WIP: Introduction to Simplicity

Christian Lewe

September 30, 2022

Chapter 1

Motivation

Simplicity is a language by Russell O'Connor that has been in development since 2015. It can be used to express spending conditions that lock coins (UTXOs) on a blockchain, such as (multi)signatures, hash- and timelocks, covenants and much more. It may be used as a replacement of Bitcoin Script in the future. In this document, we give an overview of Simplicity, starting with the design goals behind it.

1.1 Declarative

Expressing spending conditions in Bitcoin Script *(here, we refer to this expression as "script")* can be difficult without introducing errors. In response, Miniscript was developed as a language for expressing spending conditions in declarative spending policies that let users focus on who can spend what. On the blockchain, the policy is converted into an equivalent script, hiding the underlying implementation details of Bitcoin Script from the user. Therefore, Miniscript can be seen as a subset of Bitcoin Script with a human-friendly interface for expressing spending conditions. Arguably, this is a more natural approach. It prevents bugs by making scripts more human-readable and by making the semantics of a script—i.e., the conditions that it expresses—more explicit. We can statically check properties such as correctness *(whether the resulting script can be redeemed on the blockchain without violating consensus rules)*, malleability *(whether the script can be redeemed with a witness that does not satisfy all spending conditions)* and fees *(how large the resulting script plus witness is and how expensive this size is at the current rate)*. Simplicity goes one step further: Simplicity is a declarative *runtime* environment: We remove the restriction of a stack, inherent to Bitcoin Script, and instead evaluate pure mathematical functions. Spending conditions are expressed in *programs* that are themselves functions, built by combining a range of base functions as fundamental building blocks. The Simplicity Bit Machine then executes these programs for us, hiding the implementation details from the user. As a result, we get all the benefits of Miniscript and much more, as we will see in the following sections.

1.2 Verifiable

The blockchain is irreversible, so we want to absolutely make sure that our coins are locked correctly before we put them on there. Simplicity is mathematically defined, which means that the spending conditions expressed in a Simplicity program are clearly defined. They are independent of any specific implementation in any given programming language. We can use logical reasoners and theorem provers—such as the Coq proof assistant—to prove, for instance, that a signature matching a given public key is the only valid witness for a program and that

any other witness is invalid. We can also prove how many resources a program will use on the Bit Machine, which implies fees. Tasks like these are already hard to do for Miniscript and the underlying Bitcoin Script, but Simplicity allows us to prove *any* mathematical property of a given program. If a property holds for a program, then we can prove it. The use of a verifiable language on the blockchain opens up entirely new possibilities for formal verification: We can essentially *prove* that our programs have no bugs. By proving the security properties first, we can avoid catastrophes like the Parity bug or DAO hack on Ethereum, which were both caused by faulty scripts. "Don't trust. Verify." Blockstream's company motto is stronger than ever in Simplicity.

1.3 Expressive

We want to be able to express as many spending conditions as possible. Miniscript can only express what Bitcoin Script can express, and Bitcoin Script cannot express everything, so it is regularly extended with new op codes. Simplicity can express *every* clearly defined—i.e., decidable—spending condition out of the box. This enables schemes like covenants, weighted thresholds and oracles, and novel cryptography like zero-knowledge proofs and identity-based encryption. The possible use cases that could build on this foundation are endless. With great power comes great responsibility, and with powerful scripts comes the risk of rogue scripts. Simplicity provides all of its expressivity while being verifiable. We can do great things, but we can also prove that we made no mistakes. Figuratively speaking, we tame the expressivity beast with the restraints of formal proof. Also, Simplicity is *not* Turing complete, which means that programs never run into infinite loops. They finish after a short while, whose duration we can computationally predict beforehand.

1.4 Efficient

It should be as fast as possible to evaluate if a given witness satisfies the spending conditions of a given expression. Simplicity's introspection is limited to the current transaction, which means that programs can be evaluated on Bitcoin full nodes without the need for new index structures. The program size is minimized for storage on the blockchain and for transmission: Parts of a program that do the same computation are collapsed into one *(called sharing)*. Furthermore, branches of a program that are not visited by a given witness are removed *(called pruning)*. The latter is similar to how Taproot hides unused scripts, but Simplicity also hides unused branches in a *used* program. Expensive operations—such as signature checks—are implemented as *jets*: Special program expressions that call verified C code during their evaluation. Finally, being able to predict the resource usage of a program on the Bit Machine enables runtime optimizations.

Chapter 2

Declarative Circuits

Simplicity is a language for writing circuits: A Simplicity program is a circuit. In the following, we give an account of the structure Simplicity and how it relates to traditional circuits. In reality, the Simplicity Bit Machine interprets a program in terms of types and executes it in terms of bits. We ignore type theory for the moment and instead focus on how bits are modified by each operation. By focusing on bits, we can become more familiar with Simplicity without requiring extra theory.

