI into their own format called the *Truffle artifact*, which contains a lot of other information, such as the raw bytecode, source code, and deployed addresses [30].

In the case of Bitcoin, there is more information needed to interface with a contract than in the case of Ethereum. In Ethereum, the contract's bytecode is stored on the blockchain [31], while with Bitcoin's

P2SH the bytecode is stored client-side, and needs to be provided when using the contract [5]. This effectively extends the ABI to need additional fields. With these additional fields added, a Bitcoin ABI is more similar to a Truffle artifact than an Ethereum ABI. To avoid confusion, we decide to adhere to most of the Truffle artifact specification and use the name artifact rather than ABI to refer to this specification.

These artifacts can be used by libraries and SDKs to integrate CashScript contracts, as it includes all information needed to instantiate and use contracts. The JavaScript SDK is currently the only SDK that uses these artifacts, but other SDKs could be developed by third parties without the need to re-implement a compiler, as they only need to integrate the artifact format into their SDK. The actual compilation can then be done using the command line CashScript compiler.

4.4.1 Artifact specification

```
interface Artifact {
  contractName: string; // Contract name
  constructorInputs: AbiInput[]; // Arguments required to instantiate a contract
  abi: AbiFunction[]; // functions that can be called
  bytecode: string; // Compiled Script without constructor parameters added (in ASM
      format)
  source: string; // Source code of the CashScript contract
  networks: { // Dictionary per network (testnet / mainnet)
    [network: string]: { // Dictionary of contract addresses with the corresponding
        compiled script (in ASM format)
      [address: string]: string;
    };
  };
  compiler: {
    name: string; // Compiler used to compile this contract
    version: string; // Compiler version used to compile this contract
  }
  updatedAt: string; // Last datetime this artifact was updated (in ISO format)
}

interface AbiFunction {
  name: string; // Function name
  inputs: AbiInput[]; // Function inputs / parameters
}

interface AbiInput {
  name: string; // Input name
  type: string; // Input type
}
```

4.5 Extensions

To keep it within the scope of this four-month project, the language described in this chapter has intentionally omitted some functionality that could be valuable to the language. This additional functionality could be added in a future version of the language. We briefly describe the functionality that we envision to be added in the future.

The most important extension that we see fit is the addition of structs. These structs could be defined similar to JavaScript objects or Solidity structs. The compiler would translate these structs to a custom encoded byte sequence. Since it is possible to split and concatenate byte sequences in Bitcoin Script, it should be possible to retrieve the parts of this byte sequence that correspond to the struct's properties. The JavaScript SDK could provide a deterministic mapping between JavaScript objects and their Bitcoin Script representation, so a CashScript struct would be translated to the exact same byte sequence as a JavaScript object.

Another extension is enabling simple array operations. As long as the length of these arrays is bounded it should be possible to do simple array operations like iterations, getting and setting, and appending and removing. However, while this could technically be achieved, there is still the Script size limitation that needs to be accounted for. This is why these possibilities would be further researched before deciding on an implementation.

A third possible extension is the support for bitwise logic operators and more bounded 'bytes' types. BCH has support for the OP_AND, OP_OR and OP_XOR opcodes, which are disabled in BTC. These

opcodes do a bit by bit comparison of the two input variable and output the result of these bitwise operators. However, when passing inputs of unequal size, these operators cause the full Script to fail. With regard for safety, we decided to leave these operators out of the language, as they could be the cause of errors when used wrongly.

A way to enable these operators could be in combination with extended support for bounded 'bytes' types. Currently, we support the 'bytes20' and 'bytes32' types, but this can be extended to more specific bounded types, so the compiler can check for size at compile time to make sure the two inputs are absolutely of the same size.

A fourth possible extension is better datetime support. In the current design, it is possible to use an integer type to represent a Unix timestamp or a block number to do time-based operations. However, this is not a very user-friendly way to handle time operations. In the future we could add a *time* type that allows for more complex time operations. Spedn already offers support for specifying dates using a standard-formatted date string.

A fifth extension is internal functions inside contracts. This can be helpful to break up functionality into several functions, while keeping the contract's external interface the same. This somewhat improves the readability of the contract, but it is mainly a quality of life improvement, since the contracts can not get to the size where this separation of concerns would become absolutely necessary.

The final proposed extension is contract inheritance, which would allow further separation of concerns. This functionality is especially useful when combined with internal functions, as this can allow the creation of standard libraries and other shared functionality. Since we can remove any unused code from the compiler output, inheritance and internal functions shouldn't necessarily lead to bloated contracts.

Chapter 5

A compiler for CashScript

5.1 Compiler implementation

In this section we describe the implementation of the compiler that translates the contracts written in our language into Bitcoin Script. Section 2.2 describes the general structure of a compiler. In this section we describe the implementation of each of the mentioned compiler stages.

5.1.1 Used tools

One of the goals of our language is to have first class support for integration with JavaScript projects, which means that we want to be able to compile contracts from within JavaScript code. There are many tools that can be used to assist with compiler construction, such as Bison or ANTLR, but not all of these tools are compatible with JavaScript. Luckily, there are many tools available to build compilers with JavaScript, such as ANTLR, PEG.js, Jison and Chevrotrain [32].

Most tools integrate the lexical analysis and the syntax analysis compiler stages into a single tool. However, many of these tools stop at the syntax analysis phase, leaving the developer to create their own solutions for further traversals of the AST. One clear exception to this rule is ANTLR, which provides a framework for custom AST traversal to assist with semantic analysis, optimisation, and code generation. ANTLR also offers the possibility to generate parsers and lexers in multiple languages, using the same grammar definitions. This makes it easier to extend support to other popular programming languages, such as Java or Python [32].

There is a performance benchmark available for these tools [1], which measures the operations per second that the tools are capable of when parsing a JavaScript Object Notation (JSON) file. This shows that Chevrotrain is the most performant tool, while ANTLR comes in second, and other tools such as PEG.js and Jison lag behind. The benchmark, however, is created by the authors of Chevrotrain, which could indicate that the benchmark doesn't hold up in other use-cases. We do not focus on bringing down compilation time in this project, so this benchmark does not influence our choice of tools.

We do look at the popularity and maintenance of these tools. If we look at an NPM comparison between the available tools [2], we see that all of the tools have been around for years, and are updated regularly – Chevrotrain, PEG.js and ANTLR have been updated within the last week at the time of writing. In terms of downloads, PEG.js and Jison are far above the other tools, but all of them get over tens of thousands of downloads per week.

Keeping the above in mind it is difficult to name one of these tools the best for the job, so in the end it boils down to personal preference. The Ivy language is written using PEG.js, which would make it easy to extract parts of their code and fit it into our own implementation. Most of the Ivy code is independent of PEG.js though, so using parts of their algorithms is possible even without using PEG.js. In the end, we decided to go with ANTLR, as this offers just that extra bit of functionality. It also allows us to potentially offer compiler support for other languages in the future.

[1] `https://sap.github.io/chevrotain/performance/`

[2] `https://www.npmtrends.com/chevrotain-vs-pegjs-vs-antlr4-vs-jison`

5.1.2 Lexical & syntax analysis

During the design of our language and implementation of our grammar, we used a *top-down* approach. What this means is that we started with the bigger constructs of the language like contracts and functions, and we worked down to lower level constructs like assignments and expressions, ending with literals and values. This is opposed to a bottom-up-approach, which starts with the smallest possible elements of the language and works up to the bigger constructs.

We leveraged the strength of ANTLR, which allows us to put the lexer and parser definitions in one single file. On top of this, it allows us to use the actual characters included in tokens, rather than having to define tokens for every single operator or keyword, as is the case with more traditional tools such as *flex*. ANTLR also offers a simplified syntax for recursive rules, so we don't have to explicitly deal with left-recursion [20]. An example of these languages features can be seen in the snippet below, showing a simple expression grammar for arithmetic expressions that respect order of precedence.

```
expression
    : '(' expression ')'
    | expression ('*' | '/') expression
    | expression ('+' | '-') expression
    | NUMBER
    ;
```

From the grammar specification using ANTLR's file format, ANTLR automatically generates a lexer and parser in the specified target language. ANTLR offers many target languages for this, and several unofficial languages targets have been implemented by the open source community. We want to integrate our compiler in JavaScript projects, so the obvious choice would be the default JavaScript target [3]. However, the official target for JavaScript targets the ten-year-old ECMAScript 5 (ES5), which lacks support for many modern JavaScript constructs, such as classes, modules, and arrow functions.

To gain access to these modern features, as well as a more robust type system, we decided to target the unofficial TypeScript target [4]. This allows us to write the compiler itself using modern development practices with TypeScript. This also allows first-class support for TypeScript and modern JavaScript projects, as so-called declarations-files can be automatically generated. There are two drawbacks to using the TypeScript target. The first drawback is the fact that it is an unofficial target that does not have a 1.x release yet, which means there might be some unexpected behaviour. The second drawback is that the TypeScript target is not compatible with ES5, so it needs to be used in an environment that supports at least the ECMAScript 2015 (ES6) standard. Currently, all major browsers and server runtimes have support for ES6, which is why we decided it was worth the trade-off.

The parser and lexer generated by ANTLR and the ANTLR TypeScript target produce a parse tree, which we transformed into a more useful AST. This AST consists of different node types for constructs such as function calls, if-statements, binary operations and literals. The nodes are structured to hold all necessary data about the source file and to make it easily traversable. The AST was built using an implementation of ANTLR's generated *visitor* interfaces. This visitor allows us to traverse the parse tree, and return AST nodes in the place of the existing parse tree nodes. For the new AST we also implemented a generic visitor and traversal class using the visitor pattern [5]. These classes are subclassed in the later stages to define custom AST traversals.

5.1.3 Semantic analysis

The goal of lexical analysis and syntax analysis is to transform a piece of source into an AST and assert that it is syntactically correct. The goal of the subsequent semantic analysis is to use this AST to assert that the code makes semantic sense. Two important parts of this semantic analysis step that we implemented are symbol table checking and type checking.

