Bachelor's Thesis Computing Science

Proving Correctness of Cryptographical Primitives through While

Alex Feenstra
s1052638

October 30, 2023

First supervisor/assessor:
Dr. Engelbert Hubbers

Second assessor:
Dr. Ir. Bart Mennink

Radboud University

Abstract

Proving the security claims of cryptographic primitives is essential, but not sufficient. Many implementations have been shown to contain bugs, which can sometimes destroy the security claim completely. We examine a new technique to prove implementation correctness of cryptographic primitives, and show its feasibility by applying it to Keccak-f[200], and an optimized variant. By first transforming the program into so-called `While` code, the technique operates as language-agnostic as possible. Finally, we discuss its limitations and drawbacks.

Contents

Chapter 1

Introduction

In cryptography, lots of effort is spent on designing secure encryption schemes and permutation functions. Unfortunately, even a perfect primitive can fail when an implementation error is made. An example of this is the SHA-3 competition [18]. Nearly half of the submitted reference implementations contained bugs, and so did the winner, which was only discovered after seven years.
Some bugs might only slightly reduce the security claim of an implementation, while others can reduce it to effectively zero. Due to the nature of bugs (bugs being accidental errors introduced while writing a program), it is not possible to compensate for them. The only solution, then, is to reduce or completely eliminate them.
The goal of this paper is to show that this is feasible, by taking a specific implementation of a hashing algorithm, and proving that it is semantically equal to its reference implementation. To achieve this, we transform both implementations into their `While`$^{+}$ equivalents, apply the `While` derivation rules [20] on them, and show that the resulting state mappings are equal.
If the state mappings are equal, then so must the semantics of the two programs be, as well. This reduces the problem of trying to prove implementation correctness for any program, to showing that the extensions to `While` and the transformation rules are correct.

The solution we present in this paper is the first of its kind. It is a one size fits all approach; through extending `While` so that we can effectively reason about cryptographical primitives, we only need to add new transformation rules if we wish to prove semantic equality for a new language, as opposed to having to redevelop the entire system.
That is not to say it is the only approach for proving the correctness of programs which relate to cryptography. Indeed, papers have been written on the automatic verification of protocols [7, 3], and likewise on the automatic extraction of a model based on an implementation [5], but these all function

on *one* programming language or protocol model.

In Chapter 2, we introduce core concepts from software verification used in the rest of the thesis. After that, we present the entirety of the formal definition of our new technique in Chapter 3. We show the application of said technique in Chapter 4, explaining the results of the transformation and how to interpret the results of the annotated programs.
The program transformation, annotated programs and various helper scripts are present within Appendix A. We show that existing techniques do not cover the points our technique covers in Chapter 5 and finally go over the feasibility of our technique in Chapter 6.

Chapter 2

Preliminaries

In this thesis, we use terms relating to software verification, as well as (a small subset of) cryptography. We shortly explain the core concepts from each below.

2.1 `While`

`While` [20] is a language in which the most advanced construct is a while loop. It also contains if-then-else statements, integer variables, basic arithmetic operators ($+$, $-$, $*$) and basic Boolean operators ($<$, $<=$, $\neg$, $==$, $\wedge$). While it *is* Turing complete, `While` is not very expressive, which makes it unfit for normal software development, but this also makes it easy to describe its semantics. For this reason, it is used in software verification.

2.2 Derivation trees

To reason about the correctness of programs written in so-called `While`, we make use of natural semantics based on derivation trees. These trees take a program, and apply derivation rules on it to produce a predicate that describes the semantics of it. Typically, these predicates are much easier to reason about than the entirety of the source code. Because `While` is not very expressive, the soundness (being able to prove only things that are true) and completeness (being able to prove all things that are true) of its deriavtion rules as described in [20] is well known.
We provide the following excerpt from [20], which defines derivation trees:

> In a natural semantics we are concerned with the relationship between the *initial* and the *final* state of an execution. Therefore the transition relation will specify the relationship between the initial state and the final state for each statement.

We shall write a transitions as

$$\langle S, s\rangle \to s'$$

Intuitively this means that the execution of S from s will terminate and the resulting state will be s'.

A *rule* has the general form

$$\frac{\langle S_1, s_1\rangle \to s'_1, \ldots, \langle S_n, s_n\rangle \to s'_n}{\langle S, s\rangle \to s'} \text{if } \ldots$$

where $S_1, \ldots, S_n$ are *immediate constituents* of S or are statements *constructed* from the immmediate constituents of S. A rule has a number of *premises* (written above the solid line) and one *conclusion* (written below the solid line). A rule may also have a number of *conditions* (written to the right of the solid line) that have to be fulfilled whenever the rule is applied. Rules with an empty set of premises are called *axioms* and the solid line is then omitted.
Intuitively, the axiom [ass_{ns}] says that in a state s, $x := a$ is executed to yield a final state $s[x \mapsto \mathcal{A}[\![a]\!]s]$, which is as s except that x has the value $\mathcal{A}[\![a]\!]s$. This is really an *axiom schema* because x, a and s are meta-variables standing for arbitrary variables, arithmetic expressions and states but we shall simply use the term axiom for this. We obtain an *instance* of the axiom by selecting particular variables, arithmetic expressions and states. As an example, if s_0 is the state that assigns 0 to all variables then

$$\langle x := x + 1, s_0\rangle \to s_0[x \mapsto 1]$$

is an instance of [ass_{ns}] because x is instantiated to be x, a to $x + 1$, s to s_0, and the value $\mathcal{A}[\![x + 1]\!]s_0$ is determined to be 1.
Similarly [skip_{ns}] is an axiom and, intuitively, it says that **skip** does not change the state. Letting s_0 be as above we obtain

$$\langle \mathit{skip}, s_0\rangle \to s_0$$

as an instance of the axiom [skip_{ns}].
Intuitively, the rule [comp_{ns}] says that to execute $S_1; S_2$ from state s we must first execute S_1 from s. Assuming that this yields a final state s' we shall then execute S_2 from s'. The premises of the rule are concerned with the two statements S_1 and S_2 whereas the conclusion expresses a property of the composite statement itself. The following is an *instance* of the rule:

$$\frac{\langle \mathit{skip}, s_0\rangle \to s_0, \langle x := x + 1, s_0\rangle \to s_0[x \mapsto 1]}{\langle \mathit{skip}; x := x + 1, s_0\rangle \to s_0[x \mapsto 1]}$$

Here S_1 is instantiated to *skip*, S_2 to $x := x + 1$, s and s' are both instantiated to s_0 and s'' is instantiated to $s_0[x \mapsto 1]$.

For the if-construct we have two rules. The first one, $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$, says that to execute *if* b *then* S_1 *else* S_2 we simply execute S_1 provided that b evaluates to $\mathbf{tt}$ in the state. The other rule, $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]$, says that if b evaluates to $\mathbf{ff}$ then to execute *if* b *then* S_1 *else* S_2 we just execute S_2. Taking $s_0 x = 0$ the following is an instance of the rule $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$:

$$\frac{\langle \mathit{skip}, s_0\rangle \to s_0}{\langle \mathit{if}\ x = 0\ \mathit{then}\ \mathit{skip}\ \mathit{else}\ x := x + 1, s_0, \rangle \to s_0}$$

because $\mathcal{B}[\![x = 0]\!]s_0 = \mathbf{tt}$. However, had it been the case that $s_0 \neq 0$ then it would not be an instance of the rule $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$ because then $\mathcal{B}[\![x = 0]\!]s_0$ would amount to $\mathbf{ff}$. Furthermore it would not be an instance of the rule $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]$ because the premise has the wrong form.

Finally, we have one rule and one axiom expressing how to execute the *while*-construct. Intuitively, the meaning of the construct *while* b *do* S in the state s can be explained as follows:

- If the test b evaluates to true in the state s then we first execute the body of the loop and then we continue with the loop itself from the state so obtained.
- If the test b evaluates to false in the state s then the execution of the loop terminates.

The rule $[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]$ formalizes the first case where b evaluates to $\mathbf{tt}$ and it says that then we have to execute S followed by *while* b *do* S again. The axiom $[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}]$ formalizes the second possibility and states that if b evaluates to $\mathbf{ff}$ then we terminate the execution of the *while*-construct leaving the state unchanged. Note that the rule $[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]$ specifies the meaning of the *while*-construct in terms of the meaning of the very same construct so that we do *not* have a compositional definition of the semantics of statements.

When we use the axioms and rules to derive a transition $\langle S, s\rangle \to s'$ we obtain a *derivation tree.* The *root* of the derivation tree is $\langle S, s\rangle \to s'$ and the *leaves* are instances of axioms. The *internal nodes* are conclusions of instantiated rules and they have the corresponding premises as their immediate sons. We request that all the instantiated conditions of axioms and rules must be satisfied. When displaying a derivation tree it is common to have the root at the bottom rather than at the top; hence the son is *above* its father. A derivation tree is called *simple* if it is an instance of an axiom, otherwise it is called *composite*.

Note that while this excerpt contains the most important information about derivation trees, it does not cover everything present in `While` [20]. In particular, the definition of the state and syntax evaluation functions are not present.

2.2.1 State

The state can be seen as a mapping from a variable to a number. More formally,

$$\text{State} = \text{Var} \to \mathbb{Z}.$$

The authors of `While` [20] specify that a state might also be specified as a list or table. However, in this thesis, the state will always be a function.

2.2.2 Syntax evaluation functions

The link between syntax and semantics is made through evaluation functions. Usually styled as $\mathcal{A}$ (for Arithmetic expressions, in this example), these functions take a piece of syntax and a state, and then provide the resulting semantic value. More formally,

$$\mathcal{A} : \text{AExp} \to (\text{State} \to \mathbb{Z}),$$
$$\mathcal{B} : \text{BExp} \to (\text{State} \to \{\mathbf{tt}, \mathbf{ff}\}),$$

where ‘AExp’ are arithmetic expressions, and ‘BExp’ are boolean expressions.

2.3 Cryptographic primitives

A cryptographic primitive is an algorithm which functions as a building block. A cryptographic protocol will typically use several primitives to obtain some kind of security-related goal. We briefly discuss several examples of primitives below.

2.3.1 Authenticated encryption schemes

An authenticated (symmetric) encryption scheme is an algorithm which encrypts and authenticates its input using a key. For encryption, it should be infeasible to figure out what the original data was without the key. For authentication, it computes a tag which proves the data belongs to a person. In this case, it should be infeasible to generate a new tag that verifies successfully, without knowing the key. An example of an authenticated encryption scheme is Elephant [6].

2.3.2 Permutations

A cryptographic permutation takes input data, and maps it to a seemingly unrelated output. Ideally, a permutation should have first pre-image resistance, second pre-image resistance and collision resistance.
A permutation h has first pre-image resistance if it is computationally infeasible to compute a hash y for a given x such that $h(x) = y$.
A permutation h has second pre-image resistance if it is computationally infeasible to compute a value x' for a given x such that $h(x) = h(x')$, and $x \neq x'$.
A permutation h has collision resistance if it is computationally infeasible to find an x and y such that $x \neq y$, but $h(x) = h(y)$.

2.4 Keccak

Keccak is a cryptographic function. The corresponding permutation Keccak-f[200][17] is the main focus of this thesis. Below, we provide its pseudocode and list some areas where it is used in practice.

2.4.1 Pseudocode

As listed on the Keccak site[10], the pseudocode of the Keccak function is:

```
Keccak-f[b](A) {
  for i in 0...n-1
    A = Round[b](A, RC[i])
  return A
}

Round[b](A,RC) {
  # sigma step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   for x in 0...4
  D[x] = C[x-1] xor rot(C[x+1],1),                             for x in 0...4
  A[x,y] = A[x,y] xor D[x],                           for (x,y) in (0...4,0...4)

  # rho and pi steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                 for (x,y) in (0...4,0...4)

  # theta step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),  for (x,y) in (0...4,0...4)

  # iota step
  A[0,0] = A[0,0] xor RC
  return A
}
```

The way in which Keccak operates can be visualized as a three-dimensional cube. The ‘steps’ referred to in the pseudocode (which will be referred to as the diffusion functions in the rest of this thesis) break correlation in this cube by shifting the data in lanes, columns and rows, as well as adding in certain constants at the end of each round.

2.4.2 Uses

Instances of Keccak-f are used in numerous primtives and protocols. Some examples include:

- Sha-3[1] *is* an instance of Keccak-f, namely Keccak-f[1600]. The Sha family is perhaps the most well known family of hash functions.
- Isap[2] is a lightweight authenticated encryption scheme that uses Keccak-f internally. It is one of the finalists in an ongoing competition for lightweight cryptography at NIST.
- Elephant[6] is another lightweight authenticated encryption scheme that uses Keccak-f internally. Also a finalist at the aforementioned NIST competition, it features support for parallel encryption.

Chapter 3

Definition of `While`$^+$

In this chapter, we argument our choice for the extensions necessary to `While`, give the extensions, and finally provide the derivation rules for the `While`$^+$ language that is thus created.

3.1 `While`$^+$

To reason about encryption schemes in `While`, we must extend it with several concepts:

1. Bytes. When boiled down, encryption is some operation on bytes (or bits, the immediate constituents of bytes), so without the concept of bytes, proving correctness for encryption schemes is impossible.

2. Arrays. Without arrays, we would have to assign each subsequent byte to a new variable. Theoretically, this is possible; practically, it is not.

3. Functions. If we count the total source lines of code (or SLOC), then excluding any comments or white space, we get to 215 lines of code. It is very likely that the transformation from C to `While` would increase this number, as `While` is more simplistic. Without functions, however, the amount of code to apply the derivation rules to would become far too large to reason about. This is because without functions, we would have to expand each and every function definition to its body. Not only would this cost a lot of time, it would also be incredibly wasteful; for each expanded function, we would be re-proving something we had already proven before. This is why we also extend `While` with functions.

4. The modulus statement. In encryption, one often wants to work with wraparounds. That is to say, if an index i happens to fall outside of an array with length l, you really want to get the item at index $i \bmod l$. We can simulate a modulo statement with a while loop (repeatedly

subtracting l from i until i is less than l), but this is rather inefficient; in the variant of keccak-f alone, we count 21 modulus statements.

In the following sections, we give the definition of this extended `While`$^+$.

However, before we define our extended `While`$^+$, we must first explain our notation for defining functions. We define a function as having a signature, that being the name, the type of the arguments that are supplied to it and an *optional* return type. We have chosen to include both the name and the type of the arguments, because this precludes confusion as to the actual type of the argument, given only its name. Below, we give two examples of function signatures, to show what they look like. Note that the first features a return type, the second does not:

$$func_1(a', b') : A \times B \to C,$$
$$func_2(a', b') : A \times B.$$

To know what a function does, we look at its body. The body is separated from the signature with an equals sign. To avoid confusion as to which equals sign means what, we have chosen to use the $\leftarrow$ symbol to indicate that a function returns some value. We again provide two examples; one function with a return value, and one without. Note that the functions 'foo' and 'bar' are just illustrative examples; they have no special meaning.

$$\begin{aligned} &func_1(a', b') : A \times B \to C = \\ &\qquad foo(a') \\ &\qquad \leftarrow bar(b') \end{aligned}$$

$$\begin{aligned} &func_2(a', b') : A \times B = \\ &\qquad foo(a') \\ &\qquad bar(b') \end{aligned}$$

We *only* use this notation for functions we define as short-hands for semantics. For functions which evaluate syntax, like $\mathcal{BY}$, $\mathcal{ARR}$ or $\mathcal{A}$, we adopt the notation used in [20]. This is because they are always defined in terms of their arguments, that is to say, for all possible values of the input, the function is defined to give an output. An example of this is:

$$\begin{aligned} &func_3(a') : A \to B, \\ &func_3(a'_0) \;=\; b'_0, \\ &func_3(a'_1) \;=\; b'_1, \\ &func_3(a'_2) \;=\; b'_2. \end{aligned}$$

Where A and B are defined as:

$$A := \{a_0, a_1, a_2\},$$
$$B := \{b_0, b_1, b_2\}.$$

3.1.1 State

In `While`, a state is a mapping from a variable to a real integer. In `While`$^+$, a variable can be more than just an integer, so we need a more advanced state. We define three additional state functions:

$$\begin{aligned} \text{State}_{bi} &: \text{BITS} \to \mathbb{Z}_2, \\ \text{State}_{by} &: \text{BYTES} \to \mathbb{Z}_2^8, \\ \text{State}_{arr} &: \text{ARRAYS} \to \{0,1\}^{\mathbb{N}}. \end{aligned}$$

We also rename the 'normal' State function of `While`. Otherwise, it would be easy to confuse it with the other states.

$$\text{State}_x : \text{VAR} \to \mathbb{Z}$$

Using these definitions, we now define the state,

$$State = (\text{State}_{bi}, \text{State}_{by}, \text{State}_{arr}, \text{State}_x)$$

When we use the notation s or $State$ in the rest of this thesis, that is a shorthand for the appropriate state function, for the context it is being used in.
For example, $s\ bi$ is shorthand for $s_{bi}\ bi$.

3.1.2 Bits

We start by defining bits:

$$bi \quad ::= \underline{0} \mid \underline{1} \mid bi \backslash\!/ bi \mid bi /\!\backslash bi \mid bi \ \hat{}\ bi \mid \ \sim bi \mid x_{bi}$$

Where bi ranges over all bit expressions BIExpr. Then, we define the meta-variable bi to range over all bits.

Note the underline; it represents a visual distinction between real integers ($\mathbf{0}$), numerals (0) and bit expressions. We use this notation only in syntax.

Bits exist on the syntactical level (in the form of bit expressions), but also on the semantical level. There, bits are represented as values in the equivalence class of $\mathbb{Z}_2$.
We define all commonly used operators on bits; OR, AND, XOR and NEG.

All of these operate on the semantical level.

$$\begin{aligned}
\vee(x,y) &: (\mathbb{Z}_2 \times \mathbb{Z}_2) \to \mathbb{Z}_2 = \\
&\leftarrow \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0, \\ 1 & \text{otherwise.} \end{cases} \\
\wedge(x,y) &: (\mathbb{Z}_2 \times \mathbb{Z}_2) \to \mathbb{Z}_2 = \\
&\leftarrow x \cdot y \\
\oplus(x,y) &: (\mathbb{Z}_2 \times \mathbb{Z}_2) \to \mathbb{Z}_2 = \\
&\leftarrow x + y \\
\neg(x,y) &: (\mathbb{Z}_2 \times \mathbb{Z}_2) \to \mathbb{Z}_2 = \\
&\leftarrow x + 1
\end{aligned}$$

We define the $\mathcal{BI}$ function, which allows us to evaluate all bit expressions.

$$\begin{aligned}
\mathcal{BI} &: \text{BIExpr} \to (\text{State} \to \mathbb{Z}_2) \\
\mathcal{BI}[\![\underline{0}]\!]s &= 0 \\
\mathcal{BI}[\![\underline{1}]\!]s &= 1 \\
\mathcal{BI}[\![x_{bi}]\!]s &= s_{bi}\ x_{bi} \\
\mathcal{BI}[\![bi \backslash/ bi']\!]s &= \vee(\mathcal{BI}[\![bi]\!]s, \mathcal{BI}[\![bi']\!]s) \\
\mathcal{BI}[\![bi /\backslash bi']\!]s &= \wedge(\mathcal{BI}[\![bi]\!]s, \mathcal{BI}[\![bi']\!]s) \\
\mathcal{BI}[\![bi\ \hat{}\ bi']\!]s &= \oplus(\mathcal{BI}[\![bi]\!]s, \mathcal{BI}[\![bi']\!]s) \\
\mathcal{BI}[\![\sim bi]\!]s &= \neg(\mathcal{BI}[\![bi]\!]s)
\end{aligned}$$

3.1.3 Bytes

Using the definition of a bit, we can define byte expressions:

$$\begin{aligned}
by\ \ ::=\ \ &(bi_1, bi_2, bi_3, bi_4, bi_5, bi_6, bi_7, bi_8) \mid by \backslash/ by \mid by /\backslash by \\
&\mid by\ \hat{}\ by \mid \sim by \mid by \ll a \mid by \gg a \mid x_{by}
\end{aligned}$$

As well as the metavariable *by*, which ranges over all byte expressions BYExpr.

A byte expression is either an octuple of bits (which we then may call a byte), two bytes separated by an AND, OR or XOR symbol, a single byte preceded by the negation symbol, a byte separated from an arithmetic expression by the left/right shift symbol or a variable containing a byte value.

We will now define functions that describe the semantics of all byte operators. Because they operate at the semantical level, they do not take bytes as arguments, but octuples of $\mathbb{Z}_2$. This, however, leads to rather lengthy definitions. Consider the byte which has a value of $\underline{01001011}$. By converting it to its decimal representation, we save a lot of space (75), and by then overlining it, we make it visually distinct from other numbers ($\overline{75}$). Because there is a bijection between the octuple of bits and $\mathbb{Z}_{256}$, we will use the two interchangeably in the rest of this thesis. This might sometimes look strange; we can define a variable x to be of type $\mathbb{Z}_{256}$ and then proceed to reason about the value of x_3 (the third bit from the left, in the byte x). In this case, we use the bijection between $\mathbb{Z}_{256}$ and the octuple of bits to implicitly convert x.

1. The or:
$$\begin{aligned}\vee(x,y) : \mathbb{Z}_{256} \times \mathbb{Z}_{256} \to \mathbb{Z}_{256} = \\ & z = \overline{0} \\ & \forall i \in [1,8] : z_i = \vee(x_i, y_i) \\ & \leftarrow z\end{aligned}$$
In natural language: the *or* of two bytes is calculated by applying the or function to each bit of the two bytes.

2. The and:
$$\begin{aligned}\wedge(x,y) : \mathbb{Z}_{256} \times \mathbb{Z}_{256} \to \mathbb{Z}_{256} = \\ & z = \overline{0} \\ & \forall i \in [1,8] : z_i = \wedge(x_i, y_i) \\ & \leftarrow z\end{aligned}$$
In natural language: the *and* of two bytes can be calculated by applying the and function to each bit of the two bytes.

3. The xor:
$$\begin{aligned}\oplus(x,y) : \mathbb{Z}_{256} \times \mathbb{Z}_{256} \to \mathbb{Z}_{256} = \\ & z = \overline{0} \\ & \forall i \in [1,8] : z_i = \oplus(x_i, y_i) \\ & \leftarrow z\end{aligned}$$
In natural language: the *xor* of two bytes can be calculated by applying the xor function to each bit of the two bytes.

4. The negation:
$$\begin{aligned}\neg(x) : \mathbb{Z}_{256} \to \mathbb{Z}_{256} = \\ & y = \overline{0} \\ & \forall i \in [1,8] : y_i = \neg(x_i) \\ & \leftarrow y\end{aligned}$$
In natural language: the *negation* of a byte is the negation of each of its individual bits.

5. The left shift:

$$\begin{aligned}\ll (x, \mathit{shift}) : \mathbb{Z}_{256} \times \mathbb{Z}_8 \to \mathbb{Z}_{256} = \\ & \mathit{res} = \overline{0} \\ & \mathit{pivot} = 8 - \mathit{shift} \\ & \forall i \in [1, \mathit{pivot}] : \mathit{res}_i = x_{i+\mathit{pivot}} \\ & \forall i \in [\mathit{pivot}, 8] : \mathit{res}_i = 0 \\ & \leftarrow \mathit{res}\end{aligned}$$

In natural language: *shifting* a byte z bits to the left discards the z most significant bits, and introduces z zeroes at the right side.

6. The right shift:

$$\begin{aligned}\gg (x, \mathit{shift}) : \mathbb{Z}_{256} \times \mathbb{Z}_8 \to \mathbb{Z}_{256} = \\ & \mathit{res} = \overline{0} \\ & \mathit{pivot} = 8 - \mathit{shift} \\ & \forall i \in [\mathit{pivot}, 8] : \mathit{res}_i = x_{i-\mathit{pivot}} \\ & \forall i \in [1, \mathit{pivot}] : \mathit{res}_i = 0 \\ & \leftarrow \mathit{res}\end{aligned}$$

In natural language: shifting a byte z bits to the right discards the z least significant bits, and introduces z zeroes at the left side.