2.1 Traditional Circuits

Circuits consist of *gates*, which are their atomic building blocks, connected via wires. Traditionally, these are `NOT`, `AND`, `OR`, etc. Circuits compute *Boolean functions*, i.e., functions that take a certain number of bits as input and produce a certain number of bits as output. While gates already compute fundamental functions, we can combine multiple gates to compute more complex functions. An `XOR` and `AND` gate can be combined to form a half adder. `XOR`, `AND` and `OR` form a full adder, etc. Certain sets of gates can simulate other ones. Gates `NOT` and `AND` can simulate all possible gates. Alternatively, gate `NAND` alone is also enough. This enables us to write a circuit that computes *any* Boolean function using only gates from the set. We say, the set of gates is *functionally complete*.

2.2 Simplicity Circuits

Simplicity goes a slightly different route: Wires carry a certain number of bits, like voltages, and gates change the number of bits, like a change in voltage (down *or up*). Besides the number of bits, each bit is either true of false *(set or unset)*. Each gate modifies the bitstring it receives based on the function that it computes. To summarize, wires in Simplicity circuits can carry multiple bits, while traditional wires carry only one. Simplicity circuits take a bitstring as input, gates modify this bitstring and the circuit produces a certain bitstring as a result.

In addition, we work with gates that take a *variable* number of input bits and that produce a certain number of bits as result, based on a mathematical function. For instance, a gate can take $n+m$ bits as input and produce n bits as output. Here, the output function is $f(n, m) = n$. Such a variable gate is called a gate *schema*. By assigning fixed values to variables, we obtain a gate with a fixed number of inputs and outputs. In the above example, we could set $n \coloneqq 1$ and $m \coloneqq 2$ to obtain a gate that takes 3 bits of input to produce 1 bit of output. Such a fixed gate is called an *instance* of the schema.

Schemata help us design circuits, focusing on large mathematical functions that ultimately express spending conditions, instead of focusing on tiny bits that do very little in the big picture.

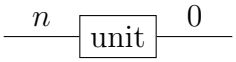
When we evaluate a circuit on some input of fixed size, then all variables are fixed and we end up with an instance. This conversion from schema to instance happens in the background. Simplicity circuits are *declarative*: Users define *(declare)* the function they want to compute, in terms of variables, and the Bit Machine does the computation in the background. Users never have to define how the evaluation is done, just what they want to be evaluated.

2.3 Combinators

For convenience, we make the wires in Simplicity circuits implicit by using *combinators*: A combinator takes zero to two circuits as argument and produces a larger circuit. By recursively nesting combinators, we can build larger and larger circuits. This turns the circuit into a tree structure where combinators have up to two children. There is a topmost combinator, called the *root*. The bottommost combinators have no children and are called *leaves*. The root of each circuit determines its input and output type. Bits are passed to the root as input, the root passes these bits to its children, the children return other bits as output, and the root returns them as output. This process of passing input through the children is repeated everywhere in the tree.

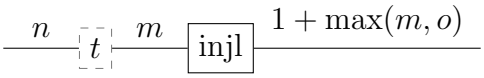
In the following, the flow of bits will become obvious due to the wires. We introduce each combinator, illustrating the circuit that it represents and explaining the function that it computes. Boxes with variables s, t stand for any circuit that is inserted into the diagram *(a circuit variable)*. The remaining boxes stand for gate schemata. Each wire has a number that stands for its number of bits. Bits always flow from left to right.

2.3.1 Unit Constant



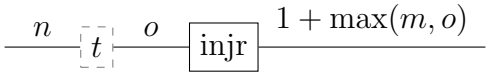
The unit combinator reads some number n of bits and writes zero bits. This is of little use on its own, but serves as the foundation for bitstrings. The number n of read bits depends on the context, i.e., how many bits are fed into "unit" via the surrounding circuit. By default, it holds that $n = 0$.

2.3.2 Left Injection



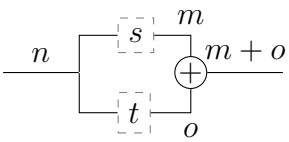
The left injection combinator takes one argument—circuit t that reads n bits and writes m bits—and places its output in a *union* as *left* value. Length o of the *right* type depends on the context, i.e., how many bits are read from "injl" via the surrounding circuit. By default, it holds that $o = 0$. A *(tagged) union* is a type that can contain values from a fixed set of types, but only one at a time. Which type is contained is encoded in the *tag* which is part of the union. In Simplicity, unions have a *left* type and a *right* type. The tag is therefore a simple bit, which is "false" if a left value is contained and "true" if a right value is contained. Given a left type of length m and a right type of length o, the length of their union is $1 + \max(m, o)$. *This helps with memory alignment. The shorter type is padded internally.*

2.3.3 Right Injection



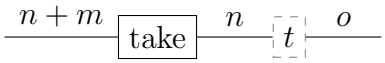
The right injection combinator takes one argument—circuit t that reads n bits and writes o bits—and places its output in a *union* as *right* value. Again, length m of the left type depends on the context and it is zero by default. With injections, we can produce bits: By wrapping m bits in a union as left value via "injl", we add a "false" bit at the front. By wrapping o bits in a union as right value via "injr", we add a "true" bit. If circuit t is "unit", then we obtain a circuit that produces a constant "false" or "true" no matter the input.

