SCIENCE CHINA
Information Sciences

• Supplementary File •

The Verification of Conversion Algorithms between Finite Automata

Dongchen JIANG[1*] & Wei LI[2]

[1]*School of Information Science & Technology, Beijing Forestry University, Beijing* 100083*, China;*
[2]*State Key Laboratory of Software Development Environment, Beihang University, Beijing* 100191*, China*

Automata theory is fundamental to computer science, and it has broad applications in lexical analysis, compiler implementation, software design and program verification [8, 9]. As an indispensable part of automata theory, the conversion algorithms between finite automata are frequently used to construct decision procedures for language recognition. In recent years, there is a tendency to formalize and verify the correctness of various conversion algorithms using interactive theorem provers. One of the motivations is to obtain verified decision procedures for language recognitions.

In 1964, Brzozowski firstly introduced the notion of derivatives and proposed an algorithm to convert a regular expression into a deterministic finite automaton [3]. Then there come many researches on this subject. In practice, the classical approach normally contains four steps: firstly, to obtain a non-deterministic finite automaton with epsilon-transitions (ε-NFA) from the regular expression; secondly, to remove the epsilon-transitions and get a non-deterministic automaton (NFA); thirdly, to convert the NFA into a DFA; and finally to minimize the DFA.

The formalization of finite automata was first accomplished by Jean-Christophe Filliâtre using Coq [6]. A constructive proof of the equivalence between regular expressions and ε-NFA was also proposed. In 2010, Braibaut and Pous first verified the complete procedure of converting the regular expressions into automata [2]. With the help of matrix, their implementation is efficient. Almeida, Moreira, Pereira and Sousa provided a constructive proof different from the matrix-based verification for a partial derivative automata construction from a regular expression, and also discussed the termination property [1]. In 2015, Lawrence introduced the new datatype of hereditarily finite set, and used it to simplify the constructive proofs of the correctness of the conversion algorithms [14]. In 2016, Doczkal and Smolka verified the conversion from two-way automata to one-way automata in Coq [5].

The purpose of this paper is towards the implementation of three proved, terminable and correct specifications of conversion algorithms in automata theory, which is part of the Computer Aided Verification of Automata project. We will construct three correct and verified conversion algorithms between finite automata using Isabelle/HOL, i.e. the algorithm that removes epsilon-transitions of an ε-NFA to obtain a normal NFA, the powerset construction algorithm to convert an NFA into a DFA and the Hopcroft's algorithm to minimize a DFA. These specifications enable the generation of correct functional programs by Isabelle/HOL Collection Framework (ICF) [10].

Different from the existing formalizations, the finite automata are formalized by using the datatype of list, and the correctness verification of the algorithms are directly conducted on list. Its main advantage lies in that we can make full use of the properties of specifications and therefore shorten the length of verification. The proofs are similar to the textbook proofs which avoid extra construction and verification of the equivalent conversion between different datatypes.

This supplementary file includes two appendixes. In Appendix A, we will introduce the preliminaries and provide the formalizations of relevant concepts in automata theory. In Appendix B, we will incorporate the specifications of the three conversion algorithms and their correctness verification.

Appendix A Preliminaries

Appendix A.1 Introduction to Isabelle/HOL

Isabelle/HOL is an interactive theorem prover, which allows mathematical formulas to be expressed in terms of formal language and provides tools to prove those formulas in a logical calculus [13]. It has been instantiated to support reasoning

[*] Corresponding author (email: jiangdongchen@bjfu.edu.cn)

in first-order logic, higher-order logic, ZermeloCFraenkel set theory, the logic for computable functions, etc. Its main application is to provide formal mathematical proofs and formal verification, and also includes proving the correctness of software or algorithm in the specification level. With the correct specifications, the correct executable codes can be generated by the code-generator of Isabelle/HOL [7].

In Isabelle/HOL, all terms are processed with types, e.g. $a :: \alpha$ means that the term a has a datatype of α. α, β, γ and so on are all commonly used type variables. Compound types are also allowed, e.g. (τ_1, τ_2) refers to the type of ordered pairs and α *list* represents the type of lists whose elements have a type of α. Functions also have types: a function f that has two arguments carries a type of $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, and its application can be written as $f\ a\ b$ where a and b are the arguments of f. Lambda terms are used to express functions and are written in standard syntax, i.e. $\lambda x.\ t$.

