Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Interpreter for Machine Programs on Arbitrary Models of Computation

Konrad Wienecke
Matrikelnummer: 10005023

2020

Erstprüfer: Prof. Dr. rer. nat. Heribert Vollmer
Zweitprüfer: Dr. rer. nat. Maurice Chandoo

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen Hilfsmittel und Quellen als angegeben verwendet habe.

Konrad Wienecke

Contents

1 Introduction

Automata have long played an important role in theoretical computer science. They are well-known for their use in the fields of formal language theory and complexity theory where they are used to describe classes of languages and problems. Their simplicity and formalism serves as a good basis for theoretical proofs. Yet automata are structurally quite dissimilar to real-world computers. Where the notions of hardware and software are clearly detached from each other in the context of computers, automata show no clear separation of these two concepts.

That is not to say that this separation is impossible. In [Sco67], the separation of automata into machine and (machine) program is first proposed. The notion of sequential models of computation is not far from that of Scott's machines. Broadly said, a sequential model of computation describes the actual capabilities of a machine as an interface. The machine has a hidden internal state, the machine state, which can be accessed by a program through the machine's buttons (operations) and indicator lamps (predicates). [1] Many deterministic automata like finite automata, pushdown automata, and Turing machines can be modelled as a combination of a sequential model of computation and a corresponding program. Typically, an automaton transitions between configurations sequentially as defined by its transition function. This process is similar to how a program sequentially applies operations to a machine state.

Note that this thesis is only concerned with sequential models of computation. Non-sequential models of computation also exist, for instance lambda calculus, Boolean circuits, and Petri nets. However, these models describe computations in much different ways. From here on, we will just use the term model of computation to refer to sequential models of computation.

Models of computation have uses beyond those of automata. For instance, they can serve as a notional machine, a concept introduced in [Sor13]. Sorva details how notional machines can be useful for teaching about programs and program execution by serving as a mental model to the programmer. Program tracing is described as another useful process for the purpose of understanding a program.

We present an interpreter which can be used for interpretation and tracing of machine programs for arbitrary models of computation. The machine programs are written in a language which allows the use of many different models of computation. The interpreter has been implemented in Haskell as this language lends itself to parsing and because Haskell's fundamental integration of higher-order functions lets us easily implement models of computation.

[1]This informal description of sequential models of computation is inspired by that given in [Cha20b].

2 Basic Definitions

2.1 Model of Computation

While different types of models of computation exist, this thesis is only concerned with sequential models of computation which we will refer to with the former term from here on.

The notion of the model of computation stems mainly from automata theory. While an automaton such as a finite state machine may compute a mathematical function, that specific machine is limited to that function only. In contrast, when we think of a computer we carry the notion that it is a general-purpose machine which can be programmed. If we separate the automaton into *program* and *machine* [2] , it structurally becomes more akin to a conventional computer while retaining its simplicity.

Models of computation describe *machines* a little bit more abstractly. We can think of a model of computation as being a sort of programming interface for a machine. It defines three parts of the machine: The machine state space, a set of operations which can change the machine's state, and a set of predicates which show information about the machine state. Formally:

Definition 2.1 ([Cha20a])**.** A model of computation $\mathcal{M}$ is a triple (S, O, P) where S is a countable set, O is a finite set of functions from S to S and P is a finite set of functions from S to $\{\texttt{true}, \texttt{false}\}$. The set S is called (machine) state space of $\mathcal{M}$ and an element s of S is a (machine) state of $\mathcal{M}$; if $\mathcal{M}$ is clear from the context we call s a machine state. A function from O is called an operation of $\mathcal{M}$ and a function from P is called a predicate of $\mathcal{M}$.

Note that in general, the operations and predicates of a model of computation are only applied to the hidden state of the same machine. We may therefore omit the argument of the function as it is implied by the context. Furthermore, as these functions will later be used as part of machine programs, we will use strings to reference operations and predicates instead of defining more mathematical symbols.

We may also parametrize models of computation. A parameter can for instance determine the number of registers of a machine with registers, or an alphabet that the machine works with. Parameters may directly affect the state space and the sets of functions of the model of computation. In theory however, we could make a parameter completely change the behaviour of a model of computation; two models of computation of the same type but with different parameter values do not necessarily have to have anything in common.

[2]This concept is one of the main ideas discussed in [Sco67].

The following sections outline some models of computation of which all but the last two are related to types of automata from the field of formal language theory.

2.1.1 Deterministic Finite Automaton

The deterministic finite automaton (DFA) is perhaps the most basic automaton, the concept first appearing in [MP43]. As a model of computation (deterministic finite automaton model of computation, DFAMOC), its machine state consists of a list of symbols representing the input, and a Boolean value, the *accept*-value, that represents whether the input has been accepted or not. The input consists of symbols from a fixed alphabet and this alphabet must be defined as a string parameter for the model of computation.

The DFAMOC specifies just two operations: The operation `NXT` removes the first symbol of the input given that the input is not already empty. If the input is found to be empty, this operation does not change the machine state. The other operation, `ACC`, deletes the remainder of the input and sets the Boolean value to `true`.

The set of predicates consists of one predicate `I=s` for each symbol `s` in the alphabet, which is `true` if and only if the first symbol of the remaining input is `s`. Another predicate with the name `I=` is `true` in case the input is empty.

2.1.2 Real-Time Deterministic Pushdown Automaton

The real-time deterministic pushdown automaton (RDPA) is a somewhat unconventional and slightly weaker version of the deterministic pushdown automaton (DPA) which does not allow for ϵ-transitions [3] . This automaton is originally presented in [PY83].

As a model of computation (real-time deterministic pushdown automaton model of computation, RDPAMOC), its machine state is similar to that of the DFAMOC: The RDPAMOC has two symbol lists called the input and the stack, and an *accept*-Boolean value as its machine state. One list of symbols, the shared alphabet, is defined as a parameter with both the input and the stack only being allowed to hold symbols from this alphabet.

The RDPAMOC shares the `ACC`-operation with the DFAMOC, which just empties the input list and sets the *accept*-value to `true` (however, the stack is not changed by this operation). Additionally, there is one operation for each finite string of alphabet symbols, including the empty string. Given such a string `y` $= s_0 s_1 \ldots s_k$, the operation `Wy` does the following changes only if the input is non-empty:

1. The first symbol of the (remaining) input is removed
2. If the stack is not empty, the top-most symbol of the stack is popped
3. If `y` is not empty, the symbols $s_0 s_1 \ldots s_k$ are pushed onto the stack so that s_0 is the new top-most symbol

[3] ϵ-transitions allow a traditional DPA to transition in between states and write symbols onto the stack without reading a symbol from the input.

In case the input is found to be empty, the operation does not change the machine state.

Finally, the set of predicates includes two functions for each symbol `s`: `I=s` is `true` if and only if the first symbol of the remaining input is `s`, and `S=s` is `true` if and only if the top-most symbol on the stack is `s`. As with the DFAMOC, the predicate `I=` also exists and is `true` in case the input is empty whereas the additional predicate `S=` is `true` in case the stack is empty.

2.1.3 Turing Machine

The machine state of the Turing machine model of computation (TMMOC) consists of a finite list of symbols which represents the tape, together with an index for the position of the tape head. As with the RDPAMOC, we only define one list of symbols as a shared alphabet instead of separating the input and tape alphabets from each other. The first symbol in this list is the special blank symbol. The tape can only hold symbols from this alphabet at any moment.

For each symbol `s`, there is one predicate, `=s`, which is `true` if and only if `s` is the symbol on the tape at the current head position. Additionally, there is one operation for each symbol `s` and each direction $\mathtt{d} \in \{\mathtt{L}, \mathtt{R}, \mathtt{N}\}$ which we call `WsL`, `WsR`, and `WsN`. These operations first write the given symbol onto the tape at the current position before moving the head either left, right, or not at all. Since we are using a finite list as the tape, the head can end up at one of the ends. In this case the list is extended in that direction with a blank symbol.

2.1.4 Linear Bounded Automaton

As the linear bounded automaton model of computation (LBAMOC) is very similar to the TMMOC, it also uses a finite list of symbols and an index to represent the tape and head. Again, one alphabet is used to represent both the input and tape symbols, this time without a special blank symbol. The tape can only hold symbols from this alphabet at any moment and it can not grow past the extends of the original input.

The predicates of the LBAMOC are extended compared to the TMMOC's: Alongside the symbol predicates exist two new predicates, `LE` and `RE`. These are `true` if and only if the head is currently positioned one cell to the left or one cell to the right of the first or last cell of the input. These cells are called the left bound and the right bound.