We define the function $\mathcal{BY}$. This function, given a byte expression, evaluates the semantics of the expression. All the previously defined functions for operators and bytes are used here.

$$\begin{aligned}\mathcal{BY} &: \text{BYExpr} \to (\text{State} \to \mathbb{Z}_{256}) \\ \mathcal{BY}[\![by]\!]s &= (\mathcal{BI}[\![by_1]\!]s, \mathcal{BI}[\![by_2]\!]s, \mathcal{BI}[\![by_3]\!]s, \mathcal{BI}[\![by_4]\!]s, \\ &\qquad \mathcal{BI}[\![by_5]\!]s, \mathcal{BI}[\![by_6]\!]s, \mathcal{BI}[\![by_7]\!]s, \mathcal{BI}[\![by_8]\!]s) \\ \mathcal{BY}[\![by \backslash / by']\!]s &= \vee(\mathcal{BY}[\![by]\!]s, \mathcal{BY}[\![by']\!]s) \\ \mathcal{BY}[\![by / \backslash by']\!]s &= \wedge(\mathcal{BY}[\![by]\!]s, \mathcal{BY}[\![by']\!]s) \\ \mathcal{BY}[\![by \;\hat{}\; by']\!]s &= \wedge(\mathcal{BY}[\![by]\!]s, \mathcal{BY}[\![by']\!]s) \\ \mathcal{BY}[\![\sim by]\!]s &= \neg(\mathcal{BY}[\![by]\!]s) \\ \mathcal{BY}[\![by << a]\!]s &= \ll (\mathcal{BY}[\![by]\!]s, \mathcal{A}[\![a]\!]s) \\ \mathcal{BY}[\![by >> a]\!]s &= \gg (\mathcal{BY}[\![by]\!]s, \mathcal{A}[\![a]\!]s)\end{aligned}$$

3.1.4 Arrays

Using the definition of a byte, we can now define arrays as

$$\mathit{arr} \quad ::= (by', \ldots, by'^{\cdots\prime}) \mid \mathit{arr}[a] \mid \mathit{arr}[a] = by \mid x_{\mathit{arr}}$$

where arr ranges over all array expressions ArrExpr.
Note that the notation of $by'^{\cdots'}$ indicates that an arbitrary amount of bytes may be present within the array. We define an array as either an n-tuple of bytes, an accessor statement (returning the *value* of the item at index x), or an assignment statement (assigning a new value on index x). Finally, note that arrays are defined to only hold bytes, and not bits or numbers.

Representation of arrays

To formally reason about arrays, we must provide them with a proper type. A tuple of arbitrary length, while nice to reason about, is not quite well defined, so we use another approach. The infinite sequence $\{0,1\}^{\mathbb{N}}$ contains only ones and zeroes. Each element of an array will be translated into eight zeroes or ones (representing the value of the element), and one trailing zero or one. This last value indicates whether it is the final element in the array. Let us consider an example array:

$$a := (\overline{34}, \overline{55}, \overline{77})$$

In this case, we translate $\overline{34}$ to 001000100, where the first eight digits are the binary representation of 34, and the final zero indicates it is not the last value in the array.
Then, $\overline{55}$ becomes 001101110. Finally, $\overline{77}$ becomes 010011011, where the final one indicates there are no other values after it.
Thus, we can represent the array a as 001000100001101110010011011 . . . .
The dots represent arbitrary values, because the final nonuplet has a 1 as its last value, so we know the array 'ends' there.
Of course, reasoning about long strings of binary for each array is cumbersome, so we make use of the surjection between $\{0,1\}^{\mathbb{N}}$ and n-tuples of bytes.

We define functions which operate on the semantical level, and describe the getting and setting of array elements:

$$\begin{gathered} get(arr', n') : \{0,1\}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{Z}_2^8 = \\ \leftarrow arr'_{n'} \\ set(arr', n', by') : \{0,1\}^{\mathbb{N}} \times \mathbb{N} \times \mathbb{Z}_2^8 = \\ arr'_{n'} = by' \end{gathered}$$

The way we currently write arrays is not very compact, especially when writing down a new array (which has all of its elements set to $\overline{0}$), we have to write it as $arr' = (\overline{0}, \overline{0}, \ldots, \overline{0}, \overline{0})$. Because we define new arrays to always be created with each of its elements set to $\overline{0}$, we can shorten the creation of new arrays to just $x = arr\{n\}$, where n is the size of the array. To avoid ambiguity as to the accessor statement or this new initialization statement, we use curly braces.

Now, we define the *arrlength* function. It maps from an array variable to the length of the array.

$$arrlength(arr') : (\mathbb{Z}_{256} \times \ldots \times \mathbb{Z}_{256}) \to \mathbb{N}$$

We only give the signature. For every array that is created, we define an additional case to the function. An example of an array with a content of (by', by'', by''') would be:

$$arrlength((by', by'', by''')) = 3$$

This allows us to enforce array bounds checking.

Finally, we give the $\mathcal{ARR}$ function, which defines the semantics of an array:

$$\begin{aligned}
\mathcal{ARR} &: \text{ArrExpr} \to (\text{State} \to \{0,1\}^{\mathbb{N}}) \\
\mathcal{ARR}[\![(by', \ldots, by'^{\cdots\prime})]\!]s &= (\mathcal{BY}[\![by']\!]s, \ldots, \mathcal{BY}[\![by'^{\cdots\prime}]\!]s) \\
\mathcal{ARR}[\![arr[a]]\!]s &= get(\mathcal{ARR}[\![arr]\!]s, \mathcal{A}[\![a]\!]s) \\
\mathcal{ARR}[\![arr[a] = by]\!]s &= set(\mathcal{ARR}[\![arr]\!]s, \mathcal{A}[\![a]\!]s, \mathcal{BY}[\![by]\!]s)
\end{aligned}$$

3.1.5 Modulus

We extend the definition of arithmetic expressions to include the modulus:

$$a \ ::= \ n \,|\, x \,|\, a_1 + a_2 \,|\, a_1 * a_2 \,|\, a_1 - a_2 \,|\, a_1 \,\%\, a_2 \,.$$

Then, we define a function which describes the modulus operation on the semantical level:

$$\begin{aligned}
mod(x, y) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} = \\
&quotient \leftarrow \left\lceil \frac{x}{y} \right\rceil \\
&rem \leftarrow x - (quotient \cdot y) \\
&\leftarrow rem,
\end{aligned}$$

and finally, we provide the extension to the $\mathcal{A}$ function

$$\mathcal{A}[\![a' \,\%\, a'']\!]s = mod(\mathcal{A}[\![a']\!]s, \mathcal{A}[\![a'']\!]s).$$

3.1.6 Boolean evaluation function

The boolean evaluation function $\mathcal{B}$ is defined in `While` for numbers. Because `While`$^+$ introduces bits and bytes, we need to expand $\mathcal{B}$ to deal with these concepts as well.

$$\begin{aligned}
\mathcal{B}[\![bi < bi']\!]s &= s_{bi}\ bi < s_{bi}\ bi' \\
\mathcal{B}[\![bi > bi']\!]s &= s_{bi}\ bi > s_{bi}\ bi' \\
\mathcal{B}[\![bi == bi']\!]s &= s_{bi}\ bi = s_{bi}\ bi' \\
\mathcal{B}[\![bi >= bi']\!]s &= s_{bi}\ bi = s_{bi}\ bi' \vee s_{bi}\ bi > s_{bi}\ bi' \\
\mathcal{B}[\![bi <= bi']\!]s &= s_{bi}\ bi = s_{bi}\ bi' \vee s_{bi}\ bi < s_{bi}\ bi' \\
\mathcal{B}[\![by < by']\!]s &= s_{by}\ by < s_{by}\ by' \\
\mathcal{B}[\![by > by']\!]s &= s_{by}\ by > s_{by}\ by' \\
\mathcal{B}[\![by == by']\!]s &= s_{by}\ by = s_{by}\ by' \\
\mathcal{B}[\![by >= by']\!]s &= s_{by}\ by = s_{by}\ by' \vee s_{by}\ by > s_{by}\ by' \\
\mathcal{B}[\![by <= by']\!]s &= s_{by}\ by = s_{by}\ by' \vee s_{by}\ by < s_{by}\ by'
\end{aligned}$$

3.1.7 Functions

Most programming languages make use of functions. In this context, functions are a fixed sequence of operations, which are carried out on a number of parameters, and which may (or may not) return some result.
We provide a definition for functions in `While`$^+$ for the simple reason that if we did not, we would have to expand each function definition to its function body.

We define a function as a name, followed by a certain number of arguments (between brackets), and finally a number of statements, which are applied upon the arguments:

$$funcname(arg)(S)$$

Where we define 'arg' as:

$$arg \quad ::= x_{var} = var \mid x_{bi} = bi \mid x_{by} = by \mid x_{arr} = arr \mid arg, arg$$

With this definition, we state that an argument is some variable of a certain type, where the value is unknown. Functions do *not* have a return type. They also consider arguments to be passed by reference. This combination means that we can simulate a non-void function by adding a 'ret' variable, to which the return value is assigned. We provide the following example:

```
foo(x = var) (x := x + 3)

bar() (x := 7; foo(x))
```

Once the execution of the function 'bar' has terminated, the value of 'x' will be 10. First, it is assigned the value of 7, then, when foo is called, 3 is

added to this. When we define the derivation rules in section 3.2, we will elaborate on this.
We also define a function call as

$$funccall := funcname(funcargs),$$

where we define 'funcargs' as

$$funcargs := var \mid var,\ funcargs.$$

3.1.8 Statements

In `While`, we have the following statements:

$$S \ ::= \ x{:=}a \mid \ skip \mid \ S_1 ; S_2 \mid \quad if\, b \ \ then\, S_1 \ \ else\, S_2 \ \mid \\ while\, b \ \ do\, S$$

Following the definitions in Subsections 3.1.2, 3.1.3, 3.1.4 3.1.5 and 3.1.7 we extend this to include all our new constructs.

$$S \ ::= \ x{:=}a \mid skip \mid S_1 ; S_2 \mid \quad if\, b \ \ then\, S_1 \ \ else\, S_2 \ \mid \\ while\, b \ \ do\, S \ \mid x_{arr} \ := \ arr \mid x_{arr}\, [a]{=}by \mid \ x_{bi} \ := \ bi \mid \\ x_{by} \ := \ by \mid \ funccall(funcargs)$$

3.1.9 Substitutions of variables

In `While`, there is a notion of *state*. A state is some mapping from variables to values, and so to update the state, we must know how to update each new type we have introduced. Specifically, for each type, we are interested in how to replace an exsiting variable y with a new expression.

Substitution of bits

$$\begin{aligned}
n[y \mapsto bi'] &= n \\
a[y \mapsto bi'] &= a \\
x[y \mapsto bi'] &= x \\
bi[y \mapsto bi'] &= \begin{cases} bi' \text{ if } bi = y \\ bi \text{ if } bi \neq y \end{cases} \\
by[y \mapsto bi'] &= (bi_1[y \mapsto bi'], bi_2[y \mapsto bi'], bi_3[y \mapsto bi'], bi_4[y \mapsto bi'], \\
&\quad bi_5[y \mapsto bi'], bi_6[y \mapsto bi'], bi_7[y \mapsto bi'], bi_8[y \mapsto bi']) \\
arr[y \mapsto bi'] &= (by'[y \mapsto bi'], \ldots, by'^{\cdots'}[y \mapsto bi'])
\end{aligned}$$

This should make sense; if we are substituting some bit variable y for the bit expression bi', there is no change if the expression we are substituting on is a numeral or arithmetic expression. If it is a byte or array, we forward the substitution onto their components. If it is a bit, we change the value if it is the bit that is being substituted.

$$\begin{aligned}
(bi' \vee bi'')[y \mapsto bi] &= (bi'[y \mapsto bi] \vee bi''[y \mapsto bi]) \\
(bi' \wedge bi'')[y \mapsto bi] &= (bi'[y \mapsto bi] \wedge bi''[y \mapsto bi]) \\
(bi' \oplus bi'')[y \mapsto bi] &= (bi'[y \mapsto bi] \oplus bi''[y \mapsto bi]) \\
(\neg bi')[y \mapsto bi] &= (\neg bi'[y \mapsto bi])
\end{aligned}$$

Above, we define how any bit expression handles its substitution. Similarly to the definitions for the other types, we send the substitution on to each byte expression that exists within the byte expression. This recursive definition allows for an arbitrary amount of nested operators.

Substitution of bytes

$$\begin{aligned}
n[y \mapsto by'] &= n \\
a[y \mapsto by'] &= a \\
x[y \mapsto by'] &= x \\
bi[y \mapsto by'] &= bi \\
by[y \mapsto by'] &= \begin{cases} by' & \text{if } by' = y \\ by & \text{if } by' \neq y \end{cases} \\
arr[y \mapsto by'] &= (by'[y \mapsto by'], \ldots, by'^{\cdots'}[y \mapsto by'])
\end{aligned}$$

As with the bits, substituting a byte variable for another byte expression on any type that is *not* a byte doesn't result in any change. For the array, we send the substitution on to each of its constituent bytes.

$$\begin{aligned}
(by'' \vee by''')[y \mapsto by']) &= (by''[y \mapsto by'] \vee by'''[y \mapsto by']) \\
(by'' \wedge by''')[y \mapsto by']) &= (by''[y \mapsto by'] \wedge by'''[y \mapsto by']) \\
(by'' \oplus by''')[y \mapsto by']) &= (by''[y \mapsto by'] \oplus by'''[y \mapsto by']) \\
(\neg by'')[y \mapsto by'] &= \neg(by''[y \mapsto by']) \\
(by'' \gg a_0)[y \mapsto by'] &= (by''[y \mapsto by'] \gg a_0[y \mapsto by']) \\
(by'' \ll a_0)[y \mapsto by'] &= (by''[y \mapsto by'] \ll a_0[y \mapsto by'])
\end{aligned}$$

Here we again see the recursive definition of substitution on expressions, this time for bytes. Two interesting cases here are the last two, as they involve arithmetic expressions. Therefore, we must also define what happens if we substitute an arithmetic expression on those byte expressions:

$$\begin{aligned}
(by'' \gg a_0)[y \mapsto a_1] &= (by''[y \mapsto a_1] \gg a_0[y \mapsto a_1]) \\
(by'' \ll a_0)[y \mapsto a_1] &= (by''[y \mapsto a_1] \gg a_0[y \mapsto a_1])
\end{aligned}$$

Which, similarly to the other substitutions, is achieved by forwarding it onto its immediate constituents.

Substitution of arrays

As before, we define how an array variable is substituted for a new array expression. There are no special cases here, because no other type contains an array.

$$\begin{aligned}
n[y \mapsto arr'] &= n \\
a[y \mapsto arr'] &= a \\
x[y \mapsto arr'] &= x \\
bi[y \mapsto arr'] &= bi \\
by[y \mapsto arr'] &= by \\
arr[y \mapsto arr'] &= \begin{cases} arr' \text{ if } arr = y \\ arr \text{ if } arr \neq y \end{cases}
\end{aligned}$$

Then, we also show this for array expressions:

$$\begin{aligned}
(by', \ldots, by'^{\cdots\prime})[y \mapsto arr'] &= (by'[y \mapsto arr'], \ldots, by'^{\cdots\prime}[y \mapsto arr']) \\
(arr[a])[y \mapsto arr'] &= ((arr)[y \mapsto arr'])[a[y \mapsto arr']] \\
(arr[a] = by)[y \mapsto arr'] &= ((arr)[y \mapsto arr'])[a[y \mapsto arr']] = by[y \mapsto arr']
\end{aligned}$$

And finally, we see cases where arithmetic expressions and byte expressions are present within array expressions. Thus, we also define how substitution of those types happen on array expressions:

$$\begin{aligned}
(arr[a])[y \mapsto a_0] &= ((arr)[y \mapsto a_0])[a[y \mapsto a_0]] \\
(arr[a] = by)[y \mapsto a_0] &= ((arr)[y \mapsto a_0])[a[y \mapsto a_0]] = by[y \mapsto a_0] \\
(arr[a])[y \mapsto by'] &= ((arr)[y \mapsto by'])[a[y \mapsto by']] \\
(arr[a] = by)[y \mapsto by'] &= ((arr)[y \mapsto by')[a[y \mapsto by']] = by[y \mapsto by']
\end{aligned}$$

Which, similarly to the other substitutions, is achieved by forwarding it onto its immediate constituents.

3.2 Derivation rules

Let us look at the following toy example:

```
a(x = var)(b(x));
b(x = var)(c(x));
c(x = var)(x = x + incr);
incr = 3
```

If we look at the code, it seems intuitively clear what each function does; function `a` calls function `b`, function `b` calls function `c`, and function `c` increments the variable passed to it with `incr`, which is defined to be 3.
Derivation rules are the way in which we can formally parse `While`$^+$ code, to obtain the semantics of the given code. In `While`, as defined in the book [20], there is one set of derivation rules, because there are no functions or global constants. In `While`$^+$, however, those do exist. The 'intuitive' non-formal approach assumed that:

1. Functions need not be defined in order. That is to say, we can define function `a` to call function `b`, even though function `b` has not been defined before function `a`.

2. The variables assigned outside of a function are considered constants, and cannot be changed (if they could be, we would need to somehow include them in our state as well).

To formalize those assumptions, we use two sets of derivation rules. The first set identifies what functions exist, what arguments they have and what functions they call in turn. This information is then used to create a derivation order and in doing so, we avoid the problem where one might start by deriving a function that calls a function that has not been derived yet (going back to the toy example above, starting by deriving `a` would fail, because `b` is not derived yet). It also reads in all global constants to create a constant state (which is identical to a normal state, except that we cannot change the values stored within). The second set is then applied on the results of the first set, to obtain the semantics of each function.

3.2.1 First pass derivation rules

We define the first set of derivation rules, show their use on the toy example, and explain how to apply the results to obtain a working derivation order.
To store the information we obtain from these derivation rules, we define an 'environment':

$$\begin{aligned}\text{environment} := (&\text{functions}, \text{function args}, \text{function bodies},\\ &\text{function calls}, \text{current function})\end{aligned}$$

We shall briefly explain what each element is, and why we need it in the environment.
The first element, 'functions', is a set of all the function names that have been encountered so far in the derivation process. We use this set in the process of finding a derivation order; if the current derivation order has as many elements as the set has, it is finished.
The second element, 'function args', is a function that maps from a function

name to a set of pairs. Each pair represents a single argument (where the first element of the pair is the name, and the second the type). We use these in the second pass, to initialize the state with all arguments at the start of a function.
Similarly, 'function bodies' is a function that maps from a function name to the corresponding body of the function. We also use these in the second pass, to obtain the function body we want to derive.
The second-to-last element, 'function calls', is a function that maps from a function name to the set of function names that are called within the function it maps from. We use this in the process of obtaining a derivation order; at any given step, a function can be derived if its mapping in 'function calls' is the empty set.
The last element, 'current function', stores the name of the function that the derivation process currently is in. It is used to distinguish local variables within a function from global variables.

Before we give the derivation rules themselves, we first provide a number of functions to alter the environment with. Similarly to the way functions are defined on Section 3.1.2, a function that does more than one thing has every statement on a new line. Furthermore, we refer to elements in the environment tuple (called 'env', here) by their index, for conciseness.

$$\begin{aligned}
\text{updfunc}(\text{env}, \text{funcname}) := {} & \text{env}_1 = \text{env}_1 \cup \{\text{funcname}\} \\
& \text{env}_5 = \text{funcname} \\
& \leftarrow \text{env} \\
\text{updarg}(\text{env}, \text{name}, \text{type}) := {} & \text{env}_2[\text{env}_5] = \text{env}_2[\text{env}_5] \cup \{(\text{name}, \text{type})\} \\
& \leftarrow \text{env} \\
\text{endfunc}(\text{env}) := {} & \text{env}_5 = \text{""} \\
& \leftarrow \text{env} \\
\text{funccall}(\text{env}, \text{func}, \text{args}) := {} & \text{env}_4[\text{env}_5] = \text{env}_4[\text{env}_5] \cup \text{func} \\
& \leftarrow \text{env} \\
\text{updbody}(\text{env}, \text{statement}) := {} & \text{env}_3[\text{env}_5] = \text{env}_3[\text{env}_5] + \text{statement} \\
& \leftarrow \text{env}
\end{aligned}$$

We briefly explain what each function does. The first function, 'updfunc', updates the environment to reflect that the derivation rules have encountered a new function declaration, and the statements inside should be read in as belonging to said function. Next, 'updarg', updates the environment to indicate that the current function has an argument, consisting of a type and a name. Thirdly, 'endfunc' ends the current function, by setting the last element in the environment tuple to nothing. Penultimately , 'funccall' indicates that the current function has a statement that *is* a function call. This is separate, because we want to store the function calls made per function, but also because we need to know the arguments that are passed within a function call (i.e. a function call statement of the form `func(args)` is needed for the function body mapping, but only the `func` part for the function call mapping). Lastly, ‘updbody’ indicates that the current function has an additional statement belonging to it, that is *not* a function call to another function. Finally, note that each function returns the environment it alters. This is so that we can chain these calls together (for instance, to properly record a function call, we need to call both ‘updarg’ and ‘funccall’).

Having defined the functions to manipulate the environment, we give the first set of derivation rules. We give these in four parts; rules that deal with composition, rules that deal with function arguments, rules that deal with function bodies and finally, rules that deal with global variables.

$$\textbf{1}\text{-}\mathrm{COMP}^{1}_{\mathrm{NS}}\qquad \frac{\mathrm{env} \vdash \langle \mathrm{S}_1, s\rangle \longrightarrow s' \qquad \mathrm{env} \vdash \langle \mathrm{S}_2, s'\rangle \longrightarrow s''}{\mathrm{env} \vdash \langle \mathrm{S}_1; \mathrm{S}_2, s\rangle \longrightarrow s''}\ \text{if } \mathrm{env}_5 = \text{‘’}$$

$$\textbf{1}\text{-}\mathrm{COMP}^{2}_{\mathrm{NS}}\qquad \frac{\begin{array}{c}\mathrm{updfunc}(\mathrm{env}, \mathrm{func}) \vdash \langle \mathrm{func}, s\rangle \longrightarrow s \qquad \mathrm{env}' \vdash \langle \mathrm{args}, s\rangle \longrightarrow s\\ \mathrm{env}' \vdash \langle \mathrm{S}_1, s\rangle \longrightarrow s \qquad \mathrm{endfunc}(\mathrm{env}') \vdash s \longrightarrow s \qquad \mathrm{env}'' \vdash \langle \mathrm{S}_2, s\rangle \longrightarrow s'\end{array}}{\mathrm{env} \vdash \langle \mathrm{func}(\mathrm{args})(\mathrm{S}_1); \mathrm{S}_2, s\rangle \longrightarrow s'}\ \text{if } \mathrm{env}_5 = \text{‘’}$$

$$\textbf{1}\text{-}\mathrm{COMP}^{3}_{\mathrm{NS}}\qquad \frac{\mathrm{env} \vdash \langle \mathrm{S}_1, s\rangle \longrightarrow s \qquad \mathrm{updbody}(\mathrm{env}, ;) \vdash s \longrightarrow s \qquad \mathrm{env}'''\langle \mathrm{S}_2, s\rangle \longrightarrow s}{\mathrm{env} \vdash \langle \mathrm{S}_1; \mathrm{S}_2, s\rangle \longrightarrow s}\ \text{if } \mathrm{env}_5 \neq \text{‘’}$$

Table 1: First pass derivation rules for composition.

Additionally, we define:

$$\begin{aligned}\mathrm{env}' &= \mathrm{updfunc}(\mathrm{env}, \mathrm{func}),\\ \mathrm{env}'' &= \mathrm{endfunc}(\mathrm{env'}),\\ \mathrm{env}''' &= \mathrm{upbody}(\mathrm{env},\ ;).\end{aligned}$$

The rules in table 1 deal with the composition of statements. In particular, the first rule deals with the case where the first statement is *not* a function declaration, and the statements are not part of a function. The second rule deals with the case that the first statement *is* a function declaration. Then,

we must create a number of branches to handle updating the environment accordingly. To be specific; we need a branch for the function name, its arguments, the statement it contains, a seemingly empty branch to ‘close’ or ‘end’ the function, and a last branch for the statement that comes after the function declaration. These need to occur in specifically that order, because the first branch updates the environment to indicate that we are inside of a function, and the next-to-last branch updates the environment to indicate that we are no longer inside of a function. If the order were to be changed (by for instance swapping the last two branches), one might change which statements are considered to be part of a function, and which not.
The last rule is applicable only inside of a function declaration. The main difference between it and the first rule is that here, we must make a branch to add the character ‘;’ to the appropriate function body mapping, in between the branches of the first and second statement.

$$\textbf{1}\text{-ARG-COMP}_{ns}\quad \frac{\text{updarg}(\text{env}, \text{var}, \text{type}) \vdash s \longrightarrow s \qquad \text{env}' \vdash \langle \text{arg}, s\rangle \longrightarrow s}{\text{env} \vdash \langle \text{var} = \text{type}, s\rangle \longrightarrow s}$$

$$\textbf{1}\text{-ARG}_{ns}\quad \text{updarg}(\text{env}, \text{var}, \text{type}) \vdash \langle \text{var} = \text{type}, s\rangle \longrightarrow s$$

Table 2: First pass derivation rules for function arguments.

Additionally, we define

$$\text{env}' = \text{updarg}(\text{env}, \text{var}, \text{type}).$$

The rules in table 2 deal with function arguments. Because they are defined recursively, we need to handle two cases; where the argument consists of two arguments, and where the argument consists of only itself. With each of these, we simply update the environment through the appropriate function.

$$\textbf{1-}\textsc{FuncCall}_{\textsc{ns}}\quad \text{updbody}(\text{funccall}(\text{env},\text{func}),\text{`func(args)'})\vdash\langle\text{func(args)},s\rangle\longrightarrow s$$

$$\textbf{1-}\textsc{FuncBody}_{\textsc{ns}}\quad \text{updbody}(\text{env},\text{S})\vdash\langle\text{S},s\rangle\longrightarrow s$$

$$\textbf{1-}\textsc{Conditional}_{\textsc{ns}}\quad \frac{\begin{array}{c}\text{updbody}(\text{env},\text{`if}(b)\text{then('})\vdash s\longrightarrow s\\ \text{env}'\vdash\langle\text{S}_1,s\rangle\longrightarrow s\qquad \text{updbody}(\text{env}',\text{`)else('})\vdash s\longrightarrow s\\ \text{env}''\vdash\langle\text{S}_2,s\rangle\longrightarrow s\qquad \text{updbody}(\text{env}'',\text{`)'}\vdash s\longrightarrow s\end{array}}{\text{env}\vdash\langle\text{if}(b)\text{then}(\text{S}_1)\text{else}(\text{S}_2),s\rangle\longrightarrow s}$$

$$\textbf{1-}\textsc{While}_{\textsc{ns}}\quad \frac{\begin{array}{c}\text{updbody}(\text{env},\text{`while}(b)\text{do('})\vdash s\longrightarrow s\\ \text{env}'''\vdash\langle\text{S},s\rangle\longrightarrow s\qquad \text{updbody}(\text{env}''',\text{`)'})\vdash s\rightarrow s\end{array}}{\text{env}\vdash\langle\text{while}(b)\text{do}(\text{S}),s\rangle\longrightarrow s}$$

Table 3: First pass derivation rules for function bodies.