2.3.4 Pair



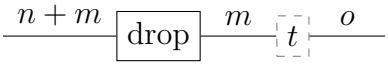
The pair combinator takes two arguments—circuit s that reads n bits and writes m bits, and circuit t that reads n bits and writes o bits—and *concatenates* their outputs. That is, the overall input becomes the input of both circuits, and the overall output is the output of the first circuit followed by the output of the second circuit. With concatenation, we can produce arbitrary bitstrings: Simply concatenate as many bits, produced by injections, as necessary. By nesting the unit, injection and pair combinators, we can write a circuit that produces any bitstring as constant output.

2.3.5 Take



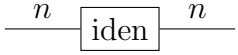
The take combinator takes one argument—circuit t that reads n bits and writes o bits—and evaluates it on the *first n bits* of the overall input. That is, the first n bits the input are *taken* and the remaining m bits are discarded. The number m depends on the context, i.e., how many bits are fed into "take" via the surrounding circuit. By default, it holds that $m = 0$.

2.3.6 Drop



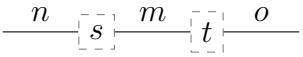
The drop combinator takes on argument—circuit t that reads m bits and writes o bits—and evaluates it on the *last m bits* of the overall input. That is, the first n bits of the input are *dropped* and the remaining m bits are used as input for t. Again, number n depends on the context and it is zero by default. By recursively nesting the take and drop combinators, we can deconstruct bitstrings in arbitrary ways: We can write circuits that take a bitstring of given length as input and return any sub-slice of it.

2.3.7 Identity



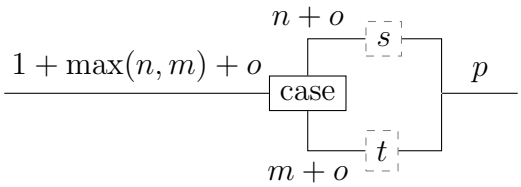
The identity combinator reads n bits and writes the same n bits. The circuit therefore computes the *identity* function. It is often used in conjunction with "take" and "drop" to produce a subslice of a given bitstring, or to keep a copy of the input in one branch of a pair while the other branch computes something. There is no other circuit that passes through the input.

2.3.8 Composition



The composition combinator takes two arguments—circuit s that reads n bits and writes m bits, and circuit t that reads m bits and writes o bits—and *composes* the two circuits. That is, the overall input becomes the input of the first circuit, whose output becomes the input of the second circuit, whose output becomes the overall output. In programming, this is also called *chaining* or *piping*.

2.3.9 Case



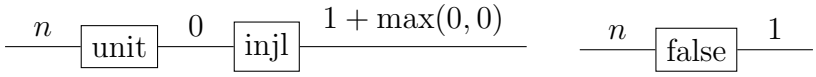
The case combinator takes two arguments—circuit s that reads $n+o$ bits and writes o bits, and circuit t that reads $m+o$ bits and writes o bits—and *branches* to either circuit depending on its input. The input consists of a union of a left type of length n and a right type of length m, concatenated with o additional bits. If the tag is zero, then the union contains a left value and circuit s is evaluated on the left value plus the trailing o bits. If the tag is one, then circuit t is evaluated on the contained right value plus the trailing bits. Note how the first bits of the input of s and t differ, while the remaining input and output are the same. In programming, the case combinator corresponds to an *if-statement*. Given the (de)construction of arbitrary bitstrings and the three functional primitives above, we can compute any Boolean function using some circuit.

2.4 Examples

Let us look at some examples of Simplicity programs that compute useful functions. As already said, Simplicity is functionally complete, i.e., for each Boolean function there is a program that computes said function. Any decidable spending condition can be expressed in a program that might or might not be small. If the program ends up large, its size can be reduced by using so-called *jets* (see Section 2.7).

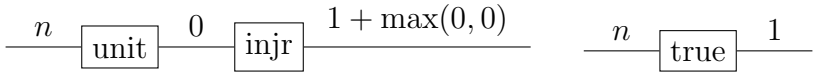
Note: We deliberately keep algebraic expressions $x + y$, $\max(x, y)$ and the parentheses as they are, even though it might seem natural to resolve them. It is important to keep them separate, because the combinators will break them down again.

2.4.1 False Constant



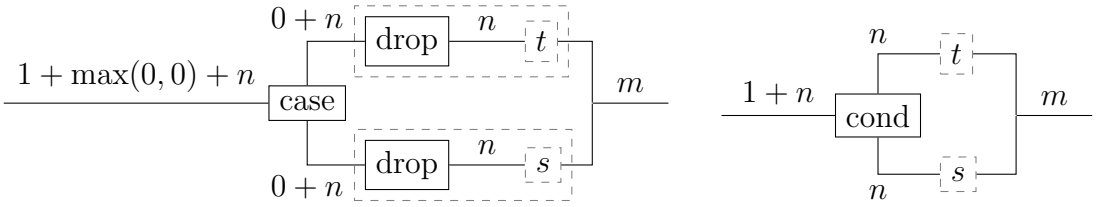
By nesting the unit constant in a left injection, we obtain a circuit that produces a constant "false" bit no matter the input *(see left)*. The n input bits are variable and depend on the surrounding circuit. The input is passed to "unit", which passes 0 bits to "injl", which in turn wraps these 0 bits in a union as left value. The length of the right value of this union is also 0 bits, so the length of the union is 1 bit, which is what we wanted. As a shorthand, we can write the "false" circuit as a gate *(see right)*.