While the operations of the LBAMOC are mostly the same as those from the TMMOC, there exist some differences when the head is positioned on one of the bounds: In these cases the writing part of the operation is skipped, and the head can not be moved further away from the original input cells.

2.1.5 Counter Machine

The counter machine model of computation (CMMOC) has at least one register as its machine state. Each register contains a single, non-negative integer. We enumerate these registers starting with 1 for the purpose of accessing them with the operations and predicates. The number of registers of one specific CMMOC is defined as a parameter.

Each register can be independently accessed by the same operations and predicates: An increment operation and a bounds-safe decrement operation that does not decrement past 0, as well as a single predicate indicating whether that register's value is 0. [4] We use the following notations for these operations and predicates:

- `Ri+` denotes incrementing the value in register i by 1
- `Ri-` denotes safely decrementing the value in register i by 1
- `Ri=0` denotes checking whether the value in register i is 0

2.1.6 Stack Machine

The stack machine model of computation (SMMOC) is a logical extension of the CMMOC, working with stacks instead of counters. Each stack holds zero or more symbols. As the name entails, only the top-most symbol is visible and can be manipulated by the model's operations and predicates. Stack machines can be modified by two parameters, one defining the number of stacks and the other defining the stack alphabet. A stack machine with an alphabet consisting of just one symbol is functionally equivalent to a counter machine with the same number of registers.

We define the following operations and predicates:

- `Ri+s` denotes pushing the symbol `s` onto the `i`-th stack
- `Ri-` denotes popping the top-most symbol from the `i`-th stack
- `Ri=s` denotes checking whether the top-most symbol on the `i`-th stack is `s`
- `Ri=_` denotes checking whether the `i`-th stack is empty

2.2 Machine Program

A program transforms a given input until it (maybe) terminates, at which point it returns an output. A machine program does the same while being confined to the specific components of a machine, or more specifically, the operations and predicates of a model of computation associated with that machine.

Definition 2.2. Let $\mathcal{M} = (S, O, P)$ be a model of computation.

We can express a $\mathcal{M}$-machine program $\mathcal{P}_{\mathcal{M}} = (Z, z_{Start}, z_{End})$ as a set of program states $Z = \{z_0, z_1, \dots, z_n\}$, a start state $z_{Start} \in Z$, and an end state $z_{End} \notin Z$. Every program state $Z \ni z = (T_{\mathcal{P}_{\mathcal{M}}}, o)$ consists of a binary decision tree $T_{\mathcal{P}_{\mathcal{M}}}$ (defined in 2.3) and an operation $o \in (O \cup \{\texttt{NOP}\})$. z_{Start} always has the identity operation `NOP`. [5]

[4] This CMMOC resembles the counter machine described by Minsky [Min61] and by Lambek [Lam61] , however it differs in that they combine the decrementing operation and the zero-check predicate into one instruction.

[5] A similar definition can be found in [Cha20a], which this one is loosely based on.

Definition 2.3. Let $\mathcal{M} = (S, O, P)$ be a model of computation, $\mathcal{P}_\mathcal{M} = (Z, z_{Start}, z_{End})$ be a $\mathcal{M}$-machine program. Let $Z' = (Z \setminus \{z_{Start}\}) \cup \{z_{End}\}$ be the set of reachable program states.

A binary decision tree $T_{\mathcal{P}_\mathcal{M}}$ (abbr. BDT) is a binary tree with the following constraints:

(i) every node in the tree has either zero or two children

(ii) every node with zero children (i.e. every leaf of $T_{\mathcal{P}_\mathcal{M}}$) is in Z'

(iii) every node with two children (i.e. every branch of $T_{\mathcal{P}_\mathcal{M}}$) is in P

The right child of a branch is associated with the branch's predicate being `true`, while the left child is associated with the branch's predicate being `false`.

As described above, a machine program primarily consists of some states which, in return, each consist of an operation and a BDT. Figure 2.1 shows an example of a BDT as it may appear in a program. Program execution follows the procedure which is defined in Algorithm 2.1.

Algorithm 2.1 Execute a machine program

```
function runMachineProgram($\mathcal{P}_\mathcal{M}, s$)
    $pstate \leftarrow z_{Start}$
    $mstate \leftarrow s$
    while $pstate \neq z_{End}$ do
        $mstate \leftarrow pstate.o(mstate)$
        $pstate \leftarrow$ evaluateBDT($pstate.T, mstate$)
    return $mstate$
function evaluateBDT($T, s$)
    $root \leftarrow$ getRoot($T$)
    if isChild($root$) then
        return $root$
    if $root(s) = true$ then
        $newroot \leftarrow$ getRightChild($T$)
        return evaluateBDT($newroot, s$)
    else
        $newroot \leftarrow$ getLeftChild($T$)
        return evaluateBDT($newroot, s$)
```

We begin at the given start state z_{Start} and with a given input machine state. In a repeating process, we first transform the current machine state using the program state's operation (although this will just be an identity-transformation in z_{Start}) and then evaluate the resulting machine state through a series of predicates which are structured in the BDT of the program state. We terminate if we reach the end state z_{End}.

The binary decision trees are evaluated by recursively traversing the tree until a leaf node is reached. At each branch, one of the two children is chosen by applying the predicate in the node to the machine state: If the predicate is `true`, we continue with the right child, otherwise we continue with the left child.

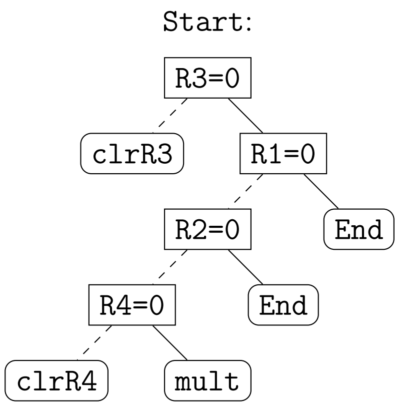


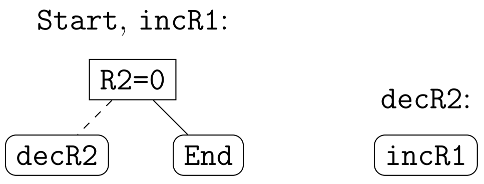
Figure 2.1: The start state's BDT in a program with states $Z \supseteq \{\texttt{clrR3}, \texttt{clrR4}, \texttt{mult}\}$, for a counter machine with at least four registers.

2.2.1 Exemplary Machine Program

We want to write a machine program for our counter machine from section 2.1.5 with two registers. The program shall add the two registers up and write the result into the first register while setting the second register to 0. This program can be implemented with three states $Z = \{\texttt{Start}, \texttt{decR2}, \texttt{incR1}\}$, where `Start` is the start state and `End` the end state of $\mathcal{P}_{\mathcal{M}}$. Each state's operation is given in Table 2.2a and the binary decision trees can be seen in Figure 2.2b. The binary decision trees for the states `Start` and `incR1` are identical and shown in one diagram.

State	Operation
Start	NOP
decR2	R2-1
incR1	R1+1

(a) Operations of $\mathcal{P}_{\mathcal{M}}$



(b) Binary Decision Trees of $\mathcal{P}_{\mathcal{M}}$

Figure 2.2: Schematic representation of a machine program $\mathcal{P}_{\mathcal{M}}$

2.2.2 Machine Programs in Text Form

With the goal of interpreting our machine programs in mind, we need to find a fitting textual representation that lends itself to parsing. We define a programming language *MP* that lets us define everything necessary to describe a machine program. Note that this section only describes the features of *MP* needed for this task. More advanced features are described in detail in section 4.2.

Firstly, the model of computation for which the program is written needs to be defined. A model of computation is identified by a name and zero or more parameters. Here, we use the abbreviations from section 2.1 as names. The specific parameters and their order is also detailed in that section. The model of computation definition is always contained in the first non-empty, non-comment line of the source file and is introduced by the keyword `#MOC`. After that follow the name and the parameters, all separated by spaces. For example, this is the definition for a stack machine that has 3 registers and uses the alphabet {`A`, `B`, `C`, `x`}:

```
#MOC SMMOC 3 "ABCx"
```

A file may hold multiple programs. All programs share the same model of computation with each other and are introduced by the line `#PROGRAM <pname>`, where `<pname>` is the name uniquely identifying that program.