As one purpose of this paper is to provide correct specifications for code generation by ICF, the datatype of some basic concepts should be selective. For example, selection and traversal operations are required in some specifications, which makes me formalize most sets in automata theory by distinct *list*. In Isabelle/HOL, lists come with the following functional applications: *size* (a function to calculate the length of a list), *set* (a function to convert a list into a set of its elements), [] (the empty list), # (a list constructor to add an element in front of an existing list), @ (a function to append a list to another), *hd* (the function to acquire the first element of *xs*) and $xs!i$ (the i^{th} element of list *xs*).

Appendix A.2 Finite Automata

Normally, a finite automaton (FA) is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, δ is the transition function which describes the state transition behavior of the finite automaton (in a DFA δ is a function from $Q \times \Sigma$ to Q, while in an NFA δ is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$), q_0 is the initial state, and F is a subset of Q which represents the set of accept states.

In our formalization, a record of five fields is used to describe the datatype of finite automata. In Isabelle/HOL, records merely generalize the concept of tuples, where components may be addressed by labels instead of positions. Then, the datatype of a finite automaton is constructed as follows:

$$\begin{aligned} \textit{Record}\ (\alpha, s) fa = \quad & Q ::\ s\ \textit{list} \\ & \Sigma ::\ \alpha\ \textit{list} \\ & \delta ::\ (s, \alpha, s)\ \textit{list} \\ & q_0 ::\ s \\ & F ::\ s\ \textit{list}. \end{aligned}$$

Here, $(\alpha, s)fa$ is the datatype for finite automata, α is the datatype for characters, and s is the datatype for states.

In this formalization, all sets of finite automata carry a datatype of *list*, which is different from the definition in textbooks, because the traversal of states or characters is required in all the conversion algorithms and it is more practical to pick up elements from a list than from a set.

Out of the similar reason, the transition function is also formalized by a list of 3-tuples. The first and the third element of each tuple are states of Q, and the second element is a character of Σ. Then the definition of finite automata can be formalized by the following specification:

$$\begin{aligned} \textit{Finite\_Automata}\ (|Q, \Sigma, \delta, q_0, F|) \ = \ & (\textit{distinct}\ Q \ \wedge \textit{distinct}\ \Sigma \wedge \textit{distinct}\ \delta \wedge \textit{distinct}\ F \\ & \wedge\ q_0 \in \textit{set}\ Q \ \wedge \textit{set}\ F \subseteq \textit{set}\ Q \\ & \wedge (\forall (q, a, q') \in \textit{set}\ \delta.\ q \in \textit{set}\ Q \wedge a \in \textit{set}\ \Sigma \wedge q' \in \textit{set}\ Q)). \end{aligned}$$

In this definition, Q, Σ, δ and F are all distinct lists used to formalize sets, q_0 is the initial state and F represents the final state set. δ is a list of 3-triples which formalizes the transition function. If (q, a, q') is a 3-tuple of δ, then there is a transition from q to q' labeled by a. In this paper, we call (q, a, q') a transition triple, and δ contains all the transition triples of an finite automaton.

The advantage of formalizing a transition function with a 3-tuple list lies in that we can use a unified specification to describe the general properties of both DFA and NFA: if an automaton is a DFA, every state has only one exact transition triple for each character; if an automaton is an NFA, a character can lead to only one, more than one or no transition triple for a given state. Thus, no matter what kind of automaton it is, the transition function can always be formalized by the list of transition triples.

For an NFA, for each $a \in \Sigma$ and $q \in Q$, there can be one, more than only one or no transition triple with a form of $(q, a, *)$ in δ ($*$ stands for an arbitrary state of Q). No other requirements are needed. Thus, the specification of finite automata is the specification of NFA, i.e. $(|Q, \Sigma, \delta, q_0, F|)$ is an NFA if and only if it is a *Finite_Automata*.