Additionally, we define

$$\begin{aligned}\text{env}' &= \text{updbody}(\text{env},\text{`if}(b)\text{then}('),\\ \text{env}'' &= \text{updbody}(\text{env}',\text{`)else}('),\\ \text{env}''' &= \text{updbody}(\text{env},\text{`while}(b)\text{do}(').\end{aligned}$$

The rules in table 3 deal with all function bodies. For the purpose of the first pass, there is only one thing in a function body which we are interested in; function calls. The first rule handles this case, and the second rule handles the case where the statement is not a function call. The last two rules deal with if statements and while loops, because these are statements that contain *other* statements inside of them.
Note that calls to 'updbody' have the syntax that is being added wrapped in quotation marks. These quotation marks do not have any meaning, and are only there to distinguish closing parentheses in *syntax* from closing parentheses that actually close the arguments passed to 'updbody'.

1-ASS-BI$_{\text{NS}}$ $\text{env} \vdash \langle x_{\text{bi}} = \text{bi}, s\rangle \longrightarrow s[x_{\text{bi}} \to \mathcal{BI}[\![\text{bi}]\!]s]$ if $\text{env}_5 = $ ‘’
1-ASS-BY$_{\text{NS}}$ $\text{env} \vdash \langle x_{\text{by}} = \text{by}, s\rangle \longrightarrow s[x_{\text{by}} \to \mathcal{BY}[\![\text{by}]\!]s]$ if $\text{env}_5 = $ ‘’
1-ASS-ARR$_{\text{NS}}$ $\text{env} \vdash \langle x_{\text{arr}} = \text{arr}, s\rangle \longrightarrow s[x_{\text{arr}} \to \mathcal{ARR}[\![\text{arr}]\!]s]$ if $\text{env}_5 = $ ‘’
1-ASS-ARRELEM$_{\text{NS}}$ $\text{env} \vdash \langle x_{\text{arr}} = \text{by}, s\rangle \longrightarrow \mathcal{ARR}[\![x_{\text{arr}}[a] = \text{by}]\!]s]$ if $\text{env}_5 = $ ‘’
1-ASS-VAR$_{\text{NS}}$ $\text{env} \vdash \langle x_{\text{var}} = \text{var}, s\rangle \longrightarrow s[x_{\text{var}} \to \mathcal{A}[\![\text{var}]\!]s]$ if $\text{env}_5 = $ ‘’

Table 4: First pass derivation rules for global variables.

These rules handle variable assignment *outside* of functions (indicated by the requirement that $\text{env}_5 = $ ‘’).

We can now apply the rules we have defined to obtain a derivation tree. For the sake of brevity, we will refer to elements of the environment tuple by their index, as is also done in the definition of the environment update functions. The obtained derivation tree is given below:

$$\cfrac{\cfrac{\begin{array}{c}\text{updfunc}(\text{env}, a) \vdash \langle a, ()\rangle \longrightarrow () \\ \text{updarg}(\text{env}', x, var) \vdash \langle x = var, ()\rangle \longrightarrow () \\ \text{updbody}(\text{funccall}(\text{env}'', b), b(x)) \vdash \langle b(x), ()\rangle \longrightarrow () \\ \text{endfunc}(\text{env}''') \vdash () \longrightarrow () \\ \text{Tree 2}\end{array}}{\text{env} \vdash \langle b(x = var)(c(x)); c(x = var)(x = x + incr); incr = 3, ()\rangle \longrightarrow (incr \to 3)}}{\text{env} \vdash \langle a(x = var)(b(x)); b(x = var)(c(x)); c(x = var)(x = x + incr); incr = 3, ()\rangle \longrightarrow (incr \to 3)}$$

Where ‘Tree 2’ is defined as:

$$\cfrac{\cfrac{\begin{array}{c}\text{updfunc}(\text{env}'''', b) \vdash \langle b, ()\rangle \longrightarrow () \\ \text{updarg}(\text{env}''''', x, var) \vdash \langle x = var, ()\rangle \longrightarrow () \\ \text{updbody}(\text{funccall}(\text{env}'''''', c), c(x)) \vdash \langle c(x), ()\rangle \longrightarrow () \\ \text{endfunc}(\text{env}''''''') \vdash () \to () \\ \text{Tree 3}\end{array}}{\text{env} \vdash \langle c(x = var)(x = x + incr); incr = 3, ()\rangle \longrightarrow (incr \to 3)}}{\text{env} \vdash \langle b(x = var)(c(x)); c(x = var)(x = x + incr); incr = 3, ()\rangle \longrightarrow (incr \to 3)}$$

Where ‘Tree 3’ is defined as:

$$\frac{\dfrac{\begin{array}{c}\text{updfunc}(\text{env}'''''''', c) \vdash \langle c, ()\rangle \longrightarrow () \\ \text{updarg}(\text{env}''''''''', x, var) \vdash \langle x = var, ()\rangle \longrightarrow () \\ \text{updbody}(\text{env}'''''''''', x = x + incr) \vdash \langle x = x + incr, ()\rangle \longrightarrow () \\ \text{endfunc}(\text{env}''''''''''') \vdash () \to ()\end{array}}{\text{env} \vdash \langle c(x = var)(x = x + incr), ()\rangle \longrightarrow ()} \quad \text{env} \vdash \langle incr = 3, ()\rangle \longrightarrow (incr \to 3)}{\text{env} \vdash \langle c(x = var)(x = x + incr); incr = 3, ()\rangle \longrightarrow (incr \to 3)}$$

And where we define

$$\begin{aligned}
\text{env}' &= \text{updfunc}(\text{env}, a), \\
\text{env}'' &= \text{updarg}(\text{env}', x, var), \\
\text{env}''' &= \text{updbody}(\text{funccall}(\text{env}'', b), b(x)), \\
\text{env}'''' &= \text{endfunc}(\text{env}'''), \\
\text{env}''''' &= \text{updfunc}(\text{env}'''', b), \\
\text{env}'''''' &= \text{updarg}(\text{env}''''', x, var), \\
\text{env}''''''' &= \text{updbody}(\text{funccall}(\text{env}'''''', c), c(x)), \\
\text{env}'''''''' &= \text{endfunc}(\text{env}'''''''), \\
\text{env}''''''''' &= \text{updfunc}(\text{env}'''''''', c), \\
\text{env}'''''''''' &= \text{updarg}(\text{env}''''''''', x, var), \\
\text{env}''''''''''' &= \text{updbody}(\text{env}'''''''''', x = x + incr).
\end{aligned}$$

Annotated program style

It is immediately obvious that this style of derivation trees is rather verbose. Indeed, the toy example, which contains only four lines of code (or, if they are expanded to fit with the normal styling of `While`$^+$ code, ten), results in a derivation tree which contains nineteen rule applications. If we consider that just the reference implementation consists of 580 lines, it should be clear that we require a more compact visualization. For this, we provide the annotated program style. This style, inspired by ‘decorated programs’ as defined by Benjamin Pierce [21] allows us to show a derivation tree much more compactly.
There is one distinct difference between decorated programs and our styling; decorated programs are defined for Hoare logic, whereas our derivation rules fall under Natural Semantics.
Below, we provide the annotated equivalent of each derivation rule, starting with composition rules:

[$\mathbf{1}$-comp$^1_{ns}$]
$env = (funcs, func_{args}, func_{bodies}, func_{calls}, curr_{func})$
$state = s$
S_1; $\{ \vdash \ \delta_1 \ \}$
$env = (funcs, func_{args}, func_{bodies}, func_{calls}, curr_{func})$
$state = s'$
S_2 $\{ \ \delta_2 \ \}$
$env = (functions', func'_{args}, func'_{bodies}, func'_{calls}, curr'_{func})$
$state = s''$

[$\mathbf{1}$-comp$^2_{ns}$]
$env = (funcs, func_{args}, func_{bodies}, func_{calls}, curr_{func})$
$state = s$
$func$ $\{ \ funcs \mathrel{+}= func, curr_{func} = func \ \vdash \}$
(
$args$ $\{ \ \delta_1 \ \vdash \}$
)
(
S_1 $\{ \ \delta_2 \ \vdash \}$
); $\{ \ curr_{func} ='' \vdash \}$
$env = (funcs', func'_{args}, func'_{bodies}, func'_{calls}, curr'_{func})$
$state = s'$
S_2 $\{ \ \delta_3 \ \vdash \ \delta_4 \ \}$
$env = (functions'', func''_{args}, func''_{bodies}, func''_{calls}, curr''_{func})$
$state = s''$

[$\mathbf{1}$-comp$^3_{ns}$]
$env = (funcs, func_{args}, func_{bodies}, func_{calls}, curr_{func})$
$state = s$
S_1; $\{ \ \delta_1, \ func_{bodies}[curr_{func}] + = ; \vdash \}$
S_2 $\{ \ \delta_2 \ \vdash \}$
$env = (funcs', func'_{args}, func'_{bodies}, func'_{calls}, curr'_{func})$

Table 5: First pass derivation rules for composition, annotated program style.

Here, 'δ_i' indicates the differences in environment or state caused by a specific statement. If a turnstile is present, it denotes that the delta only affects the environment (if the delta is before the turnstile) or only the state (if it is present after).
For instance, in the rule **1**-comp$^1_{ns}$, 'funcs$'$' is obtained by applying δ_2 on 'funcs'.
Furthermore, in this rule, the first statement cannot be a function declaration. Thus, it cannot alter the environment, which is why the turnstile indicates that the delta only applies to the state.

Next, we provide the annotated variants of the rules that deal with function arguments:

[**1**-arg-comp$_{ns}$]
$var = type,\quad \{func_{args}[curr_{func}] \mathrel{+}= (var, type)\ \vdash\}$
$arg\quad \{\ \delta_1\ \vdash\}$

[**1**-arg$_{ns}$]
$var = type\quad \{func_{args}[curr_{func}] \mathrel{+}= (var, type)\ \vdash\}$

Table 6: First pass derivation rules for function arguments, annotated program style.

Note that for the first rule, the δ will denote all subsequent arguments that are to be processed, due to our definition of 'args'.

We now provide the annotated variants of the function body rules:

[**1**-funccall$_{ns}$]
$func(args) \quad \{\ func_{calls}[curr_{func}] \mathrel{+}= func,\ func_{body}[curr_{func}] + = func(args)\ \vdash \}$

[**1**-funcbody$_{ns}$]
$S \qquad \{\ func_{body}[curr_{func}] \mathrel{+}= S\ \vdash \}$

[**1**-conditional$_{ns}$]
$if(b)\ then\ (\quad \{func_{body}[curr_{func}] \mathrel{+}= if(b)then(\vdash \}$
$\quad S_1 \qquad \{\ \delta_1\ \vdash \}$
$)\ else\ (\quad \{func_{body}[curr_{func}] \mathrel{+}=)else(\vdash \}$
$\quad S_2 \qquad \{\ \delta_2\ \vdash \}$
$) \qquad \{func_{body}[curr_{func}] \mathrel{+}=)\ \vdash \}$

[**1**-whileloop$_{ns}$]
$while(b)do(\quad \{func_{body}[curr_{func}] \mathrel{+}= while(b)do(\vdash \}$
$\quad S \qquad \{\delta_1\ \vdash \}$
$) \quad \{func_{body}[curr_{func}] \mathrel{+}=)\ \vdash \}$

Table 7: First pass derivation rules for function bodies, annotated program style.

Finally, we provide the annotated variants of the rules that deal with global variables.

[**1**-ass-bi$_{ns}$]
$x_{bi} = bi \quad \{ \vdash \; x_{bi} = \mathcal{BI}[\![bi]\!]s \}$

[**1**-ass-by$_{ns}$]
$x_{by} = by \quad \{ \vdash \; x_{by} = \mathcal{BY}[\![by]\!]s \}$

[**1**-ass-arr$_{ns}$]
$x_{arr} = arr \quad \{ \vdash \; x_{arr} = \mathcal{ARR}[\![arr]\!]s \}$

[**1**-ass-arrelem$_{ns}$]
$x_{arr}[a] = by \quad \{ \vdash \; x_{arr} = \mathcal{ARR}[\![x_{arr}[a] = by]\!]s \}$

[**1**-ass-var$_{ns}$]
$x_{var} = a \quad \{ \vdash \; x_{var} = \mathcal{A}[\![a]\!]s \}$

Table 8: First pass derivation rules for global variables, annotated program style.

An observant reader will note that these rules are not quite the same as one would expect. Indeed, we are only defining the *difference* in the environment on every rule application. This is a consequence of the fact that there are only two ways we can alter the environment; either we change the value of 'current function', or we add some element to a set or tuple. Because this is so restrictive, there is only one way in which we can interpret those deltas. Note that we explicitly state the environment and state at the start of the program, but also at the end. We do this, because it allows any reader to verify that the result of the annotated program is valid. Using this annotated program style, we can simplify our toy example derivation tree:

```
env = ( ∅, ∅, ∅, ∅, '')
state = ()
a            { funcs += a, curr_func = a ⊢ }
(x = var)    { func_args[a] += (x, var) ⊢ }
(
{ func_body[a] += b(x), func_calls[a] += b ⊢ }
    b(x)
);          { curr_func = ‘‘” ⊢ }
b(          { funcs += b, curr_func = b ⊢ }
    x = var { func_args[b] += (x, var) ⊢ }
)(
{ func_{body}[b] += c(x), func_calls[b] += c ⊢ }
    c(x)
);          { curr_func = ‘‘” ⊢ }
c(          { funcs += c, curr_func = c ⊢ }
    x = var { func_args[c] += (x, var) ⊢ }
)(
    { func_body[c] += x = x+incr ⊢ }
    x = x + incr
);          { curr_func = ‘‘” ⊢ }
incr = 3    { ⊢ incr = 3}
env = (
    { a, b, c },
    {
        a ↦ {(x, var)},
        b ↦ {(x, var)},
        c ↦ {(x, var)}
    },
    {
        a ↦ b(x),
        b ↦ c(x),
        c ↦ x=x+incr
    },
    {
        a ↦ { b },
        b ↦ { c },
        c ↦ ∅
    },
    ””)
state = { incr ↦ 3 }
```

In order to avoid ambiguity as to which delta applies to which statement, we added additional newlines to the code. Even with those, however, we can now simply read the resulting state and environment by applying all deltas from top to bottom (starting with the empty environment and the empty state).

Results of application

We see that after the application, both the environment and the state contain useful information. Let us make the information contained within slightly more explicit:

$$\begin{aligned}
\text{functions} &= \{a, b, c\} \\
\text{function args} &= \{a \to ((x, var)), b \to ((x, var)), c \to ((x, var))\} \\
\text{function bodies} &= \{a \to b(x), b \to c(x), c \to x = x + incr\} \\
\text{function calls} &= \{a \to \{b\}, b \to \{c\}, c \to \emptyset\} \\
\text{state} &= \{incr \mapsto 3\}
\end{aligned}$$

Note that we have not listed the 'current function' part of the environment, because it contains no useful information; after constructing a derivation tree of a valid `While`$^+$ program, it will always be empty.
We now need to decide upon a valid order of derivation, for we cannot derive a function that calls a function that we have not yet derived (because we would not know what the called function does). To do this, we can look at 'function calls'. Indeed, any function which maps to the empty set in this mapping can be derived at the beginning. A general procedure for obtaining a valid derivation order is as follows:

1. Apply the first pass derivation rules on a `While`$^+$ program, to obtain the resulting environment and state.
2. Create an ordered tuple, which will hold the names of the functions to be derived, in some order. Let us call this tuple 'derivation order'.
3. Take each function which, in the mapping of 'function calls', currently maps to $\emptyset$, and add it to the end of 'derivation order'. Afterwards, remove this element from the domain of 'function calls'.
4. Update the mapping of 'function calls'. For each element, we remove those parts of the co-domain that are also present in 'derivation order'. By doing so, 'function calls' no longer represents the function calls that are made per function, but only those that are not yet (simulated to be) derived.
5. If the size of 'functions' is equal to 'derivation order', or there are no elements in 'function calls' that currently map to the empty set, stop. Otherwise, go back to step 3.

Note that this procedure may fail. Indeed, if there are functions that refer to themselves (or refer to other functions, which in turn refer back to the first function), there is no valid derivation order. Similarly, if there are function calls to functions that do not exist, there will be no valid derivation order. This is the case when the length of the 'functions' set is not equal to the length of the 'derivation order' tuple.

A second important thing to note is that this procedure is not exactly deterministic. If we arrive at step 3, and we have *multiple* functions with an empty co-domain, the procedure does not specify which one should be added first. However, this does not influence the correctness of the outcome; the only thing we wish to ensure is that if we derive each function according to a given ordering, no function calls to non-derived functions are encountered.

Let us apply the procedure on the toy example. We have already applied step 1, and now apply step 2.

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \{a \to \{b\}, b \to \{c\}, c \to \emptyset\}\\ \text{derivation order} &= ()\end{aligned}$$

We apply step 3:

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \{a \to \{b\}, b \to \{c\}\}\\ \text{derivation order} &= (c)\end{aligned}$$

Step 4:

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \{a \to \{b\}, b \to \emptyset\}\\ \text{derivation order} &= (c)\end{aligned}$$

Now, we are in step 5, but b maps to the empty set, so we go back to step 3.

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \{a \to \{b\}\}\\ \text{derivation order} &= (c, b)\end{aligned}$$

We again apply step 4:

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \{a \to \emptyset\}\\ \text{derivation order} &= (c, b)\end{aligned}$$

We are again in step 5, but now a maps to the empty set, so we go back to step 3.

$$\begin{aligned}\text{functions} &= \{a, b, c\}\\ \text{function calls} &= \emptyset\\ \text{derivation order} &= (c, b, a)\end{aligned}$$

Because there are no elements in 'function calls' anymore, we can skip step 4, and immediately arrive in step 5. Now, we see that the length of 'functions' and 'derivation order' are equal, so we may stop, as this means that all functions are present in the derivation order. We have found a derivation order of (c, b, a), of the functions $\{a, b, c\}$.

3.2.2 Second pass derivation rules

The second pass deals with the creation of function mappings for each function in the 'derivation order' tuple. The general process for doing so is as follows:

1. Remove the first element from the 'derivation order' tuple. Let us call this element 'current function'.
2. Use 'current function' on 'function args' to obtain the arguments that belong to this function. Let us call these arguments 'args'.
3. Use 'current function' on 'function bodies' to obtain the function body that belongs to this function. Let us call this 'body'.
4. Create an initial state, that contains as its read only part the state of the first pass. Add every argument pair in 'args' to the state, but with arbitrary values. Let us call this state 's'.
5. Apply the second pass derivation rules on 'body', with state 's', to obtain the resulting function mapping. Add that function mapping to the environment.
6. If there are still functions left in 'derivation order', go to step 1.

An arbitrary value, in this context, means a value that is within the domain of the variable type, but not yet known to be any *specific* value. Additionally, if the type is made up of component types, we may specify the arbitrary value in terms of the component types.
As an example, we show what arbitrary values look like for numbers, bits and bytes:

$$\begin{aligned}(number, var) &\mapsto x\\ (bitval, bit) &\mapsto b\\ (byteval, byte) &\mapsto (b', b'', b''', b'''', b''''', b'''''', b''''''', b'''''''')\end{aligned}$$

Initializing each argument as an arbitrary value in step 3 might seem like an odd choice, but it has far-reaching consequences. If we initialize arguments as concrete values, the results of a derivation only show that for some *specific* input, we get some other specific output. By taking an arbitrary input, we show that a function mapping holds for *all* possible inputs. This makes the equivalence proof a lot stronger.

Restrictions

A function mapping is defined in terms of its variables, but is not always applicable. Consider the following example:

```
foo(a = var)
(
    if (a > 3) then (
        a = 7
    ) else (
        a = 0
    )
)
```

If we were to apply the second pass rules to it, we would obtain two function mappings. The first has as restriction (or, equivalently, requirement) that $a > 3$, and maps a to 7 . The second has as restriction that $\neg(a > 3)$, or the equivalent $a \leq 3$, and maps a to 3.
Formally, we define a restriction as:

$$r_i := (x, s) \mapsto \{\text{True}, \text{False}\}$$
$$\forall i \in \{arr, bi, by, var\}$$

Where x is the variable of type i, and s the state.
Going back to the toy example defined earlier; our two function mappings are formally defined as:

$$(\{a > 3\}, (a \to 7))$$
$$(\{a \leq 3\}, (a \to 3))$$

In the environment, we would visualize the function mappings as:

$$\begin{aligned}&\{foo \longrightarrow \{\\ &\quad(\{a > 3\}, (a \to 7)),\\ &\quad(\{a \leq 3\}, (a \to 3))\}\\ &\}\end{aligned}$$

So when function 'foo' is called, the function mapping that is applied is the one that holds true given the provided state 's'.
Note that when the rules are applied properly, we always get cases that are complete and not overlapping. In some edge cases[1], however, we could end

[1]These edge cases are not invalid, but can never influence the semantics of the program. Thus, we do not really care about them.

up with a restriction which can never be true. Take for instance $x \leq 3 \wedge x \geq 8$. This simply means that the function mapping is never applicable. An example piece of code that would lead to such a restriction would be:

```
if( x <= 3 ) then (
    if ( x >= 8 ) then (
        S_1
    ) else (
        S_2
    )
) else (
    S_3
)
```

The statement S_1 would then be unreachable with the above restriction because although our rule definitions state that the statement is reached when the requirement is true, it is intuitively clear that $x \leq 3 \wedge x \geq 8$ will never be true.

With all the rules defined, we show their application on a toy example, and show that it is equivalent to creating a derivation tree for every valuation of the arguments.

```
foo(bar=bit, ret=bit)
(
    baz(bar, ret)
);
baz(bar=bit, ret=bit)
(
    ret = bar ∨ 1̲
)
```

Application of the first pass derivation rules yields the following annotated program:

```
env = ( ∅, ∅, ∅, ∅, "")
state = ()
foo { functions += foo, current function = foo ⊢ }
(
    bar = bit { function args[foo] += (bar, bit) ⊢ }
    ,
    ret = bit { function args[foo] += (ret, bit) ⊢ }
)(
    { function calls[foo] += baz, function body[foo]
      += baz(bar, ret) ⊢ }
    baz(bar, ret)
); { current function = "" ⊢ }
baz { functions += baz, current function = baz ⊢ }
```

```
(
    { function body[baz] += ret=bar\/1̲ ⊢ }
    ret = bar\/1̲
)   { current function = "" ⊢ }

env = (
    {foo, baz},
    {
        foo ↦ {(bar, bit), (ret, bit)},
        baz ↦ {(bar, bit), (ret, bit)}
    },
    {
        foo ↦ baz(bar, ret),
        baz ↦ ret = bar ∨ 1̲
    },
    {
        foo ↦ { baz },
        baz ↦ ∅
    },
    ""
)
state = ()
```

Applying the procedure for obtaining a valid derivation order (omitted for conciseness), we get:

$$(\texttt{baz}, \texttt{foo}).$$

We can manually quickly check this for correctness; `baz` is called from `foo`, but not the other way around. Thus, we can safely derive `baz` before deriving `foo`.

Now we may apply the second pass derivation rules on them. Applying the procedure:

$$\begin{aligned}\text{current function} &= \texttt{baz}\\ \text{args} &= \{((bar, bi), (ret, bi)\}\\ \text{body} &= ret = bar\backslash/\underline{1}\\ \text{s} &= (bar \mapsto bi', ret \mapsto bi'')\end{aligned}$$

Now, we may make an annotated program:

```
env = ( ∅ )
state = ()
ret = bar\/1̲ { ⊢ ret = 1̲}
env = ( { } )
state = (ret ↦ 1̲)
```

Where we manually simplify $bar \vee \underline{1}$ to $\underline{1}$, using the definition of $\vee$ (i.e., because both $\underline{0} \vee \underline{1}$ and $\underline{1} \vee \underline{1}$ yield $\underline{1}$).
Note as well that the environment, which contains the function mappings, is completely empty. This should make sense, as this is the first function for which we are creating an annotated program.
Then we update the environment with the new function mapping:

$$\begin{aligned}&\{\\ &\qquad baz \longrightarrow \{\{\}, ((bar, ret) \rightarrow (bar, \underline{1}))\},\\ &\}\end{aligned}$$

where the empty set represents the fact that there are no restrictions on this function.
We will now repeat this with the second function:

$$\begin{aligned}\text{current function} &= \texttt{foo}\\ \text{args} &= \{((bar, bi), (ret, bi)\}\\ \text{body} &= ret = baz(bar, ret)\\ \text{s} &= (bar \mapsto bi', ret \mapsto bi'')\end{aligned}$$

Creating another annotated program:

```
env = ({baz ⟶ {∅, ((bar,ret) ↦ (bar,1))}})
state = ( )
ret = baz(bar, ret) { ⊢ ret=1 }
env = ({baz ⟶ {∅, ((bar,ret) ↦ (bar,1))}})
state = ( )
```

In the third line, a lot of things are happening at once. Let us go through them step by step, to explain the process clearly:

- We apply the rule that handles function calls. Specifically, the function is ‘baz’, and it is called with the parameters ‘bar’ and ‘ret’, which are both bytes[2].