Following the program definition, the programs' states are listed, each together with its operation (except for the start state where the operation is omitted) and BDT. The following lines show how the state `<name>` = (`<BDT>` , `<operation>`) is written in *MP*:

```
<name> / <operation>:
    <BDT>
```

The last basic program component to be described is the BDT. The nodes are written line by line, in pre-order. This means that the `false`-associated child of a predicate node always comes before the `true`-associated child. The root node is indented by at least one space or tabular character, which is the first indentation level. All remaining nodes are indented by one more level than their parent. For example, this is the BDT shown in Figure 2.1 as written in *MP*:

```
Start:
    R3=0
        clrR3
        R1=0
            R2=0
                R4=0
                    clrR4
                mult
            End
        End
```

The complete textual representation of the machine program from section 2.2.1 can be viewed in Figure 1 in the appendix.

3 Relations to Classical Definitions in Automata Theory

In automata theory, programs for different automata can be given in widely different formats. In fact, when defining an automaton as a tuple, the program is not even differentiated from the rest of the machine. In [Sco67], these two concepts are separated. To show how programs for arbitrary machines can be written in *MP*, we look at how Scott's programs can be translated. Scott defines programs as a set of instructions which each take on one of the following patterns:

$$\texttt{start} \;:\; \texttt{go to } L \tag{1}$$
$$L \;:\; \texttt{do } F\texttt{; go to } L' \tag{2}$$
$$L \;:\; \texttt{if } P \texttt{ then go to } L' \texttt{ else go to } L'' \tag{3}$$
$$L \;:\; \texttt{halt} \tag{4}$$

where L, L' and L'' are labels in the code (which fulfil the same purpose as our program states), F is an operation (symbol) and P is a predicate (symbol). The special label `start` and the instruction `halt` correspond to the start and end states of a machine program. Figure 3.1 shows how each of these instructions can be expressed as a program state in *MP*.

When talking about Scott's programs and machines, one difference to our models of computation and machine programs that should be mentioned is that Scott defines two additional sets, the input set and the output set, as well as functions to convert input instances into machine states and machine states into output instances. It should be

```
Start:
    L
```

(a) *MP*-translation for (1)

```
L / F:
    L'
```

(b) *MP*-translation for (2)

```
L / NOP:
    P
        L''
        L'
```

(c) *MP*-translation for (3)

```
L / NOP:
    End
```

(d) *MP*-translation for (4)

Figure 3.1:
MP-program state definitions for Scott's instructions

noted that our machine programs are fed machine states directly as the input and also just return their machine state.

The idea with programming in *MP* is that program states can hold both an operation and a whole tree of predicates and subsequent program states. Looking again at the instructions by Scott and their translations, we can make some observations:

Firstly, 3.1d is an instance of 3.1b, since `NOP` is an operation and `End` is a state. Secondly, the binary decision trees of any state can be extended (or shrunk) at will, allowing for the use of any number of predicates in one tree. Thus, the "concatenation" of multiple states of scheme 3.1c can be combined into one state by substitution of program states by their decision trees.[6] And thirdly, the concatenation of a state of scheme 3.1b into one of scheme 3.1c can be combined into one state that takes the operation `F` from the first state and the BDT of the second.

We end up with just two general schemes for program states, shown in Figure 3.2. Next, we will look at how translating some automata can be done directly, without first having to abstract between machine and program.

```
Start:
    T
```

(a) Generic scheme for the `Start` state

```
L / F:
    T
```

(b) Generic scheme for any state `L` $\in Z \setminus \{z_{Start}\}$

Figure 3.2: Generic program state schemes in *MP*. `T` abbreviates a BDT of any size.

3.1 Finiteness of Machine Programs

Machine programs can generally run forever without terminating regardless of what model of computation is used. For instance, we can simply omit the `End`-program state from every BDT of a machine program and it will never terminate. While non-terminating behaviour is certainly needed to model a Turing machine, the DFA and the RDPA will always terminate after a maximum of n transitions when given an input of length n.

Although machine programs for the DFAMOC or the RDPAMOC may not always terminate, they possess a characteristic similar to termination as long as no `NOP`-operations are used. If this is the case, the number of steps (one step is the transition from one program state to another) that a machine program for these models of computation can run for before the machine state stays constant is in fact limited. Both of these models of computation define two types of operations: The `ACC`-operation deletes the rest of the input and sets the *accept*-value to `true`. In the case of the RDPAMOC, the stack is unchanged. As for the other type of operation, the DFAMOC defines the operation `NXT`, which simply removes one symbol from the input. The RDPAMOC defines a whole set of

[6]Scheme 3.1a can also be concatenated with following states of scheme 3.1c.

operations `W`Y for any finite string Y of input symbols. Essentially, these `W`Y-operations all work the same and only change the machine state if the remaining input is not empty. Similarly to the `NXT`-operation of the DFAMOC however, one input symbol is always removed in case the input is not already empty.

Clearly, machine programs for these models of computation will empty their given input in at most n steps. With an empty input, the *accept*-value is the only part of the machine state that can still change. If the program has k program states, at most k steps can be taken in the worst case to reach a program state with an `ACC`-operation, visiting each other program state before transitioning into it. Thus, the limit for a machine program for either the DFAMOC or the RDPAMOC to compute a final machine state is limited by $n + k$ steps.

3.2 Reduced Binary Decision Tree

Some models of computation like the DFAMOC and the TMMOC offer sets of predicates which are mutually exclusive, i.e. at most one predicate can hold for any machine state. Intuitively, we do not need to check any predicates after we find that one holds; no predicate needs to appear in a BDT as the right descendant of another predicate. The resulting tree structure is that of a maximally imbalanced tree where all included predicates are chained together on the very left side (the left-most path) of the tree. A BDT which only holds each predicate at most once and only holds predicates on the left-most path shall be called a reduced binary decision tree.

Definition 3.1. Let $\mathcal{M} = (S, O, P)$ be a model of computation. Let $\mathcal{P}_\mathcal{M}$ be a machine program. Let $T_{\mathcal{P}_\mathcal{M}}$ be the BDT of any program state of $\mathcal{P}_\mathcal{M}$.

The left-most leaf in a BDT is the leaf node that is reached by starting at the root and then continuously following the left edge, until a leaf node is reached.

The left-most path in a BDT is the path that starts at the root and always follows the left edge from there on, until it stops at the left-most leaf.

Binary decision trees for models of computations with mutually exclusive predicates can be reduced in the following manner: We consider subtrees that are (right) children of nodes on the left-most path, and incorporate more than one node. During BDT evaluation, a subtree is entered as a result of its parent predicate being `true`. As all predicates are mutually exclusive, we can infer the truth value for every predicate and find the leaf node that will always be the result of BDT evaluation when that subtree is entered. We can therefore replace the whole subtree with that leaf node.

Replacing all the previously mentioned subtrees results in a new BDT which is maximally imbalanced. However, predicates might still appear twice. Trivially, we simply replace every non-first occurrence of a predicate with its left sub-tree without changing the semantics of the tree. The result is a *reduced* binary decision tree.

3.3 Conversion of Deterministic Finite Automata

Definition 3.2 ([HU79])**.** A deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set called the state set,
- Σ is a finite set called the alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accepting states,
- $\delta : Q \times \Sigma \to Q$ is a total function called the transition function.

Definition 3.3. Computation of a DFA M starts with M being in the start state q_0. The first symbol to be read from the input is the left-most symbol.

In any step, the automaton is aware of its current state $q \in Q$ and the current input symbol $a \in \Sigma$. The automaton transitions into the state $q' = \delta(q, a)$ and moves the reading head one space to the right to read the next symbol.

The automaton terminates when the input is fully read. At this point, the automaton accepts the input if an accepting state $q \in F$ has been reached. The language $L(M)$ that is accepted by M is the set of all inputs $w \in \Sigma^*$ which are accepted by M, where $*$ is the Kleene star.

Definition 3.4. A DFAMOC-machine program P accepts the input w if the *accept*-value is set to `true` after the program has run for at most $|w| + k$ steps with w being the initial input and the *accept*-value being set to `false` at the start. The language $L(P)$ that is accepted by P is the set of all valid inputs (inputs consisting only of symbols defined by the model of computation) which are accepted by P.

3.3.1 DFA to MP

Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = \{s_0, s_1, \ldots, s_m\}$, an equivalent machine program for the DFAMOC can be written as follows: First of all, the model of computation is defined using the given symbols, and the program is then introduced:

```
#MOC DFAMOC "s0s1...sm"
#PROGRAM P
```

The accepting behaviour of a DFA is modelled with the `ACC`-operation. We make use of this operation by defining a single program state with the same name:

```
ACC / ACC:
    End
```

Next, we consider each state $q_i \in Q$. For each state q_i, we define one program state which we simply name z_i here. The operation for each program state z_i will always be `NXT`. Its decision tree is formed using the transition function δ: For each symbol $s_j \in \Sigma$, the transition function defines a succeeding state $z_{i,j} = \delta(z_i, s_j)$. The symbols and their related states are put into the BDT of z_i such that the program state $z_{i,j}$ is chosen exactly when s_j is the first of the remaining symbols in the input. If we find that none of these predicates hold, the input must be empty. In this case we either switch into the `End`-program state if $q_i \notin F$, or we switch into the `ACC`-Program state if $q_i \in F$:

```
$z_i$ / NXT:
      I=$s_0$
            I=$s_1$
                  ...
                        I=$s_m$
                              End or ACC
                              $z_{im}$
                        ...
                  $z_{i1}$
            $z_{i0}$
```

Last but not least, the mandatory `Start`-program state is defined with the same BDT that z_0 has.