Symbol table checking

We implemented symbol table checking by creating a tree-like structure of symbol tables, where every symbol table represents a lexical scope, and contains a reference to the symbol table of its parent's scope. This allows symbols to be looked up in the current scope as well as higher scopes. Each symbol

[3] https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md
[4] https://github.com/tunnelvisionlabs/antlr4ts
[5] https://en.wikipedia.org/wiki/Visitor_pattern

table is implemented with a Map data structure that maps string identifiers to their corresponding *symbol* objects. These symbols contains type of the identifier and a parameter signature in the case of a function.

To check the symbol tables, we implemented an AST traversal that adds a new symbol to the current scope's symbol table for every found parameter or variable declaration. Before adding the new symbol, it is asserted that it is not a redefinition. We decided to disallow variable shadowing, as this provides an extra vector for human error. Because the contracts are limited in size, we do not believe variable shadowing is something that will be missed.

It then looks up every identifier it finds in the current scope's symbol table to assert that it is defined. We decided to disallow unused variables as this is most likely unintentional and another vector for human error. After the symbol table checks we know that all symbols are defined, and the symbol table can be used in the type checking step of semantic analysis.

Type checking

For type checking we added an optional 'type' field on all expression nodes, which we set in the type check traversal. The leaf nodes of any expression are either literals or identifiers. For literals the types are known, while the types of identifiers are extracted from the symbol table. From these leaves, all expressions such as binary operations and function calls, are visited bottom up. They are checked to have the correct argument types, and the resulting type is stored on the corresponding nodes. Finally, these expressions are used in statements such as assignments or if-statements, where they are checked again to have the correct type.

5.1.4 Code generation

Code generation is the final stage of the compiler. After all syntactical and semantic checks have passed, we are absolutely sure that the input code is valid, which makes it suitable to be translated into the target code.

Basic expressions and statements

Many of the expressions and statements in the CashScript language have a one to one mapping, such as the '>' operator that corresponds with the OP_GREATERTHAN opcode, or the 'require' function that corresponds with the OP_VERIFY opcode. Since the Bitcoin Script language is a left-to-right stack-based language, the way this code is generated is by adding the arguments to these functions or operators first, followed by their corresponding opcode. Not all of CashScript's functionality is compiled that easily though, as there is some more complex functionality than only mapping operators and functions to their corresponding opcodes.

Variable management

The first thing is that Script has no native support for variables, registers, or other forms of memory management. The memory management in Script consists of one single stack where values can be stored, as well as a so called alt-stack that can be used as a secondary stack. The primary stack has several stack operations, such as OP_PICK, which copies an item from anywhere in the stack to the top, while the alt-stack only has the OP_TOALTSTACK and OP_FROMALTSTACK operations.

We handle variables in CashScript by simulating a stack during the code generation. This stack keeps track of all variables and values on the stack, although it doesn't keep track of their actual values, only the position. Whenever a variable is needed inside an expression, it is then retrieved from the stack using the OP_PICK opcode. This means that we can always retrieve the correct variable as long as we keep track of their stack position. Whenever we reassign a variable, we store the new value on the top of the stack, and we leave the old value on the stack as well. The first found reference is used at all times, so it doesn't matter if there are additional occurrences of the variable deeper in the stack.

Conditional blocks

Storing variables this way immediately causes another difficulty though, in the form of conditional blocks. It is impossible to know at compile time which execution path will be taken, but if local variables are assigned or other variables are reassigned, the stack can grow in one execution path while staying the same in the second. In order to deterministically retrieve variables from the stack, we need to make sure that the stack positions are equal after all branches of an if-statement.

For local variables this is easily solved, as they are not needed after the conditional block, and can be dropped from the stack. However, when existing variables are reassigned inside a scope, we do want this change to persist outside of the scope. This is why we decide to change our reassignment method inside conditional blocks. Instead of adding the new value to the top of the stack, we do a drop-in replacement of the old value at its original position. There is no straightforward way to do this in Script, so this ends up being quite costly in terms of operation count.

The way this replacement algorithm works is by removing the old value of the variable from the stack with an OP_ROLL OP_DROP combination. This is followed by a number of OP_SWAP OP_TOALTSTACK, which puts the values above the variable on the alt-stack and moves the new value down the stack. Finally when the new value has reached the correct position, the other values are retrieved from the alt-stack again with OP_FROMALTSTACK. The cost of this algorithm is specified in equation 5.1.

$$cost = 1 + 1 + max(d - 1, 0) + max(d - 2, 0) + max(d - 2, 0) = max(3 * d - 3, 2) \tag{5.1}$$

This algorithm can be optimised for certain hardcoded stack depths by using more specific stack operations. Nonetheless it will stay a costly operation until a reverse OP_ROLL is added to the Script language. Because this is so costly, we decide to only implement it inside conditional blocks – where it is necessary – rather than for all reassignments.

Multi-function contracts

The final challenge in code generation is the fact that Script has no built in support for multiple functions within a single Script. This is why we had to encode these functions using if-statements and so-called *function selectors*. Every function gets assigned a number, and to select the function later on, its number needs to be the first value in the unlocking script.

This is implemented by encoding these functions if-else-statements that check whether the selector parameter is equal to any of the functions' selectors. The function bodies' contents are then added to the corresponding blocks inside these if-else statements. When there is only one function inside a contract, this process is skipped, and the Script expects no selector parameter, as it always executes the one function.

5.1.5 Possible optimisations

Because of time limitations, the CashScript compiler does not contain any optimisations that reduce the size of the output script. For this project the priority was on creating superior integration capabilities through the SDK described in section 5.3. However, there are several optimisations that could be implemented in order to reduce the output size. In this subsection we describe several of these optimisations that could be included in the compiler at a later time.

Merging OP_VERIFY with preceding operation

There are many operations that return a Boolean value, such as OP_EQUAL or OP_CHECKSIG. Almost all of the operations that return Booleans have a variant that immediately executes an OP_VERIFY afterwards, such as OP_EQUALVERIFY or OP_CHECKSIGVERIFY. The occurrences of OP_VERIFY in the output script could be combined with the preceding operation (e.g. `OP_EQUAL OP_VERIFY` becomes `OP_EQUALVERIFY`). This would save a byte per merge, as two opcodes are merged into one.

Using OP_ROLL instead of OP_PICK

There are two different opcodes that retrieve a value from an arbitrary stack depth. OP_PICK copies a value to the top of the stack, while OP_ROLL moves it to the top of the stack. CashScript currently uses OP_PICK at all times, which results in the need to clean up left-over values at the end of the script using OP_DROP. OP_ROLL could be used for the final occurrence of a variable though, as it is not required after. This results in less cleanup, and in a reduced average stack size at any point in execution. This in turn makes the average replace operation cheaper, which depends on stack depth of variables.

Using more specific stack operations

Currently OP_PICK is used every time a value needs to be retrieved from the stack. There are more specific stack operations that are hardcoded for specific stack depths, such as OP_OVER that copies the second stack item to the top. By hardcoding these specific stack operations, some bytes can be saved since the *depth* parameter of OP_PICK does not need to be included.

Removing final OP_VERIFY

A script is considered to be valid if the script stack contains just one truthy value after execution. In other words, there is an implicit execution of OP_VERIFY at the end of each contract. Currently CashScript adds a truthy value as the final operation of the script. However, in some cases the final occurrence of OP_VERIFY can be removed instead, as a final OP_VERIFY is implicitly executed at the end of the script any way. This saves a few bytes since the OP_VERIFY and the final data push can be removed from the script.

Improved stack ordering

The compiler currently keeps track of the stack positions of every variable and retrieves it using OP_PICK whenever it is needed. It is known at compile time though when a variable is needed, so theoretically, the stack could be ordered in such a way that all variables are at the correct place when they are needed. This is usually done when manually writing Bitcoin Script. It is not clear how much can be gained from improved stack ordering, so this would need to be researched further.

Simplification

Certain operations could be simplified at compile time (e.g. `sha256('CashScript')` could be replaced with the hash result). This would result in fewer operations and generally also a smaller script size, but it is possible that it results in a bigger script size for some cases, such as the hashing example. These considerations could still be made at compile time, but most well-implemented contracts are not likely to contain expressions that would warrant simplification. There is an added complication that we need to be sure that operations such as string concatenation and type casting happen in the exact same way as they are done by a Bitcoin node, lest we change the semantic meaning of the contract.

5.1.6 Test suite

To ensure that our compiler works as expected we have implemented a test suite with nearly 100% test coverage. We structured this test suite by compiler stage. For each of the stages we created source files targeted at different test classes. The different classes have source files that include something from their test class, such as redefinition errors. The tests iterate all source files for a class and assert that they behave in the expected way. For each of the compiler stages one of the test classes is success, which includes source files with correct uses of the language.

For syntax and lexical analysis, the only additional case is Syntax Error, as these errors are not further subdivided. This test case includes many different source files with different syntactical errors, such as incorrectly terminated comments or functions, or assignments instead of comparison inside an if-statement. For symbol table analysis, there are three different classes of errors: redefinition, undefined reference, and unused variable. Type checking has many classes of errors, such as unequal type or unsupported type, and other type errors.

During code generation, the fixture data is more than only source files, as we need to test that the output matches the expectations. This is why we add the expected output to the fixture data for each of these files. The test suite still loops over these source files, but asserts that the compiler output matches the expected output that is stored in a separate file.

5.2 Command line tool

For standalone compilation of CashScript files, we created the `cashc` command line tool and NPM package. This command line tool can be used to compile a CashScript file to a JSON file containing the corresponding artifact, as described in chapter 4. This artifact can be imported into the JavaScript SDK or other SDKs, might they be implemented in the future.