- In our environment, we find some mapping for the function ‘baz’. This maps to a set of pairs, each containing a set of restrictions, and a function mapping.

- There is exactly one set of pairs.

- The set of restrictions is empty. Therefore, the restrictions are vacuously met[3].

[2]If the name of the function is not found, or the number or types of the parameters does not match, the function rule would not have been applicable.

[3]If there would be restrictions, this implies that there are multiple variants of the same

- The function mapping specifies that 'bar' retains the value it had before, and 'ret' maps to $\underline{1}$. We do not need to specify that something does not change, so in the annotation, we only need to specify that 'ret' maps to $\underline{1}$.

Then the output of the derivation rule procedure is the following set of mappings:

$$
\begin{aligned}
&\{\\
&\qquad baz \longrightarrow \{\{\}, ((bar, ret) \rightarrow (bar, \underline{1}))\}\\
&\qquad foo \longrightarrow \{\{\}, ((bar, ret) \rightarrow (bar, \underline{1}))\}\\
&\}
\end{aligned}
$$

Note that the arrows used to represent the various mappings are indeed explicitly different; the outermost mappings are represented by $\longrightarrow$, whereas the inner mappings are represented by $\rightarrow$. In ASCII format, these are similarly represented by $\longrightarrow$ and $\mapsto$.
In this manner, we reduce the chance of people misreading the scoping of the mappings.

Rule definition

We define the second set of derivation rules, show their use on the same toy example from Subsection 3.2.1, and explain how to interpret the results.
To store the information we obtain from applying the rules, we define an 'environment':

$$\text{environment} := (\text{mappings}, \text{current function})$$

The first element of the environment stores the state mapping(s) of each function, while the second stores the current function to which the rules are being applied. For conciseness, we will refer to 'env[funcname]' when accessing the mappings of function 'funcname', and to 'func' when referring to the current function.

In the first set of derivation rules, our state was a quadruplet. In the second set, the state will be a *pair* of quadruplets. The first element of this pair is the read-only state, obtained from the first pass (containing all global variables). The second element is the state of the current function that is being derived. Consequently, assigning a variable in a function is only possible if

functions (i.e. there are one or more conditional statements in the function, so it is split into one variant where the true branch is always taken, and one where the false branch is always taken). Then, we would have had to check for which variant all restrictions hold. If there would have been no such variant, the function call rule would not have been applicable.

it is *not* present in the read-only state.
No rules enforce this requirement, so it is up to the creator of the derivation tree to ensure it is not violated.

Before we give the derivation rules themselves, we first provide a number of functions to interact with the environment. As in Subsection 3.2.1, a function that does multiple things has each statement on a new line, and the environment is regarded to be a tuple that may be accessed through its index.

```
$valid_{call}(env, call_{func}, state) := result = \perp$
    $intermediary = \perp$
    $call_{func} \in env \wedge$
            $(\exists (r, intermediary) \in env[call_{func}] : \forall r' \in r :$
                        $r'(state)) \rightarrow result = intermediary$
  $\leftarrow result$

$arr_{restriction}(env, arr, len) := env[func]_1 = env[func]_1 \cup \{length(arr) \geq len\}$
  $\leftarrow env$

$arr_{creation}(env, arr, len) := env[func]_1 = env[func]_1 \cup \{length(arr) = len\}$
  $\leftarrow env$

$if_{true}(env, cond) := env[func]_1 = env[func]_1 \cup \{cond\}$
  $\leftarrow env$

$if_{false}(env, cond) := env[func]_1 = env[func]_1 \cup \{\neg cond\}$
  $\leftarrow env$
```

The first function, '$valid_{call}$', returns the function mapping for a specific function iff '$call_{func}$' exists in the function mapping, and there is at least one mapping for '$call_{func}$' for which all of its restrictions hold in the provided *state*. If not, the function is undefined.
This is enforced by capturing the current mapping to be considered in 'm'. If all restrictions hold, then the right side of the implication is met, meaning we assign the result to 'r'. If not, 'r' retains its undefined value. The second

and third functions both add a restriction to the set of restrictions for the current function. To be more precise, it adds a restriction with respect to the length of an array. This is usually used to enforce that an array can be read from or written to within a specific function, if it is a parameter.
The last two functions also add requirements, which are to be used for conditionals; they add requirements to enforce that an if-then-else statement always falls within its false or true case.

Having defined several functions to manipulate the environment, we give the second set of derivation rules. We give these in five parts; assignments, skip, compositions, conditionals, while loops and function calls. Note that each rule name starts with **2** to indicate that it is part of the rules that handle the second pass.

2-ASS-VAR$_{\text{NS}}$
$$\text{env} \vdash \langle x_{\text{var}} = \text{a}, s\rangle \longrightarrow s[x_{\text{var}} \to \mathcal{A}[\![\text{a}]\!]s] \text{ if } \text{env}_2 \neq \text{`'}$$

2-ASS-BI$_{\text{NS}}$
$$\text{env} \vdash \langle x_{\text{bi}} = \text{bi}, s\rangle \longrightarrow s[x_{\text{bi}} \to \mathcal{BI}[\![\text{bi}]\!]s] \text{ if } \text{env}_2 \neq \text{`'}$$

2-ASS-BY$_{\text{NS}}$
$$\text{env} \vdash \langle x_{\text{by}} = \text{by}, s\rangle \longrightarrow s[x_{\text{by}} \to \mathcal{BY}[\![\text{by}]\!]s] \text{ if } \text{env}_2 \neq \text{`'}$$

2-ASS-ARR$_{\text{NS}}$
$$\text{arr}_{\text{creation}}(\text{env}, \text{arr}, \mathcal{A}[\![\text{len}]\!]s) \vdash \langle x_{\text{arr}} = \text{arr}\{len\}, s\rangle \longrightarrow s[x_{\text{arr}} = \mathcal{ARR}[\![\text{arr}]\!]s]$$
$$\text{if } \text{env}_2 \neq \text{`'}$$

2-ASS-ARRELEM$_{\text{NS}}$
$$\text{arr}_{\text{restriction}}(\text{env}, x_{\text{arr}}, \mathcal{A}[\![a]\!]s) \vdash \langle x_{\text{arr}}[a] = \text{by}, s\rangle \longrightarrow s[x_{\text{arr}} = \mathcal{ARR}[\![x_{\text{arr}}[a]]\!]s$$
$$\text{if } \text{env}_2 \neq \text{`'}$$

Table 8: Second pass derivation rules for variable assignment.

The first three rules, **2**-ass-var$_{\text{ns}}$, **2**-ass-bi$_{\text{ns}}$ and **2**-ass-by$_{\text{ns}}$ are essentially identical to the ones for assignment in the first set of derivation rules, except for the fact that they now update the second quadruple of the state as opposed to the first. However, **2**-ass-arr$_{\text{ns}}$ and **2**-ass-arrelem$_{\text{ns}}$ differ, in the sense that they add restrictions; if an array is created with size x, it trivially follows that it must be of size x. It might seem strange to create a restriction for this, but this can then be used to reduce additional restrictions.
If an array is read from or written to at index x, then it follows that it must be of length $\geq x$. If the array is local, we can always directly show that it holds or that it does not hold (in which case the `While`$^{+}$ code is invalid), but if the array is a parameter, the restriction will remain in the function mapping.

$$\textbf{2}\text{-}\textsc{skip}_{\text{NS}}$$
$$\text{env} \vdash \langle \text{skip}, s\rangle \longrightarrow s$$

Table 9: Second pass derivation rules for the skip statement.

The skip statement, like in normal `While`, does nothing.

$$\textbf{2}\text{-}\textsc{comp}_{\text{NS}}$$
$$\frac{\text{env} \vdash \langle S_1, s\rangle \longrightarrow s',\ \text{env} \vdash \langle S_2, s'\rangle \longrightarrow s''}{\text{env} \vdash \langle S_1; S_2, s\rangle \longrightarrow s''}$$

Table 10: Second pass derivation rules for composition.

Next, the composition rule. Unlike in the first pass, there is only one case to deal with, because the second pass always applies to a function body[4].

$$\textbf{2}\text{-}\textsc{if}^{\text{TT}}_{\text{NS}}$$
$$\frac{\text{env} \vdash \langle S_1, s\rangle \longrightarrow s'}{\text{env} \vdash \langle \text{if}(b)\text{then}(S_1)\text{else}(S_2), s \rightarrow s'} \text{ if } \mathcal{B}[\![b]\!]s = \text{tt}$$

$$\textbf{2}\text{-}\textsc{if}^{\text{FF}}_{\text{NS}}$$
$$\frac{\text{env} \vdash \langle S_2, s\rangle \longrightarrow s'}{\text{env} \vdash \langle \text{if}(b)\text{then}(S_1)\text{else}(S_2), s \rightarrow s'} \text{ if } \mathcal{B}[\![b]\!]s = \text{ff}$$

$$\textbf{2}\text{-}\textsc{if}^{\perp}_{\text{NS}}$$
$$\frac{\text{if}_{\text{false}}(\text{env}, b) \vdash \langle S_1, s\rangle \longrightarrow s', \quad \text{if}_{\text{true}}(\text{env}, b) \vdash \langle S_2, s\rangle \longrightarrow s''}{\text{env} \vdash \langle \text{if}(b)\text{then}(S_1)\text{else}(S_2), s\rangle \Rightarrow (s', s'')} \text{ if } \mathcal{B}[\![b]\!]s = \perp$$

Table 11: Second pass derivation rules for conditionals.

There are three rules for if statements. The first two deal with the cases where the state allows us to immediately determine whether the condition is true. As a toy example; a condition of `x < 6`, with a state of $(x \mapsto 8)$ clearly makes the condition evaluate to ff.
The $\textbf{2}\text{-if}^{\perp}_{\text{ns}}$ rule, however, deals with the case where the condition refers to variables which have an arbitrary value, or are partially made up of an arbitrary value. Consider the example where the condition is `x < 6`, and

[4]Recall that the second pass rules apply upon the contents of ‘function bodies’, for each function name in ‘functions’.

the state is $(x \mapsto y + 3)$, where all we know of y is that it is also a number. Clearly, we cannot apply the first two rules, but the third allows us to overcome the problem by introducing restrictions. Effectively, it splits the current function into two parts. We then apply the second pass rules to each part to obtain two mappings, with differing restrictions. This is why the result of this derivation rule is (s', s'') as opposed to a singular state. To make this more visually distinct, we use $\Rightarrow$ instead of $\rightarrow$.
To refer back to the example, we would split the derivation tree into two branches, where one branch continues with the restriction that $(y + 3) < 6$, or the equivalent $y < 3$, and the other that $\neg(y < 3)$, or the equivalent $y \geq 3$. This then trivially allows us to verify in which case the if statement is.

$\mathbf{2}$-WHILE$^{\text{TT}}_{\text{NS}}$

$$\frac{\text{env} \vdash \langle S, s\rangle \longrightarrow s', \langle \text{while}(b)\text{do}(S), s'\rangle \longrightarrow s''}{\text{env} \vdash \langle \text{while}(b)\text{do}(S), s\rangle \longrightarrow s''} \text{ if } \mathcal{B}[\![b]\!]s = \text{tt}$$

$\mathbf{2}$-WHILE$^{\text{FF}}_{\text{NS}}$

$\text{env} \vdash \langle \text{while}(b)\text{do}(S), s\rangle \longrightarrow s \text{ if } \mathcal{B}[\![b]\!]s = \text{ff}$

Table 12: Second pass derivation rules for while loops.

The while rules are identical to those in `While`.

$\mathbf{2}$-FUNCCALL$_{\text{NS}}$

$\text{env} \vdash \langle \text{func}(\text{args}), s\rangle \rightarrow m(s) \text{ if } \text{valid\_call}(\text{env}, \text{func}, s) = m$

Table 13: Second pass derivation rules for function calls.

The function call rule applies the specified mapping to the state, if one is produced by the 'valid_call' environment function. If not, the rule itself will not be applicable.

Annotated program style

Similar to the first pass rules, the second pass rules are also verbose. To solve this, we also introduce an annotated program style for the second pass rules.

[**2**-ass-var$_{ns}$]
$x_{var} = a \quad \{ \vdash \; x_{var} = \mathcal{A}[\![a]\!]s \; \}$

[**2**-ass-bi$_{ns}$]
$x_{bi} = bi \quad \{ \vdash \; x_{bi} = \mathcal{BI}[\![bi]\!]s \; \}$

[**2**-ass-by$_{ns}$]
$x_{by} = by \quad \{ \vdash \; x_{by} = \mathcal{BY}[\![by]\!]s \; \}$

[**2**-ass-arr$_{ns}$]
$x_{arr} = arr\{len\} \quad \{ \; len(x_{arr}) = \mathcal{A}[\![len]\!]s \;\; \vdash \;\; x_{arr} = \mathcal{ARR}[\![arr]\!]s \; \}$

[**2**-ass-arrelem$_{ns}$]
$x_{arr}[a] = by \quad \{ \; len(x_{arr}) \geq \mathcal{A}[\![a]\!]s \;\; \vdash \;\; \mathcal{ARR}[\![x_{arr}[a] = by]\!]s \; \}$

Table 14: Second pass derivation rules for variable assignment, annotated program style.

Similar to the first pass, the annotated style replaces calls to environment functions with their effects on the environment.
And also similar to the first pass, the effects on the environment and state are represented by the statements before and after the turnstile, respectively. They provide us with the *differences* in state and environment that occurs from the statement the annotation is applied to.
As an example; the first rule does not alter the environment in any way, but changes the stored value of the variable that is assigned to whatever the $\mathcal{A}$ function says the arithmetic expression a evaluates to.

[**2**-skip$_{ns}$]
$skip \quad \{ \vdash \}$

Table 14: Second pass derivation rule for the skip statement, annotated program style.

[$\mathbf{2}$-comp$_{ns}$]
env $=$ (restrictions, func)
state $= s$
S_1; $\{\ \delta_1\ \vdash\ \delta_2\ \}$
env $=$ (restrictions$'$, func)
state $= s'$
S_2; $\{\ \delta_3\ \vdash\ \delta_4\ \}$
env $=$ (restrictions$''$, func)
state $= s''$

Table 15: Second pass derivation rules for the composition statement, annotated program style.

```
[$\mathbf{2}$-if$^{\mathrm{tt}}_{ns}$]
$if\ (b)\ then($
    $S_1 \quad \{\ \delta_1\ \vdash \delta_2\ \}$
$)else($
    $S_2$
$)$

[$\mathbf{2}$-if$^{\mathrm{ff}}_{ns}$]
$if\ (b)\ then($
    $S_1$
$)else($
    $S_2 \quad \{\ \delta_1\ \vdash \delta_2\ \}$
$)$

[$\mathbf{2}$-if$^{\perp}_{ns}$]
$if\ (b)\ then($
    $S_1 \quad \{$ restrictions $\mathrel{+}= b\ \vdash \delta_1\ \}$
$)else($
    $S_2 \quad \{$ restrictions $\mathrel{+}= \neg b\ \vdash \delta_2\ \}$
$)$
```

For **2**-if$^{\perp}_{ns}$, the annotations might not make it entirely clear that this rule causes the derivation tree to be split. Therefore, we introduce an alternative notation:

```
[$\mathbf{2}$-if$^{\perp}_{ns}$]
%$true\ case$ :
$if\ (true)\ then($
    $S_1$   { restrictions += $b \vdash \delta_1$ }
$)else($
    $S_2$
)
%$false\ case$ :
$if\ (false)\ then($
    $S_1$
$)else($
    $S_2$   { restrictions += $\neg b \vdash \delta_2$ }
)
```

Table 16: Second pass derivation rules for conditionals, annotated program style.

Note that the lines starting with a '%' have no semantic meaning, and are there solely to indicate the 'split'. Furthermore, because the split has introduced a restriction, we may replace the conditions by true or false to make it exceedingly clear which case the split is in.

```
[$\mathbf{2}$-while$_{ns}$]
$while\ (b)\ do$
(
    { $\delta_1 \vdash \delta_2$ }
    ...
    { $\delta_{n-1} \vdash \delta_n$ }
    $S$
)
```

Table 17: Second pass derivation rules for while loops, annotated program style.

Note that in this rule, there are as many annotations as there are iterations of the loop. If there are nested loops, this might become confusing, which is why we introduce the following notation:

```
$while\ (b)\ do$
(
  $while\ (b')\ do$
  (
    $\%outer\ loop\ 1$
    $\{\ \delta_1\ \vdash\ \delta_2\ \}$
    $\dots$
    $\{\ \delta_{n-1}\ \vdash\ \delta_n\ \}$
    $\%outer\ loop\ ...$
    $\{\ \delta_1\ \vdash\ \delta_2\ \}$
    $\dots$
    $\{\ \delta_{n-1}\ \vdash\ \delta_n\ \}$
    $\%outer\ loop\ m$
    $\{\ \delta_1\ \vdash\ \delta_2\ \}$
    $\dots$
    $\{\ \delta_{n-1}\ \vdash\ \delta_n\ \}$
    $S$
  )
)
```

Where the lines starting with '%' have no semantic meaning, but are used to indicate the start of a new outer loop. Note that if a higher level of nesting occurs, the creator of the derivation tree is free to change the specific wording of the lines starting with '%'.

```
[$\mathbf{2}$-funccall$_{ns}$]
func(args)$\{\ \vdash\ valid_{call}(\text{env}, \text{func}, s)(s)\ \}$
```

Having defined all annotated variants, we now show their application on the toy example from earlier.

The derivation order we obtained before is:

$$(\mathtt{c}, \mathtt{b}, \mathtt{a}).$$

Thus, following the general process laid out, we first select the function `c` to be derived. Recall that our read-only part of the state is $(\mathit{incr} \mapsto 3)$, the argument to `c` is (x, var), and that the body is `x = x + incr`.
As we are just starting to derive programs, the first element of the environment (containing function mappings) will be empty. The second element is the name of the function being derived. Thus, the annotated program for the function `c` is:

```
state = ()
env = ( ∅, c)
{ ⊢ x = x + 3 }
x = x + incr
state = (x ↦ x = x + 3)
env = ( ∅, c)
```

There are no restrictions in this function, and the resulting state trivially is $(x \mapsto x + 3)$. Thus, the state mapping of `c` is:

```
c ↦ { ({ ∅, (x ↦ x + 3)}) }
```

Next, we select the function `b` to be derived.
The argument to `b` is (x, var), and the body is `c(x)`.
We have derived `c`, and so we have now added it to the first element of the environment. Furthermore, we update the second element of the environment to be `b`. The annotated program for the function `b` is:

```
state = ()
env = ( {c ↦ { ({ ∅, (x ↦ x + 3)}) }}, b)
{ ⊢ x = x + 3}
c(x)
state = (x ↦ x + 3)
env = ( {c ↦ { ({ ∅, (x ↦ x + 3)}) }}, b)
```

Note that although there is only one annotation, a lot happens in it. In order:

1. We see the function being called is `c`, so we look up `c` in the function mapping. It exists.

2. We check if there exists any pair from the set of function mappings of `c` for which the restriction holds. There is one; the first and only pair has an empty set of restrictions, so all restrictions vacuously hold.

3. We apply the state mapping of the pair we just found to the current state, resulting in the delta that is visible in the annotation.

There are again no restrictions, and the resulting state is again trivially $(x \mapsto x + 3)$. Thus, the state mapping of `b` is:

```
b ↦ { ({ ∅, (x ↦ x + 3)}) }
```

Finally, we select the function `a` to be derived.
The argument to `a` is (x, var), and the body is `b(x)`.
We have derived `c` and `b`, and so we have now have two function mappings in the environment. Furthermore, we update the second element of the environment to be `a`. Then, the annotated program for the function `a` is:

```
state = ()
env = ( {c ↦ { ({ ∅, (x ↦ x + 3)}) },
         b ↦ { ({ ∅, (x ↦ x + 3)}) }}, a)
{ ⊢ x = x + 3}
b(x)
state = (x ↦ x + 3)
env = ( {c ↦ { ({ ∅, (x ↦ x + 3)}) },
         b ↦ { ({ ∅, (x ↦ x + 3)}) }}, a)
```

Like before, there are no restrictions, and the resulting state is again $(x \mapsto x + 3)$. The state mapping of `a` is then:

```
a ↦ { ({ ∅, (x ↦ x + 3)}) }
```

And thus the full state mapping of this toy example is:

```
{
    c ↦ { ({ ∅, (x ↦ x + 3)}) },
    b ↦ { ({ ∅, (x ↦ x + 3)}) },
    a ↦ { ({ ∅, (x ↦ x + 3)}) }
}
```

Results of application

The result of a successful application of the derivation rules is the mapping of function names to mappings of state and associated restrictions. This, however, is not a direct answer to the question we aim to answer in this thesis. Therefore, this last step must allow us to determine whether two programs are equivalent, given the results of a successful program derivation.

There are a number of ways to go about showing equivalence. Each of those is useful in different applications, and so we provide them all below:

1. When all function names and arguments are equal, we can just check whether the mappings for each are the same. This approach can be useful when an existing program has to be updated, and we want to ensure that its semantics remain intact.

2. When only some functions are designated ‘interesting’[5], we can do the same as in the previous point, except only for these functions. This can be useful when looking at the semantics of two equivalent APIs (not necessarily in the same language). They might have a lot of differing internal functions, but their publicly available ones should then be identical.

3. Designate all functions that are not called by other functions as ‘interesting’. We can do the same as approach 1, except only for those functions that do not call others. This can be useful for comparing programs or libraries; in programs, the ‘main’ function is not explicitly called, and in libraries, one is usually interested in verifying the semantics of functions that are exposed to the end-user.

4. Disregard function names, but attempt to find a matching function mapping in the variant for every function in the reference program. This is a sort of catch-all; it is more complicated to do (and more computationally costly), but should take care of every case that is not mentioned in the previous approaches.

In the case that the chosen approach tells us that the programs are *not* equivalent, we would like to be able to tell more about what differs. How specific we can be depends on the approach that is chosen. For every approach, we will show how to identify *what* part of the program differs.

1. In approach 1, we either have a mismatch in the amount of function names / arguments, or in the mapping itself. In both cases, it is immediately clear what function is not equal, as the approach itself already enforces a one to one pairing.

2. In approach 2, we can only tell that a mismatch occurs on one of the functions that is designated to be ‘interesting’. For any of the ‘uninteresting’, we do not have a guarantee that there is some variant function mapping that we can compare them to.

3. Approach 3 is, at least for the purposes of determining where a difference occurs, completely identical to approach 2.

4. With this approach, we cannot provide a *mis*match on any function pair, only that there is no match for some (set of) function(s). This is because we do not know what function is ‘supposed’ to match to any other function with this approach, so we cannot conclude that some function should have more arguments or a differing state mapping; it might just be another unrelated function.

[5] In this context, an ‘interesting’ function is determined to be such by the discretion of the user of the technique. In the thesis, the ‘interesting’ functions are `permutation` and `permutation_4`, because they are the functions that will be called by other developers.

Even in the most restrictive case (the first approach, where there is a fixed one-to-one mapping), we cannot be more precise than to say that the difference occurs in some function pair. This is because we do not have the guarantee that the states must be equal at any point in the function, just at the *end* of the functions. Consider the following toy example, with four arbitrary statements a, b, c, d:

```
foo() (
    a;
    b;
    c;
    d;
)
bar() (
    b;
    a;
    d;
)
```

Clearly, these functions are not equivalent; c is missing in $bar()$. But if we simply look at the first spot where the states are not equal, we would naively say that the first statement results in a differing state. However, it might be the case that we obtain the same state by executing $a; b$ as compared to $b; a$ (for instance, if we choose a to be the statement `x = 3`, and b to be `y = 2`).

3.3 Complexity

The derivation rules is made up of two passes. We will consider the complexity of each, to determine the worst-case complexity of creating an entire derivation tree.
The first pass transforms syntax into semantics, but in a very limited sense. Mostly, it keeps track of function bodies (and with it, the function calls per function). However, global constants (which must exist of just variable assignments) are read into a state. A consequence of this is that in the first pass, there is no branching or looping. Thus, the complexity of the first pass is $\mathcal{O}(n)$, where n is the number of statements.

The second pass is applied on each function. However, it splits functions on each if statement, and can (depending on circumstances) also split on while loops. As a result, its complexity is $\mathcal{O}(2^n)$, where n is the number of if rule *applications*. This is not quite the same as the number of statements, because a loop might iterate more than once, and therefore require more than one split. Thus, the overall complexity of applying the derivation rules is $\mathcal{O}(2^n)$.
As we do not expand function bodies to simulate function calls, and the n is

counted per function as opposed to the entire program, it is likely this n will not be too large. Additionally, in our personal experience, the density of if statements is relatively low in most programs. However, the poor scaling does have implications for manual application of the rules.