Theorem 3.5. The translated machine program from Section 3.3.1 accepts the same language as the original DFA.

Proof. The machine program starts execution in the program state `Start`. The BDT is evaluated to derive the next program state.

If the input is empty, the succeeding state will be `End` if $q_0 \notin F$ or `ACC` if $q_0 \in F$. In the first case, the program terminates with a machine state that is not accepting, while in the second case, the resulting machine state has the *accept*-value set to `true`.

Otherwise, let s be the current symbol, i.e. the first symbol of the (remaining) input. If $\delta(q, s) = q_i$, the chosen program state after BDT evaluation will be z_i. Subsequently the operation of z_i, which is `NXT`, is executed and the current symbol is removed from the input.

The following steps are the same: After executing the `NXT`-operation of any program state $z \in \{z_i | q_i \in Q\}$, the program always reads the current symbol and swaps into the according program state as dictated by the original δ function and terminates with the same *accept*-value when the input has been fully read. $\square$

3.3.2 MP to DFA

Definition 3.6. Let $\mathcal{M} = (S, O, P)$ be a DFAMOC. Let $\mathcal{P}_\mathcal{M} = (Z, z_{Start}, z_{End})$ be a machine program for $\mathcal{M}$. Let $T_{\mathcal{P}_\mathcal{M}}$ be a reduced BDT of any state in Z.

We define the set $L := \{\texttt{I=}\hat{s_i} \mid \texttt{I=}\hat{s_i} \in P,\ \texttt{I=}\hat{s_i} \text{ is included in } T_{\mathcal{P}_\mathcal{M}}\}$. Each symbol $\{\hat{s_i} \mid \texttt{I=}\hat{s_i} \in L\}$ is paired up with one program state $z_{\hat{s_i}}$ which is the right child of $\texttt{I=}\hat{s_i}$ in $T_{\mathcal{P}_\mathcal{M}}$. The left-most leaf of the whole tree $z_{\check{s}} \in Z \cup \{z_{End}\}$ is paired up with every remaining symbol $\{\check{s_j} \mid \texttt{I=}\check{s_i} \in P \wedge \texttt{I=}\check{s_j} \notin L\}$. The resulting set of pairs includes one such pair for every symbol from the alphabet of $\mathcal{M}$.

Let $\mathcal{P}_{\mathcal{M}} = (Z = \{z_0, z_1, \ldots, z_k\}, z_0, \texttt{End})$ be a machine program for a DFAMOC $\mathcal{M}$ with the alphabet $s_0, s_1, \ldots, s_m$. Each program state's BDT shall be in reduced form and no program state (except for the start state z_0) shall have the identity operation `NOP`. Furthermore, the program shall always terminate after reading the whole input, i.e. if the remaining input is found to be empty, the program either switches into the `End`-program state immediately, or just after it switches into a program state with the operation `ACC`. We can define a semantically equivalent DFA $M = (Q, \Sigma, \delta, q_0, F)$ in the following way:

First of all, Σ is simply defined as the alphabet of the DFAMOC:

$$\Sigma = \{s_0, s_1, \ldots, s_m\}$$

The set of states Q is closely related to the set of program states and includes the start state q_0:

$$Q = \{q_0, q_1, \ldots, q_k, q_{REJ}\}$$

The set of final states is defined as follows. We first set F to be the set of all states q_i where the operation of the related program state z_i is `ACC`. We then add to this set all the states q_j where, when given a machine state where the input list is empty, the succeeding program state z'_j of the related program state z_j is an accepting program state, i.e. it has the operation `ACC`. The resulting set of final states correlates to the set of program states which will set the *accept*-value to `true`, or will transition into an accepting program state if and when the remaining input is empty. Moreover, the machine program will always accept the input when it ends up in one of these program states with an empty remaining input.

Defining the transition function δ for q_{REJ} and all states q_{ACC} where the related operation is `ACC` is trivial for every symbol $s \in \Sigma$:

$$\delta(q_{REJ}, s) = q_{REJ}$$
$$\delta(q_{ACC}, s) = q_{ACC}$$

The reject-state q_{REJ} will be used to model cases where the machine program switches into the `End`-program state even if not all of the input has been processed yet. Conversely, accepting program states will delete the rest of the input and then swap into an end state because of our limitation on termination with an empty remaining input.

All in all, the original machine program is not concerned with the remaining input in both of these cases, which is why we model our DFA to just stay in the same state until termination.

To define the transition function for the remaining states, we consider each program state z_i and its related state $q_i \in Q$ individually. Each symbol $s_j \in \Sigma$ is allotted one program state z_{s_j} for our program state z_i as per Definition 3.6. That program state has a related state in $Q \cup \{\texttt{End}\}$, which we call q_{s_j}.

$$\delta(q_i, s_j) = \begin{cases} q_{REJ} & z_{s_j} = \texttt{End} \\ q_{s_j} & \text{otherwise} \end{cases}$$

Theorem 3.7. The translated DFA from Section 3.3.2 accepts the same language as the original machine program.

Proof. Operation of the DFA starts in state q_0 and execution of the DFAMOC-machine program starts in program state z_0. We consider the behaviour of the DFA in any given state q_i and that of the machine program in the related program state z_i, starting with the evaluation of the BDT.

If the (remaining) input is empty, the machine program will either directly terminate in the `End`-program state or switch into an accepting program state, i.e. a program state that has the operation `ACC`, before also terminating in the `End`-program state. The DFA will simply terminate in the current state in this circumstance. This state is a final state if and only if the related program state would result in a `true` *accept*-value under the premise that the input is empty.

In the case that the input is not empty, the machine program will switch into a program state z_s depending on the current symbol s. If z_s is the `End`-program state, the machine program will terminate and reject the input. In this case, the δ function dictates that the DFA switches into the state q_{REJ}, in which it will loop until the whole input has been processed, rejecting it. Should z_s not be the `End`-program state, the machine program will switch into it and execute its operation, continuing with the next symbol or accepting the input. Similarly, the DFA will switch into the related state q_s and continue with the next symbol or loop in this state until the whole input has been processed, this time accepting it. $\square$

3.4 Conversion of Real-Time Deterministic Pushdown Automata

Definition 3.8. [7] A real-time deterministic pushdown automaton (RDPA) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite set called the state set,
- Σ is a finite set called the input alphabet,
- Γ is a finite set called the stack alphabet,
- q_0 is the start state,
- $B \in \Gamma$ is the start symbol which is the only symbol on the RDPA's stack initially,

[7] This definition is equivalent to the original definition of a "realtime DPDA" in [PY83].

- $F \subseteq Q$ is the set of accepting states,
- $\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$ is a partial function called the transition function.

Definition 3.9. Computation of a RDPA starts with M being in the start state q_0 and the start symbol B solely residing on the stack. The first symbol to be read from the input is the left-most symbol.

In any step, the automaton is aware of its current state $q \in Q$, the current input symbol $a \in \Sigma$ and the top-most stack symbol $x \in \Gamma$. If $\delta(q, a, x) = (q', y_0y_1 \dots y_n)$ is defined, the automaton does three things:

- The automaton transitions into the state q'
- The symbol x is popped from the stack and the symbols $y_1y_2 \dots y_k$ are pushed onto it so that y_1 is the new top-most symbol
- The input reading head moves one space to the right to read the next input symbol

The automaton terminates if $\delta(q, a, x)$ is not defined for the current state, input symbol, and stack symbol, or when the input is fully read. It accepts the current input if an accepting state $q \in F$ has been reached after reading the whole input. The language $L(M)$ that is accepted by M is the set of all inputs $w \in \Sigma^*$ which are accepted by M.

Definition 3.10. A RDPAMOC-machine program P accepts the input w if the *accept*-value is set to `true` after the program has run for at most $|w| + k$ steps (k being the number of program states in the machine program) with this initial machine state: w is written onto the input, and the *accept*-value is set to `false`. Additionally, the stack holds one symbol, the start symbol, which is the first symbol from the shared alphabet. The language $L(P)$ that is accepted by P is the set of all valid inputs (inputs consisting only of symbols defined by the model of computation) which are accepted by P.