While for a DFA, δ should also satisfy the uniqueness requirement, i.e.

$$\forall q \in \textit{set}\ Q.\ \forall a \in \textit{set}\ \Sigma.\ \exists! q' \in \textit{set}\ Q.\ (q, a, q') \in \textit{set}\ \delta. \tag{A1}$$

Here, $\exists! q' \in \textit{set}\ Q$ means that there exists a unique q' in *set* Q which satisfies its following property. Then an automaton is a DFA if and only if it is a *Finite_Automata* that satisfies Property (A1).

Appendix A.3 Word Acceptance

As we have discussed earlier, an alphabet is formalized by a distinct nonempty list in this paper. A list Σ of characters is an alphabet if and only if

$$\Sigma \neq [] \ \wedge \ distinct\ \Sigma.$$

If Σ is an alphabet, a language L over Σ is a set of finite sequence (or list) of characters over Σ. If Σ^* denotes the set of all sequences over Σ, then a language L over Σ is a subset of Σ^*, i.e. $L \subseteq \Sigma^*$. We call w a word of L if $w \in L$ holds.

For each regular language L, there is only one empty word ε which contains no character and satisfies the following property for arbitrary word $w \in L$:

$$\varepsilon \cdot w = w \cdot \varepsilon = w.$$

Conversion algorithms are used to simplify automata without changing the recognizable language. Generally, an automaton recognize a language L if and only if it accepts every word of L. In this paper, a finite automaton fa accepts a word w can be defined as follows:

$$\begin{aligned} accept\ fa\ w = (&\exists slist.\ size\ slist = size\ w + 1 \\ &\wedge slist!0 = q_0 \\ &\wedge (\forall i < size\ w.\ (slist!i, w!i, slist!Suc\ i) \in set\ \delta) \\ &\wedge slist!(size\ w) \in set\ F). \end{aligned}$$

Here, fa accepts w if there is a state list $slist$ whose first element is the initial state, whose last element is a final state, and in which every two adjacent states together with the corresponding character of the word constitute a transition triple of fa.

Thus, to show that the automata fa_1 and fa_2 recognize the same language is to show they accept the same words, i.e.

$$\forall w.\ accept\ fa_1\ w = accept\ fa_2\ w.$$

Appendix A.4 Non-deterministic Loops

In Isabelle/HOL, a non-deterministic *while* combinator is defined as *while_option*. It takes a check function $b :: \alpha \Rightarrow bool$, a process function $c :: \alpha \Rightarrow \alpha$ and a variable $s :: \alpha$ as inputs, and iterates by following recursion equation:

$$while\_option\ b\ c\ s \ = \ (if\ b\ s\ then\ while\_option\ b\ c\ (c\ s)\ else\ Some\ s).$$

This equation is executable, but may not terminate. To show a *while_option* terminates, it is necessary to construct a monotonically decreasing function $f :: s \Rightarrow nat$ which decreases when the process function c is applied on s. Then the termination theorem of *while_option* comes as follows:

Theorem 1. $(\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s) \wedge f\ (c\ s) < f\ s) \Longrightarrow P\ s \Longrightarrow EX\ t.\ while\_option\ b\ c\ s = Some\ t.$

while_option provides the basic non-deterministic loop operation for the conversion algorithms. But for specific conversion algorithms, a worklist-based non-deterministic loop is always required.

In automata theory, a worklist-based procedure is always applied to collect some kind of reachable states from a given state. For example, in the conversion algorithm from an NFA to a DFA, a worklist-based procedure is used to collect all the reachable states of the target DFA. And in the conversion algorithm from an ε-NFA to an NFA, the procedure is not only used for collecting all the reachable states of the target NFA but also for calculating all the ε-reachable states from a given state.

In a worklist-based procedure, a worklist is used for storing all the unprocessed elements. The procedure repeatedly picks an element from the worklist, processes it to acquire the intermediate result, and removes it from the worklist. The procedure terminates when the worklist becomes empty, and the final result (the intermediate result) is obtained. However, the worklist-based procedure may not terminate, as processing an element may generate new unprocessed elements to add to the worklist.