Chapter 4

Application of `While`$^+$

In this chapter, we provide and argument our choice for a cryptographical algorithm to verify. We create derivation trees of most the algorithm's reference implementation and its chosen variant. Finally, we provide a pairing of the state mappings obtained from the derivation trees, to show semantic analogy between the reference and variant.

4.1 Choice of algorithm

In an effort to choose a simple cryptographic primitive, and so provide a proof that is not so large that it is difficult to comprehend, we have chosen to use the Keccak-f primitive. While it is not a large primitive, it is probably *the* most used family of permutations.
For the reference program, we use keccak-f[200] [1]. For a variant, we use the RISC-V optimized variant[9], whose implementation can be found on github.

4.2 Preprocessing

CIL, which is an abbreviation for C Intermediate Language, is a simplified subset of C [19]. Its simplicity allows us to make it easier to transform the code to `While`$^+$. There are some caveats to using it, but the main one is that it cannot transform *all* C code that compiles. The majority of the C code that cannot be transformed is code that contains undefined behaviour (that is, code which is not specified by the C standard, and whose behaviour therefore completely depends on the compiler that is used). Because we cannot properly obtain the semantics of code with undefined behaviour with our technique, the fact that this code cannot be transformed is of no concern. For the purpose of defining which code is valid for our technique, we say that

C code is valid if and only if it can be transformed by cilly[1] (or, equivalently, if it contains no undefined behaviour).

We have written a short script which simplifies the transformation process. The script, the result of its application on the reference implementation and the variant are all given in the appendix, in Sections A.1, A.2 and A.6 respectively.

4.2.1 CIL post-processing

When we look at the code that CIL generates, we can observe that there are lines that are meaningless with respect to this thesis (and the general purpose of transforming C code into `While`$^+$ code). Thus, we can discard them and, in doing so, make the transformation process simpler. We give all categories of meaningless lines below:

1. The '#line' and '#merge' pragma statements. These indicate where the code below it originates from, which is nice to know for debugging or validating correctness of CIL, but not required in our case.

2. The two lines of comments at the top. These indicate by which CIL version the file has been generated, and to which level the verbosity is set. This information is not useful for creating `While`$^+$ code.

3. Prototype statements. Prototype statements are function signatures, that do *not* have an accompanying body. In C, these exist to tell the compiler that some function with this signature will exist in another file. When merging with CIL, these statements are kept, though they have become meaningless; the merging merges everything to one file, so prototypes do not add anything, as the actual implementation will *also* be in that file.
   An observant reader might note that simplifying CIL code in the manner described above can create possible issues. For instance, if there is code that contains a prototype, but no accompanying function implementation, the only trace of the function are the function calls that are made to it in the rest of the program. This is correct, but it turns out that that creates no problem. If this situation occurs, then one of the things listed below will be the case:

   i. There is some reference to the missing function in the rest of the CIL code. Then the original C code does not compile and the resulting `While`$^+$ code cannot be used to create a derivation tree. At some point, we will come across a function call to the missing function, and then the derivation tree cannot be completed, so it fails.

[1]Note that cilly is the driver for CIL, which can be found in Section A.6.

ii. There is no reference to the missing function in the rest of the CIL code. Then the original C code does compile, assuming that optimizations are turned on (because unused code will be omitted). The resulting `While`$^+$ code can then also be used to create a derivation tree, because even though there is a 'missing' function, it is never called, and so the creation of the derivation tree does not fail.

Lastly, we may take type definitions, and replace each instance of them by the original type. We have automated this process in a short C# program, which is given in the appendix, in section A.4. The result of applying this post-processor on the implementations are given in the appendix as well, in Sections A.1 and A.2, respectively.

4.3 Transformation rules

The transformation rules are the rules which, when applied to some code, yield semantically equivalent `While`$^+$ code. For our reference and variant, we can say that the rules can be applied to *C* code, but in general, that need not be the case.
In general, there are two possible approaches to define transformation rules, namely:

1. The input consists of tokens of the source language, and all tokens must be transformed through a rule application to `While`$^+$. In this case, if any token of the source language remains after application of all rules, the transformation fails.

2. The input consists of tokens, and all tokens that are *not* syntax in `While`$^+$ must be transformed through a rule application to `While`$^+$. In this case, if any token that is not part of the `While`$^+$ syntax remains after application of all rules, the transformation fails.

Both approaches are correct, but the first is more verbose. Let us demonstrate this with an example.

```
(3 + 7)
```

Let us take the piece of C code above, and write rewrite rules for it with both approaches:

1.

$$\begin{aligned} (&\mapsto (\\ ) &\mapsto) \\ + &\mapsto + \\ &\mapsto \\ 3 &\mapsto 3 \\ 7 &\mapsto 7 \end{aligned}$$

2. For this piece of code, we need no rewrite rules. It is immediately valid `While`$^+$ code.

Note that this piece of code is not a *line* of code, and it is generally not the case that we need no rewrite rules with the second approach. In particular, though, it does show that the first approach is extremely verbose, and it also shows us that we need a lot of new characters to distinguish between tokens in C and tokens in `While`$^+$ (also, observe the second to last rule, which appears empty; it actually maps a space to a space).
Therefore, we choose the second approach.

4.3.1 Rule definitions

A rule is defined as a mapping from a string to a string. Note that the *input* string cannot be empty (because it could then be applied everywhere, infinitely), but the *output* string might be (which typically happens when information is actively discarded due to being superfluous). We define a couple of meta-variables:

$$\begin{aligned} \mathbb{I} &= \text{Any constant integer value.} \\ \mathbb{B} &= \text{Any constant byte value.} \\ \mathbb{C} &= \text{The smallest non-constant piece of code,} \\ &\qquad \text{such that the provided pattern holds.} \\ \mathbb{V} &= \text{A variable name.} \\ \mathbb{V}^+ &= \text{A variable name that does not yet exist in this scope.} \\ \emptyset &= \text{The empty string.} \end{aligned}$$

These allow us to define rules that are applicable on code, irrespective of the constant values like integer declarations or variable names (i.e., we do not need different rules for replacing some piece of C code with a constant 1, or the same piece of code with a constant 2 in it).
To discern one distinct meta variable from another, we may add a subscript to it. In that case, a meta variable is considered to be identical to another if both its symbol *and* its subscript are equal (i.e., $\mathbb{I}_0$ is distinct from $\mathbb{I}_1$).

Integer transformation

The rules below handle the transformation of integers from C to `While`$^+$.

$$\begin{aligned}
\mathbb{I}_0 U &\mapsto \mathbb{I}_0 \\
\text{unsigned int } \mathbb{V}_0; &\mapsto \mathbb{V}_0 = 0; \\
\text{unsigned int } \mathbb{V}_0 = \mathbb{I}_0; &\mapsto \mathbb{V}_0 = \mathbb{I}_0; \\
\text{int } \mathbb{V}_0; &\mapsto \mathbb{V}_0 = 0; \\
\text{int } \mathbb{V}_0 = \mathbb{I}_0; &\mapsto \mathbb{V}_0 = \mathbb{I}_0; \\
\mathbb{I}_0\, UL &\mapsto \mathbb{I}_0 \\
(\text{int})(\mathbb{C}_0) &\mapsto \mathbb{C}_0 \\
(\text{unsigned long})(\mathbb{C}_0) &\mapsto \mathbb{C}_0 \\
(\text{unsigned int const})(\mathbb{C}_0) &\mapsto \mathbb{C}_0 \\
(\text{unsigned long})\mathbb{I}_0 &\mapsto \mathbb{I}_0 \\
\mathbb{V}_0 + + &\mapsto \mathbb{V}_0 = \mathbb{V}_0 + 1 \\
\mathbb{V}_0 + = \mathbb{I}_0 &\mapsto \mathbb{V}_0 = \mathbb{V}_0 + \mathbb{I}_0
\end{aligned}$$

We show the application of these rules on a toy example.

```
unsigned int x;
unsigned int y = (unsigned long)33;
x = x + 3U;
y = y - 7UL + 3;
int z;
int a = (unsigned int)3;
z = y - a;
```

Which yields the following code:

```
x = 0;
y = 33;
x = x + 3;
y = y - 7 + 3;
z = 0;
a = 3;
z = y - a;
```

These rules mainly discard information that is not required in `While`$^+$. For instance, a 'U' written after some integer indicates it is unsigned. That does not quite mean it can be discarded without losing information, of course. If an unsigned integer exceeds it maximum value, it will not wrap around to the smallest possible value, but to 0. Similarly, a 'const' value cannot be reassigned, and so any assignment to a constant value should result in the derivation tree becoming stuck. However, this does not occur in the reference (not in the reference nor in the variant, to be precise), so these rules are valid for this case. We will elaborate on what this means with respect to generality in chapter 6.

Byte transformation

The next rules handle the transformation of unsigned chars in C to bytes in `While`$^+$.

$$\begin{aligned}
\text{unsigned char } \mathbb{V}_0; &\mapsto \mathbb{V}_0 = \overline{0};\\
\text{unsigned char } \mathbb{V}_0 = \mathbb{I}_0; &\mapsto \mathbb{V}_0 = \overline{\mathbb{I}_0};\\
\text{sizeof(unsigned char)} &\mapsto \overline{1}\\
\text{(unsigned char)}\mathbb{B}_0 &\mapsto \overline{\mathbb{B}_0}\\
\text{(unsigned char const)}\mathbb{B}_0 &\mapsto \overline{\mathbb{B}_0}\\
\text{(unsigned char)}\mathbb{C} &\mapsto \mathbb{C}
\end{aligned}$$

We again provide some sample code to which these rules are applied.

```
unsigned char a = 3;
unsigned char b;
b = a - sizeof(unsigned char) * (unsigned char
    const)3;
a = (unsigned char)5;
```

And the resulting `While`$^+$ code:

```
a = 3;
b = byte(0);
b = a - 1 * 3;
a = 5;
```

Note that a byte is some value between 0 and 255. A ‘char’, in C, is some value between -128 and 127. Thus, an unsigned char is exactly a byte; it does not need to use the sign bit to denote the sign. Similarly to in the previous subsection, we may discard the casts because the fact that they might be constant does not matter in the Keccak implementations.
Finally, note that in order to store `While`$^+$ code in ASCII format, we represent a byte value i as $byte(i)$.

Array transformation

Now, we give rules that handle the transformation of arrays (or, equivalently, arrays of unsigned chars) in C to byte arrays in `While`$^+$.

$$\begin{aligned}
*(\mathbb{V}_0 + \mathbb{C}_0) &\mapsto \mathbb{V}_0[\mathbb{C}_0] \\
\text{unsigned char } \mathbb{V}_0[\mathbb{I}_0]; &\mapsto \mathbb{V}_0 = arr[\mathbb{I}_0]; \\
\text{unsigned char } *\mathbb{V}_0; &\mapsto \mathbb{V}_0 = arr[25]; \\
(\text{unsigned char const } *)\mathbb{V}_0 &\mapsto \mathbb{V}_0 \\
(\text{unsigned char const } *)(\mathbb{V}_0) &\mapsto \mathbb{V}_0 \\
\text{unsigned char const } *\mathbb{V}_0; &\mapsto \mathbb{V}_0 = arr[25]; \\
\text{unsigned char const}\mathbb{V}_0[\mathbb{I}_0] = \{\mathbb{B}_0, \mathbb{B}_1, \ldots, \mathbb{B}_{\mathbb{I}_0 - 1}, \mathbb{B}_{\mathbb{I}_0}\} &\mapsto \mathbb{V}_0 = arr[\mathbb{I}_0]; \\
& \mathbb{V}_0[0] = \mathbb{B}_0; \\
& \mathbb{V}_0[1] = \mathbb{B}_1; \\
& \ldots \\
& \mathbb{V}_0[\mathbb{I}_0 - 1] = \mathbb{B}_{\mathbb{I}_0 - 1}; \\
& \mathbb{V}_0[\mathbb{I}_0] = \mathbb{B}_{\mathbb{I}_0};
\end{aligned}$$

The only type of arrays in `While`$^+$ are arrays of bytes. Thus, arrays of unsigned chars in C correspond to arrays in `While`$^+$. Furthermore, de-referencing a pointer plus some offset i in C corresponds to accessing an array at index i. There is one rule that, to most, will probably look odd. Indeed, why do we choose to map a pointer to an array of length 25? The answer is that in the reference implementation, whenever a pointer variable is made, it always happens to point toward an array of length 25. Though this sounds unreasonable, it is (mostly) a consequence of the fact that the language we are transforming from is C. In C, a pointer is no more than some memory address that can be de-referenced, and an array is no more than a contiguous block of memory that holds some number of values. Having a pointer, therefore, does not tell you anything about the length of the array that exists where the pointer points to.
The last two rules handle the mapping of arrays that are explicitly initialized. This occurs twice, in the constant offsets defined for two of the diffusion functions of Keccak. The dots ('. . . ') indicate a clearly repeating pattern, in this case, the array indices that range from 0 up until the length of the array.

If statements

We provide rules that handle the transformation of if statements, if-else statements, and if statements containing a return statement.

$$\begin{aligned}
\text{if}(\mathbb{C}_0)\{\mathbb{C}_1\}\text{ else }\{\mathbb{C}_2\} &\mapsto \text{if}(\mathbb{C}_0)\text{ then }(\mathbb{C}_1)\text{ else }(\mathbb{C}_2);\\
\text{if}(\mathbb{C}_0)\{\mathbb{C}_1\} &\mapsto \text{if}(\mathbb{C}_0)\text{ then }(\mathbb{C}_1)\text{ else (skip)};\\
\text{if}(\mathbb{C}_0)\{\text{return;}\}\mathbb{C}_1 &\mapsto \text{if}(\mathbb{C}_0)\text{ then (skip) else }(\mathbb{C}_1);\\
\text{if}(\mathbb{C}_0)\{\text{if}(\mathbb{C}_1)\{\mathbb{C}_2;\text{return};)\}\}\mathbb{C}_3 &\mapsto \text{if}(\mathbb{C}_0\&\&\mathbb{C}_1)\text{ then }(\mathbb{C}_2)\text{ else }(\mathbb{C}_3);
\end{aligned}$$

These rules are valid for *all* possible values of the meta-variables in them; we do not assume anything about them. That we choose to restructure if statements with the 'return' keyword in them is due to the fact that `While`$^+$ has no concept of a return statement.

While loops

We provide the rules that handle the transformation of while loops. These are fairly simple rules, because while loops are syntactically almost identical in `While`$^+$ and C.

$$\begin{aligned}
\text{while}(\mathbb{C}_0)\{\mathbb{C}_1\text{while}(\mathbb{C}_2)\{\mathbb{C}_3\}\mathbb{C}_4\} &\mapsto \text{while}(\mathbb{C}_0)\text{do}(\mathbb{C}_1;\text{while}(\mathbb{C}_2)\text{do}(\mathbb{C}_3);\mathbb{C}_4);\\
\text{while}(\mathbb{C}_0)\{\mathbb{C}_1\} &\mapsto \text{while}(\mathbb{C}_0)\text{do}(\mathbb{C}_1);
\end{aligned}$$

Function arguments

Similarly to normal variables, we must transform the arguments of each function.

$$\begin{aligned}
\text{unsigned char }\mathbb{V}_0 &\mapsto \mathbb{V}_0 = \mathit{byte}\\
\text{unsigned char} * \mathbb{V}_0 &\mapsto \mathbb{V}_0 = \mathit{arr}\\
\text{unsigned char const} * \mathbb{V}_0 &\mapsto \mathbb{V}_0 = \mathit{arr}\\
\text{int }\mathbb{V}_0 &\mapsto \mathbb{V}_0 = \mathit{var}\\
\text{unsigned int }\mathbb{V}_0 &\mapsto \mathbb{V}_0 = \mathit{var}
\end{aligned}$$

Functions

We now handle the mapping of functions themselves. Note that because functions do not return anything directly in `While`$^+$, void functions can be mapped trivially. Functions that have a return value must introduce a new return variable, to which the return value is assigned.

$$\begin{aligned}
\text{void}\mathbb{V}_0(\mathbb{C}_0)\{\mathbb{C}_1\{\mathbb{C}_2\}\} &\mapsto \mathbb{V}_0(\mathbb{C}_0)(\mathbb{C}_1\mathbb{C}_2)\\
\text{return}) &\mapsto)
\end{aligned}$$

The double curly braces are a consequence of the fact that CIL introduces a new scope that separates variable declaration from the rest of the function body. We do not need this separation, so we discard it. We also discard the return statements of void functions, because those that are left are only those at the end of functions. Since the code we are transforming only consists of void functions, we do not need to worry about non-void functions. This does mean these rules only work for void functions; we will discuss what this means with respect to generality in chapter 6.

Variant rules

All the rules given above apply to both the reference and the variant implementation. The variant, however, makes use of `uint32` variables to hold data, instead of `uint8`.
Unfortunately, however, these rules are rather lengthy. Therefore, we have not included them completely in this thesis, but uploaded them to Zenodo[15], and refer to them in Section A.3.
Below, we shall briefly explain the variant rules.

The first four rules all deal with the 'splitting' of array elements. This is because the variant uses `__uint_32_t` as type, which has 32 bits, as opposed to the 8 bits in a `__uint_8_t`. Since `While`$^+$ only defines arrays to hold 8-bit bytes, we choose to split each element into four subsequent elements that are all 8 bits.
The fifth rule is relatively simple; it uses a compound xor assignment. This does not exist in `While`$^+$, so we transform it to its equivalent normal assignment.
The sixth rule removes a bunch of attributes that are applied to the function signature to make the compiler inline them properly, since `While`$^+$ does not require these concepts.
The seventh rule and onward all deal with the slightly more difficult case of array assignments. Essentially, each 'index' variable is multiplied by four, and then the whole assignment is duplicated four times. Each duplication, we increase the overall index by one to create four subsequent assignments. If rotation is present, we ensure it rotates cyclically in the aforementioned four elements.
The last rule is notably longer than the other rules, this is because it deals with rotations that can rotate from 1 bit to 31 bits. To deal with that, we need to distinguish four cases (1-8, 9-16, 17-24, 25-32), which we do with nested if-then-else statements.

Note that the rules defined above are by no means exhaustive for C, or even CIL. They are just the ones we need in order to transform the two Keccak implementation in CIL, therefore, it is likely that one would need to add additional rules to succesfully transform a different program.
Additionally, note that many of these rules make use of specific ways in which code is written in the reference and variant, and so are not necessar-

ily reusable on other programs.

4.3.2 Application of rules

We have written a program that mostly handles the transformation, by applying the rules defined in Section 4.3.1. There are, however, some rules that it cannot handle. Amongst these are the rules handling the implicit conversion of types. As such, we have applied those rules manually. The underlying mechanism (and the reason for its partial failure) and the code of this program are given in the Appendix A.5. The transformed `While`$^+$ code is also given in Sections A.1 and A.2.

4.3.3 Transformation post-processing

When looking at the `While`$^+$ code in the Appendix A.1, we can see and infer patterns that we can simplify further, by discarding redundant information. Although all of the simplifications can be applied automatically, for the purpose of this thesis, we apply them manually.
We distinguish the following categories below:

1. An assignment of the form $a = a$. Whatever the a might be, this never does anything of use. Typically, one can see this where a ternary assignment of the form $a = b\ ?\ c : a$ exists, as CIL lowers that to an if-else statement. An example can be seen in the function `rho`.

2. An if-then-else statement of the form `if(b) then (S_1)else (S_2)`, where b is either always true, or always false. This can be simplified to S_1 (if b is always true) or S_2 (if b is always false).

3. A while loop that is never entered, because its condition is never met. To simplify this, we remove the entire while loop.

4. A modulus statement $a \ \% \ b$, where a is always strictly smaller than b. We can see this in the reference implementation, because it contained a macro to easily traverse a one-dimensional array of size 25 as if it were a two-dimensional array of size five by five.
   Keccak requires wraparounds in their calculations, and so modulo statements are present in this macro. They are, however, not always actually necessary, as can be seen in the function `rho`.

5. An array initialization loop. In C code, uninitialized arrays contain whatever values happened to be stored at the memory addresses before the array was created. To have more predictable behaviour, array initialization is done (either through an initialization *statement*, or by looping through the entire array, but CIL lowers the first option to the second). In `While`$^+$, however, we define arrays as having each element

at value $\overline{0}$ by default. Therefore, if an array initialization loop is found which initializes an array with value $\overline{0}$, it can be removed.

6. An extension to the previous point; an array initialization need not happen in a loop. If an array is fully iterated over and all assigned a value of $\overline{0}$, when the state of the array is fully zeroed, we may remove the initialization statements.

7. An assignment to a variable that already has the same value. This happens mostly where CIL has separated the declaration of a variable and the usage, but the first usage of the variable was a declaration with its default value, as can be seen in the function 'permutation'.

It is important to note that the last four of these categories rely on constant expressions. Because of this, we can simply scan through the code for any statement that *might* be meaningless, while keeping track of all parameter names. If such a statement does not have a parameter in it, and meets the conditions listed above, we can simplify it.
For the first category, we can simply scan through all assignment statements, and remove them if they are of the form $a = a$, where a is an arbitrary statement[2].
The resulting `While`$^+$ code, after applying this post-processing, is given in the appendix, in Sections A.1 and A.2 respectively.

We can see that applying these simplifications does indeed have an effect; in the reference implementation the line count reduces from 191 to 174. This, of course, does not include all of the simplifications, as a modulus being removed does not mean a line is removed, as well.

4.4 Derivation trees

We apply the derivation rules on the obtained `While`$^+$ code to obtain the derivation trees.
We show the complete trees in the appendix, but go over the most important steps here.

[2]Note that we do *not* catch chases where $a = b$, such that $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![b]\!]s$, for some state s. This is because in the transformation, we do not wish to interpret the semantics of the code; that happens during the creation of derivation trees.

4.4.1 Keccak reference implementation

First pass

Because of its size, we provide the annotated program[3] in Appendix A.1. From this, we have obtained a state and environment, which we will use to obtain a valid derivation order.
First, we write out the environment slightly more explicitly, while at the same time leaving the parts of it out that we do not need to determine the order.

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\quad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \{\text{theta} \mapsto \emptyset, \text{rho} \mapsto \emptyset, \text{pi} \mapsto \emptyset, \text{chi} \mapsto \emptyset, \text{iota} \mapsto \emptyset,\\ &\quad \text{KeccakP200Round} \mapsto \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota}\},\\ &\quad \text{permutation} \mapsto \{\text{KeccakP200Round}\}\}\end{aligned}$$

Now we apply the general procedure for obtaining a derivation order defined in Section 3.2.1. We have already applied step 1, by applying the first pass derivation rules. For step 2, we define:

$$\text{derivation order} = ()$$

We apply step 3. There are several functions that can immediately be added to the derivation order. We end up with:

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\quad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \{\text{KeccakP200Round} \mapsto \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota}\},\\ &\quad \text{permutation} \mapsto \{\text{KeccakP200Round}\}\}\\ \text{derivation order} &= (\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota})\end{aligned}$$

We now apply step 4; we remove each element that is currently present in ‘derivation order’ from the co-domain of ‘function calls’. We end up with:

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\quad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \{\text{KeccakP200Round} \mapsto \emptyset,\\ &\quad \text{permutation} \mapsto \{\text{KeccakP200Round}\}\}\\ \text{derivation order} &= (\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota})\end{aligned}$$

There are now functions that map to the empty set, and the length of ‘derivation order’ (5) is not equal to the size of ‘functions’ (7), so we go back

[3]Recall that annnotated programs are a compact way to display derivation trees, which we define in Section 3.2.1

to step 3. After again applying step 3, we get:

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\qquad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \{\text{permutation} \mapsto \{\text{KeccakP200Round}\}\}\\ \text{derivation order} &= (\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota}, \text{KeccakP200Round})\end{aligned}$$

We apply step 4 once more:

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\qquad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \{\text{permutation} \mapsto \emptyset\}\\ \text{derivation order} &= (\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota}, \text{KeccakP200Round})\end{aligned}$$

Once again, there are still elements in 'function calls' that have the empty set as co-domain, and the length of 'function calls' (6) is not equal to those of 'functions' (7), so we go back to step 3. Then, we get:

$$\begin{aligned}\text{functions} &= \{\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota},\\ &\qquad \text{KeccakP200Round}, \text{permutation}\}\\ \text{function calls} &= \emptyset,\\ \text{derivation order} &= (\text{theta}, \text{rho}, \text{pi}, \text{chi}, \text{iota}, \text{KeccakP200Round},\\ &\qquad \text{permutation})\end{aligned}$$

And now, we see that there are no more mappings in 'function calls' to the empty set[4], and that the size of 'functions' equals that of 'derivation order'. We have therefore found a valid derivation order.