3.4.1 RDPA to MP

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ with $\Sigma = \Gamma = \{B, s_0, s_1, \dots, s_m\}$ be a RDPA. To formulate a semantically equivalent machine program for the RDPAMOC, we first define the model of computation with its alphabet, and establish a program:

```
#MOC RDPAMOC "Bs0s1...sm"
#PROGRAM P
```

For each state $q \in Q$, input symbol $a \in \Sigma$, and stack symbol $x \in \Gamma$ for which the transition

$$\delta(q, a, x) = (p, Y), \; p \in Q, \; Y = y_0y_1 \dots y_n, \; n \geq 0, \; y_i \in \Gamma \; \forall i$$

is defined, we create one program state which we give the name $q\#a\#x$ and the operation `W`Y:

```
q#a#x / WY:
    T
```

Its BDT `T` depends on all the pairs of input symbols $a_i \in \Sigma$ and stack symbols $x_i \in \Gamma$ for which $\delta(p, a_i, x_i)$ is defined: Using the predicates `I=`a_i and `S=`x_i, we add a branch to the BDT for each of these pairs so that during BDT evaluation, the program state $p\#a_i\#x_i$ is reached if and only if the aforementioned predicates hold. [8] Additionally, we add one branch using the predicate `I=` which will lead to the program state `ACC` if $p \in F$, or `End` otherwise. All remaining required leaf nodes will be set as the `End`-program state as well; these leaves can represent undefined instances of δ, or simply unreachable leaves in the BDT.

As for the program state `Start`, we define its BDT by looking up all the symbols $a_i \in \Sigma$ for which the transitions $\delta(q_0, a_i, B)$ are defined. As with the other program states, we add a branch to the BDT for each a_i so that the program state $q_0\#a_i\#B$ is reached if and only if the predicate `I=`a_i holds. Again, we may omit stack predicates as we assume that B is the symbol on the stack at the start.

The program state `ACC` is defined with the operation of the same name, and a BDT that always results in the `End`-program state.

Theorem 3.11. The translated machine program from Section 3.4.1 accepts the same language over the original input alphabet Σ as the original RDPA.

Proof. A step of program execution starts with evaluation of the BDT of the `Start`-program state. The input is written in the input list, the stack holds the start symbol, and the *accept*-value is `false`. Depending on the current input symbol a, we switch into the program state $q_0\#a\#B$ if for $p \in Q$, $Y = y_0y_1 \dots y_n$, $n \geq 0$, $y_i \in \Gamma\ \forall i$, $\delta(q_0, a, B) = (p, Y)$ is defined, otherwise the program terminates as shall be described later. After switching into this program state $q_0\#a\#B$, the operation `W`Y is performed. At this point, we find the machine state to be semantically equivalent to that of the original RDPA: The first symbol of the input has been removed, and the stack now consists of the symbols Y with y_0 being the top-most symbol given that Y is not empty in the first place. The *accept*-value stays `false`.

For as long as the program does not terminate or switch into the `ACC`-program state because the input has been fully processed, the program will evaluate the BDT of the current program state q and assume some program state $p\#a\#x$, given that $\delta(q, a, x) = (p, Y)$ is defined. In this case, we know that the current input symbol is a and the current stack symbol is x. The program state's operation `W`Y is then performed. The first input symbol is again removed, and the symbols Y are written onto the stack after removing the top-most stack symbol, directly correlating to the change of the RDPA's machine state under the given transition.

[8]Note that the stack-predicates are not always needed: As long as Y consists of at least one symbol, the top-most stack symbol is known at the time of BDT evaluation. Thus, the current stack symbol will only need to be checked in binary decision trees of program states where we define the operation to be `W`. We can also leave out some transitions $\delta(p, a_i, x_i)$ if we know that x_i is not the top-most symbol on the stack during BDT evaluation.

If symbols a and x are found in the input and on the stack for which $\delta(q, a, x)$ is not defined for the current program state q, the machine program will directly terminate in the `End`-program state. The *accept*-value is always `false` in this case, meaning that the program rejects the current input. Similarly, the original RDPA will also reject the input in such a case.

If the input is found to be empty during the evaluation of the BDT of some program state $q\#a\#x$ and $\delta(q, a, x) = (p, Y)$ is defined, the program will find itself in one of two states: If $p \in F$, the machine program switches into the `ACC`-program state, setting the *accept*-value to `true` and terminating. If $p \notin F$, the program simply terminates in the `End`-program state with the *accept*-value still set to `false`.

This is semantically equivalent to the RDPA finding itself in the state p with an empty input, at which point it will accept or reject the input depending on whether p is a final state. $\square$

3.4.2 MP to RDPA

Let $\mathcal{P}_{\mathcal{M}} = (Z = \{z_0, z_1, \ldots, z_k\}, \texttt{Start} = z_0, \texttt{End} = z_{REJ})$ be a machine program for a RDPAMOC $\mathcal{M}$ with the alphabet $B, s_0, s_1, \ldots, s_m$. No program state (save `Start`) shall specify the identity operation `NOP` and the program shall always terminate after reading the whole input, i.e. if the remaining input is found to be empty, the program either switches into the `End`-program state immediately, of just after it switches into a program state with the operation `ACC`. To add to that, the program is also not allowed any writing operation on an empty stack during execution. Furthermore, we assume that no BDT has any leaves that can not be reached during evaluation of the BDT on any valid machine state, also considering that the top-most stack symbol can be inferred for any program state that does not have the operation `W` or `ACC`. [9] We can define a semantically equivalent RDPA $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ in the following way, starting with the alphabets Σ and Γ:

$$\Sigma = \{s_0, s_1, \ldots, s_m\}$$
$$\Gamma = \Sigma \cup \{B\}$$

Again, the set of states Q is closely related to the original set of program states Z and includes the start state q_0. Each program state $z_i \in Z$ has one corresponding state $q_i \in Q$ and the `End`-program state z_{REJ} corresponds to the state $q_{REJ} \in Q$:

$$Q = \{q_0, q_1, \ldots, q_k, q_{REJ}\}$$

Similarly to the translation of a DFAMOC-machine program, the set of final states F is to be made up of those states q_i whose related program states z_i have the operation `ACC`, and those states q_j where, when given a machine state where the input list is empty,

[9] In the start program state z_0 the stack always holds the start symbol B. For states with an operation $\texttt{W}Y$ where Y is not empty, the top-most symbol during BDT evaluation is always the first symbol from Y.

the succeeding program state z'_j of the related program state z_j is an accepting program state, i.e. it has the operation ACC.

Considering each original program state $v \in Z \cup \{z_{REJ}\}$ and its corresponding state $q \in Q$, we define the following transitions: Any program state $z_i \in Z$ may mention v in its BDT as a leaf node which can be reached with at least one combination

$$(a_{i,j}, x_{i,j}),\ a_{i,j},\ x_{i,j} \in \{s_0, s_1, \ldots, s_m\}$$

of an input symbol and a stack symbol, given that $a_{i,j}$ is the current input symbol and $x_{i,j}$ the top-most stack symbol during BDT evaluation. For each possible pair $(a_{i,j}, x_{i,j})$ for all of those program states z_i, we define one entry in the transition function:

$$\delta(q_i, a_{i,j}, x_{i,j}) = \begin{cases} (q, Y) & \text{the operation of } v \text{ is of the form } \texttt{W}Y \\ (q, x_{i,j}) & \text{the operation of } v \text{ is } \texttt{ACC} \text{ or } v \text{ is the } \texttt{End}\text{-program state} \end{cases}$$

Additionally, for each program state z_i whose operation is ACC, we define the entries

$$\delta(q_i, a, x) = (q_i, x) \qquad \forall a \in \Sigma\ \forall x \in \Gamma$$

which simply define a loop back to the same state which the RDPA follows until accepting the input, should it find itself in this state. Note that for the state $q_{REJ} \notin F$, no transitions from that state are defined.

Theorem 3.12. The translated RDPA from Section 3.4.2 accepts the same language over the alphabet Σ as the original machine program.

Proof. Operation of the RDPA starts in state q_0 with a stack consisting only of the start symbol, and execution of the RDPAMOC-machine program starts in program state z_0, also with a stack consisting only of the start symbol. We consider the behaviour of the RDPA in any given state q_i and that of the machine program in the related program state z_i, starting with the evaluation of the BDT.

If the (remaining) input is empty, the machine program will either (1) directly terminate in the End-program state or (2) switch into an accepting program state, i.e. a program state that has the operation ACC, before also terminating in the End-program state (We do not consider these program states in the first case.). The RDPA will simply terminate in the current state in this circumstance. This state is a final state if and only if the related program state would result in a `true` *accept*-value under the premise that the input is empty, which is the case only in case (2).