2.4.2 True Constant



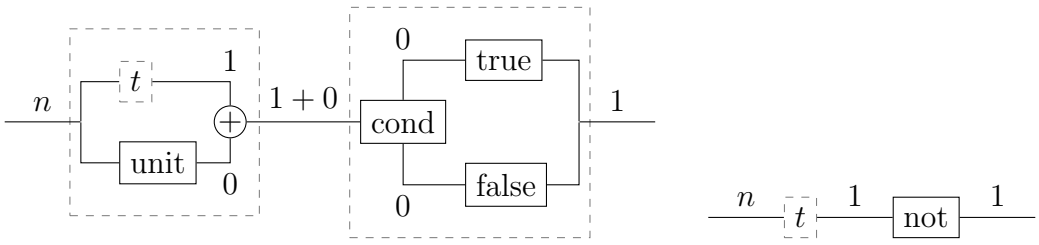
By nesting the unit constant in a right injection, we obtain a circuit that produces a constant "true" bit no matter the input *(see left)*. The computation is analogous to that of the false constant. As a shorthand, wer can write the "true" circuit as a gate *(see right)*.

2.4.3 Condition



By nesting drop circuits in a case circuit, we obtain a circuit that resembles an "if-else-statement" *(see left)*: The circuit takes two arguments—circuits s and t that read n bits and write m bits, respectively—and branches to either of them based on the tag bit of the input. Circuit s corresponds to the "if" clause and circuit t to the "else" clause. *To avoid confusion, we consistently put negative branches above positive ones, like for "case", so the order of s and t is reversed.* If the tag bit is false, then $0+n$ bits are passed to the top "drop", which passes n bits to t, which in turn produces m output bits. If the tag bit is true, then the bottom branch produces m output bits. As a shorthand, we can write the "cond" circuit as a gate *(see right)*.

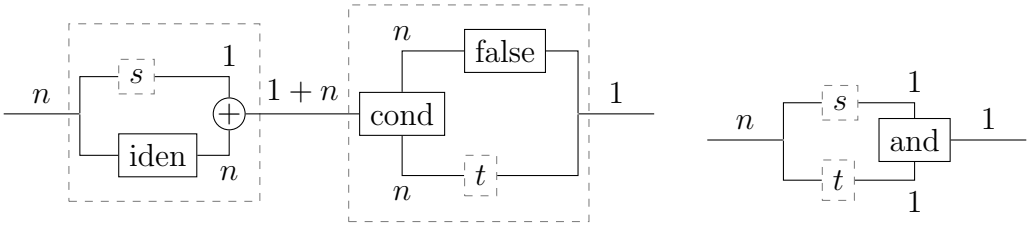
2.4.4 Negation



Now we show how traditional gates are represented as Simplicity circuits: The negation circuit takes one argument—circuit t that reads n bits and writes 1 bit—and negates this output *(see left)*. First, a pair joins the output of t with zero bits. Second, a condition evaluates the first bit and passes the remaining zero bits to either top or bottom branch. If the first bit is false, then

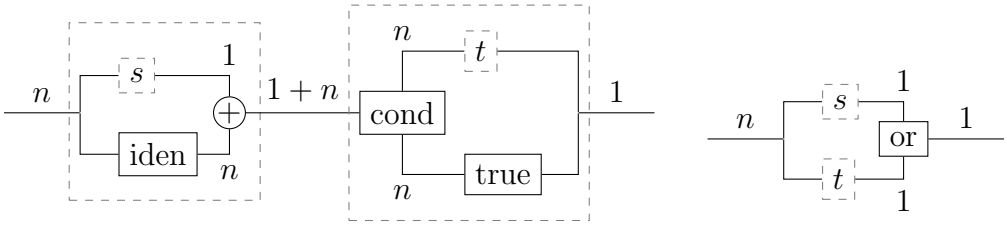
the top branch produces a constant "true". Otherwise, the bottom branch produces "false". The zero bits are ignored. As you can see, the output is "true" if t produces "false" and vice versa. As a shorthand, we can write the "not" circuit as a gate *(see right)*.