Second pass

Similarly to the first pass, the annotated proof is too lengthy to properly display in this thesis. Therefore, it is stored in the appendix (section A.1), but we will discuss the results of each function mapping here.
Recall that our derivation order is:

$$\begin{aligned}\text{derivation order} = (&\texttt{theta}, \texttt{rho}, \texttt{pi}, \texttt{chi}, \texttt{iota}, \texttt{KeccakP200Round},\\ &\texttt{permutation})\end{aligned}$$

The first function, `theta`, uses two array variables of length five. It xors each 'lane' of the input into the first array, and then mixes together the results of the first array variable into the second. We apply the xor operation to

[4]Recall that with (in)direct recursion, or function calls to functions that do not exist, we might end up with an undecidable derivation order. In this case, we would have had mappings in 'function calls', but none that map to the empty set.

the contents of the second array variable and the input array, cyclically (so we apply the xor operation to the sixth element of the input array and the first element of the second array variable, and so on).
In the state mapping, this simplifies into the xor operation being applied on each element, with a number of other elements, some of which are also rotated one bit.
The annotation results in a large number of requirements, but they all reduce to the requirements that:

1. The first array variable is of length five,
2. The second array variable is of length five,
3. The input array has at least a length of twenty-five.

The first two requirements can be trivially shown to always hold, so the sole requirement for this function is that the input array has a length of at least twenty-five.

The second function, rho, rotates the contents of the input array with an amount specified in a global variable (`KeccakRhoOffsets`).
As such, the state mapping is not very complex, and 'simply' shows each element being rotated by the specified amount (if any). Although one of the requirements that remains after simplification is that the input array has at least a length of twenty-five, there are more requirements than in the previous function. However, the additional requirements simplify to the requirement that the `KeccakRhoOffsets` array must have a length of at least twenty-five. Because the `KeccakRhoOffsets` array is a global variable, we know its length already, and can therefore show that the requirement holds irrespective of the parameters chosen. Thus, we can omit this requirement entirely.

The next function is `pi`. It shuffles the order of the input array using an array variable of length twenty-five. As such, the state mapping simplifies into a simple re-ordering of elements. It comes as no surprise that the requirements also simplify to the sole requirement that the input array has at least a length of twenty-five.

Then, we come across `chi`. This function attempts to break any correlation within 'lanes' by combining data from three subsequent elements, and assigning it to a single element in the input array. This is done using a single array variable, which has a length of five.
The state mapping shows each element is indeed transformed into an expression that depends on its own value, and the next two elements, in a cyclic manner.

As before, the requirements simplify to the sole requirement that the input array must have a length of at least twenty-five.

Next in the derivation order is the function iota. This function is relatively short; it xors the first element with some value contained in `KeccakRoundConstants`, as defined by the second parameter.
As such, the function mapping is exceedingly short. However, we do have two requirements. The first requirement is that the input array has a length of at least one, but the second is that the indexRound variable is no larger than seventeen.

The penultimate function is `KeccakP200Round`.
It 'simply' calls each of the previous functions, in order, to execute a single round of the permutation. The requirements of this function are thus the requirements of all previous functions combined:

1. That the input array has a length of at least twenty-five,
2. That the `indexRound` parameter is at most seventeen.

However, the mapping of this function is lengthy to the point that we cannot properly reason about it. We will discuss possible solutions to this in Section 6.1.

The last function is permutation. However, this function applies `KeccakP200Round` eighteen times upon the input array. This results in a state mapping so large that the tool we use to automate the process cannot handle it. Therefore, we cannot apply the second pass derivation rules on this function. As with the previous function, we will discuss possible solutions to this problem in Section 6.1.

4.4.2 Keccak variant implementation

First pass

Because of its size, we provide the annotated program in the Appendix A.2. From this, we have obtained a state and environment, which we will use to obtain a valid derivation order.
First, we write out the environment slightly more explicitly, while at the same time leaving the parts of it out that we do not need to determine the

order.

$$\begin{aligned}
\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\
&\quad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\
\text{function calls} &= \{\text{theta\_4} \mapsto \emptyset,\\
&\quad \text{rho\_4} \mapsto \emptyset,\\
&\quad \text{pi\_4} \mapsto \emptyset,\\
&\quad \text{chi\_4} \mapsto \emptyset,\\
&\quad \text{iota\_4} \mapsto \emptyset,\\
&\quad \text{KeccakP200Round\_4} \mapsto \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4}\},\\
&\quad \text{permutation\_4} \mapsto \{\text{KeccakP200Round\_4}\}\}
\end{aligned}$$

As with the reference implementation, we start with an empty derivation order, to obtain a valid order:

$$\text{derivation order} = ()$$

We have already applied the first two steps, and so arrive at the third:

$$\begin{aligned}
\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\
&\quad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\
\text{function calls} &= \{\text{KeccakP200Round\_4} \mapsto \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4}\},\\
&\quad \text{permutation\_4} \mapsto \{\text{KeccakP200Round\_4}\}\},\\
\text{derivation order} &= (\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4})
\end{aligned}$$

Where we have immediately added a number of functions that mapped to $\emptyset$ to the derivation order. Next, we apply step 4.

$$\begin{aligned}
\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\
&\quad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\
\text{function calls} &= \{\text{KeccakP200Round\_4} \mapsto \emptyset,\\
&\quad \text{permutation\_4} \mapsto \{\text{KeccakP200Round\_4}\}\},\\
\text{derivation order} &= (\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4})
\end{aligned}$$

Having done so, we have removed all elements of the derivation order from any elements in the co-domain of 'function calls'. We now arrive at step 5, but the length of the derivation sequence does not yet equal the length of the 'functions' set and there are still functions that map to the empty set in 'function calls', so we go back to step 3.

$$\begin{aligned}
\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\
&\quad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\
\text{function calls} &= \{\text{permutation\_4} \mapsto \{\text{KeccakP200Round\_4}\}\},\\
\text{derivation order} &= (\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4}, \text{KeccakP200Round\_4})
\end{aligned}$$

We have added another function to the derivation order, so we may now apply step 4 again:

$$\begin{aligned}\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\ &\qquad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\ \text{function calls} &= \{\text{permutation\_4} \mapsto \emptyset\},\\ \text{derivation order} &= (\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4}, \text{KeccakP200Round\_4})\end{aligned}$$

Again, we are in step 5. And since we can see that the last element in 'function calls' maps to the empty set, we go back to step 3 one last time:

$$\begin{aligned}\text{functions} &= \{\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4},\\ &\qquad \text{KeccakP200Round\_4}, \text{permutation\_4}\}\\ \text{function calls} &= \emptyset,\\ \text{derivation order} &= (\text{theta\_4}, \text{rho\_4}, \text{pi\_4}, \text{chi\_4}, \text{iota\_4}, \text{KeccakP200Round\_4},\\ &\qquad \text{permutation\_4})\end{aligned}$$

Now we apply step 4 again - though we do so on an empty set of function calls - and arrive at step 5. Here, we find that the length 'derivation order' is equal to the length of the 'functions' set. So we have found our valid derivation order.

Second pass

Similarly to the first pass, the annotated proof is too lengthy to properly display in this thesis. Therefore, it is stored in the appendix (section A.2), but we will discuss the results of each function mapping here.
Recall that our derivation order is:

$$\begin{aligned}\text{derivation order} = (&\text{theta}_4, \text{rho}_4, \text{pi}_4, \text{chi}_4, \text{iota}_4, \text{KeccakP200Round}_4,\\ &\text{permutation})\end{aligned}$$

The first function, theta_4, uses two array variables of length twenty. It xors each 'lane' of the input into the first array, and then mixes together the results of the first array variable into the second. The xor operation is applied to the contents of this second array variable, and the input array, cyclically (so the xor operation is applied to the 21st element of the input array and the first element of the array variable, and so on).
In the state mapping, this simplifies into the xor operation being applied on each element directly with a number of other elements, which the additional elements are sometimes rotated by four bits.
The annotation results in a large number of requirements, but they all reduce to the requirements that:

1. The first array variable is of length twenty,

2. The second array variable is of length twenty,
3. The input array has at least a length of one hundred.

The first two requirements can be trivially shown to always hold, so the sole requirement for this function is that the input array has a length of at least one hundred.

The second function, rho_4, rotates the contents of the input array with an amount specified in a global variable (KeccakRhoOffsets).
However, to deal with the fact that a single word is 32 bits, rotation is more complex. A single rotation therefore affects four consecutive elements in the input array. Additionally, if a rotation happens to align with byte boundaries (8, 16, 24 or 32), the mapping does not explicitly shift, but reorders the bytes that make up the word.
Although one of the requirements that remains after simplification is that the input array has at least a length of one hundred, there are more requirements than in the previous function. However, the additional requirements simplify to the requirement that the KeccakRhoOffsets array must have a length of at least twenty-five. Because that array is a global variable read in in the first pass, we knonw its length. As such, we can show the requirement holds irrespective of the chosen parameters and therefore omit it.

The next function is pi_4. It shuffles the order of the input array using an array variable of length one hundred. As such, the state mapping simplifies into a simple re-ordering of elements. It comes as no surprise that the requirements also simplify to the sole requirement that the input array has at least a length of one hundred.

Then, we come across chi_4. This function attempts to break any correlation within 'lanes' by combining data from three subsequent elements, and assigning it to a single element in the input array. This is done using a single array variable, which has a length of twenty.
The state mapping shows each element is indeed transformed into an expression that depends on its own value, and the next two elements, in a cyclic manner.
As before, the requirements simplify to the sole requirement that the input array must have a length of at least one hundred.

Next up is the function iota_4. This function is relatively short; it xors the first four elements with some values contained in KeccakRoundConstants_4, as defined by the second parameter indexRound.
As such, the function mapping is exceedingly short. However, we do have two requirements. The first requirement is that the input array has a length of at least four, but the second is that the indexRound variable is no larger

than seventeen.

The penultimate function is KeccakP200Round_4.
It 'simply' calls each of the previous functions, in order, to execute a single round of the permutation. The requirements of this function are thus the requirements of all previous functions combined:

1. That the input array has a length of at least one hundred,
2. That the indexRound parameter is at most seventeen.

However, the mapping of this function is lengthy to the point that we cannot properly reason about it. We will discuss possible solutions to this in Section 6.1.

The last function is permutation_4. However, this function applies KeccakP200Round_4 eighteen times upon the input array. This results in a state mapping so large, that the tool we use to automate the process cannot handle it. Therefore, we cannot apply the second pass derivation rules on this function. Similarly to the previous function, we will discuss possible solutions to this problem in Section 6.1.

4.5 Comparing mappings

After completing the derivation trees of both variants up until their respective permutation functions, we now have two sets of state mappings (and associated read-only states containing constants!) which we must compare to determine whether the implementations operate the same way, or not.
Due to the technical limitation that is mentioned previously (and which will be elaborated on in the discussion in Section 6), we have made state mappings of all functions, except `permutation` and `permutation_4`.
However, both `permutation` and `permutation_4` call their respective round functions 18 times on the provided state. So if we can show that their respective round functions operate identically, then so must these functions.

Here, however, it seems our hopes of proving semantical equivalence are futile; the requirements of `KeccakP200Round_4` and `KeccakP200Round` differ! Indeed, the requirement of `KeccakP200Round_4` is that $len(A) \geq 100 \wedge indexRound < 18$, whereas the requirement of `KeccakP200Round` is that $len(A) \geq 25 \wedge indexRound < 18$.

Whatever their function mappings, it is intuitively obvious that they cannot operate identically; it is trivial to create an array that will meet the requirements for one function, but not the other (say, an array of length 25).
But this difference should not come as a surprise. Indeed, the optimized

variant is just that; an *optimized* variant. There are multiple optimizations at play, but the one that is semantically noteworthy is that the word size of the optimized variant is four times as large as the reference implementation. If we can show that all differences in state mapping are explainable through this fact, then we can show that the optimization is semantically analogous to the reference.

In order to do this, it is easier to look at the individual diffusion functions than to look at the round function as a whole, due to the fact that the state mapping of the round function is not simplified whatsoever.

4.5.1 Iota

Starting with the shortest function, let us compare the state mappings of `iota` to `iota_4`.

$$\begin{aligned}
iota_4 \mapsto \{&len(A) \geq 4 \wedge indexRound \leq 17, (A, indexRound) \rightarrow \\
&\quad (a_0\hat{\ } KeccakRoundConstants_4[indexRound * 4], \\
&\quad\ a_1\hat{\ } KeccakRoundConstants_4[indexRound * 4 + 1], \\
&\quad\ a_2\hat{\ } KeccakRoundConstants_4[indexRound * 4 + 2], \\
&\quad\ a_3\hat{\ } KeccakRoundConstants_4[indexRound * 4 + 3] \\
&)\} \\
iota \mapsto \{&len(A) \geq 1 \wedge indexRound \leq 17, (A, indexRound) \rightarrow \\
&\quad (a_0\hat{\ } KeccakRoundConstants[indexRound])) \\
&\}
\end{aligned}$$

Of course, we can explain the difference in the first requirement by the factor of four; arrays in `While`$^+$ are of bytes, and thus we have split up an array of 32-bit elements, into an array of 8-bit elements that has four times as many elements.
The second requirement is identical. This is due to the fact that regardless of the variant, we always have 18 rounds. Though the round constant array has 72 elements in the optimized variant (as opposed to 18), we do not mention this in the requirement because its length is a constant. Thus, the requirement simplifies from $indexRound * 4 + 3 < len(KeccakRoundconstants_4) * 4 + 4$ to $indexRound * 4 < 68$ and finally $17 \geq indexRound$.
The state mapping itself is also analogous, as we clearly see that instead of xor-ring just the first element, the variant applies the xor operation on the first four elements.
One thing remains to be shown; are the elements of `KeccakRoundConstants` analogous to those of `KeccakRoundConstants_4`? Let us investigate.

$$\begin{aligned}
KeccakRoundConstants &\mapsto (\overline{1}, \overline{130}, \overline{138}, \overline{0}, \overline{139}, \overline{1}, \overline{129}, \overline{9}, \overline{138}, \overline{136}, \overline{9}, \overline{10}, \overline{139}, \overline{139}, \overline{137}, \overline{3}, \overline{2}, \overline{128}),\\
KeccakRoundConstants_4 &\mapsto (\overline{255}, \overline{0}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{0}, \overline{240}, \overline{240}, \overline{0}, \overline{240}, \overline{240}, \overline{0}, \overline{0}, \overline{0}, \overline{0}, \overline{240}, \overline{0},\\
&\overline{240}, \overline{255}, \overline{15}, \overline{0}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{0}, \overline{15}, \overline{240}, \overline{15}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{240}, \overline{240},\\
&\overline{240}, \overline{0}, \overline{240}, \overline{0}, \overline{240}, \overline{15}, \overline{0}, \overline{0}, \overline{240}, \overline{240}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{240}, \overline{255}, \overline{240}, \overline{240},\\
&\overline{0}, \overline{255}, \overline{240}, \overline{0}, \overline{240}, \overline{255}, \overline{255}, \overline{0}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{0}, \overline{0}, \overline{240}, \overline{0}, \overline{0}, \overline{0})
\end{aligned}$$

First, we shall look at the first element of each array. For the reference, this is $\overline{1}$. For the variant, this is $(\overline{255}, \overline{0}, \overline{0}, \overline{0})$. Interpreting that value as a single 32-bit value yields 255, so the factor of four is not a simple multiplication. But it is also not a case of repeating the same bits 4 times; then we would have seen $(\overline{1}, \overline{1}, \overline{1}, \overline{1})$.
Does this mean they are not analogous? Not necessarily. As can be seen in the paper that describes the variant [9], the environment for which it is optimized works by interleaving bits. And indeed, we can see this pattern emerge:

$$\begin{aligned}
\overline{0}, \overline{0}, \overline{0}, \overline{255} &= (\underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}), (\underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}), (\underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}), (\underline{1}, \underline{1}, \underline{1}, \underline{1}, \underline{1}, \underline{1}, \underline{1}, \underline{1})\\
\overline{1} &= (\underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{0}, \underline{1}).
\end{aligned}$$

So we may conclude that these functions are analogous.

4.5.2 Chi

On to the second function; we will compare chi and chi_4. As before, we first display the state mappings:

$$
\begin{aligned}
&chi_4 \mapsto \{len(A) \geq 100, (A) \rightarrow (\\
&\quad a_0 \;\hat{}\; \neg a_4 \wedge a_8, a_1 \;\hat{}\; \neg a_5 \wedge a_9, a_2 \;\hat{}\; \neg a_6 \wedge a_{10}, a_3 \;\hat{}\; \neg a_7 \wedge a_{11},\\
&\quad a_4 \;\hat{}\; \neg a_8 \wedge a_{12}, a_5 \;\hat{}\; \neg a_9 \wedge a_{13}, a_6 \;\hat{}\; \neg a_{10} \wedge a_{14}, a_7 \;\hat{}\; \neg a_{11} \wedge a_{15},\\
&\quad a_8 \;\hat{}\; \neg a_{12} \wedge a_{16}, a_9 \;\hat{}\; \neg a_{13} \wedge a_{17}, a_{10} \;\hat{}\; \neg a_{14} \wedge a_{18}, a_{11} \;\hat{}\; \neg a_{15} \wedge a_{19},\\
&\quad a_{12} \;\hat{}\; \neg a_{16} \wedge a_0, a_{13} \;\hat{}\; \neg a_{17} \wedge a_1, a_{14} \;\hat{}\; \neg a_{18} \wedge a_2, a_{15} \;\hat{}\; \neg a_{19} \wedge a_3,\\
&\quad a_{16} \;\hat{}\; \neg a_0 \wedge a_4, a_{17} \;\hat{}\; \neg a_1 \wedge a_5, a_{18} \;\hat{}\; \neg a_2 \wedge a_6, a_{19} \;\hat{}\; \neg a_3 \wedge a_7,\\
&\quad a_{20} \;\hat{}\; \neg a_{24} \wedge a_{28}, a_{21} \;\hat{}\; \neg a_{25} \wedge a_{29}, a_{22} \;\hat{}\; \neg a_{26} \wedge a_{30}, a_{23} \;\hat{}\; \neg a_{27} \wedge a_{31},\\
&\quad a_{24} \;\hat{}\; \neg a_{28} \wedge a_{32}, a_{25} \;\hat{}\; \neg a_{29} \wedge a_{33}, a_{26} \;\hat{}\; \neg a_{30} \wedge a_{34}, a_{27} \;\hat{}\; \neg a_{31} \wedge a_{35},\\
&\quad a_{28} \;\hat{}\; \neg a_{32} \wedge a_{36}, a_{29} \;\hat{}\; \neg a_{33} \wedge a_{37}, a_{30} \;\hat{}\; \neg a_{34} \wedge a_{38}, a_{31} \;\hat{}\; \neg a_{35} \wedge a_{39},\\
&\quad a_{32} \;\hat{}\; \neg a_{36} \wedge a_{20}, a_{33} \;\hat{}\; \neg a_{37} \wedge a_{21}, a_{34} \;\hat{}\; \neg a_{38} \wedge a_{22}, a_{35} \;\hat{}\; \neg a_{39} \wedge a_{23},\\
&\quad a_{36} \;\hat{}\; \neg a_{20} \wedge a_{24}, a_{37} \;\hat{}\; \neg a_{21} \wedge a_{25}, a_{38} \;\hat{}\; \neg a_{22} \wedge a_{26}, a_{39} \;\hat{}\; \neg a_{23} \wedge a_{27},\\
&\quad a_{40} \;\hat{}\; \neg a_{44} \wedge a_{48}, a_{41} \;\hat{}\; \neg a_{45} \wedge a_{49}, a_{42} \;\hat{}\; \neg a_{46} \wedge a_{50}, a_{43} \;\hat{}\; \neg a_{47} \wedge a_{51},\\
&\quad a_{44} \;\hat{}\; \neg a_{48} \wedge a_{52}, a_{45} \;\hat{}\; \neg a_{49} \wedge a_{53}, a_{46} \;\hat{}\; \neg a_{50} \wedge a_{54}, a_{47} \;\hat{}\; \neg a_{51} \wedge a_{55},\\
&\quad a_{48} \;\hat{}\; \neg a_{52} \wedge a_{56}, a_{49} \;\hat{}\; \neg a_{53} \wedge a_{57}, a_{50} \;\hat{}\; \neg a_{54} \wedge a_{58}, a_{51} \;\hat{}\; \neg a_{55} \wedge a_{59},\\
&\quad a_{52} \;\hat{}\; \neg a_{56} \wedge a_{40}, a_{53} \;\hat{}\; \neg a_{57} \wedge a_{41}, a_{54} \;\hat{}\; \neg a_{58} \wedge a_{42}, a_{55} \;\hat{}\; \neg a_{59} \wedge a_{43},\\
&\quad a_{56} \;\hat{}\; \neg a_{40} \wedge a_{44}, a_{57} \;\hat{}\; \neg a_{41} \wedge a_{45}, a_{58} \;\hat{}\; \neg a_{42} \wedge a_{46}, a_{59} \;\hat{}\; \neg a_{43} \wedge a_{47},\\
&\quad a_{60} \;\hat{}\; \neg a_{64} \wedge a_{68}, a_{61} \;\hat{}\; \neg a_{65} \wedge a_{69}, a_{62} \;\hat{}\; \neg a_{66} \wedge a_{70}, a_{63} \;\hat{}\; \neg a_{67} \wedge a_{71},\\
&\quad a_{64} \;\hat{}\; \neg a_{68} \wedge a_{72}, a_{65} \;\hat{}\; \neg a_{69} \wedge a_{73}, a_{66} \;\hat{}\; \neg a_{70} \wedge a_{74}, a_{67} \;\hat{}\; \neg a_{71} \wedge a_{75},\\
&\quad a_{68} \;\hat{}\; \neg a_{72} \wedge a_{76}, a_{69} \;\hat{}\; \neg a_{73} \wedge a_{77}, a_{70} \;\hat{}\; \neg a_{74} \wedge a_{78}, a_{71} \;\hat{}\; \neg a_{75} \wedge a_{79},\\
&\quad a_{72} \;\hat{}\; \neg a_{76} \wedge a_{60}, a_{73} \;\hat{}\; \neg a_{77} \wedge a_{61}, a_{74} \;\hat{}\; \neg a_{78} \wedge a_{62}, a_{75} \;\hat{}\; \neg a_{79} \wedge a_{63},\\
&\quad a_{76} \;\hat{}\; \neg a_{60} \wedge a_{64}, a_{77} \;\hat{}\; \neg a_{61} \wedge a_{65}, a_{78} \;\hat{}\; \neg a_{62} \wedge a_{66}, a_{79} \;\hat{}\; \neg a_{63} \wedge a_{67},\\
&\quad a_{80} \;\hat{}\; \neg a_{84} \wedge a_{88}, a_{81} \;\hat{}\; \neg a_{85} \wedge a_{89}, a_{82} \;\hat{}\; \neg a_{86} \wedge a_{90}, a_{83} \;\hat{}\; \neg a_{87} \wedge a_{91},\\
&\quad a_84 \;\hat{}\; \neg a_88 \wedge a_92, a_85 \;\hat{}\; \neg a_89 \wedge a_{93}, a_{86} \;\hat{}\; \neg a_{90} \wedge a_{94}, a_{87} \;\hat{}\; \neg a_{91} \wedge a_{95},\\
&\quad a_{88} \;\hat{}\; \neg a_{92} \wedge a_{96}, a_{89} \;\hat{}\; \neg a_{93} \wedge a_{97}, a_{90} \;\hat{}\; \neg a_{94} \wedge a_{98}, a_{91} \;\hat{}\; \neg a_{95} \wedge a_{99},\\
&\quad a_{92} \;\hat{}\; \neg a_{96} \wedge a_{80}, a_{93} \;\hat{}\; \neg a_{97} \wedge a_{81}, a_{94} \;\hat{}\; \neg a_{98} \wedge a_{82}, a_{95} \;\hat{}\; \neg a_{99} \wedge a_{83},\\
&\quad a_{96} \;\hat{}\; \neg a_{80} \wedge a_{84}, a_{97} \;\hat{}\; \neg a_{81} \wedge a_{85}, a_{98} \;\hat{}\; \neg a_{82} \wedge a_{86}, a_{99} \;\hat{}\; \neg a_{83} \wedge a_{87}\\
&)\},
\end{aligned}
$$

$chi \mapsto \{len(A) \geq 25, (A) \rightarrow ($

$a_0 \,\hat{}\, \neg a_1 \wedge a_2, a_1 \,\hat{}\, \neg a_2 \wedge a_3, a_2 \,\hat{}\, \neg a_3 \wedge a_4, a_3 \,\hat{}\, \neg a_4 \wedge a_0, a_4 \,\hat{}\, \neg a_0 \wedge a_1,$

$a_5 \,\hat{}\, \neg a_6 \wedge a_7, a_6 \,\hat{}\, \neg a_7 \wedge a_8, a_7 \,\hat{}\, \neg a_8 \wedge a_9, a_8 \,\hat{}\, \neg a_9 \wedge a_5, a_9 \,\hat{}\, \neg a_5 \wedge a_6,$

$a_{10} \,\hat{}\, \neg a_{11} \wedge a_{12}, a_{11} \,\hat{}\, \neg a_{12} \wedge a_{13}, a_{12} \,\hat{}\, \neg a_{13} \wedge a_{14}, a_{13} \,\hat{}\, \neg a_{14} \wedge a_{10},$

$a_{14} \,\hat{}\, \neg a_{10} \wedge a_{11}, a_{15} \,\hat{}\, \neg a_{16} \wedge a_{17}, a_{16} \,\hat{}\, \neg a_{17} \wedge a_{18}, a_{17} \,\hat{}\, \neg a_{18} \wedge a_{19},$

$a_{18} \,\hat{}\, \neg a_{19} \wedge a_{15}, a_{19} \,\hat{}\, \neg a_{15} \wedge a_{16}, a_{20} \,\hat{}\, \neg a_{21} \wedge a_{22}, a_{21} \,\hat{}\, \neg a_{22} \wedge a_{23},$

$a_{22} \,\hat{}\, \neg a_{23} \wedge a_{24}, a_{23} \,\hat{}\, \neg a_{24} \wedge a_{20}, a_{24} \,\hat{}\, \neg a_{20} \wedge a_{21}$

$)\}$

As before, the requirements differ by exactly a factor of four, and each element in the reference code matches up to four elements in the variant.
For example, look at the first element in the reference mapping ($a_0 \,\hat{}\, \neg a_1 \wedge a_2$) and the first four elements in the variant mapping ($a_0 \,\hat{}\, \neg a_4 \wedge a_8$, $a_1 \,\hat{}\, \neg a_5 \wedge a_9$, $a_2 \,\hat{}\, \neg a_6 \wedge a_{10}$, $a_3 \,\hat{}\, \neg a_7 \wedge a_{11}$). As we can see, the indices of the variant are exactly those of the reference, multiplied by four and then repeated four times such that each subsequent index is increased by one each time. Since the mapping does not refer to read-only variables, we can say with certainty that these two functions are semantically analogous.