If the input is not empty, the machine program will switch into a program state $z_{a,x}$ depending on the current input symbol a and the top-most stack symbol x. If $z_{a,x}$ is the End-program state, the machine program will terminate and reject the input. In this case, $\delta(q_i, a, x) = (q_{REJ}, x)$ and the RDPA also rejects the input. Should $z_{a,x}$ not be the End-program state, the machine program will switch into it and execute its operation: If the operation is of the form WY, the current input symbol and the top-most stack symbol are removed, and Y is pushed onto the stack. If the operation is ACC, the machine program will terminate after setting the *accept*-value to `true`. Similarly, the RDPA will switch into the related state $q_{a,x}$ and change its input and stack in the same way, or loop in a final state until the whole input has been consumed. $\square$

3.5 Conversion of Turing Machines

Definition 3.13. [10] A (deterministic) Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite set called the state set,
- Σ is a finite set called the input alphabet,
- $\Gamma \supset \Sigma$ is a finite set called the tape alphabet,
- $q_0 \in Q$ is the start state,
- $B \in \Gamma \setminus \Sigma$ is the blank symbol,
- $F \subseteq Q$ is the set of final states,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ is a partial function called the transition function. L means left shift, R means right shift, and N means no shift.

Definition 3.14. The input for a Turing machine consists of a finite string of input symbols on an infinite tape that has the blank symbol written in every other cell. The Turing machine starts in the start state and its head is positioned on the first tape cell of the input, i.e. the first non-blank cell from the left. In one step, the Turing machine will be aware of the current state q and the symbol s currently under the head. If $\delta(q, s) = (q', s', d)$ is defined, the machine will change its state to q', replace the current symbol with s' and then shift its head in the given direction d. If δ is not defined for the current state and symbol, the machine crashes. If the current state is a final state, the machine halts.

We say that the language $L(M)$ that is accepted by M is the set of all inputs $w \in \Sigma^*$ for which M halts.

Definition 3.15. A TMMOC-machine program P accepts the input w if the program terminates with this initial machine state: w is written onto the tape and the tape head is resting over the first symbol of w.

The language $L(P)$ that is accepted by P is the set of all valid inputs (inputs consisting only of symbols defined by the model of computation) which are accepted by P.

[10] This definition strongly resembles that from [HU79]. The only notable difference is the addition of the "no shift"-transition which does not affect the machine's computational capability.

3.5.1 Turing Machine to MP

Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ with $\Sigma = \{s_0, s_1, \ldots, s_m\}$, and $\Gamma = \{\Box\} \cup \Sigma$, an equivalent machine program can be written as follows: First, the model of computation is defined with the tape alphabet, and a single program with some name is defined:

```
#MOC TMMOC "□s_0s_1...s_m"
#PROGRAM P
```

Following that, we need to define a program that mimics the semantics of the transition function δ of M. Each entry which does not result in a final state

$$\delta(q, s) = (q', s', d), q' \notin F$$

will be translated into one program state which we name $q\#s$. The program state's decision tree is dependent on the symbols $\{\hat{s_0}, \hat{s_1}, \ldots, \hat{s_n}\} \subseteq \Gamma$ for which $\delta(q', \hat{s_i})$ is defined:

```
q#s / Ws'd:
    =ŝ_0
        =ŝ_1
            ...
                =ŝ_n
                    CRASH
                    q#ŝ_n
                ...
            q'#ŝ_1
        q'#ŝ_0
```

In the case where δ is not defined for following state and *all* symbols, M can crash in this state. To emulate this, we create a program state CRASH which will write some arbitrary symbol s into the current cell, stay at the position and then loop forever:

```
CRASH / WsN:
    CRASH
```

For entries in δ where the subsequent state is a final state

$$\delta(q, s) = (q', s', d), q' \in F$$

the corresponding program state's decision tree simply consists of the program state End:

```
q#s / Ws'd:
    End
```

The last program state we need to define is the Start state. This program state does two things: It finds out which symbol s is currently at the tape head position and moves into the according state $q_0\#s$. Again, $\{\hat{s_0}, \hat{s_1}, \ldots, \hat{s_n}\} \subseteq \Gamma$ are the symbols for which

$\delta(q_0, \hat{s_i})$ is defined:

```
Start:
    =$\hat{s_0}$
        =$\hat{s_1}$
            ...
                =$\hat{s_n}$
                    CRASH
                    $q_0\#\hat{s_n}$
                ...
            $q_0\#\hat{s_1}$
        $q_0\#\hat{s_0}$
```

In the trivial case that $q_0 \in F$, the whole program can be written with just the program state `Start` that transitions into the program state `End`.

Theorem 3.16. The translated machine program from Section 3.5.1 accepts the same language over the original input alphabet Σ as the original Turing machine.

Proof. The machine program starts execution in the program state `Start`. The BDT is evaluated to derive the next program state. If $\delta(q_0, s) = (q' \notin F, s', d)$ is not defined for the current symbol s the next program state will be `CRASH`, which results in an endless loop. Otherwise, the next program state will be $q_0\#s$. After arriving in this state, the program executes that state's operation $\mathtt{W}s'd$: The symbol at the current head position is replaced by s' and the head is moved into the direction d.

The following steps are all the same: If $\delta(q, s) = (q' \notin F, s', d)$ is not defined for some states q and symbols s, the program ends up in an endless loop; otherwise the program switches into the program state $q\#s$ and executes its operation $\mathtt{W}s'd$. If at any point, a final state would be transitioned into with $\delta(q, s) = (q' \in F, s', d)$, the next program state will be `End` and the program will terminate. In the trivial case that $q_0 \in F$, the program transitions into the `End` state in the first step and terminates. □

An example for this conversion is given in the appendix in the figures 2 (the original Turing machine) and 3 (the translated machine program). The machine program shown in the latter figure is also available in the arbipreter repository (see chapter 4) under the path `machine_programs/tmmoc/reverse_word_ab.mp` .

3.5.2 MP to Turing Machine

Definition 3.17. Let $\mathcal{M} = (S, O, P)$ be a Turing model of computation. Let $\mathcal{P}_\mathcal{M} = (Z, z_{Start}, z_{End})$ be a machine program for $\mathcal{M}$. Let $T_{\mathcal{P}_\mathcal{M}}$ be a reduced BDT of any state in Z.

We define the set $L := \{\mathtt{=}\hat{s_i} \mid \mathtt{=}\hat{s_i} \in P,\ \mathtt{=}\hat{s_i} \text{ is included in } T_{\mathcal{P}_\mathcal{M}}\}$. Each symbol $\{\hat{s_i} \mid \mathtt{=}\hat{s_i} \in L\}$ is paired up with one program state $z_{\hat{s_i}}$ which is the right child of $\mathtt{=}\hat{s_i}$ in $T_{\mathcal{P}_\mathcal{M}}$. The left-most leaf of the whole tree $z_{\check{s}} \in Z \cup \{z_{End}\}$ is paired up with every remaining symbol $\{\check{s_j} \mid \mathtt{=}\check{s_i} \in P \wedge \mathtt{=}\check{s_j} \notin L\}$. The resulting set of pairs includes one such pair for every symbol from the alphabet of $\mathcal{M}$.

Let $\mathcal{P}_{\mathcal{M}} = (Z = \{z_{Start}, z_0, z_1, \dots, z_k\}, z_{Start}, z_{End})$ be a TMMOC-machine program with the alphabet $s_0, s_1, \dots, s_m$. Each program state's BDT shall be in reduced form.

We can define an equivalent Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ in the following way: To start with, we define all symbol-related components of M:

$$\begin{aligned}\Sigma &= \{s_1, s_2, \dots, s_m\}\\ \Gamma &= \Sigma \cup \{s_0\}\\ B &= s_0\end{aligned}$$

F will consist of one final state q_{End}. For each program state $z \in Z$ we define two states q_z and q'_z with the start state being q_{Start}:

$$\begin{aligned}Q &= \{q_{Start}, q'_{Start}, q_{z_0}, q'_{z_0}, \dots, q_{z_k}, q'_{z_k}, q_{End}\}\\ q_0 &= q_{Start}\\ F &= \{q_{End}\}\end{aligned}$$

The transition function δ is defined using the operation o and BDT T of every program state z:

$$\begin{aligned}\forall s_i \in \Gamma : \delta(q_z, s_i) &= \begin{cases}(q'_z, s, d) & , o \text{ is of the form } \mathtt{W}sd\\ (q'_z, s_i, N) & , \text{otherwise}\end{cases}\\ \forall \hat{s_i} \in L : \delta(q'_z, \hat{s_i}) &= (q_{z_{\hat{s_i}}}, \hat{s_i}, N)\\ \forall \check{s_j} \in \Gamma - L : \delta(q'_z, \check{s_j}) &= (q_{z_{\check{s}}}, \check{s_j}, N)\end{aligned}$$

Theorem 3.18. The translated Turing machine from Section 3.5.2 accepts the same language over the alphabet Σ as the original machine program.