The command line tool executes all compiler stages. It starts at lexing and parsing with ANTLR, then builds the AST and goes through the AST traversals as specified in section 5.1. From the final AST and the generated target code, and artifact is generated, which is exported to the specified file. The functionality in the command line tool is also exported in the NPM package, so compilation can be done from the JavaScript SDK as well.

5.3 JavaScript SDK implementation

The JavaScript SDK allows anyone to easily interact with CashScript contracts with a high level of abstraction. The SDK defines Contract objects, which are JavaScript representations of a CashScript contract / artifact. These Contract objects can be used to create new Instance objects, which represent instantiated versions of these contracts, meaning they have had their constructor parameters added. Contract functions can be called from this instance like a regular JavaScript function and return a Transaction instance. These Transaction objects can be used to send a specified amount of money to an address or multiple addresses.

5.3.1 Contract

Contract objects can be created through two static functions on the `Contract` class. The first is `Contract.fromCashFile`, which uses the `cashc` package to compile a specified CashScript file and generate an artifact for it. The second function is `Contract.fromArtifact`, which imports an existing artifact JSON file. For both of these files, the resulting artifact is stored in a new Contract object, and three functions are generated on this object: `contract.export`, `contract.new`, `contract.deployed`. `export` is used simply to export the contract's artifact to a file. The other two are more complex.

`contract.new` uses the constructor inputs specified in the artifact as parameters, which get encoded and added to the front of the compiled bytecode stored in the artifact. This represents the full redeem script, and gets passed into the Instance constructor with the full artifact and network string to create a new contract instance. It also updates the artifact to store the full redeem script under the deployed addresses.

`contract.deployed` takes an optional address, and searches the deployed addresses inside the artifact. If no address is passed, it takes the first deployed address listed for the current network. It then passes the the full redeem script, full artifact, and network string into the Instance constructor in the same way as `contract.new`.

5.3.2 Instance

The section above explains how to use the Contract class to create new contract instances and retrieve earlier created instances. This instance object has an `address` field with its P2SH address, and a `getBalance` function that retrieves the amount of money locked inside the contract instance. Besides these functions, an Instance object also has all the functions that were defined on the contract in the CashScript file.

The contract functions are extracted from the artifact by looking at the `abi` field, which has a specification of all functions, their parameters, and the corresponding function selector, as described in chapter 4. Together with the passed redeem script this is everything that is needed to call these functions on the contract. The Instance object has a member field `functions` under which all generated functions can be found. These functions return a Transaction object that can be used to send money to an address or multiple addresses, as can be seen in the example below.

```
const tx = await instance.functions.spend(pk, new Sig(keypair, 0x01))
  .send(instance.address, 10000);
```

A note on transaction signatures

Some cash contract functions require a signature parameter, which needs to be a valid transaction signature for the current transaction. The current transaction details are unknown at the time of calling a contract function, so it can not be generated yet. This is why we have a separate `Sig` class made up of a keypair and hashtype, that represents a placeholder for these signatures. These placeholders are automatically replaced by the correct signature during the transaction building phase.

5.3.3 Transaction

Calling any of the functions on a contract instance results in a Transaction object, which can be sent by specifying a recipient and amount - or a list of these pairs - using the `send` method. `send` function calls can also be replaced by `meep` function calls with the same signature. The `meep` function doesn't send the transaction, but outputs the debug command to debug the transaction using the meep debugger [6].

The `transaction.send` method is where the core functionality of the SDK lies. The transaction's locktime is automatically set by looking at the current block count, so the user does not have to deal with manually setting this value to us the `tx.time` variable in their contracts. Next, a UTXO selection algorithm is used that keeps track of the running cost of a transaction while adding new UTXOs until the input amount is higher than the target amount.

The selected UTXOs and the transaction outputs are added to a transaction builder, which is used to build an incomplete transaction from these inputs and outputs. It is incomplete because the full input scripts have not been added yet. The corresponding input scripts are then generated from the redeem script and the function arguments, while replacing the placeholder `Sig` objects by the correct signatures. These signatures are created by signing the incomplete transaction using the `Sig` object's keypair and hashtype.

After generating these input scripts they are added to the incomplete transaction, completing the transaction object. This transaction object is then converted to a hexadecimal format and sent to the Bitcoin.com mining nodes through the BITBOX library that communicates with the `rest.bitcoin.com` Application Programmer Interface (API). The BITBOX library returns an object with the transaction details that can be used to look up the transaction on a block explorer or programmatically.

[6] `https://github.com/gcash/meep`

Chapter 6

Evaluation

6.1 Considered languages

In chapter 3 we discussed the existing languages that compile to Bitcoin Script. Of these languages Ivy has the most sophisticated tooling, while Spedn has full support for BCH. BitAuth Script is the relative newcomer and opts to stay close to the original syntax of Script. BALZaC is an academic project that allows anyone to formally describe Bitcoin transactions, mainly for the sake of formal reasoning about its behaviour.

Since BitAuth Script is intertwined with the rest of the BitAuth project, it is difficult to evaluate it on its own. Besides this, it is more like a templating engine on top of Bitcoin Script rather than a full-fledged language. BALZaC is only usable within their own online IDE, and does not allow the creation of standalone contracts. There is no current use case for it outside of formal reasoning about Bitcoin transactions. This is why we do not consider BitAuth Script and BALZaC in the evaluation.

Ivy has a good deal of standalone tooling, but it has been abandoned for a long time, so there is little practical value in considering it. We still think it would be interesting to consider the language, as its integration capabilities are more extensive than the rest of the related work, specifically for JavaScript applications. However, we use participants in our evaluation and we need to be mindful of their time, so we do not consider Ivy in the evaluation.

This leaves Spedn, the most extensive language among the related work when looking at features, although it lacks some of Ivy's integration capabilities. We compare Spedn against the language we describe in chapter 4, CashScript. We hypothesise that Spedn and CashScript offer similar functionality, but that CashScript has superior integration capabilities, specifically for JavaScript applications.

The underlying assumption in the motivation for both Spedn and CashScript is that Bitcoin Script is not accessible and is difficult to write and use. To challenge this assumption, we consider Bitcoin Script in our evaluation as well.

6.2 Metrics

At the start of this project, we conducted a literature study into studies that compare multiple programming languages with each other. In this literature study we looked at common metrics that are often used in programming language comparisons, and we reasoned about their applicability to Bitcoin Script languages. We excluded the metrics that were not applicable or not valuable, while the applicable metrics are listed below. Of the applicable metrics only development time and correctness relate to CashScript's goal of making Bitcoin Cash development more simple and accessible, making them the most important. The other applicable metrics are measured for reference purposes, rather than a means of evaluation.

The full literature study on programming language comparisons is included in appendix D.

- Development time
- Implementation correctness
- Executable size
- Code size
- Compilation time

To answer the research questions we also evaluate the integration capabilities of CashScript through

integration time and **integration correctness** metrics. These metrics are considered more important than the others, as they align most with this project's research questions.

6.3 Setup & methods

6.3.1 Contract implementation

The goal of both Spedn and CashScript is to make it easier and more accessible to write Bitcoin Script. On top of this, CashScript's goal is also to make it easier and more accessible to integrate these scripts into JavaScript applications. The main two metrics for this project are therefore the ease-of-writing, and the ease-of-integration into JavaScript applications. Ease-of-use metrics like this are always difficult to evaluate as they tend to be subjective.

To measure the ease-of-writing, we ask participants to implement a contract with a given specification. This contract is a *toy program*, a small program with semi-complex functionality, but that is simple enough to implement consistently across multiple languages [33]. When more complex applications are used for these comparisons, it is difficult to assert that the implementations are actually equivalent across different languages [33], which is why a simple toy program is used. We measure the time it takes the participants to complete the implementations – this corresponds with the development time metric. After the implementation, we ask the participants to rank their preferred languages during the contract implementation.

We collect the participants' implementations and compile them on the same machine for consistency. From this we measure the executable size, code size, and compilation time metrics. To measure the correctness of the implementations, we compare it to our reference implementations – included in appendix C – and manually assert that they are semantically equivalent. If they are not, we count the number of errors that are made in the implementation. Consistently repeated errors are not counted more than once (such as using the underscore character in a variable name in Spedn).

6.3.2 Contract integration

We test CashScript's integration capabilities through the integration time metric. For this we provide the same participants with our CashScript reference implementation. We then ask them to integrate this into a boilerplate CLI application that allows the contract to be used by its participants. We measure the time it takes the participants to complete this integration. We also test the correctness of this integration through manual testing.

For the evaluation of CashScript's integration capabilities we do not compare CashScript against Spedn and Script. Preferably we would compare this as well, but the steps required to integrate Spedn or Script are currently much more extensive than for CashScript. While it would definitely be doable for experienced Spedn or Script developers, a big part of our participants is inexperienced, and we do not expect them to be able to make this integration in a reasonable amount of time.

Since Spedn compiles to Bitcoin Script, the steps for integrating these two technologies into JavaScript are similar. Making this integration would consist of repeating a lot of code that is similar to the implementation of the CashScript JavaScript SDK. This includes UTXO selection, building incomplete transactions, and generating transaction signatures and input scripts for these incomplete transactions.

While the required implementation does not need the same level of generality that is included in the CashScript SDK, someone unfamiliar with Bitcoin's transaction model and scripting language will have a hard time even knowing where to begin. Some of the evaluation participants do have this knowledge, but quite a few don't and the main audience that stands to benefit from CashScript are those without this domain knowledge. To be mindful of these participants' time we decide to forego integration with Spedn and Script.

There are people who have integrated either Spedn or Script into their applications, so to illustrate the work that goes into it these can be inspected. Notably, Karol Trzeszczkowski has created a *Last Will* contract with Spedn, and integrated the compiled Script into a plugin for the popular Electron Cash wallet, written in Python [1]. Tobias Ruck has created a decentralised exchange for SLP tokens in Rust. This exchange uses a contract that he wrote by hand with Script [2] and integrated into his Rust codebase.