2.4.5 Conjunction



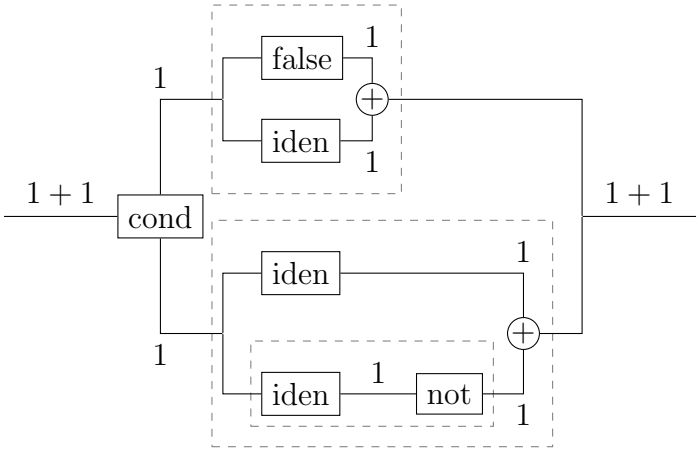
The conjunction circuit takes two arguments—circuits s and t that read n bits and write 1 bit, respectively—and computes the logical conjunction of both outputs *(see left)*. First, a pair joins the output of s with a copy of the original n input bits. Second, a condition evaluates the first bit and passes the remaining n bits to either top or bottom branch. If the first bit is false, then the top branch produces a constant "false". Otherwise, the bottom branch evaluates t on the n input bits. As you can see, the output is "true" if both s and t produce "true" and "false" otherwise. As a shorthand, we can write the "and" circuit as a gate *(see right)*.

2.4.6 Disjunction



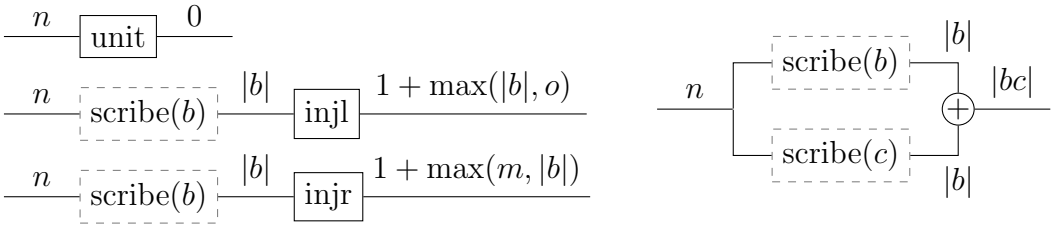
The disjunction circuit takes two arguments—circuits s and t that read n bits and write 1 bit, respectively—and computes the logical disjunction of both outputs *(see left)*. The pair-condition construction is just as for conjunction, but with different branches: If the first bit (from s) is false, then the top branch evaluates t on the n input bits. Otherwise, the bottom branch produces a constant "true". As you can see, the output is "false" if both s and t produce "false" and "true" otherwise. As a shorthand, we can write the "or" circuit as a gate *(see right)*.

2.4.7 Half Adder



We can combine the above gates to construct arithmetic circuits of any kind, like adders, subtractors and multipliers. For brevity sake, we limit ourselves to a very simple circuit, namely the half adder: The circuit reads 2 bits a and b, adds them and writes 2 output bits c and s as a result. The first output c is the carry bit while the second output s is the sum modulo two. If a is "false", then the top branch receives b. A pair joins a constant "false" with b and passes that to the output. That is, the carry bit is always false and the sum bit equals b. This makes sense, since $0 + b = b$ carry 0. If a is "true", then the bottom branch receives b. A pair joins a copy of b with the negation of b and passes that to the output. That is, the carry bit equals b and the sum is the negation of b. This makes sense, since $1 + 0 = 1$ carry 0 and $1 + 1 = 0$ carry 1.

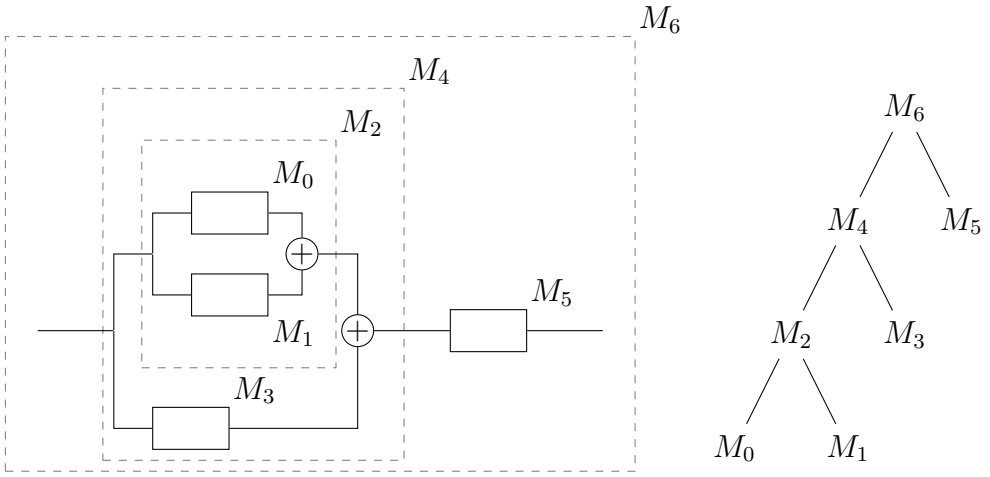
2.4.8 Scribe



Simplicity's functional completeness includes one special case, namely the ability to produce any constant in a circuit no matter the input. For each bitstring a that we want to produce, there exists such a circuit which we call scribe(a). If a is the empty string, then scribe(a) is the unit constant *(see top left)*. If a is a "false" bit followed by bitstring b, then we can produce it by passing the output of scribe(b), which produces b, into a left injection *(see middle left)*. This adds a false tag bit in front of b. Note that in principle, the length o of the right type of this union could be nonzero, but for our purposes we may assume it to be zero. If a is a "true" bit followed by bitstring b, then we pass the output of scribe(y) into a right injection, analogous to the above *(see bottom left)*. Again, we assume the length m to be zero. If a is the concatenation of two bitstrings b and c, then we join the outputs of scribe(b) and scribe(c) in a pair *(see right)*. In all cases, the input is completely ignored and a constant output is produced.