4.5.3 Pi

The next function to look at is `pi`. It shuffles the contents of the array it receives as argument. As before, we look at the state mappings:

$$
\begin{aligned}
& pi_4 \mapsto \{len(A) \geq 100, (A) \rightarrow (\\
& \quad a_0, a_1, a_2, a_3, a_{24}, \\
& \quad a_{25}, a_{26}, a_{27}, a_{48}, a_{49}, \\
& \quad a_4, a_5, a_6, a_7, a_{74}, \\
& \quad a_{75}, a_{96}, a_{97}, a_{98}, a_{99}, \\
& \quad a_{12}, a_{13}, a_{14}, a_{15}, a_{36}, \\
& \quad a_{37}, a_{38}, a_{39}, a_{40}, a_{41}, \\
& \quad a_{42}, a_{43}, a_{64}, a_{65}, a_{66}, \\
& \quad a_{67}, a_{88}, a_{89}, a_{90}, a_{91}, \\
& \quad a_{40}, a_{41}, a_{42}, a_{43}, a_{28}, \\
& \quad a_{29}, a_{30}, a_{31}, a_{52}, a_{53}, \\
& \quad a_{54}, a_{55}, a_{76}, a_{77}, a_{78}, \\
& \quad a_{79}, a_{80}, a_{81}, a_{82}, a_{83}, \\
& \quad a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, \\
& \quad a_{21}, a_{22}, a_{23}, a_{44}, a_{45}, \\
& \quad a_{46}, a_{47}, a_{68}, a_{69}, a_{70}, \\
& \quad a_{71}, a_{92}, a_{93}, a_{94}, a_{95}, \\
& \quad a_8, a_9, a_{10}, a_{11}, a_{32}, \\
& \quad a_{33}, a_{34}, a_{35}, a_{56}, a_{57}, \\
& \quad a_{58}, a_{59}, a_{60}, a_{61}, a_{62}, \\
& \quad a_{63}, a_{84}, a_{85}, a_{86}, a_{87}, \\
&)\} \\
& pi \mapsto \{len(A) \geq 25, (A) \rightarrow (\\
& \quad a_0, a_6, a_{12}, a_{18}, a_{24}, \\
& \quad a_{15}, a_{21}, a_2, a_8, a_{14}, \\
& \quad a_5, a_{11}, a_{17}, a_{23}, a_4, \\
& \quad a_{20}, a_1, a_7, a_{13}, a_{19}, \\
& \quad a_{10}, a_{16}, a_{22}, a_3, a_9 \\
&)\}
\end{aligned}
$$

Similarly to the mappings of `chi` and `chi_4`, these are analogous. In fact, this is probably the easiest function to compare, as the mapping does not introduce any new values, just shuffling them around. As such, we can trivially see that the first index on the reference (a_0) matches up to the first four indices on the variant mapping (a_0, a_1, a_2, a_3) as expected.

4.5.4 Rho

The next functions to compare are `rho` and `rho_4`. They are more interesting, due to the fact they rotate each element by a fixed offset. This not only means the mapping refers to a read-only variable, but it also means that the mappings differ in more respects. For instance, if a 32-bit variable is rotated 16 bits, then the mapping should denote four elements, *none* of which are rotated themselves. Instead, the ordering of the four elements should be “rotated” two elements to the left.

Let us observe the state mappings:

$rho_4 \mapsto \{len(A) \geq 100, (A) \rightarrow ($

$a_0, a_1, a_2, a_3, a_4 \ll 4 \;\hat{}\; a_5 \gg 4,$

$a_5 \ll 4 \;\hat{}\; a_6 \gg 4, a_6 \ll 4 \;\hat{}\; a_7 \gg 4, a_7 \ll 4 \;\hat{}\; a_4 \gg 4, \overline{0} \;\hat{}\; a_{11}, \overline{0} \;\hat{}\; a_8,$

$\overline{0} \;\hat{}\; a_9, \overline{0} \;\hat{}\; a_{10}, a_{13} \ll 4 \;\hat{}\; a_{14} \gg 4, a_{14} \ll 4 \;\hat{}\; a_{15} \gg 4,$

$a_{15} \ll 4 \;\hat{}\; a_{12} \gg 4, a_{12} \ll 4 \;\hat{}\; a_{13} \gg 4, \overline{0} \;\hat{}\; a_{18}, \overline{0} \;\hat{}\; a_{19}, \overline{0} \;\hat{}\; a_{16}, \overline{0} \;\hat{}\; a_{17},$

$\overline{0} \;\hat{}\; a_{22}, \overline{0} \;\hat{}\; a_{23}, \overline{0} \;\hat{}\; a_{22}, \overline{0} \;\hat{}\; a_{21}, \overline{0} \;\hat{}\; a_{26},$

$\overline{0} \;\hat{}\; a_{27}, \overline{0} \;\hat{}\; a_{24}, \overline{0} \;\hat{}\; a_{25}, \overline{0} \;\hat{}\; a_{31}, \overline{0} \;\hat{}\; a_{27},$

$\overline{0} \;\hat{}\; a_{28}, \overline{0} \;\hat{}\; a_{29}, a_{35} \ll 4 \;\hat{}\; a_{32} \gg 4, a_{32} \ll 4 \;\hat{}\; a_{33} \gg 4, a_{33} \ll 4 \;\hat{}\; a_{34} \gg 4,$

$a_{34} \ll 4 \;\hat{}\; a_{35} \gg 4, a_{37} \ll 4 \;\hat{}\; a_{38} \gg 4, a_{38} \ll 4 \;\hat{}\; a_{39} \gg 4,$

$a_{39} \ll 4 \;\hat{}\; a_{36} \gg 4, a_{36} \ll 4 \;\hat{}\; a_{37} \gg 4,$

$\overline{0} \;\hat{}\; a_{42}, \overline{0} \;\hat{}\; a_{43}, \overline{0} \;\hat{}\; a_{40}, \overline{0} \;\hat{}\; a_{41}, a_{45} \ll 4 \;\hat{}\; a_{46} \gg 4,$

$a_{46} \ll 4 \;\hat{}\; a_{47} \gg 4, a_{47} \ll 4 \;\hat{}\; a_{44} > 4, a_{45} \ll 4 \;\hat{}\; a_{46} \gg 4, a_{48}, a_{49},$

$a_{50}, a_{51}, a_{52} \ll 4 \;\hat{}\; a_{53} \gg 4, a_{53} \ll 4 \;\hat{}\; a_{54} \gg 4, a_{54} \ll 4 \;\hat{}\; a_{55} \gg 4,$

$a_{55} \ll 4 \;\hat{}\; a_{52} \gg 4, a_{59} \ll 4 \;\hat{}\; a_{56} \gg 4, a_{56} \ll 4 \;\hat{}\; a_{57} \gg 4,$

$a_{57} \ll 4 \;\hat{}\; a_{58} \gg 4, a_{58} \ll 4 \;\hat{}\; a_{59} \gg 4,$

$a_{60} \ll 4 \;\hat{}\; a_{61} \gg 4, a_{61}, a_{62} \ll 4 \;\hat{}\; a_{63} \gg 4, a_{63} \ll 4 \;\hat{}\; a_{60} \gg 4, a_{66} \ll 4 \;\hat{}\; a_{67} \gg 4,$

$a_{67} \ll 4 \;\hat{}\; a_{64} \gg 4, a_{64} \ll 4 \;\hat{}\; a_{65} \gg 4, a_{65} \ll 4 \;\hat{}\; a_{66} \gg 4, a_{71} \ll 4 \;\hat{}\; a_{68} \gg 4,$

$a_{68} \ll 4 \;\hat{}\; a_{69} \gg 4,$

$a_{69} \ll 4 \;\hat{}\; a_{70} \gg 4, a_{70} \ll 4 \;\hat{}\; a_{71} \gg 4, a_{74} \ll 4 \;\hat{}\; a_{75} \gg 4, a_{75} \ll 4 \;\hat{}\; a_{72} \gg 4,$

$a_{72} \ll 4 \;\hat{}\; a_{73} \gg 4,$

$a_{73} \ll 4 \;\hat{}\; a_{74} \gg 4, a_{76}, a_{77}, a_{78}, a_{79},$

$\overline{0} \;\hat{}\; a_{81}, \overline{0} \;\hat{}\; a_{82}, \overline{0} \;\hat{}\; a_{83}, \overline{0} \;\hat{}\; a_{80}, \overline{0} \;\hat{}\; a_{85},$

$\overline{0} \;\hat{}\; a_{86}, \overline{0} \;\hat{}\; a_{87}, \overline{0} \;\hat{}\; a_{84}, a_{90} \ll 4 \;\hat{}\; a_{91} \gg 4, a_{91} \ll 4 \;\hat{}\; a_{88} \gg 4,$

$a_{88} \ll 4 \;\hat{}\; a_{89} \gg 4, a_{89} \ll 4 \;\hat{}\; a_{90} \gg 4, a_{92}, a_{93}, a_{94},$

$a_{95}, a_{96}, \overline{0} \;\hat{}\; a_{97}, \overline{0} \;\hat{}\; a_{98}, \overline{0} \;\hat{}\; a_{99}$

$)\}$

$rho \mapsto \{len(A) \geq 25, (A) \rightarrow ($

$a_0, (a_1 \ll 1) \;\hat{}\; (a_1 \gg 7), (a_2 \ll 6) \;\hat{}\; (a_2 \gg 2), (a_3 \ll 4) \;\hat{}\; (a_3 \gg 4), a_4,$

$(a_5 \ll 4) \;\hat{}\; (a_5 \gg 4), (a_6 \ll 4) \;\hat{}\; (a_6 \gg 4), (a_7 \ll 6) \;\hat{}\; (a_7 \gg 2),$

$(a_8 \ll 7) \;\hat{}\; (a_8 \gg 1), (a_9 \ll 4) \;\hat{}\; (a_9 \gg 4),$

$(a_{10} \ll 3) \;\hat{}\; (a_{10} \gg 5), (a_{11} \ll 2) \;\hat{}\; (a_{11} \gg 6), (a_{12} \ll 3) \;\hat{}\; (a_{12} \gg 5),$

$(a_{13} \ll 1) \;\hat{}\; (a_{13} \gg 7), (a_{14} \ll 7) \;\hat{}\; (a_{14} \gg 1),$

$(a_{15} \ll 1) \;\hat{}\; (a_{15} \gg 7), (a_{16} \ll 5) \;\hat{}\; (a_{16} \gg 3), (a_{17} \ll 7) \;\hat{}\; (a_{17} \gg 1),$

$(a_{18} \ll 5) \;\hat{}\; (a_{18} \gg 3), a_{19},$

$(a_{20} \ll 2) \;\hat{}\; (a_{20} \gg 6), (a_{21} \ll 2) \;\hat{}\; (a_{21} \gg 6), (a_{22} \ll 5) \;\hat{}\; (a_{22} \gg 3),$

$a_{23}, (a_{24} \ll 6) \;\hat{}\; (a_{24} \gg 2)$

$)\}$

It might seem confusing that there is no reference to `KeccakRhoOffsets` and `KeccakRhoOffsets_4` in the mapping, since it is used in the functions. Like with the Iota functions, the requirements can be shown to always hold and can therefore be omitted. However, because we do not access elements within this array through an index that is a parameter, we do not need to refer to it in the mapping at all.

Beyond this, the one requirement the functions have is again clearly analogous.
Showing that the state mapping itself is analogous is not quite as straightforward. We can see that the ordering of the elements is the same as with the other functions, but the rotations applied on them look differently. We shall show that for each form of these rotations, the elements remain analogous.

1. Let us look at the element a_0 in the reference mapping, and the associated (a_0, a_1, a_2, a_3) in the variant. Having not been rotated at all, we can see that the mapping is analogous here.
2. Let us look at the element $(a_1 \ll 1)$ ^ $(a_1 \gg 7)$ in the reference mapping, and the associated $(a_4 \ll 4$ ^ $a_5 \gg 4, a_5 \ll 4$ ^ $a_6 \gg 4, a_6 \ll 4$ ^ $a_7 \gg 4, a_7 \ll 4$ ^ $a_4 \gg 4)$ in the variant. The element has been rotated by one bit in the reference, and thus is rotated by four in the variant. Note that where the rotation transcends byte boundaries, it still remains cyclic! Thus, the mapping is also analogous here.
3. Let us look at the element $(a_{20} \ll 2)$ ^ $(a_{20} \gg 6)$ in the reference mapping, and the associated $(\overline{0}$ ^ $a_{81}, \overline{0}$ ^ $a_{82}, \overline{0}$ ^ $a_{83}, \overline{0}$ ^ $a_{80})$ in the variant. The element has been rotated by two bits in the reference, and thus is rotated by eight in the variant. Because eight bits aligns with the byte boundaries, we see that the order of elements is shuffled, but no rotations are present in the variant.

So, the functions `rho` and `rho_4` are semantically analogous.

4.5.5 Theta

Finally, we look at `theta` and `theta_4`. This function compresses twenty-five elements into five, and then xors the resulting array element wise, repeated five times. For the variant, of course, this is one hundred elements and twenty, respectively. Let us look at the state mappings:

$theta_4 \mapsto \{len(A) \geq 100, (A) \rightarrow ($

$a_0 \;\hat{}\; (\overline{0} \;\hat{}\; a_3 \;\hat{}\; a_{23} \;\hat{}\; a_{43} \;\hat{}\; a_{63} \;\hat{}\; a_{83} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_4 \;\hat{}\; a_{24} \;\hat{}\; a_{44} \;\hat{}\; a_{64} \;\hat{}\; a_{84} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{15} \;\hat{}\; a_{35} \;\hat{}\; a_{55} \;\hat{}\; a_{75} \;\hat{}\; a_{95},$

$a_1 \;\hat{}\; (\overline{0} \;\hat{}\; a_4 \;\hat{}\; a_{24} \;\hat{}\; a_{44} \;\hat{}\; a_{64} \;\hat{}\; a_{84} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_5 \;\hat{}\; a_{25} \;\hat{}\; a_{45} \;\hat{}\; a_{65} \;\hat{}\; a_{85} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{16} \;\hat{}\; a_{36} \;\hat{}\; a_{56} \;\hat{}\; a_{76} \;\hat{}\; a_{96},$

$a_2 \;\hat{}\; (\overline{0} \;\hat{}\; a_5 \;\hat{}\; a_{25} \;\hat{}\; a_{45} \;\hat{}\; a_{65} \;\hat{}\; a_{85} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_6 \;\hat{}\; a_{26} \;\hat{}\; a_{46} \;\hat{}\; a_{66} \;\hat{}\; a_{86} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{17} \;\hat{}\; a_{37} \;\hat{}\; a_{57} \;\hat{}\; a_{77} \;\hat{}\; a_{97},$

$a_3 \;\hat{}\; (\overline{0} \;\hat{}\; a_6 \;\hat{}\; a_{26} \;\hat{}\; a_{46} \;\hat{}\; a_{66} \;\hat{}\; a_{86} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_7 \;\hat{}\; a_{27} \;\hat{}\; a_{47} \;\hat{}\; a_{67} \;\hat{}\; a_{87} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{18} \;\hat{}\; a_{38} \;\hat{}\; a_{58} \;\hat{}\; a_{78} \;\hat{}\; a_{98},$

$a_4 \;\hat{}\; (\overline{0} \;\hat{}\; a_7 \;\hat{}\; a_{27} \;\hat{}\; a_{47} \;\hat{}\; a_{67} \;\hat{}\; a_{87} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_8 \;\hat{}\; a_{28} \;\hat{}\; a_{48} \;\hat{}\; a_{68} \;\hat{}\; a_{88} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{19} \;\hat{}\; a_{39} \;\hat{}\; a_{59} \;\hat{}\; a_{79} \;\hat{}\; a_{99},$

$a_5 \;\hat{}\; (\overline{0} \;\hat{}\; a_8 \;\hat{}\; a_{28} \;\hat{}\; a_{48} \;\hat{}\; a_{68} \;\hat{}\; a_{88} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_9 \;\hat{}\; a_{29} \;\hat{}\; a_{49} \;\hat{}\; a_{69} \;\hat{}\; a_{89} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_0 \;\hat{}\; a_{20} \;\hat{}\; a_{40} \;\hat{}\; a_{60} \;\hat{}\; a_{80},$

$a_6 \;\hat{}\; (\overline{0} \;\hat{}\; a_9 \;\hat{}\; a_{29} \;\hat{}\; a_{49} \;\hat{}\; a_{69} \;\hat{}\; a_{89} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{10} \;\hat{}\; a_{30} \;\hat{}\; a_{50} \;\hat{}\; a_{70} \;\hat{}\; a_{90} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_1 \;\hat{}\; a_{21} \;\hat{}\; a_{41} \;\hat{}\; a_{61} \;\hat{}\; a_{81},$

$a_7 \;\hat{}\; (\overline{0} \;\hat{}\; a_{10} \;\hat{}\; a_{30} \;\hat{}\; a_{50} \;\hat{}\; a_{70} \;\hat{}\; a_{90} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{11} \;\hat{}\; a_{31} \;\hat{}\; a_{51} \;\hat{}\; a_{71} \;\hat{}\; a_{91} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_2 \;\hat{}\; a_{22} \;\hat{}\; a_{42} \;\hat{}\; a_{62} \;\hat{}\; a_{82},$

$a_8 \;\hat{}\; (\overline{0} \;\hat{}\; a_{11} \;\hat{}\; a_{31} \;\hat{}\; a_{51} \;\hat{}\; a_{71} \;\hat{}\; a_{91} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{12} \;\hat{}\; a_{32} \;\hat{}\; a_{52} \;\hat{}\; a_{72} \;\hat{}\; a_{92} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_3 \;\hat{}\; a_{23} \;\hat{}\; a_{43} \;\hat{}\; a_{63} \;\hat{}\; a_{83},$

$a_9 \;\hat{}\; (\overline{0} \;\hat{}\; a_{12} \;\hat{}\; a_{32} \;\hat{}\; a_{52} \;\hat{}\; a_{72} \;\hat{}\; a_{92} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{13} \;\hat{}\; a_{33} \;\hat{}\; a_{53} \;\hat{}\; a_{73} \;\hat{}\; a_{93} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_4 \;\hat{}\; a_{24} \;\hat{}\; a_{44} \;\hat{}\; a_{64} \;\hat{}\; a_{84},$

$a_{10} \;\hat{}\; (\overline{0} \;\hat{}\; a_{13} \;\hat{}\; a_{33} \;\hat{}\; a_{53} \;\hat{}\; a_{73} \;\hat{}\; a_{93} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{14} \;\hat{}\; a_{34} \;\hat{}\; a_{54} \;\hat{}\; a_{74} \;\hat{}\; a_{94} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_5 \;\hat{}\; a_{25} \;\hat{}\; a_{45} \;\hat{}\; a_{65} \;\hat{}\; a_{85},$

$a_{11} \;\hat{}\; (\overline{0} \;\hat{}\; a_{14} \;\hat{}\; a_{34} \;\hat{}\; a_{54} \;\hat{}\; a_{74} \;\hat{}\; a_{94} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{15} \;\hat{}\; a_{35} \;\hat{}\; a_{55} \;\hat{}\; a_{75} \;\hat{}\; a_{95} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_6 \;\hat{}\; a_{26} \;\hat{}\; a_{46} \;\hat{}\; a_{66} \;\hat{}\; a_{86},$

$a_{12} \;\hat{}\; (\overline{0} \;\hat{}\; a_{15} \;\hat{}\; a_{35} \;\hat{}\; a_{55} \;\hat{}\; a_{75} \;\hat{}\; a_{95} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{16} \;\hat{}\; a_{36} \;\hat{}\; a_{56} \;\hat{}\; a_{76} \;\hat{}\; a_{96} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_7 \;\hat{}\; a_{27} \;\hat{}\; a_{47} \;\hat{}\; a_{67} \;\hat{}\; a_{87},$

$a_{13} \;\hat{}\; (\overline{0} \;\hat{}\; a_{16} \;\hat{}\; a_{36} \;\hat{}\; a_{56} \;\hat{}\; a_{76} \;\hat{}\; a_{96} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{17} \;\hat{}\; a_{37} \;\hat{}\; a_{57} \;\hat{}\; a_{77} \;\hat{}\; a_{97} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_8 \;\hat{}\; a_{28} \;\hat{}\; a_{48} \;\hat{}\; a_{68} \;\hat{}\; a_{88},$

$a_{14} \;\hat{}\; (\overline{0} \;\hat{}\; a_{17} \;\hat{}\; a_{37} \;\hat{}\; a_{57} \;\hat{}\; a_{77} \;\hat{}\; a_{97} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{18} \;\hat{}\; a_{38} \;\hat{}\; a_{58} \;\hat{}\; a_{78} \;\hat{}\; a_{98} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_9 \;\hat{}\; a_{29} \;\hat{}\; a_{49} \;\hat{}\; a_{69} \;\hat{}\; a_{89},$

$a_{15} \;\hat{}\; (\overline{0} \;\hat{}\; a_{18} \;\hat{}\; a_{38} \;\hat{}\; a_{58} \;\hat{}\; a_{78} \;\hat{}\; a_{98} \ll 4) \;\hat{}\; (\overline{0} \;\hat{}\; a_{19} \;\hat{}\; a_{39} \;\hat{}\; a_{59} \;\hat{}\; a_{79} \;\hat{}\; a_{99} \gg 4)$
$\;\hat{}\; \overline{0} \;\hat{}\; a_{10} \;\hat{}\; a_{30} \;\hat{}\; a_{50} \;\hat{}\; a_{70} \;\hat{}\; a_{90},$