[1] `https://github.com/KarolTrzeszczkowski/Electron-Cash-Last-Will-Plugin`
[2] `https://github.com/EyeOfPython/slpagora`

6.3.3 Similarity to Ethereum's workflow

A secondary objective of CashScript is creating a language and SDK with a similar workflow. To measure this goal, we make sure that part of the test group has experience with Ethereum, while another part does not have this experience. This allows us to look for differences in outcome between these two groups. To do so, we ask the participants to rate their Ethereum knowledge, as well as their knowledge of other technologies to discover possible relations.

After doing the whole evaluation, we ask all the participants that are familiar with Ethereum how similar they think the CashScript language is to Solidity and how similar they think integration into a JavaScript application is for CashScript and Solidity. We ask them to rate this on a five-point likert scale ranging from very different to very similar.

6.4 Participants

We collected participants from several groups. For Ethereum developers we approached people we met at several Ethereum-related conferences or hackathons. We also approached more experienced BCH developers who already understand and actively use Script, and BCH developers that don't actively use Script. We approached these participants through a Telegram channel dedicated to BCH compilers and Script development. These are high-profile developers in the BCH community that are well-known for their skills in Script development or general BCH development. We finally also approached a few people that have both experience with Ethereum and BCH or Bitcoin Script.

In total we have six participants for this evaluation study. Two of these participants are skilled with Ethereum, but have never done anything with BCH before. Two participants have extensive experience with both Ethereum and BCH. Two participants are skilled with BCH, but have never done anything with Ethereum before. Furthermore, all participants are skilled in JavaScript, to ensure that JavaScript skills are not a bottleneck in the integration assignment.

With this distribution we aim to have a sufficiently varied group of participants, while also including enough Ethereum developers so the Ethereum-related questions can be answered. The participants receive a compensation for the time they spent on this evaluation, as the evaluation assignment is quite long.

Chapter 7

Results

7.1 Overview of participants

We asked all participants to rate their experience levels of several relevant technologies, using the included scale. An overview of the participants and their experience levels is presented in table 7.1.

```
1 = Never looked into it or never heard of it
2 = Looked into it, but never tried it
3 = Played around with it (e.g. followed tutorial, ran examples)
4 = Have used it once in a non-trivial professional or personal project
5 = Have used it regularly in non-trivial professional or personal projects for less
    than a year
6 = Have used it regularly in non-trivial professional or personal projects for over a
    year
```

Table 7.1: Overview of evaluation participants and their experience levels.

	Ethereum	Bitcoin Cash	JavaScript	Bitcoin Script	CashScript	Spedn
P1	6	2	4	1	1	1
P2	2	6	6	3	1	2
P3	2	5	6	5	1	2
P4	5	5	6	1	1	1
P5	6	2	6	3	1	1
P6	6	5	6	3	3	3

7.2 Contract implementation

We asked all participants to implement a contract with Bitcoin Script, CashScript and Spedn. We measured the time it took them, the number of errors, the executable size, source code size, and compilation time. We then asked them which of the languages was their preferred way of writing the contract. The results of this are presented in table 7.2, table 7.3 and table 7.4.

Table 7.2: Results of the contract implementation in Bitcoin Script.

	Development time (min)	Error count	Executable size (opcodes)	Rank of preference
P1	35	4	12	3
P2	60	3	17	1
P3	70	0	13	1
P4	68	4	23	3
P5	54	2	16	3
P6	38	3	14	3
Reference			13	
Average (SD)	54.17 (14.86)	2.67 (1.51)	15.43 (3.78)	2.33 (1.03)

Table 7.3: Results of the contract implementation in Spedn.

	Development time (min)	Error count	Executable size (opcodes)	Rank of preference	Code size (bytes)	Compilation time (seconds)
P1	33	2	70	1	345	
P2	31	1	84	3	448	0.58
P3	7	0	69	2	332	0.56
P4	16	2	67	1	331	0.56
P5	11	1	68	2	350	0.56
P6	7	2	68	1	362	0.56
Reference			69		349	0.56
Average (SD)	17.5 (11.73)	1.33 (0.82)	70.71 (5.94)	1.67 (0.82)	359.57 (40.45)	0.56 (0.01)

Table 7.4: Results of the contract implementation in CashScript.

	Development time (min)	Error count	Executable size (opcodes)	Rank of preference	Code size (bytes)	Compilation time (seconds)
P1	14	0	74	1	354	0.54
P2	16	0	74	2	308	0.53
P3	9	0	74	3	340	0.54
P4	6	1	74	1	347	0.53
P5	17	0	74	1	354	0.54
P6	10	0	74	2	365	0.53
Reference			74		348	0.54
Average (SD)	12 (4.34)	0.17 (0.41)	74 (0)	1.67 (0.82)	345.14 (18.11)	0.53 (0.00)

7.3 Contract integration

We asked all participants to integrate our reference contract into a boilerplate JavaScript application that allows the contract to be used from a CLI. We measured the time it took and the number of errors were made during the integration. The results of this are presented in table 7.5.

Table 7.5: Results of the contract integration.

	Integration time (min)	Error count
P1	97	0
P2	57	0
P3	50	0
P4	44	0
P5	85	0
P6	96	0
Average (SD)	71.5 (23.92)	0 (0)

7.4 Similarity to Ethereum's workflow

We asked the participants who had used Ethereum before whether they thought that the CashScript language was similar to the Solidity language, and we asked them whether they thought that the CashScript integration process was similar to the Solidity integration process. We asked them to use the included scale. The results of this are presented in table 7.6.

```
1 = Very different
2 = Somewhat different
3 = Not really different, not really similar
4 = Somewhat similar
5 = Very similar
```

Table 7.6: Results on similarity between the workflows of CashScript and Ethereum.

	Language similarity	Integration similarity
P1	5	4
P2		
P3		
P4	4	4
P5	5	2
P6	4	4
Average (SD)	4.5 (0.58)	3.5 (1)

Chapter 8

Discussion

We did research into existing solutions to write smart contracts on BTC and BCH. From this we concluded that BTC has no functional high-level languages [1] that can be used for this, while BCH has Spedn [3]. Both have Bitcoin Script, but as confirmed by our evaluation, this does not offer the same level of developer experience as a high-level language. In addition, BCH has more functionality in both the base layer as well as the high-level languages through re-enabled opcodes and the newly introduced OP_CHECKDATASIG [34]. This makes BCH a better candidate than BTC when developing smart contracts.

In this work we presented CashScript, a new high-level language that sets itself apart from the related work in high-level languages in several ways. Most importantly, the existing work lacked a good way to integrate their compiled code into applications. The compiled Bitcoin Script needs a lot of additional work to be integrated, as discussed in chapter 6. Additionally, the related work disregards the Ethereum community and makes no effort to engage this body of developers. CashScript is created to be more similar in workflow to Ethereum and to offer superior JavaScript integration capabilities, without sacrificing on functionality.

For the evaluation of CashScript we used six participants in total, which is quite a small sample size. Because of this the conclusions and implications discussed in this chapter carry less weight than with a larger number of participants.

8.1 Contract implementation

In our evaluation six participants implemented a simple smart contract using Bitcoin Script, Spedn and CashScript. For these implementations we measured the time needed to develop them, the number of errors in the implementation, the size of the executable, the size of the code, and the compilation time. Bitcoin Script does not need to be compiled, so the final two metrics don't apply to it.

As can be expected, the executable size for hand-crafted Bitcoin Script is the lowest by far, as this can be precisely optimised for the use-case. Spedn and CashScript compile to very inefficient executable code that requires around five times the number of opcodes to run a similar contract, where Spedn is slightly more efficient than CashScript. When looking at code size and compilation time, the difference between Spedn and CashScript is negligible.

When looking at the development time, we see a different story. Bitcoin Script development takes the most time, while Spedn is already developed significantly faster, and CashScript contracts have the shortest development time. However, we believe that this improvement in development time is partially due to an unfair evaluation setup, as we found out that there is a CashScript example available online that is very similar to the evaluation assignment [2]. This was an oversight during the evaluation design. While only a minority of the participants ended up using this example, we do believe this has caused CashScript to come out as somewhat more favourable than it is in reality. Besides, the variance in development time is much higher for Spedn, ultimately making it difficult to draw significant conclusions from these figures.

We observe a similar effect when looking at the error count in the implementations. Bitcoin Script experiences the most errors, while Spedn still experiences a few and CashScript experiences next to none. Again part of this could be due to the evaluation setup and the existing CashScript example. However,

[1] Considering that Ivy has been abandoned

[2] `https://github.com/Bitcoin-com/cashscript/blob/master/examples/htlc.cash`

in this case we did observe that the errors in the Spedn implementations were mainly due to violations of expectations – such as disallowing underscores in variable names and enforcing unnecessary typecasting. Part of the design goals of CashScript was to adhere to developer expectations, making these errors unlikely to occur.

We also observed errors in Spedn and Bitcoin Script that were due to a confusion between `checkLockTime` and `checkSequence`. These functions are not at all descriptive of their intended use, so it can be expected that mistakes are made with this. In contrast, with CashScript, we used `tx.time` and `tx.age`, which are more indicative of their intended use. Again this makes these kind of errors unlikely to occur in CashScript, which could be a legitimate reason for CashScript's lack of errors.

After using all three languages, we asked the participants to rank them by their preference. In general most participants preferred the high-level languages over Bitcoin Script, but there was no strong consensus in preference between CashScript and Spedn, as they have a similar syntax. We did observe more errors being made with Spedn. Since the participants are not aware of this difference in the amount of errors, it makes sense that these errors have not influenced their preference.

8.2 Contract integration

Our evaluation shows that JavaScript developers are able to integrate CashScript into their applications within one and a half hours. This is true regardless of their experience with Bitcoin Cash or Ethereum. We did not evaluate the integration capabilities against the existing work, but we illustrated the amount of work needed to make this integration through examples. When looking at the code for the CashScript integration as completed by our participants, it is evident that CashScript integration is more accessible than the related work.