2.5 Merkle Roots

Programs have a tree structure: The outermost combinator, called *root*, forms the entire circuit and determines the program's input and output type. Other combinators are recursively nested in the root as arguments, according to the respective combinator's definition (see Section 2.3). At the bottom of this tree are combinators without arguments, called *leaves*.



Above is an illustration of a circuit with its nested combinators (dashed boxes). Each combinator has a Merkle root which uniquely identifies it and which is computed from its arguments. On the right is an illustration of the corresponding tree structure of the circuit.

2.5.1 Commitment Merkle Root

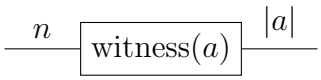
The commitment Merkle root (CMR) uniquely identifies a program at *commitment time*. At this time, we define a program that should lock a UTXO and *commit* to the program by writing its CMR in a transaction output in a block. Because the program is a Merkle tree, the tiniest change in its structure will lead to a completely different Merkle root. Later at *redemption time*, we unlock the UTXO by providing a program whose CMR equals the UTXO's CMR and that successfully runs on the Bit Machine.

$$
\begin{aligned}
\mathrm{CMR}(x) &:= \mathrm{tag}(x) && \text{for nullary combinator } x \\
\mathrm{CMR}(x\ t) &:= \mathrm{SHA256}_{\mathrm{Block}}(\mathrm{tag}(x), 0\ldots0 || \mathrm{CMR}(t)) && \text{for unary combinator } x \\
\mathrm{CMR}(x\ st) &:= \mathrm{SHA256}_{\mathrm{Block}}(\mathrm{tag}(x), \mathrm{CMR}(s) || \mathrm{CMR}(t)) && \text{for binary combinator } x
\end{aligned}
$$

The CMR is roughly defined as above. Each combinator is associated with a unique tag, which is a 256-bit number. The CMR of a nullary combinator *(with no arguments)* is simply this tag. For unary combinators *(with one argument)*, we compute the CMR via the SHA256 block compression function: The initial value (first argument) is the tag, while the block (second argument) consists of 256 zeroes concatenated with the CMR of the combinator's argument t. The CMR of binary combinators *(with two arguments)* is computed analogously, but we replace the 256 zeroes with the CMR of s.

There is one problem: We do not know the witness that will be used to unlock the UTXO at commitment time! The program restricts the possible witnesses that can be used for unlocking—this is what it means for a program to express a spending condition—but the individual witness that will be used in the future is unknown. We must be able to add witness data to a program without changing its CMR. A special combinator is needed, namely the *witness* combinator.

2.5.2 Witness



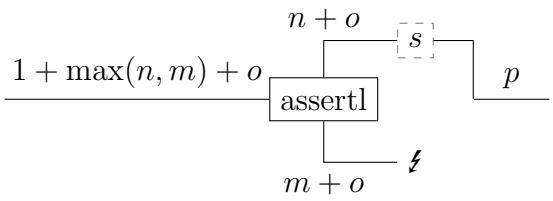
The witness combinator reads n bits and writes bitstring a no matter the input. Just like "scribe" (see Subsection 2.4.8), the witness combinator reads an arbitrary number of bits and writes a constant output. However, the former is one gate that writes its output in one step and the latter is many gates that write their output in many steps. But the deciding difference is the unique property of "witness": Changing its contained bitstring does *not* change its CMR. This enables the following workflow: At commitment time, the UTXO writer defines a program which includes a witness combinator. The combinator may contain any bitstring. He commits to the program via its CMR. At redemption time, the UTXO redeemer changes the program to include her witness and puts it in her transaction input. This way, the "same" program can be used at both commitment and redemption time while the witness can be changed. There is one important restriction, namely that the bitstring of a witness combinator must always be of the same length. The UTXO writer specifies how long the bistring should be and the UTXO redeemer must provide a bitstring of this length. To summarise, "scribe" produces fixed constants while "witness" produces variable constants that are provided as witness data.

2.6 Assertions and Failure

Programs might fail during execution. This has two use cases: First, validating spending conditions involves a sequence of Boolean checks: *Does the signature match the public key? Does the preimage hash to the image? Is the current block height less or equal to the locktime?* By making the entire program fail if one of its checks fails, we can do each check once and then forget about it. We do not need to keep track Boolean outputs. With this approach, it is possible to write programs that read zero bits, validate through hard-coded *(scribe)* and variable constants *(witness)*, and write zero bits as a result. The number of Boolean checks inside the program does not influence its output type.