Proof. Operation of the Turing machine starts in state q_{Start}. Execution of the machine program starts in program state z_{Start}. At each step, the operation of the current program state z is applied: Unless the current state is the start state or the operation is `NOP`, an operation of the form $\mathtt{W}sd$ is performed. This behaviour is imitated by the Turing machine, which in the state q_z, writes the symbol s and then moves the head into the direction d in case the original operation of the program state was $\mathtt{W}sd$, otherwise the tape and head position are not changed. The state, however, always changes from the current state q_z to q'_z.

After this, the BDT is evaluated: The succeeding state is either $z_{\hat{s_i}}$ if $\hat{s_i} \in L$ or $z_{\check{s}}$ otherwise, depending on the current symbol s_i. If that chosen program state is the end state, the program terminates. If not, the program continues by again applying the successor's operation, and so on. The Turing machine emulates this behaviour in the state q'_z by swapping into the state $q_{z_{\hat{s_i}}}$ or $q_{z_{\check{s}}}$ accordingly, while not changing the tape and head position. If in the machine program the succeeding program state is the end state, the new state in the Turing machine is q_{End} which is a final state, terminating the program. $\square$

3.6 Conversion of Linear Bounded Automata

Definition 3.19 ([HU79])**.** A (deterministic) linear bounded automaton (LBA) is a special Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ with the following constraints:

- $\Gamma - \Sigma$ includes two additional special symbols $\triangleright$ and $\triangleleft$ which are the left and right bounds of the tape.
- $\triangleright$ and $\triangleleft$ are always the first and last symbols of the input, can not be moved past and can not be overwritten.

As a LBA is just a Turing machine with additional constraints, its translation from and to a LBA-machine program is analogous. While the bounds are not represented by actual symbol on the tape in our model of computation, they present themselves as the predicates `LE` and `RE` which can be regarded as an equivalent representation.

4 The Program Arbipreter

Arbipreter is a command-line interface program which can interpret *MP*-programs for an expandable set of models of computation. All models of computation from Section 2.1 are implemented and can be used to interpret machine programs.

The source code for *Arbipreter* is available at `https://github.com/kwie98/arbitrary-interpreter` and can be compiled and used with the Haskell Stack build tool (`https://docs.haskellstack.org/`). To build the project, the command `stack build` can be used. This creates an executable in a stack-specific subdirectory which can be accessed from the project root with the `stack exec`-command. Alternatively, the command `stack install` additionally copies the executable to the local bin path to make it more easily accessible.

4.1 Usage

If the executable is run without sufficient arguments, it will print a usage text which lists all available options for the user, some of which are mandatory for every use-case (except showing the usage text, of course). The short forms for all options can also be found in the usage text.

The first argument is the *MP*-source file path which can contain one or multiple programs for one specific model of computation. Some example programs are included in the repository in the folder `machine_programs`.

The next information to be passed to the program is the input machine state, which is parsed by Haskell's `read` function and needs to conform to the machine state format of the program's model of computation. The input can be given directly on the command line using the `--input`-option, or it can alternatively be read from a file using the `--long-input`-option.

If the passed source file includes more than one program, the user additionally needs to pass the name of the specific program using which is to be interpreted. This is done using the `--program`-option.

The option `--max-steps` allows the user to limit the maximum number of steps that the program should be interpreted for. One step is counted at each transition from one program state to another, so if `--max-steps=0` is specified, interpretation stops in the `Start`-state.

By default, the interpreter executes the program until termination and then outputs the final program state and machine state. To be shown more information during interpretation, the user can use the `--trace`-option to have the interpreter print a program execution trace to the standard output. This trace shows the following information about each step:

- The current program state
- The machine state after the program state's operation has been applied
- The list of predicates that were evaluated and their truth values

The trace is printed in a `.csv`-format with semicolon separators. For the CMMOC and the SMMOC, the individual registers and stacks are further separated into individual columns.

4.2 Advanced Machine Program Features

4.2.1 Program Calls

Program calls give the programmer the ability to combine multiple programs from one file to write more complex programs more easily. In any given program, another program from that same source file can be referenced as a program state's operation with a dollar sign (`$`) and that program's name. When such a state is reached during interpretation, execution of the original program is momentarily stopped as the referenced program is run on the current machine state until it halts. Then, the original program continues with the now changed machine state.

For example, the programmer can create a rather simple program for a TMMOC with the name `MOVETOLEFT` . This program moves the tape head to the left until it encounters a blank symbol, at which point it moves the tape head to the right by one symbol. The programmer can then use this program as an operation in one or more states of another program by using `$MOVETOLEFT` as the operation of the state.

Note that steps taken in called programs are not counted for the main program. Furthermore, program calls are not traced, so the program trace will only show that a program has been called; the steps taken in that other program are not recorded. Also, program calls are not to be confused with function calls, which typically execute another program in a separate environment. With program calls, each program works with the same shared machine state. To add to that, recursive or looping program calls are not possible, and programs can only reference other programs that appear before them in the same source file.

4.2.2 Program Calls with Register Permutation

Register-based models of computation like the CMMOC and SMMOC allow for even more advanced program calls. Register permutation provides the option of changing the order of registers before executing the referenced program. This order is specified as a sequence of integers which correlate to register indices. Mathematically, this order corresponds to the inverse of the applied permutation, given in one-line notation.

Register permutation is mostly useful for referencing programs which only change a subset of all the registers. For example, the programmer can define an addition program `ADD` for the CMMOC which adds the first two registers up and writes the sum into the first register. If the CMMOC has more than just two registers however, the program can also be used for any other two registers. If the CMMOC has five registers, the third and fifth register can be summed up into the latter register with the program call `$ADD 5 3`, which is equivalent to the program call `$ADD 5 3 1 2 4`. Note that programs are never limited in how many registers they can change, even if the permutation order mentions just one register.

4.3 Implementation Details

4.3.1 Parsing

This section explains some of the inner workings of the interpreter using Haskell's type signatures. Type signatures specify the types of data constructs (data types) and functions. For functions, the types of all parameters and the returned value are defined. All custom data types used in the interpreter are defined in one module, `ArbitraryInterpreter.Defs`.

Fundamentally, a *MP*-source file consists of two parts: The definition of the model of computation, and one or more programs which all use that common model of computation. These two parts are parsed independently from each other and in this process, the following two structures are created:

Model of Computation

Models of computation are implemented as a data type `MoC`. A `MoC` is made up of three functions: First and second, the functions `ops` and `preds` take the name of an operation or predicate and, if the name is valid, return the corresponding operation or predicate in the form of a function. An operation has the type `MachineState -> MachineState`, and a predicate has the type `MachineState -> Bool`. The `MachineState` type is just a synonym for the type `String`. Generally, these strings are `show`-representations of other data types, meaning that any data type that is an instance of Haskell's `Show`-class can be used as a machine state. To assure that machine states are valid, the `validState` function is defined as the third component of a model of computation. This function has the type `MachineState -> Maybe String` and returns an error message if the machine state is invalid. This function is used to check the machine state before and during interpretation. If the state is valid, the function simply returns `Nothing`.

During parsing, the type of model of computation is decided based on the given name. A model of computation is then constructed based on the given parameters. Optionally, the model of computation is extended with additional information and functions for register permutation and customised machine state printing.

Machine Programs

The source code describing programs is organised as a hierarchy. The whole file contains one or more programs. Each of these programs contains one or more states, and each state consists of a state name, an operation (except for the `Start`-state) and a BDT. The parser follows this hierarchy, splitting up the source code into smaller and smaller parts until each program state is parsed individually. A map is created to represent each program, mapping state namen to their corresponding operations and binary decision trees. Each program is then added to a program map one by one. In this process, the programs are also added as operations to the `ops` function of the model of computation.

4.3.2 Program Evaluation

Programs are evaluated in an iterative process. Given a program state and a machine state, the succeeding program state is first determined by evaluating the BDT. If the `End`-program state is reached, evaluation stops without any further manipulation of the machine state. Otherwise, the new program state's operation is looked up and applied to the machine state. The new program state and machine state, as well as a trace of the BDT evaluation, is returned. If the `--trace`-option is active, this information is printed at every iteration. Otherwise, only the last program state and the resulting machine state are printed. Program evaluation may be prematurely stopped by defining a maximum number of iterations with the `--max-steps`-option.