We consulted with Tobias Ruck, who is the author of SLP Agora and SLPDEX. Tobias stated that integrating CashScript is an order of magnitude simpler than integrating Bitcoin Script. He also said that using it would be a no-brainer if the output was optimised and if it had full support for his use-case and languages of choice.

In the final weeks of writing this work, the creator of Spedn released a JavaScript library of their own [3] that allows developers to integrate their Spedn contracts into JavaScript applications. It is possible that this library offers the same integration capabilities as CashScript. Because the Spedn JavaScript library was released after a big part of our evaluation was already finished, we did not include it in this evaluation.

8.3 Similarity to Ethereum's workflow

We evaluated the similarity between CashScript's workflow and Ethereum's workflow. To do this we included four participants in the evaluation that are skilled with Ethereum and we asked them to rate the similarity between CashScript's and Solidity's syntax, and the similarity between integrating each of the languages into a JavaScript application.

CashScript as a language was rated as somewhat to very similar to the Solidity language as CashScript's syntax was directly derived from Solidity's. When asked about the similarity of the overall workflow of integrating CashScript, the reactions ranged from somewhat similar to somewhat different. This shows that there is still work to be done to bring these closer together. A big part of Ethereum development is currently done with the help of Truffle, so this contributes to the workflow that many Ethereum developers are used to. Tooling that offers Truffle's functionality for CashScript could be a contribution to bringing these workflows closer together.

We had initially thought that the participants would show a correlation between their skill levels and their performances in development time, error count, or integration time. By looking at the data no significant correlations can be found between the participants' skill levels and their development performance. So we cannot say that CashScript is better suited for Ethereum developers than the related work. We have not applied sophisticated data analysis to come to this conclusion though.

[3] `https://spedn.readthedocs.io/en/latest/bitbox.html`

8.4 Threats to validity

When looking at the results we see a significant improvement in development time and error count for CashScript as compared to the related work. Part of these results could be inflated by an oversight in the evaluation setup, as discussed earlier in this chapter. The example was only used by a minority of the participants though. Additionally, the errors that were made in the Spedn and Bitcoin Script implementations are very unlikely or even impossible to occur in CashScript. These points mitigate the possible threats to validity caused by this oversight in the evaluation design.

8.5 Conclusions

In this research we have presented a new high-level contract language for BCH that compiles to Bitcoin Script. This language was evaluated against the most important related work, namely Bitcoin Script and Spedn. From this evaluation we conclude that CashScript offers the same functionality as the related work, but is easier to integrate into JavaScript applications. It is developed much faster than Bitcoin Script, but not significantly faster than Spedn. It is less error-prone to write, but the executable size is slightly bigger than Spedn's and much bigger than Bitcoin Script's. CashScript's syntax is somewhat to very similar to Solidity's, but this does not make it more suited for Ethereum developers. The overall integration workflow also needs revision to offer more similarity to Ethereum's.

8.6 Recommendations for further research

Because the evaluation study has only been performed with six participants, the results only present an indication, rather than conclusive evidence. It can be interesting to repeat a similar research with a larger body of participants to verify these results. This can be a challenge though, as the current evaluation study took three to four hours per participant. It can be difficult to find enough participants willing to dedicate that amount of time.

In chapter 4 we present several extensions to the language such as user-defined structs and contract inheritance. Additional research could be done into the possibilities for including these kinds of extensions to the language. Additionally we present several possible optimisations. Some of these optimisations are easily implemented, but others such as improved stack ordering and expression simplification could have deeper consequences. Therefore these kinds of optimisations could be a suitable subject for further research.

We had no time to compare the CashScript library to Spedn's newly released JavaScript library. This could be another good candidate for further research as the two libraries both offer improved integration, but function differently.

Acknowledgements

I want to thank Bitcoin.com for providing me with the opportunity to conduct my research and create CashScript in Tokyo. I want to thank my supervisors Adam Belloum and Gabriel Cardona for their input and thoughts on the thesis and implementation. I also want to thank the rest of the developer services team at Bitcoin.com for sharing their thoughts during the standups, and my co-workers in Tokyo for the great working environment. Finally I want to thank Kevser, for leaving her own work and life behind her and coming with me to the other side of the world on this four-month adventure.

References

[1] Electric Capital. (Mar. 2019). Dev report, [Online]. Available: https://medium.com/@ElectricCapital/dev-report-476df4ff1fd2 (visited on 04/04/2019).

[2] Chain. (2018). Ivy documentation, [Online]. Available: https://docs.ivy-lang.org/bitcoin/ (visited on 04/12/2019).

[3] T. Pein. (2018). Spedn documentation, [Online]. Available: https://spedn.readthedocs.io/en/latest/ (visited on 11/20/2018).

[4] Ethereum Foundation. (2019). Web3js documentation, [Online]. Available: https://web3js.readthedocs.io/en/1.0/ (visited on 06/25/2019).

[5] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*, 2nd. O'Reilly Media, Inc., Jun. 2017, ISBN: 978-1-491-95438-6.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", Mar. 2019, [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 04/05/2019).

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", Oct. 2008, [Online]. Available: https://bitcoin.com/bitcoin.pdf (visited on 04/10/2019).

[8] R. Kalis and A. Belloum, "Validating data integrity with blockchain", in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2018, pp. 272–277. DOI: 10.1109/CloudCom2018.2018.00060. [Online]. Available: https://ieeexplore.ieee.org/document/8591029 (visited on 04/05/2019).

[9] R. Kalis, "Using blockchain to validate audit trail data in private business applications", University of Amsterdam, Jun. 2018. [Online]. Available: https://esc.fnwi.uva.nl/thesis/centraal/files/f1051832702.pdf (visited on 04/05/2019).

[10] G. E. Moore *et al.*, *Cramming more components onto integrated circuits*, 1965.

[11] L. Shin, "Will this battle for the soul of bitcoin destroy it?", *Forbes*, Oct. 2017. [Online]. Available: https://www.forbes.com/sites/laurashin/2017/10/23/will-this-battle-for-the-soul-of-bitcoin-destroy-it/ (visited on 12/20/2018).

[12] D. De Rosa. (Apr. 2015). A developer-oriented series about bitcoin, [Online]. Available: https://davidederosa.com/basic-blockchain-programming/ (visited on 04/09/2019).

[13] A. Stone. (Nov. 2018). Why bitcoin (cash) script is nearly useless (and what to do about it), [Online]. Available: https://medium.com/@g.andrew.stone/why-bitcoin-cash-script-is-nearly-useless-and-what-to-do-about-it-b47adbfeceec (visited on 04/10/2019).

[14] G. Walker. (2015). Learn me a bitcoin, [Online]. Available: http://learnmeabitcoin.com/ (visited on 04/10/2019).

[15] J. Fyookball, J. Cramer, Unwriter, M. B. Lundeberg, C. Culianu, and R. X. Charles. (Aug. 2018). Slp token type 1 protocol specification, [Online]. Available: https://github.com/simpleledger/slp-specifications/blob/master/slp-token-type-1.md (visited on 04/10/2019).

[16] E. Oldenbrg. (Nov. 2018). Taking op_checkdatasig out for a test drive, [Online]. Available: https://www.yours.org/content/taking-op_checkdatasig-out-for-a-test-drive-68687aa8e3b9 (visited on 04/04/2019).

[17] M. Möser, I. Eyal, and E. G. Sirer, "Bitcoin covenants", in *International Conference on Financial Cryptography and Data Security*, Springer, 2016, pp. 126–141. [Online]. Available: https://maltemoeser.de/paper/covenants.pdf (visited on 08/08/2019).

[18] A. Zegers. (Dec. 2018). The story of op_checkdatasig, [Online]. Available: https://medium.com/@Mengerian/the-story-of-op-checkdatasig-c2b1b38e801a (visited on 07/19/2019).

[19] T. Pein. (Dec. 2018). Spending constraints with op_checkdatasig, [Online]. Available: https://honest.cash/v2/pein%5C_sama/spending-constraints-with-op%5C_checkdatasig-172 (visited on 07/19/2019).

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN: 0-201-10088-6.

[21] Ethereum. (2019). Solidity documentation, [Online]. Available: https://solidity.readthedocs.io/en/latest/ (visited on 04/19/2019).

[22] Chain. (Dec. 2017). Ivy for bitcoin: A smart contract language that compiles to bitcoin script, [Online]. Available: https://blog.chain.com/ivy-for-bitcoin-a-smart-contract-language-that-compiles-to-bitcoin-script-bec06377141a (visited on 11/20/2018).

[23] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. (2018). Balzac documentation, [Online]. Available: https://blockchain.unica.it/balzac/docs/ (visited on 05/31/2019).

[24] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of bitcoin transactions", Jan. 2018. [Online]. Available: https://eprint.iacr.org/2017/1124.pdf (visited on 06/03/2019).

[25] J. Dreyzehner. (Feb. 2019). Just released: Bitauth ide – write and debug custom bitcoin scripts, [Online]. Available: https://blog.bitjson.com/bitauth-ide-write-and-debug-custom-bitcoin-scripts-aad51f6e3f44 (visited on 04/12/2019).

[26] L. McIver and D. Conway, "Seven deadly sins of introductory programming language design", in *Proceedings 1996 International Conference Software Engineering: Education and Practice*, Jan. 1996, pp. 309–316. DOI: 10.1109/SEEP.1996.534015. [Online]. Available: https://ieeexplore.ieee.org/document/534015 (visited on 04/18/2019).

[27] W. Bright. (Jan. 2014). So you want to write your own language, [Online]. Available: http://www.drdobbs.com/architecture-and-design/so-you-want-to-write-your-own-language/240165488 (visited on 04/18/2019).