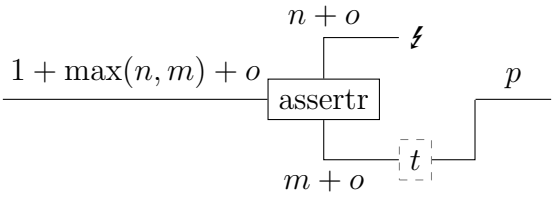
Second, a given witness determines exactly which branches of the program will be explored during execution. We can compress the program by removing all unused branches. This is called *pruning* and saves valuable space on the blockchain.

2.6.1 Left Assertion



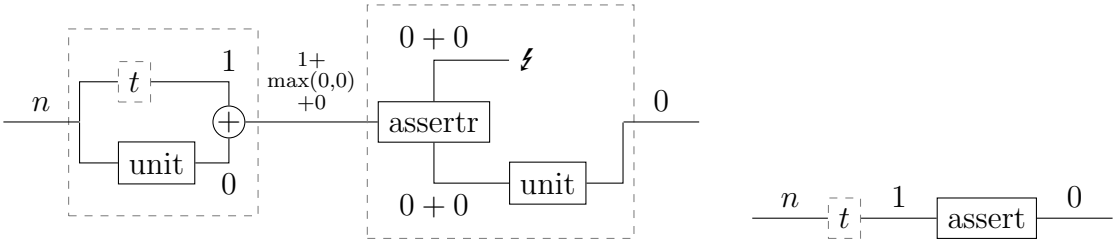
The left assertion combinator takes one argument—circuit s that reads $n + o$ bits and writes p bits—branches to it if the input tag bit is false and fails otherwise. The combinator is exactly like "case" (see Subsection 2.3.9), but with a failure branch on the bottom. Length m of the right type of the union depends on the surrounding circuit. The CMR of "assertl" is defined as the CMR of "case", where circuit t is a free variable. For each left assertion, we must choose such a t to set the CMR.

2.6.2 Right Assertion



The right assertion combinator takes one argument—circuit t that reads $m + o$ bits and writes p bits—branches to it if the input tag bit is true and fails otherwise. The combinator is exactly like "case" (see Subsection 2.3.9), but with a failure branch on the top. Length n of the left type of the union depends on the surrounding circuit. The CMR of "assertr" is defined as the CMR of "case", where circuit s is a free variable. For each right assertion, we must choose such a s to set the CMR.

2.6.3 Assertion



We can turn a Boolean check into an assertion by using the assertion circuit *(see left)*. It takes a circuit t that reads n bits and writes 1 bit, writes 0 bits if this bit was "true" and fails otherwise. First, a pair joins the output of t with zero bits. Second, an assertion evaluates the first bit, fails on "false" and passes the remaining zero bits to the bottom branch on "true". Finally, the unit combinator writes zero bits as output. As a shorthand, we can write the "assert" circuit as a gate *(see right)*.

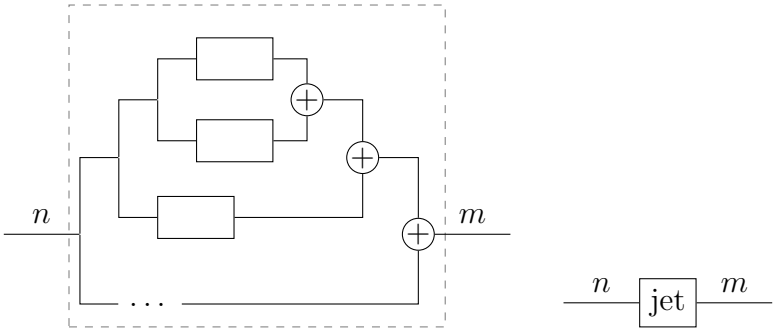
2.6.4 Pruning

We can compress a program by pruning all branches that are not explored during execution for a given witness. Note that this works only for programs that take no input. Also note that the output of this compression depends on the witness. First, we execute the program and observe which branches are explored. Next, we replace case combinators (1) with left assertions if the right branch was unused, and (2) with right assertions if the left branch was unused.

Because "assertl" and "assertr" have exactly the same parameters as "case", the numbers n, m, o and p remain the same. Because the CMR of "assertl" and "assertr" is the same as for "case" *(if we keep the same s and t)*, the CMR of the overall program remains the same. The UTXO writer can commit to an unpruned program, the UTXO redeemer prunes it according to her witness and both programs will be equal according to their CMR. This is how we compress programs at redemption time while keeping equality to the original program commitment.

2.7 Jets

Simplicity can verify any function, but because of the generic nature of its combinators, useful programs are typically large in size. SHA256 hashing involves many steps, so does Schnorr signature checking, etc. Meanwhile, evaluating Simplicity circuits is not as fast as evaluating a program written in C. The latter works like a computer works and enjoys hardware acceleration, so it is naturally magnitudes faster. Blocks must be verified quickly, so we must speed up circuit evaluation. Block space is also valuable, so circuits must be as small as possible.