4.4 Defining Additional Models of Computation

Support for additional models of computation can be added by following these general steps, of which all but the first are outlined in more detail after this list:

- Creating a module for the model of computation in `ArbitraryInterpreter.MoC` and importing the new module in `ArbitraryInterpreter.Parse.ParseMoC`
- In this module, writing a function which returns a `MoC` given a list of parameters: `newMOC :: [String] -> MoC`
- Optionally, if the model of computation works on registers: Writing custom permutation and printing functions in `ArbitraryInterpreter.MoC.Permuters` and `ArbitraryInterpreter.MoC.Prettifiers`
- Choosing a uniquely identifying name for the model of computation and adding a case to `ArbitraryInterpreter.Parse.ParseMoC.parseMoC`

Defining the constructor. A `MoC` is constructed purely from a list of zero or more parameters passed as `[String]`. For safety, these parameters should be checked first to see that they are valid. Following after, the components of the `MoC` which are the `ops`, the `preds`, and the `validState` functions should be specified.

The functions

```
ops :: OpName -> Maybe (MachineState -> MachineState)
```

and

```
preds :: PredName -> Maybe (MachineState -> MachineState)
```

should return `Nothing` for invalid operation or predicate names (strings), and `Just` the respective operation or predicate otherwise. The special operation `NOP` is always added to the `MoC` and thus should not be implemented here; otherwise a runtime error will occur during parsing of the model of computation. Operations should also not start with the dollar symbol (`$`) as it is used for program calls. The function

```
validState :: MachineState -> Maybe String
```

should be defined to return `Nothing` for valid machine states, and `Just` an error message otherwise.

Defining permuters and prettifiers. If the machine state is a data structure with many ordered values of the same type, it is possible to add support for register permutation and special printing. These features are realised independently as two functions, `permuteMState` and `printMState`, as part of the `MoCInfo` construct. This construct also includes a mandatory field for the number of registers. Existing functions for other models of computation are implemented in the `ArbitraryInterpreter.MoC.Permuters` and `ArbitraryInterpreter.MoC.Prettifiers` modules.

The function

```
permuteMState :: Maybe ([Int] -> MachineState -> MachineState)
```

takes a permutation in one-line notation and applies the inverse of that permutation to the machine state.

The function

```
printMState ::  Maybe (MachineState -> String)
```

takes a machine state and outputs a `.csv`-formatted string where all values are separated by semicolons.

Integration into the parser. In order to make the new model of computation accessible to the parser, it needs to be integrated into the `parseMoC` function as a new case-branch. The model's unique code-name should be matched against here and the `MoC` and `MoCInfo` constructs should be integrated like the other models of computation.

References

[Cha20a] Maurice Chandoo. "A Systematic Approach to Programming". In: *CoRR* abs/1808.08989 (2020). arXiv: 1808.08989.

[Cha20b] Maurice Chandoo. "Separating Algorithmic Thinking and Programming". In: (2020). DOI: 10.15488/9177.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Reading, Mass: Addison-Wesley, 1979. ISBN: 020102988X.

[Lam61] Joachim Lambek. "How to Program an Infinite Abacus". In: *Canadian Mathematical Bulletin* 4.3 (1961), pp. 295–302. DOI: 10.4153/cmb-1961-032-6.

[Min61] Marvin L. Minsky. "Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines". In: *The Annals of Mathematics* 74.3 (1961), pp. 437–455. DOI: 10.2307/1970290.

[MP43] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/bf02478259.

[PY83] Jan Pittl and Amiram Yehudai. "Constructing a realtime deterministic push-down automaton from a grammar". In: *Theoretical Computer Science* 22.1-2 (1983), pp. 57–69. DOI: 10.1016/0304-3975(83)90138-x.

[Sco67] Dana Scott. "Some definitional suggestions for automata theory". In: *Journal of Computer and System Sciences* 1.2 (1967), pp. 187–212. DOI: 10.1016/s0022-0000(67)80014-x.

[Sor13] Juha Sorva. "Notional machines and introductory programming education". In: *ACM Transactions on Computing Education* 13.2 (2013), pp. 1–31. DOI: 10.1145/2483710.2483713.

Appendix

```
#MOC CMMOC 2

// R1 <- R1 + R2
#PROGRAM ADD
Start:
    R2=0
        decR2
        End

decR2 / R2-1:
    incR1

incR1 / R1+1:
    R2=0
        decR2
        End
```

Figure 1: Textual representation of a machine program

$$M = (\{q_0, q_a, q_{a2}, q_b, q_{b2}, q_L, q_F\}, \{\mathrm{a}, \mathrm{b}\}, \{\mathrm{B}, \mathrm{a}, \mathrm{b}\}, \delta, q_0, B, F)$$

$$\begin{aligned}
\delta(q_0, \mathrm{B}) &= (q_F, \mathrm{B}, \mathrm{N}) & \delta(q_L, \mathrm{B}) &= (q_0, \mathrm{B}, \mathrm{R}) \\
\delta(q_0, \mathrm{a}) &= (q_a, \mathrm{B}, \mathrm{R}) & \delta(q_L, \mathrm{a}) &= (q_L, \mathrm{a}, \mathrm{L}) \\
\delta(q_0, \mathrm{b}) &= (q_b, \mathrm{B}, \mathrm{R}) & \delta(q_L, \mathrm{b}) &= (q_L, \mathrm{b}, \mathrm{L}) \\
\\
\delta(q_a, \mathrm{B}) &= (q_{a2}, \mathrm{B}, \mathrm{L}) & \delta(q_b, \mathrm{B}) &= (q_{b2}, \mathrm{B}, \mathrm{L}) \\
\delta(q_a, \mathrm{a}) &= (q_a, \mathrm{a}, \mathrm{R}) & \delta(q_b, \mathrm{a}) &= (q_b, \mathrm{a}, \mathrm{R}) \\
\delta(q_a, \mathrm{b}) &= (q_a, \mathrm{b}, \mathrm{R}) & \delta(q_b, \mathrm{b}) &= (q_b, \mathrm{b}, \mathrm{R}) \\
\delta(q_{a2}, \mathrm{a}) &= (q_L, \mathrm{B}, \mathrm{L}) & \delta(q_{b2}, \mathrm{b}) &= (q_L, \mathrm{B}, \mathrm{L})
\end{aligned}$$

Figure 2: A Turing machine which accepts the language $\{ww^R \mid w \in \Sigma^*\}$

Figure 3: A TMMOC-machine program which accepts the language $\{ww^R \mid w \in \Sigma^*\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

```
#MOC TMMOC "Bab"

#PROGRAM WORD_REVERSEWORD

Start:
    =B
        =a
            =b
                crash
                q0#b
            q0#a
        q0#B

crash / WBN:
    End

q0#B / WBN:
    End

q0#a / WBR:
    =B
        =a
            =b
                crash
                qa#b
            qa#a
        qa#B

q0#b / WBR:
    =B
        =a
            =b
                crash
                qb#b
            qb#a
        qb#B
```

Figure 3: A TMMOC-machine program which accepts the language $\{ww^R \mid w \in \Sigma^*\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

```
qa#B / WBL:
    =a
        crash
        qa2#a

qa#a / WaR:
    =B
        =a
            =b
                crash
                qa#b
            qa#a
        qa#B

qa#b / WbR:
    =B
        =a
            =b
                crash
                qa#b
            qa#a
        qa#B

qa2#a / WBL:
    =B
        =a
            =b
                crash
                qL#b
            qL#a
        qL#B
```

Figure 3: A TMMOC-machine program which accepts the language $\{ww^R \mid w \in \Sigma^*\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

```
qb#B / WBL:
    =b
        crash
        qb2#b

qb#a / WaR:
    =B
        =a
            =b
                crash
                qb#b
            qb#a
        qb#B

qb#b / WbR:
    =B
        =a
            =b
                crash
                qb#b
            qb#a
        qb#B

qb2#b / WBL:
    =B
        =a
            =b
                crash
                qL#b
            qL#a
        qL#B
```

Figure 3: A TMMOC-machine program which accepts the language $\{ww^R \mid w \in \Sigma^*\}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$

```
qL#B / WBR:
    =B
        =a
            =b
                crash
                q0#b
            q0#a
        q0#B

qL#a / WaL:
    =B
        =a
            =b
                crash
                qL#b
            qL#a
        qL#B

qL#b / WbL:
    =B
        =a
            =b
                crash
                qL#b
            qL#a
        qL#B
```