[28] ConsenSys. (May 2019). Solidity is twice as popular as the next blockchain coding language, [Online]. Available: Solidity%20is%20Twice%20as%20Popular%20as%20the%20Next%20Blockchain%20Coding%20Language (visited on 05/30/2019).

[29] T. Ho. (Mar. 2008). How safe is your programming language, [Online]. Available: http://tobyho.com/2008/03/30/how-safe-is-your-programming/ (visited on 04/19/2019).

[30] Truffle. (2018). Truffle documentation, [Online]. Available: https://www.trufflesuite.com/docs (visited on 06/19/2019).

[31] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform", 2013, [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 04/05/2019).

[32] F. Tomassetti. (Jul. 2017). Parsing in javascript: Tools and libraries, [Online]. Available: https://tomassetti.me/parsing-in-javascript/ (visited on 04/17/2019).

[33] Unknown. (2008). The computer language benchmarks game, [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (visited on 04/11/2019).

[34] bitcoincash.org. (2019). Bitcoin cash spec, [Online]. Available: https://github.com/bitcoincashorg/bitcoincash.org/tree/master/spec (visited on 12/20/2018).

[35] R. Henderson and B. Zorn, "A comparison of object-oriented programming in four modern languages", *Software: Practice and Experience*, vol. 24, no. 11, pp. 1077–1095, 1994. [Online]. Available: https://pdfs.semanticscholar.org/ddc2/942bb7b7c87b803355d1942924a28644128b.pdf (visited on 01/30/2019).

[36] L. Prechelt, "An empirical comparison of seven programming languages", *Computer*, no. 10, pp. 23–29, 2000. [Online]. Available: https://ieeexplore.ieee.org/document/876288 (visited on 01/30/2019).

[37] L. Mannila and M. de Raadt, "An objective comparison of languages for teaching introductory programming", in *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, ACM, 2006, pp. 32–37. [Online]. Available: https://eprints.usq.edu.au/1701/3/Mannila_DeRaadt_KOLI2006_PV.pdf (visited on 01/30/2019).

[38] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic, and P. S. Center, "An experiment in measuring the productivity of three parallel programming languages", in *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*, 2006, pp. 30–36.

[39] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics", *BMC bioinformatics*, vol. 9, no. 1, p. 82, 2008. [Online]. Available: `https://link.springer.com/article/10.1186/1471-2105-9-82` (visited on 01/30/2019).

[40] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects", in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, IEEE, 2013, pp. 303–312.

[41] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages", in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, IEEE, 2013, pp. 183–192.

[42] S. B. Aruoba and J. Fernández-Villaverde, "A comparison of programming languages in macroeconomics", *Journal of Economic Dynamics and Control*, vol. 58, pp. 265–273, 2015. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0165188915000883` (visited on 01/30/2019).

[43] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code", in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, IEEE, vol. 1, 2015, pp. 778–788. [Online]. Available: `https://ieeexplore.ieee.org/document/7194625` (visited on 01/31/2019).

Acronyms

ABI Application Blockchain Interface. 22, 23
ANTLR Another Tool For Language Recognition. 19, 25, 26, 30
API Application Programmer Interface. 31
AST Abstract Syntax Tree. 13, 25–27, 30
BCH Bitcoin Cash. 1, 6, 9–17, 19, 23, 32, 34, 39, 41, 52, 54
BTC Bitcoin. 1, 6, 9, 10, 14–17, 19, 23, 39, 52
CLI Command Line Interface. 15, 17, 33, 37, 51, 52, 54
CTOR Canonical Transaction Ordering. 9
DAG Directed Acyclic Graph. 4, 10
ES5 ECMAScript 5. 26
ES6 ECMAScript 2015. 26
HTLC Hash Time Locked Contract. 22
IDE Integrated Development Environment. 15–17, 32
JSON JavaScript Object Notation. 25, 29, 30
NPM Node Package Manager. 6, 15, 25, 29, 30, 51, 52, 54, 55
oclif Open CLI Framework. 54
P2MS Pay-to-Multi-Signature. 11
P2PK Pay-to-Public-Key. 11
P2PKH Pay-to-Public-Key-Hash. 11, 12
P2SH Pay-to-Script-Hash. 4, 11–13, 19, 23, 30
PoW Proof-of-Work. 8
SDK Software Development Kit. 6, 7, 14–17, 19, 23, 28–31, 33, 34, 52, 55
SLP Simple Ledger Protocol. 11, 12, 33
UTXO Unspent Transaction Output. 4, 10–13, 15, 31, 33

Appendix A

Code repository

The full implementation of the CashScript compiler and the CashScript SDK can be found in the code repository at `https://github.com/Bitcoin-com/cashscript`. This repository contains the full code for the CashScript compiler, CLI tool, and SDK, as well as usage examples and detailed READMEs. The full documentation for CashScript can be found at `https://developer.bitcoin.com/cashscript/docs/getting-started`. The code used in the evaluation can be found at `https://github.com/rkalis/thesis-evaluation`.

Appendix B

CashScript grammar

```
grammar CashScript;

sourceFile
    : contractDefinition EOF
    ;

contractDefinition
    : 'contract' Identifier parameterList '{' functionDefinition* '}'
    ;

functionDefinition
    : 'function' Identifier parameterList '{' statement* '}'
    ;

parameterList
    : '(' (parameter (',' parameter)*)? ')'
    ;

parameter
    : typeName Identifier
    ;

block
    : '{' statement* '}'
    | statement
    ;

statement
    : variableDefinition
    | assignStatement
    | timeOpStatement
    | requireStatement
    | ifStatement
    ;

variableDefinition
    : typeName Identifier '=' expression ';'
    ;

assignStatement
    : Identifier '=' expression ';'
    ;

timeOpStatement
    : 'require' '(' TxVar '>=' expression ')' ';'
    ;

requireStatement
    : 'require' '(' expression ')' ';'
    ;

ifStatement
    : 'if' '(' expression ')' ifBlock=block ('else' elseBlock=block)?
    ;
```

```
functionCall
    : Identifier expressionList // Only built-in functions are accepted
    ;

expressionList
    : '(' (expression (',' expression)*)? ')'
    ;

expression
    : '(' expression ')' # Parenthesised
    | typeName '(' expression ')' # Cast
    | functionCall # FunctionCallExpression
    | expression '[' index=NumberLiteral ']' # TupleIndexOp
    | expression '.length' # SizeOp
    | obj=expression '.split' '(' index=expression ')' # SplitOp
    // | left=expression op=('++' | '--')
    // | op=('!' | '~' | '+' | '-' | '++' | '--') right=expression
    | op=('!' | '-') expression # UnaryOp
    // | expression '**' expression --- No power
    // | expression ('*' | '/' | '%') expression --- OP_MUL is still disabled
    | left=expression op=('/' | '%') right=expression # BinaryOp
    | left=expression op=('+' | '-') right=expression # BinaryOp
    // | expression ('>>' | '<<') expression --- OP_LSHIFT 7 RSHIFT are disabled
    | left=expression op=('<' | '<=' | '>' | '>=') right=expression # BinaryOp
    | left=expression op=('==' | '!=') right=expression # BinaryOp
    // | left=expression op='&' right=expression --- Disabled bitwise logic for now
    // | left=expression op='^' right=expression
    // | left=expression op='|' right=expression
    | left=expression op='&&' right=expression # BinaryOp
    | left=expression op='||' right=expression # BinaryOp
    | '[' (expression (',' expression)*)? ']' # Array
    | Identifier # Identifier
    | literal # LiteralExpression
    ;

literal
    : BooleanLiteral
    | numberLiteral
    | StringLiteral
    | HexLiteral
    ;

numberLiteral
    : NumberLiteral NumberUnit?
    ;

typeName
    // : 'int' | 'bool' | 'string' | 'address' | 'pubkey' | 'sig' | Bytes
    : 'int' | 'bool' | 'string' | 'pubkey' | 'sig' | 'datasig' | Bytes
    ;

Bytes
    : 'bytes' ('20' | '32')?
    ;

BooleanLiteral
    : 'true' | 'false'
    ;

NumberUnit
    : 'satoshis' | 'sats' | 'finney' | 'bits' | 'bitcoin'
    | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks'
    ;

NumberLiteral
    : [-]?[0-9]+ ([eE] [0-9]+)?
    ;

StringLiteral
    : '"' ('\\"' | ~["\r\n])*? '"'
    | '\'' ('\\\'' | ~['\r\n])*? '\''
    ;
```

```
HexLiteral
    : '0' [xX] [0-9A-Fa-f]+
    ;

TxVar
    : 'tx.age'
    | 'tx.time'
    ;

Identifier
    : [a-zA-Z] [a-zA-Z0-9_]*
    ;

WHITESPACE
    : [ \t\r\n\u000C]+ -> skip
    ;

COMMENT
    : '/*' .*? '*/' -> channel(HIDDEN)
    ;

LINE_COMMENT
    : '//' ~[\r\n]* -> channel(HIDDEN)
    ;
```

Appendix C

Evaluation Assignment

C.1 Contract implementation

Alice, Bob, and Carol come to an agreement that they encode in a cash contract on Bitcoin Cash. They lock up an amount of BCH inside this contract and make the following agreements:

- Alice can always spend the money by providing her transaction signature.
- Bob can spend the money when a timeout block has been reached, by providing his transaction signature.
- Carol can spend the money by providing her transaction signature and the preimage of a hash stored in the contract.

C.1.1 Specification

Implement a cash contract that takes the following parameters:

- `alicePk` : alice's public key
- `bobPk` : bob's public key
- `carolPk` : carol's public key
- `dataHash` : the sha256 hash of a secret that carol knows
- `timeout` : the block number after which bob can spend

For the high-level languages Spedn and CashScript, these parameters should be defined as contract constructor parameters. Bitcoin Script lacks the concept of a constructor or constructor parameters, so they should be included as a placeholder at the correct spot in the Script (e.g. <alicePk>, <dataHash>).