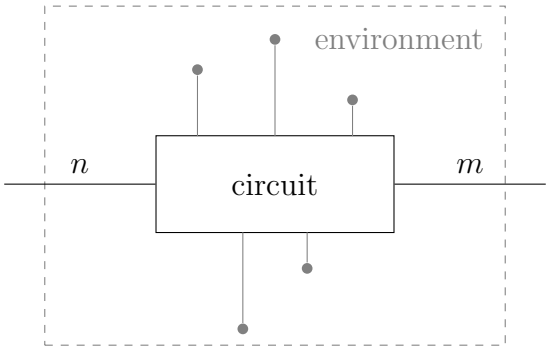
We solve the problems of speed and size by using *jets*: combinators that are associated with a given C program that behaves exactly as some Simplicity circuit. Whenever we want to use a

circuit that is rather large and expensive *(see left)*, we instead use its jet *(see right)*. The jet is evaluated by calling the C code, so the computed function remains the same, including the input and output type as well as the computation result. A catalogue includes jets for a range of computations, such as arithmetic, hashing and secp256k1 cryptography, which we constantly work on expanding.

Jets are a single gate, so they are much smaller than the circuit that they replace. This solves the problem of block space and fees. Meanwhile, jets use C code that behaves exactly as the circuit that it replaces. Simplicity is mathematically defined, so we can write a specification of the input a program takes (pre condition) and of the output it produces (post condition). The C code is written in Verifiable C, so we can prove mathematical properties about it that also hold for the resulting assembly. We use the Coq proof assistant to show that the C code produces the same output (post condition) upon a given input (pre condition). This solves the problems of speed and correctness.

In the future, programs will consist almost exclusively of jets. Simplicity circuits will almost become a theoretical construct, while the real computation happens in verified C code. However, the real picture is more nuanced than that. We design jets to solve cryptographic primitives, functions that can be combined to form protocols and locking schemes, so more than one jet is required to express a useful spending condition. Jets also cannot contain witness data, so witness combinators are required. Jets will do the heavy cryptographic lifting while the remaining circuit combines this cryptography in interesting ways, like conjunctions, disjunction and thresholds.

2.8 Computational Environment



Simplicity circuits evaluate a function upon an input. This input comes from four sources: (1) the input to the circuit itself *(see n bits)*, (2) constants hard-coded inside the circuit *(see Subsection 2.4.8)*, (3) constants provided as witness data *(see Subsection 2.5.2)* and (4) constants read from the computational environment *(see gray dots)*. A circuit is evaluated from left to right: Values are passed into it and passed out of it. As we go through the circuit, some parts produce hard-coded constants. Further data comes from outside the circuit: witness and environment data. This data lives outside and creates a context in which a circuit is evaluated. For the same input, the circuit may produce a different output in different contexts. Only if the input and context are the same, then the output is guaranteed to also be the same.

Both the witness and the environment are variable, but the latter has nothing to do with program commitments. Neither the UTXO writer *(at commitment time)* nor the UTXO redeemer *(at redemption time)* has any influence over it. The environment commonly contains the current transaction, but it can contain any data in principle, as long as it is fixed and agreed-upon by all participants of the blockchain.

Constants are read from the environment by using *primitives*: combinators that are associated with a parameter of the environment and return its value. Primitives are a window

into the environment. They read zero bits and write their value in as many bits as necessary. For this to work, the kinds of parameters that exist in the environment must be known when writing a program. We define Simplicity *applications* and their type of environment. A Simplicity program written *for a given application* is always evaluated in an environment of the same type. For each parameter, there is a primitive that returns its value. Naturally, the available primitives differ for each application. Prominent Simplicity applications include *Core*, which has an empty environment; *Bitcoin*, which has a Bitcoin transaction environment; and *Elements*, which has an Elements transaction environment.

For technical reasons, we treat primitives as jets. Both are evaluated by calling C code: one to read from the environment and the other to execute the jet. Both have a known input and output type. The difference is that primitives read from the environment, while (genuine) jets compute the same result as their associated circuit.

Chapter 3

Verifiable Circuits

Chapter 4

Efficient Circuits

4.1 Merkle Root

4.1.1 Identity Merkle Root

The identity Merkle root (IMR) uniquely identifies a program at redemption time. It is the counterpart of the CMR and is defined just as the latter, with one important exception: The IMR depends on the bitstring contained in witness combinators.

$$\begin{aligned} \text{CMR}(\text{witness } a) &\coloneqq \text{tag}(\text{witness}) \\ \text{IMR}(\text{witness } a) &\coloneqq \text{SHA256}_{\text{Block}}(\text{tag}(\text{witness}), a||\text{len}(a)||0\ldots0) \end{aligned}$$

The CMR of a witness combinator is simply its tag, independent of the bitstring. Meanwhile, the IMR of a witness combinator is computed as SHA256 of bistring a with the tag as initial value. *(This computation is a simplification, because we deal with bitstrings. In reality, Simplicity works in terms of Simplicity values. See Chapter 3 for more details.)* As a consequence, changing the witness changes the IMR.

4.1.2 Sharing