$a_{16} \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_{19} \mathbin{\hat{}} a_{39} \mathbin{\hat{}} a_{59} \mathbin{\hat{}} a_{79} \mathbin{\hat{}} a_{99} \ll 4) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_0 \mathbin{\hat{}} a_{20} \mathbin{\hat{}} a_{40} \mathbin{\hat{}} a_{60} \mathbin{\hat{}} a_{80} \gg 4)$
$\mathbin{\hat{}} \overline{0} \mathbin{\hat{}} a_{11} \mathbin{\hat{}} a_{31} \mathbin{\hat{}} a_{51} \mathbin{\hat{}} a_{71} \mathbin{\hat{}} a_{91},$
$a_{17} \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_0 \mathbin{\hat{}} a_{20} \mathbin{\hat{}} a_{40} \mathbin{\hat{}} a_{60} \mathbin{\hat{}} a_{80} \ll 4) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_1 \mathbin{\hat{}} a_{21} \mathbin{\hat{}} a_{41} \mathbin{\hat{}} a_{61} \mathbin{\hat{}} a_{81} \gg 4)$
$\mathbin{\hat{}} \overline{0} \mathbin{\hat{}} a_{12} \mathbin{\hat{}} a_{32} \mathbin{\hat{}} a_{52} \mathbin{\hat{}} a_{72} \mathbin{\hat{}} a_{92},$
$a_{18} \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_1 \mathbin{\hat{}} a_{21} \mathbin{\hat{}} a_{41} \mathbin{\hat{}} a_{61} \mathbin{\hat{}} a_{81} \ll 4) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_2 \mathbin{\hat{}} a_{22} \mathbin{\hat{}} a_{42} \mathbin{\hat{}} a_{62} \mathbin{\hat{}} a_{82} \gg 4)$
$\mathbin{\hat{}} \overline{0} \mathbin{\hat{}} a_{13} \mathbin{\hat{}} a_{33} \mathbin{\hat{}} a_{53} \mathbin{\hat{}} a_{73} \mathbin{\hat{}} a_{93},$
$a_{19} \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_2 \mathbin{\hat{}} a_{22} \mathbin{\hat{}} a_{42} \mathbin{\hat{}} a_{62} \mathbin{\hat{}} a_{82} \ll 4) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_3 \mathbin{\hat{}} a_{23} \mathbin{\hat{}} a_{43} \mathbin{\hat{}} a_{63} \mathbin{\hat{}} a_{83} \gg 4)$
$\mathbin{\hat{}} \overline{0} \mathbin{\hat{}} a_{14} \mathbin{\hat{}} a_{34} \mathbin{\hat{}} a_{54} \mathbin{\hat{}} a_{74} \mathbin{\hat{}} a_{94},$
$)\}$
$theta \mapsto \{len(A) \geq 25, (A) \to ($
$a_0 \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_1 \mathbin{\hat{}} a_6 \mathbin{\hat{}} a_{11} \mathbin{\hat{}} a_{16} \mathbin{\hat{}} a_{21} \ll 1) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_1 \mathbin{\hat{}} a_6 \mathbin{\hat{}} a_{11} \mathbin{\hat{}} a_{16} \mathbin{\hat{}} a_{21} \gg 7) \mathbin{\hat{}}$
$(\overline{0} \mathbin{\hat{}} a_4 \mathbin{\hat{}} a_9 \mathbin{\hat{}} a_{14} \mathbin{\hat{}} a_{19} \mathbin{\hat{}} a_{24}),$
$a_1 \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_2 \mathbin{\hat{}} a_7 \mathbin{\hat{}} a_{12} \mathbin{\hat{}} a_{17} \mathbin{\hat{}} a_{22} \ll 1) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_2 \mathbin{\hat{}} a_7 \mathbin{\hat{}} a_{12} \mathbin{\hat{}} a_{17} \mathbin{\hat{}} a_{22} \gg 7) \mathbin{\hat{}}$
$(\overline{0} \mathbin{\hat{}} a_0 \mathbin{\hat{}} a_5 \mathbin{\hat{}} a_{10} \mathbin{\hat{}} a_{15} \mathbin{\hat{}} a_{20}),$
$a_2 \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_3 \mathbin{\hat{}} a_8 \mathbin{\hat{}} a_{13} \mathbin{\hat{}} a_{18} \mathbin{\hat{}} a_{23} \ll 1) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_3 \mathbin{\hat{}} a_8 \mathbin{\hat{}} a_{13} \mathbin{\hat{}} a_{18} \mathbin{\hat{}} a_{23} \gg 7) \mathbin{\hat{}}$
$(\overline{0} \mathbin{\hat{}} a_1 \mathbin{\hat{}} a_6 \mathbin{\hat{}} a_{11} \mathbin{\hat{}} a_{16} \mathbin{\hat{}} a_{21}),$
$a_3 \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_4 \mathbin{\hat{}} a_9 \mathbin{\hat{}} a_{14} \mathbin{\hat{}} a_{19} \mathbin{\hat{}} a_{24} \ll 1) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_4 \mathbin{\hat{}} a_9 \mathbin{\hat{}} a_{14} \mathbin{\hat{}} a_{19} \mathbin{\hat{}} a_{24} \gg 7) \mathbin{\hat{}}$
$(\overline{0} \mathbin{\hat{}} a_2 \mathbin{\hat{}} a_7 \mathbin{\hat{}} a_{12} \mathbin{\hat{}} a_{17} \mathbin{\hat{}} a_{22}),$
$a_4 \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_0 \mathbin{\hat{}} a_5 \mathbin{\hat{}} a_{10} \mathbin{\hat{}} a_{15} \mathbin{\hat{}} a_{20} \ll 1) \mathbin{\hat{}} (\overline{0} \mathbin{\hat{}} a_0 \mathbin{\hat{}} a_5 \mathbin{\hat{}} a_{10} \mathbin{\hat{}} a_{15} \mathbin{\hat{}} a_{20} \gg 7) \mathbin{\hat{}}$
$(\overline{0} \mathbin{\hat{}} a_3 \mathbin{\hat{}} a_8 \mathbin{\hat{}} a_{13} \mathbin{\hat{}} a_{18} \mathbin{\hat{}} a_{23}),$
$\ldots$
$)\}$

To keep the mapping relatively concise, we show the first five (out of twenty-five) elements of the reference mapping, and the first twenty (out of a hundred) elements of the variant mapping.

Let us observe the first element of the reference mapping, and the first four of the variant mapping. Here, we see that all indices match up in the expected way; the variant's indices are the reference's indices, multiplied by four for the first element, and then plus one for each of the three elements after it.
Furthermore, we see that the rotation is also multiplied by four.

Lastly, we see that in both the variant and the reference mapping, each element consists of its original value, to which the xor operation is applied together with some other compound value. This compound value is equal for each five / twenty subsequent elements, for the variant and reference respectively, and as such is explained by the factor of four.
Thus, we have shown that the functions `theta_4` and `theta` are semantically

analogous.

4.5.6 KeccakP200Round

The round function consists of all diffusion functions (Iota through Theta) being applied on the state array.
However, as can be seen in the appendix[5], the state mapping of both the reference and the variant are too large to properly reason about[6].
Instead, we make use of the fact that we know the body of both variants consists of all diffusion functions being called in the same order. Given that we have already concluded that all diffusion functions are semantically analogous, this allows us to conclude that `KeccakP200Round` and `KeccakP200Round_4` are semantically analogous.

[5]Specifically in section A.1 and section A.2, for the reference and variant respectively.

[6]Note that this is not inherent to our technique, but mostly caused by an unsimplified state mapping. We discuss this point further in Section 6.1

Chapter 5

Related Work

Software verification is not a new field, and as such, a lot of research has gone into the verification of software that is related to security, in some manner. We briefly list existing studies on concepts that closely relate to our topic below, and finally explain why our approach is a new one.

5.1 ProVerif

Proverif [7] is probably the most well-known protocol verifier. It takes as input a model in the Dolev-Yao [11] format, and outputs which security properties the protocol has, or fails to have, and why.

5.2 Automatic model extraction

The paper by Aizatulin et al. [3] describes a new method to use symbolic execution on a C program, then rewrite it to a form ProVerif can accept, and finally use ProVerif to obtain the security properties of said C program. It is noted by the authors that their approach can only be applied to protocols with no significant branching, for now.
It is unfortunately not entirely clear what significant branching is in the context of the paper.

5.3 Static Analysis through semantics

The paper by Bodei et al. [8] describes how to prove the properties of a protocol implementation, without annotations, through semantical analysis. Their approach to doing so is similar to ours, but they prove properties of a protocol implementation, as opposed to proving equivalence between two implementations.

5.4 Our approach

The solutions that are available as of yet either allow one to prove the properties a model has, to extract a model from source code (in one specific language), or prove properties directly through source code. Our proposed method, however, verifies the *implementation* of a cryptographic algorithm. For this thesis, we have narrowed the scope to permutations, but the techniques can, in theory, be applied to any cryptographic primitive or protocol. We know this is the case, because `While` and `While`$^+$ are Turing complete (and as such, any algorithm can be expressed in `While`$^+$). However, the more difference between `While`$^+$ and the source language(s), the more elaborate the transformation rules have to be to yield a correct transformation.

This is the difference between the existing state-of-the-art approaches, and the new techniques proposed in this paper.

Chapter 6

Conclusions

In Chapter 3, we conclude by showing that the state mappings of the diffusion functions are semantically analogous.
From this, we may conclude that the round functions of the two implementations are semantically analogous as well, given that they consist of applications of the diffusion functions. And from this, we may conclude that the permutation functions of the two implementations are semantically analogous, because they consist of applications of the round function.

Thus, if it is the case that the the transformations do not alter the semantics of the code and the rules of the derivation system are sound and complete, then we have shown that the Keccak-f variant is indeed analogous to the reference implementation.

Below, we will briefly cover the limitations present in our current technique, go on to discuss the feasibility of its application and finally lay out what future work might be performed in this area.

6.1 Limitations

There are several limitations to the method we employ. We will go over them one by one in this subsection, and discuss how much they impact the conclusion of the application of our technique.

6.1.1 Reference implementation errors

If the reference implementation contains errors, this approach cannot catch them. In fact, it will tell the user the variant is correct if it has the same fault, and will otherwise report that there is a fault in the variant code, even if it correct!

This limitation is inherent to the way in which this method identifies mistakes. Additionally, it has a rather small impact; if we only need to verify that *one* implementation is correct manually, this is far more achieveable then having to verify that *all* implementations are correct.

6.1.2 Analogous vs identical

The two variants we compare in this thesis are not programs that are expected to be semantically identical, as one is optimized[1]. This requires us to show that the state mappings that result are analogous.
Unfortunately, there is no straightforward method for this, as analogous mappings can exist in many forms; from mappings that are equal but require rewriting, to mappings like in `rho` and `rho_4`, which differ a lot, but are relatable due to a number of factors.

It seems like this is a major issue in this approach, but it is actually not. The primary goal of this approach is to prove that two implementations are *identical.* It just so happens to be the case here that they are not, but are not supposed to be either. Thus, we can see the possibility to prove semantic analogies as an extension to the original approach, that requires more manual work.

6.1.3 Scalability

As briefly touched upon in Section 3.3, the complexity of applying the derivation rules is $\mathcal{O}(2^n)$, where n are the number of if statements. Though it seems relatively poor, it has no direct impact on our application, as it is mostly manual.
It does mean that if someone were to automate the process further, they need to be careful to do so efficiently, so as to avoid excruciatingly slow results.
Additionally, if one were to attempt proofs on bigger programs, such as Keccak-f[1600], the state might become too big to reason about, even if state mapping simplification is applied. This is because the maximum length of the state mapping is determined by the number of the individual components that make up the parameters. For instance, if the sole parameter to a mapping is a byte, then the maximum length for a single bit of that byte is 8 bits; each bit being somehow present in the final term. Thus the maximum length of the entire byte is 64 terms. In general, the worst case scaling is quadratic ($\mathcal{O}(n^2)$).
If we take Keccak-f[1600] as example again, this would lead to a maximum state mapping length of 2560000 terms.

[1]However, one would still expect identical inputs to the respective permutation function to provide identical outputs.

6.1.4 While loops

Though it does not happen in Keccak-f, there exists a particular case in while loops that the derivation rules cannot handle properly.
Consider the following `While`$^+$ program:

```
foo(bar = var)
(
    baz = 0;
    while(bar < 7) do
    (
        baz = bar * 3 + baz;
        bar + 1
    )
)
```

Here, the condition of the while loop is dependent on a parameter. Because we provide parameters with random values in the second pass, we *cannot* know when the condition is true or false.
If the condition depends on, say, a bit or a byte, we could solve the problem by enumerating all possibilities, and creating a state mapping for each. This is not a great solution, as it would need 256 mappings for a byte, but it also does not solve the problem for arrays or variables. Both have an unbounded 'maximum' value.

As stated previously, this does not impact the conclusion of this paper, because this does not occur in Keccak-f.

6.1.5 State mapping simplification

We do not provide the state mapping of `permutation` and `permutation_4`, because the mapping becomes so large that our helper program cannot handle them.
However, the fact that the helper program fails is a symptom of the underlying problem; the state mappings become incredibly large. This is because we do not simplify the state mappings at all, leading to a cascading effect every time we apply a state mapping on an existing state.

As mentioned above, this impacts our thesis in the sense that we could not create state mappings of all functions. Beyond that, however, not simplifying makes it more difficult to reason about the state mappings that we *did* make.
Simply put, the reason we did not simplify the state mappings is because we could employ a bit more manual work to circumvent these problems, whereas simplification is a process that is likely to create a significant amount of edge cases to deal with.

6.1.6 Generality of transformation rules

As is mentioned in Subsection 4.3.1, not all rules that are defined will work for any given C program. Some rules rely on the fact that arrays defined in Keccak are always of size 25, for instance.
But the biggest non-generality can be seen in the last rule of the variant rules; it transforms a single line of code into a triply nested if-then-else statement! This is an excellent example of the fact that the farther one's code strays from the concepts defined in `While`$^+$, the more work will be necessary within the transformation rules to make the transformation work.

Theoretically speaking, it is possible to define general rules for *all* cases (in the sense that Turing completeness allows us to conclude we can always simulate any program in any language, in `While`$^+$), but this requires *considerable* effort. Therefore, we decided to create general rules where possible and otherwise explained why it was not feasible.

6.2 Feasibility

If one wishes to apply our technique to a program, the steps that currently require manual work are:

1. Obtain a reference and variant.
2. Apply preprocessing to the reference and variant.
3. Write (or otherwise obtain) transformation rules for both the reference and variant.
4. Apply postprocessing to the transformed reference and variant.

4. Apply the first pass derivation rules to the reference and variant.
5. Obtain a derivation order for the reference and variant.
6. Apply the second pass derivation rules to the reference and variant, according to the derivation orders of step 5.
7. Obtain the state mappings of all functions of the reference and variant, from the derivation trees of step 6.
8. If the goal is to prove equivalence, find a strict bijection between the state mappings of the reference and variant. If the goal is to prove analogy, find a bijection between the state mappings of the reference and variant, such that any differences are explained by transformation or usage of the program.

If additional work is put in to automation of this process, the most optimal outcome would be:

1. Apply a reference and variant.
2. Apply preprocessing to the reference and variant.
3. Write (or otherwise obtain) transformation rules for both the reference and variant.
4. If the goal is to prove analogy, find a bijection between the state mappings of the reference and variant, such that any differences are explained by transformation or usage of the program[2].

Thus, we may conclude that as it stands, our technique is not feasible to apply in practice. Even if additional work is put in to improve it, the most substantial step (namely the creation of transformation rules) remains. In this case, it would be *more* feasible then it is currently, but still probably not practically so.
In the next section, we will detail the ways in which our method may be improved.

6.3 Future work

There are a number of aspects in which our method can be improved, but which were out of scope for the purposes of this thesis. We list them below, and briefly discuss what their use is.

6.3.1 While loops

As discussed in Section 6.1, certain uses of while loops are not properly handled by the derivation rules. We suspect a possible solution to this is to try a technique similar to axiomatic semantics, by attempting to describe the state delta's of the while loop not in terms of the direct state (because the state influences the while loop), but in patterns. For instance, for the following while loop:

```
x = 0;
while(x < len) do (
    y[x] = y[x] + 3;
    x = x + 1
)
```

[2]If the goal is to prove equivalence, then it is possible to do so automatically, by simplifying the state mappings and then verifying their equality.

Where *len* is a parameter that specifies the length of y, that the state mapping of the array will look something like:

$$y \rightarrow (x_0 + 3, x_1 + 3, ..., x_{len-2} + 3, x_{len-1} + 3)$$

Indicating that for however long the array happens to be, the mapping follows the pattern described.

We also briefly discuss how a while loop might be handled, if its condition depends on a variable alone:

```
while(x < 37) do (
    x = x + 5;
)
```

The number of iterations is directly dependent on the value of x. For this case, it is clear that if we can express the number of iterations in terms of x, then we can describe the value of x in terms of itself (as the reader may have spotted, the outcome will be 37 38 39 40 or 41, here). However, it is not entirely clear what may happen if the number of iterations depends on a variable, but the loop itself influences an array.

6.3.2 State mapping simplifications

Simplifying the state mapping is a matter of applying the definition of operators as much as possible, given the arbitrary state. For instance, we can simplify

$$a \rightarrow (x + 3) + 7$$

to

$$a \rightarrow x + 10.$$

We show examples for some types and operands:

$$\begin{aligned}
&a \rightarrow \overline{5} \mid a \\
&a \rightarrow (\underline{1}, a_1, \underline{1}, a_2, a_3, a_4, a_5, a_6, a_7) \\
&b \rightarrow b \ll 4\hat{\ } b \gg 4 \\
&b \rightarrow (b_4, b_5, b_6, b_7, b_0, b_1, b_2, b_3)
\end{aligned}$$

It should not be difficult on its own to apply these simplifications, but we are not sure if there are specific orders that need to be used to simplify optimally. Ideally, this simplification process should be written in term of a TRS, such that it can be shown to be terminating and confluent.

6.3.3 Automation

To improve usability, we would need to automate our approach further. Specifically, we would need to further improve the transformation step, and automate the application of the derivation rules.
We do not foresee any difficulties in this automation, apart from the aforementioned limitations of the derivation system. The specific steps we can automate are:

- The removal of modulus statements, when it is never used. In the code we look at in this thesis, this happens most often in a while loop, where the condition enforces that $x < 5$, and so a statement like $x\%5$ is semantically identical to x.
- Applying the first pass derivation rules to the reference and variant. Since the first pass is mainly intended to find the function calls made within each function, this is not too difficult.
- Finding a derivation order. Since we have already described a procedure to find a derivation order, this should not be much work to automate.
- Applying the second pass derivation rules to the reference and variant. This is also fairly straightforward, as all the rules are already described in this thesis.
- Simplifying the state mappings. This step consists entirely of applying the hypothetical TRS as described in the previous subsection.
- Proving equivalence of two sets of state mappings. Since this step occurs *after* the simplification of the state mappings, we may assume all variable names and orders are fixed, and so equal state mappings are absolutely identical.

The remainder of the process which cannot be automated typically involves some amount of human insight being required to conclude the step, such as proving the reference and variant are analogous.

6.3.4 Optimizations

We have realized that state mappings can be used to create optimized code, if used properly.
Consider the original function `pi`, from the reference implementation of the code used in this thesis. It uses a double loop to assign elements, and does so twice, essentially using an entire extra array's worth of memory.
From the state mapping of `pi`, however, we can conclude that:

1. The state mapping maps a single array.

2. This single array is mapped in terms of itself, that is, no references are made to read-only arrays, variables, bytes or bits.

3. Every element of this array is mapped from exactly one other element.

4. We can map each element in the array, using a single external byte to hold the element to be mapped to last.

In general, one could create a graph of dependencies of parameters, to determine how we may turn the state mapping into an optimized function with minimal memory or cpu overhead. In doing so, we not only improve functions by making them effectively assign the end result of what their source code does, as opposed to any interim, but it also makes it so that optimizations work far more predictably.
Currently, at least for all compilers we are aware of, adding new optimizations is incredibly complex. The order in which optimizations are applied often matters, or one optimization might depend on another working in a specific manner. By reducing functions to a state mapping and then expanding the state mapping to an assignment conforming to the mapping, we only need to optimize the different pieces of code that are used in said assignments.

Of course, a possible downside to this approach could be binary size. Consider for instance the function `pi` again. In its unoptimized form, it simply consists of a nested while loop. But if optimized, it should lead to 25 assignments instead. This is not too bad if the array size is 25, but if it happens to be 250, or even more, problems might start appearing.

6.3.5 Term rewriting systems

Term Rewriting Systems [4] (or TRS, for short) are systems that can be used to rewrite terms. If our transformation rules were to be rephrased as a TRS, it may be possible to prove it is a confluent and terminating (or, equivalently, complete) TRS. A terminating TRS is one that never endlessly rewrites its input. This is invaluable for automating the transformation step, as it means cycle detections can be omitted.
A confluent TRS is one for which each input is rewritten in exactly one way to an output. While not technically necessary to prove, it does make the result of a transformation more robust, because the reduction strategy that was used cannot impact the outcome.

Bibliography

[1] Hash Functions | CSRC. [Online accessed 30-October-2023].

[2] Leightweight Cryptography | CRSR. [Online accessed 30-October-2023].

[3] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 331–340, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Franz Baader and Tobias Nipkow. Term Rewriting and All That, 1998.

[5] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally Sound Verification of Source Code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 387–398, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Dumbo, Jumbo, and Delirium: Parallel Authenticated Encryption for the Lightweight Circus. *IACR Transactions on Symmetric Cryptology*, 2020(S1):5–30, Jun. 2020.

[7] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[8] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13:2005, 2005.

[9] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Amber Sprenkels, and Benoit Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?". In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 526–545, Cham, 2020. Springer International Publishing.

[10] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak Pseudocode and specification summary.

[11] D. Dolev and A. C. Yao. On the security of public key protocols. In *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 350–357, 1981.

[12] Alex Feenstra. CIL Postprocessor, October 2023. `https://doi.org/10.5281/zenodo.10009742`.

[13] Alex Feenstra. CIL Transformer, October 2023. `https://doi.org/10.5281/zenodo.10054395`.

[14] Alex Feenstra. Keccak reference data, October 2023. `https://doi.org/10.5281/zenodo.10009182`.

[15] Alex Feenstra. Keccak variant - transformation rules, October 2023. `https:/doi.org/10.5281/zenodo.10018399`.

[16] Alex Feenstra. Keccak variant data, October 2023. `https://doi.org/10.5281/zenodo.8435737`.

[17] Michaël Peeters Guido Bertoni, Joan Daemen and Gilles Van Assche. The Making of KECCAK. *Cryptologia*, 38(1):26–60, 2014.

[18] Nicky Mouha, Mohammad S. Raunak, D. Richard Kuhn, and Raghu Kacker. Finding Bugs in Cryptographic Hash Function Implementations. *IEEE Transactions on Reliability*, 67(3):870–884, 2018.

[19] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[20] Hanne Riis Nielson and Flemming Nielson. Semantics with Applications: A Formal Introduction, 1992.

[21] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2023. Version 6.5, `http://softwarefoundations.cis.upenn.edu`.

Appendix A

Appendix

This appendix contains code and proofs that would be too lengthy to put in the running text.

A.1 Reference data

Due to the nature of the data obtained from applying CIL and the derivation rules to the reference implementation, it is difficult to present properly in this format.
To solve this, we provide the reference data through Zenodo [14]. Zenodo is a data repository maintained by CERN. It contains:

1. The source code after being processed by CIL,
2. The CIL code after being manually processed further,
3. The `While`$^+$ code after being transformed from CIL,
4. The `While`$^+$ code, after being manually processed further,
5. The annotated program resulting from the first pass derivation rules being applied on the `While`$^+$ code,
6. The annotated programs resulting from the second pass derivation rules being applied on the results of the first pass.

As well as some plaintext that adds additional explanation to the annotated programs.

A.2 Variant data

Just as we did for the reference data, we also provide the variant data through the Zenodo repository [16]. It contains:

1. The source code after being processed by CIL,
2. The CIL code after being manually processed further,
3. The `While`$^+$ code after being transformed from CIL,
4. The `While`$^+$ code, after being manually processed further,
5. The annotated program resulting from the first pass derivation rules being applied on the `While`$^+$ code,
6. The annotated programs resulting from the second pass derivation rules being applied on the results of the first pass.

As well as some plaintext that adds additional explanation to the annotated programs.

A.3 Variant transformation rules

Due to the reasons discussed in the related subsection 4.3.1, the variant transformation rules are too lengthy to properly display here.
To solve this, we provide the variant rules through Zenodo[15].

A.4 Postprocessing program

The following program, written in C#, takes as input some amount of CIL files, applies the post-processing described in this thesis (section 4.2.1) to them, and writes the result to file. Note that although the listing includes all actual code that it uses, the accompanying *project* structure (the project file, the folders and compilation data) has been omitted, because it would take up needless space.
Because C# code does not list well in this format, we have published it on Zenodo [12].

A.5 Transformation program

Though the comments in the source code explain most of the process, we shall briefly explain it here, and highlight what functionality does not yet work as intended.
We read in the rules (which are represented in ASCII format, but otherwise completely identical to those defined in Subsection 4.3.1), and split them on newlines. Each rule is then made up of a right hand side (which will be called RHS from here on), and a left hand side (LHS). The RHS defines what must be matched in the original code, and the LHS defines what must be replaced with a match of the RHS.

We can immediately transform the LHS into a regular expression, by escaping characters that have meaning in regular expressions, and replacing each of the rule symbols by their regex equivalent.
For the RHS, we can replace every rule symbol by a placeholder (of the form '{i}', where i is some number), to turn it into a format string. Then, every captured subgroup of the LHS can be fed into the RHS, to obtain the replacement.
The failure in this process lies in the way the format string is generated; it assumes that every captured subgroup is fed into the format string *in order*. In reality, this need not be the case, and so the rules that don't comply to this fail to work through the program.

Similar to the other appendix entries, we have published the code on Zenodo [13].

A.6 CILLY script

Cilly is the driver that is used to perform code transformation outside of OCAML. We provide a short bash script to make transformation even easier below. Though it does not allow for more than two files to be merged at once, this is of no account for this thesis; we never need to merge more than two files.
Note that this script will only work if both Cilly and OCAML are installed.

Listing A.1: CILLY script

```
#!/bin/bash

cilly_location=$(which cilly)

if [ "$cilly_location" == "cilly not found" ];
   then
         echo "Error: could not find cilly!
            Please install CIL using the command
             'opam install cil'."
         exit 1
fi

if [ "$(which clang_format)" == "clang_format
   not found" ]; then
         echo "Error: could not find clang!
            Please install it before running
            this script."
         exit 1
```

```
fi

if [ "$1" == "" ]; then
        echo "Please enter at least one file to
            convert!"
        exit 1
fi

if [ "$2" == "" ]; then
        $cilly_location --save-temps $1.c #
           Calling cilly to do the
           transformation on one file.
        clang-format -i $1.cil.c # Reformatting
            the resulting file.
fi

if [ "$2" != "" ]; then
        $cilly_location --save-temps --merge $1
           .c $2.c #Calling cilly to do the
           transformation on two files, with
           the merge option enabled.
        clang-format -i a.cil.c # Reformatting
           the resulting merged file.
fi

rm a.out # Removing the final executable (which
    we are not interested in).
rm *.i #removing unneeded temp files
rm *.o #removing unneeded object files
```