The cash contract should have three functions / challenges:

1. Takes alice's signature as an argument, and executes a signature check with alice's signature and public key.
2. Takes bob's signature as an argument, and executes a signature check with bob's signature and public key. Also checks that the block number of the transaction is at least equal to the specified timeout.
3. Takes carol's signature and a raw data preimage as arguments, and executes a signature check with carol's signature and public key. Also checks that the sha256 hash of the passed preimage is equal to the stored dataHash.

For the high-level languages Spedn and CashScript, functions / challenges should be defined for each of these scenarios that take the correct parameters and execute the correct checks. Bitcoin Script lacks the concept of functions / challenges, so instead, execution paths using if-else statements should be used, as can be seen in chapter 7 of Mastering Bitcoin by Andreas Antonopoulos [1].

C.1.2 Documentation links

Documentation for Spedn can be found at `https://spedn.readthedocs.io/en/latest/` and its CLI tool can be installed with NPM at `https://www.npmjs.com/package/spedn-cli`. Its source code can

[1] `https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc#using-flow-control-in-scripts`

be found at `https://bitbucket.org/o-studio/spedn/src/master/`, where examples can be found in the examples folder.

Documentation for CashScript can be found at `https://developer.bitcoin.com/cashscript/docs/getting-started/`, its CLI tool can be installed with NPM at `https://www.npmjs.com/package/cashc`, and its JavaScript SDK can be installed with NPM at `https://www.npmjs.com/package/cashscript`. Its source code can be found at `https://github.com/Bitcoin-com/cashscript`, where examples can be found in the examples folder.

Bitcoin Script does not have a central developer documentation, nor does it have related tools, but `https://en.bitcoin.it/wiki/Script` provides a good overview of the language as well as its features. This website only includes BTC functionality, but for the purpose of this evaluation, no BCH-specific functionality is needed. Another good resource to read about Bitcoin Script is `https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch07.asciidoc`.

C.1.3 Setup

Before starting, please install Node.js as well as the Spedn and CashScript command line tools and verify that they're working correctly by trying to compile one of the examples.

```
npm install -g spedn-cli
npm install -g cashc

git clone git@bitbucket.org:o-studio/spedn.git
git clone git@github.com:Bitcoin-com/cashscript.git

spedn compile -c spedn/examples/ExpiringTip.spedn
cashc -o artifact.json cashscript/examples/transfer_with_timeout.cash
cat artifact.json
```

C.1.4 Reference Implementations

CashScript

```
contract Evaluation(
    pubkey alicePk,
    pubkey bobPk,
    pubkey carolPk,
    bytes32 dataHash,
    int timeout
) {
    function alice(sig aliceSig) {
        require(checkSig(aliceSig, alicePk));
    }

    function bob(sig bobSig) {
        require(checkSig(bobSig, bobPk));
        require(tx.time >= timeout);
    }

    function carol(sig carolSig, bytes preimage) {
        require(checkSig(carolSig, carolPk));
        require(sha256(preimage) == dataHash);
    }
}
```

Spedn

```
contract Evaluation(
    PubKey alicePk,
    PubKey bobPk,
    PubKey carolPk,
    Sha256 dataHash,
    int timeout
) {
    challenge alice(Sig aliceSig) {
        verify checkSig(aliceSig, alicePk);
    }

    challenge bob(Sig bobSig) {
        verify checkSig(bobSig, bobPk);
        verify checkLockTime(TimeStamp(timeout));
    }

    challenge carol(Sig carolSig, bin preimage) {
        verify checkSig(carolSig, carolPk);
        verify sha256(preimage) == dataHash;
    }
}
```

Bitcoin Script

```
OP_IF
    <alicePk> OP_CHECKSIG
OP_ELSE OP_IF
    <timeout> OP_CHECKLOCKTIMEVERIFY OP_DROP
    <bobPk> OP_CHECKSIG
OP_ELSE
    <carolPk> OP_CHECKSIGVERIFY
    OP_SHA256 <dataHash> OP_EQUAL
OP_ENDIF OP_ENDIF
```

C.2 Contract Integration

Alice, Bob, and Carol want to use their contract through a shared CLI. For this they created the `contract-cli` tool. In this tool they want to encode an instance of their contract with their public keys, a chosen timeout block, and a data hash. They each want their own CLI command so they can run the CLI for their own part of the contract.

They implemented a boilerplate CLI application with commands for each of them, as well as a function to display the contract's address and current balance and a function to send money to the contract from their own address. They implemented the function that allows them to send money to the contract, but now they are stuck on integrating the contract functionality into their CLI tool. They asked you to implement this functionality for them and sent you their code repository [2].

C.2.1 Participants & data

The three participants of the contract are Alice, Bob, and Carol. Their details are included below.

- Alice
  - Address: bchtest:qqftvzhuqj6z56phh3vsasq8atmchrnneu7ufc6ej5
  - Public key: 021ed9d1795a0c4a10fabb9f4678c70015b6d4ee7a39fa2d9e5eec3b6465d6eeb0
  - Private key WIF: cSU1aiAuMCc4xsSaRiuSscLQmc4uqs8bHv2riopVXiC8Wqgws7FS
- Bob
  - Address: bchtest:qqsd2c2s4hdhelzprchggm27s3lt2ngmughvnhhj4f
  - Public key: 028ba2eb3ccd6aabdd381844f6eda1bb8968bc7f433b8028ab5cd3e3a80476c521
  - Private key WIF: cVcmt7ZfnygxAuNuG15mTxfnK7cKeb3ziF6jTQKaApFc6EYqdWVs
- Carol
  - Address: bchtest:qrr49yxr30rzukvrf6gqettwr8m5qjxhnyvcrtrsn5
  - Public key: 028ee1e748dff218e8b4072251e61851ef1045f2e43509b91c8b986f27409f31e0
  - Private key WIF: cVLe9DA9SNBpbJFxPc64sWqNFXn8ANReNotXpjyMLpXj5zcoM1SZ

Bob's timeout block number is `1000000` and Carol's secret preimage is `43617368536372697074`. The necessary data is already included inside the CLI tool. All participants have an amount of BCH on their addresses, and the contract comes preloaded with funds as well.

C.2.2 Repository structure & setup

The CLI tool is created using Node.js and Open CLI Framework (oclif). All the required information can be found inside this document and the `src` directory inside the linked repository. In `src/global.js` the bitbox tool [3] is initialised and the required data is set up to initialise the contract, but the contract still needs to be imported and initialised. `src/contract.cash` contains the CashScript source code for the contract.

The CLI contains JavaScript files for the different commands of the CLI under `src/commands`: `alice`, `bob`, `carol`, `details`, `fund`. The `fund` command is already implemented, and can be used to send funds to the contract once it is initialised. The rest of the commands still need to be implemented, but have their command line arguments set up correctly.

To set up the local development environment you need to have Node.js and NPM installed locally. Then clone the repository, navigate into it, and install the dependencies.

```
git clone git@github.com:rkalis/thesis-evaluation.git
cd thesis-evaluation
npm install
```

The CLI can be used by running `./bin/run`. If everything is set up correctly, this will display the tool's usage information.

```
VERSION
  thesis-evaluation/1.0.0 darwin-x64 node-v11.15.0
```

[2] https://github.com/rkalis/thesis-evaluation
[3] https://developer.bitcoin.com/bitbox/docs/getting-started

```
USAGE
  $ contract-cli [COMMAND]

COMMANDS
  alice     Send money out of the contract by providing Alice's WIF
  bob       Send money out of the contract by providing Bob's WIF after the timeout block
  carol     Send money out of the contract by providing Carol's WIF and the correct data
      preimage
  details   Display the contract's address and balance
  fund      Send money to the contract
  help      display help for contract-cli
```

C.2.3 Specification

`src/global.js` includes the data to initialise the contract. This should be implemented inside the indicated function. When this is implemented correctly, the `contractInstance` is exported alongside the network string and bitbox instance, and imported in the different command files.

The first command is the `details` command. This should retrieve the contract instance's address as well as its balance, and display it. Next there are commands for each of the three participants that allow them to send money out of the contract. These contain flags for the recipient of the funds, the amount to send, and the private key WIF to use (included earlier in this document). Carol's command also contains a flag that allows her to pass in the correct preimage (included earlier in this document). These commands should be implemented by calling the correct contract functions with the correct parameters. It is not necessary to implement proper error handling.

During the development commands can be tested by running:

```
./bin/run [COMMAND] [FLAGS]
```

C.2.4 Documentation links

All required documentation links for CashScript are included in section C.1. You will additionally need to use the BITBOX SDK for general Bitcoin functionality. The BITBOX documentation can be found at `https://developer.bitcoin.com/bitbox/docs/getting-started` and it can be installed with NPM at `https://www.npmjs.com/package/bitbox-sdk`. Its source code can be found at `https://github.com/Bitcoin-com/bitbox-sdk`, where examples can be found in the examples folder.

Appendix D

Literature study on programming language comparison

We conducted a short literature study into programming language comparisons to get an overview of the different metrics generally used in programming language comparison studies. We compared existing studies that compare languages against each other, and we made an overview of the metrics that are used in these studies. An overview of these metrics is presented in tables D.1, D.2, and D.3.

D.1 Programming language comparison studies

D.1.1 Henderson & Zorn (1994) [35]

The research of Henderson & Zorn focuses mainly on object oriented programming languages and how object orientation is used within those languages. The research is quite old, so it compares C++, Modula-3, Sather, Oberon-2 and Self, most of which are not very widespread at this time. For their comparison they implemented a simple object-oriented that models classroom administration.

They compared these languages first on their language features regarding inheritance, dynamic dispatch, code reuse and information hiding. Next they did a performance comparison using the different implementations of the classroom administration application. They measured execution time, compile time and program size in bytes.

D.1.2 Prechelt (2000) [36]

Prechelt has published multiple papers on programming language comparisons that all use similar methods. They compare C, C++, Java, Perl, Python, Rexx, and Tcl. They compare the performance of these languages by executing the *Phonecode problem* [36]. For this he gathered many different implementations of this problem in the different languages. For this he approached people of different skill levels, so the implementations are of varying quality.

He executed these different implementations of the phonecode problem on different data sets, and compared them on the following metrics: correctness of the output, execution time, memory usage, program size in lines of code, time taken by the programmers, and programming productivity in terms of lines of code per hour. The author also state whether the languages are interpreted or compiled.

D.1.3 Mannila & De Raadt (2006) [37]

Mannila & De Raadt present a programming language comparison for introductory programming courses. They provide a large list of language properties that the languages are scored on, such as ease of learning, suitability for teaching and community support. They describe what is meant by all these properties, and present scores for all these languages. However, it is not motivated what these scorings are based on. The research also includes the results of a survey on the motivations of instructors for choosing their language for introductory courses.

D.1.4 Ebcioglu et al. (2006) [38]

Ebcioglu et al. conducted a study with multiple students that participated in a multiple day training event on parallel programming using MPI, UTC, and X10. The training concluded with an extensive programming assignment, where the students' programming productivity was measured. This was measured by keeping track of the amount of time needed to get to a correct solution to the problem. The authors include a full report on the individual students' solutions and processes, and include notes on the correctness of their implementations.

D.1.5 Fourment & Gillings (2008), [39]

Fourment & Gillings focus on programming languages for bio-informatics use cases. For this they compare C, C++, C#, Java, Perl, and Python for the execution of common applications in the bio-informatics domain (sellers algorithm, neighbour-joining algorithm, and parsing BLAST file output [39]). In contrast to the crowdsourced implementations of Prechelt [36], the authors created reference implementations for these problems themselves in each of the languages.

They executed these implementations of the common problems and compared them on the following metrics: execution time, memory usage, and program size in lines of code. They also compare these metrics across between Windows (XP) and Linux (Fedora 7). The authors have varying levels of proficiency in the tested languages, which causes varying levels of implementation quality across languages. The authors also discuss language features, most importantly the size of standard libraries, platform independence, and whether the languages are compiled, *semi-compiled* or interpreted.

D.1.6 Bissiyandé et al. (2013) [40]

Bissiyandé et al. look at a huge corpus of open source work available on GitHub. They compare the open source repositories of thirty different commonly used languages. Using these repositories, they compare the popularity of each language by looking at total lines of code, users and projects. They also measure the inter-operability of these languages by looking at multi-language repositories to see which languages are often used together. Finally they measure the success of the languages by looking at the amount of watchers, forks, and issues for the projects that use these languages.

D.1.7 Nanz et al. (2013) [41]

Nanz et al. specifically focus on performance benchmarking of the multicore languages Chapel, Cilk, TBB and Go in this work. The authors compare the languages by some of their features such as communication and programming paradigm. Next up they hire an experienced developers to create implementations of random number generation, histogram thresholding, weighted point selection, outer products, matrix-vector products, and chaining of problems for each of the languages. They combine this with code review by different experts for the respective languages to make sure the implementations are of a certain quality. For all of the implementations they measure and compare the source code size, development time, execution time, and speedup compared to a sequential version of the code.

D.1.8 Aruoba & Fernandez-Villaverde (2015) [42]

Aruoba & Fernandez-Villaverde research programming languages in the field of macroeconomics. They compare C++14, Fortran 2008, Java, Julia, Python, Matlab, Mathematica and R. They present some general information on these languages, including whether the languages are compiled or interpreted, and their standard programming paradigm. They compare these languages using implementations of the *stochastic neoclassical growth model*, which is a common algorithm in macroeconomics. They compare the language on execution time and program size in lines of code. They also state that they would like to measure code complexity and implementation time, although they state that they were unable to compare these metrics in an objective way.

D.1.9 Nanz & Furia (2015) [43]

Nanz & Furia analyse the code inside the Rosetta Code repositories [1] to compare C, C#, F#, Go, Haskell, Java, Python and Ruby. The authors downloaded, compiled (where neccesary) and ran a subset

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

of these Rosetta Code problems that had solutions for all included languages. Not all of the solutions in the repository were working correctly, and the authors explicitly did not correct mistakes in the code to get an accurate view of the actual implementations in the repository. The authors then compared the implementations on code size in lines od code, executable size in bytes, execution time, memory usage, and correctness of the implementations.

D.2 Applicability to Bitcoin Script languages

D.2.1 Implementation-based metrics

Table D.2 shows the studies that work with implementation evaluations.

One of the most mentioned metrics is execution time. Bitcoin Script transactions are executed by the Bitcoin network, meaning they do have some form of execution time. From a user's perspective, this execution time is irrelevant, as they have to wait for their transaction to be included into a block before it is considered confirmed [2].

Another popular metric is the size of the written code, as this can relate to the maintainability, expressiveness, and readability of the code. It can also be related to the efficiency of the language – how many lines of code are needed for a certain implementation. These concepts can be carried over to Bitcoin Script languages as well.

While this code size metric is quite widespread, the development time and coding productivity are used less. It is used in all studies that deal with user-made – rather than author-made – implementations tough, indicating that it is an important metric. An important objective of languages that compile to Bitcoin Script is to make it easier to write scripts for Bitcoin, highlighting the increased importance of this development time metric.

The executable size metric is disregarded in the reviewed studies, as it is mostly irrelevant on modern hardware. In the context of Bitcoin transactions though, it is an important concept. Full scripts need to be included in transactions that try to spend from the script addresses. Since Bitcoin transaction fees are paid per byte, this directly causes higher costs. As discussed in chapter 4, there are limits in place for the size and number of operations that a Bitcoin Script can contain, which places in increased importance on the executable size metric.

Memory usage and compilation time are not extensively used in the reviewed studies. In general purpose programming, compiling is a regular part of the development and deployment processes. With Bitcoin Script, it is possible to test the scripts in a test environment before deployment, but once deployed the code can not be changed. This means that compiling of code is a smaller part of Bitcoin Script development than it is of other development. The memory usage and compilation time metrics are not often included in the general purpose comparisons, and it is even less relevant for Bitcoin Script. They can still be included for reference if it does not require too much effort.

Correctness of the solution is named in a small number of studies. This metric is used to measure how difficult it is to reliably write a program with the expected output. This is extra important when dealing with Bitcoin transactions, as Bitcoin Script can be used to potentially control large amounts of funds.

D.2.2 Static metrics

Table D.3 shows the studies that consider static language features. We see a lot of different metrics being used between these studies, so we only included metrics that were named more than once. Most of these metrics are quite subjective and are measured in different ways in the different studies, making it difficult to find homogeneity among the different works.

The metric that is mentioned most in the reviewed studies is the popularity or ecosystem metric. This looks at the maturity of the surrounding ecosystem, the availability of specialised tooling, and the overall usage rates of the language. This is relevant in the context of Bitcoin Script languages, but the existing languages are very niche and young, making it difficult and less valuable to do such an analysis. An overview of the tooling is included in chapter 3 though.

All Bitcoin Script languages are compiled to Bitcoin Script, while Bitcoin Script itself is interpreted. This makes the compiled vs interpreted language irrelevant. The stateless nature of Bitcoin Script makes these languages functional by definition, although their syntax might differ. Platform independence is also irrelevant as there is no difference between the languages in that regard.

[2] Disregarding 0-confirmation transactions

The complexity and learning curve metrics are interesting in the case of Bitcoin Script language, because these languages are focused on reducing the complexity and learning curve. However, the metrics are very subjective and have no tried-and-true assessments, which makes them difficult to measure. This diminishes the usefulness of such metrics.

Table D.1: Overview of the quality and contents of the studies.

	Citation Count [3]	Conference/Journal Rating [4]	Covered Languages	Evaluates program implementations	Evaluates language features	Includes user input
[35]	34	Q2	C++, Modula-3, Sather, Oberon-2, Self	x	-	-
[36]	360	Q1	C, C++, Java, Perl, Python, Rexx, Tcl	x	-	x
[37]	68	B	C, C++, Eiffel, Haskell, Java, JavaScript, Logo, Pascal, Python, Scheme, VB	-	x	x
[38]	46	N/A	MPI, UTC, X10	x	-	x
[39]	88	Q1	C, C++, C#, Java, Perl, Python	x	x	-
[40]	68	B	30 different languages	-	x	-
[41]	36	A	Chapel, Cilk, TBB, Go	x	x	x
[42]	63	Q1	C++4, Fortran 2008, Java, Julia, Python, Matlab, Mathematica, R	x	x	-
[43]	65	A	C, C#, F#, Go, Haskell, Java, Python, Ruby	x	-	x

[3] Taken from Google Scholar
[4] Taken from `http://www.conferenceranks.com/` and `https://www.scimagojr.com/journalrank.php`

Table D.2: Overview of the metrics used in implementation evaluations.

	Programs written by	Reliability	Execution time	Compilation time	Memory usage	Code size	Executable size	Development time	Coding productivity
[35]	authors	-	x	x	-	-	x	-	-
[36]	users	x	x	-	x	x	-	x	x
[38]	users	x	-	-	-	-	-	x	x
[39]	authors	-	x	-	x	x	-	-	-
[41]	users	-	x	-	-	x	-	x	-
[42]	authors	-	x	-	-	x	-	-	-
[43]	public	x	x	-	x	x	x	-	-

Table D.3: Overview of the language features that were compared that are not based on implementations. Only features that occur in more than one paper are included.

	Compiled vs interpreted	Programming paradigm	Popularity / Ecosystem	Platform independence	Complexity	Learning curve
[37]	-	-	x	x	x	x
[39]	x	x	x	x	-	x
[40]	-	-	x	-	-	-
[41]	-	x	-	-	-	-
[42]	x	x	x	-	x	-