The worklist-based procedure is formalized as a framework *worklist* in Isabelle/HOL. The framework takes three arguments: $accum :: s \Rightarrow r \Rightarrow r$ is the function calculating the intermediate result according to the currently unprocessed element and the last intermediate result; $extend :: s \Rightarrow r \Rightarrow s\ list$ is the function calculating new unprocessed elements; and $wr :: (s\ list * r)$ is the pair of the worklist and the intermediate result. Then *worklist* can be defined as follows:

$$worklist\ accum\ extend\ sr = while\_option\ b\_wl\ (c\_wl\ accum\ extend)\ wr.$$

In this definition, there are two assistant functions: b_wl and c_wl. $b\_wl = (\lambda sr.\ fst\ sr \neq [])$ is the check function to judge when the *while_option* terminates; c_wl is the function that combines *extend* and *accum*, and the combined function $(c\_wl\ accum\ extend)$ is applied on wr.

When we formalize a conversion algorithm by this framework, the intermediate result is also a list of states. Then the datatype of wr is $(s\ list, s\ list)$, and the function *extend* and function *accum* can be concreted by a state-expansion function *expand* that calculates all kinds of reachable states from a given state. The concerted functions are formalized as follows:

$$\begin{aligned} &acc\_fa\ expand = (\lambda\ s\ rs.\ [x \leftarrow expand\ s.\ x \notin set\ rs]@rs); \\ &ext\_fa\ expand = (\lambda\ s\ rs.\ [x \leftarrow expand\ s.\ x \notin set\ rs]). \end{aligned}$$

After the above preparation, we can construct a state collection framework *collect_option* by *worklist*. It begins with a list *slist* of states, and collects all the accessible states step by step according to the expansion function *expand*.

$$\mathit{collect\_option}\ \mathit{expand}\ \mathit{slist}\ =\ \mathit{worklist}\ (\mathit{acc\_fa}\ \mathit{expand})\ (\mathit{ext\_fa}\ \mathit{expand})\ (\mathit{slist}, \mathit{slist}).$$

Framework *collect_option* calculates all the reachable states from the states of *slist* according to the state-expansion function *expand*. With this framework, what is needed is to construct the specific expansion function *expand* to collect different kinds of states of an automaton.

According to the termination theorem of *while_option*, the general termination condition of *collect_option* can be obtained by carefully constructing a measure function. And we prove that if there exists a finite set S for arbitrary s in S, any element of *expand s* belongs to S (i.e. S is closed under *expand*), and the list of the unprocessed elements is distinct, then *collect_option expand slist* terminates. Then we have

Theorem 2.

$$\mathit{finite}\ S \Longrightarrow \forall s \in S.\ \mathit{set}(\mathit{expand}\ s) \subseteq S \Longrightarrow \mathit{set}\ \mathit{slist} \subseteq S \Longrightarrow \forall s.\ \mathit{distinct}(\mathit{expand}\ s) \Longrightarrow \mathit{distinct}\ \mathit{slist}$$
$$\Longrightarrow \exists t.\ \mathit{collect\_option}\ \mathit{expand}\ \mathit{slist} = \mathit{Some}\ t.$$

Appendix B Formalization and Verification

This appendix contains the formalization of three conversion algorithms and the correctness verification of the formalized specifications. The conversion algorithms include: 1) the powerset construction algorithm to convert an NFA to a DFA; 2) the algorithm to remove all ε-transitions of an ε-NFA; and 3) the Hopcroft's algorithm to minimize a DFA. For each algorithm, the correctness verification of its specification involves: all the non-deterministic loops terminate, which is presented along with the process of formalizing; the algorithm output is an expected automaton; and the input and output automata recognize the same language.

Appendix B.1 Converting an NFA into a DFA

The standard method to convert an NFA into a DFA that accepts the same language is though the powerset construction, which was proposed by Rabin and Scott in [15]. We discuss this conversion algorithm here because it is important in practice and uncomplicated in construction.

Appendix B.1.1 *Algorithm Formalization*

The powerset construction applies directly to an NFA $(Q, \Sigma, \delta, q_0, F)$ that does not allow ε-transitions. It outputs a DFA with a form of $(\mathcal{Q}, \Sigma, \Delta, S_0, \mathcal{F})$. The DFA has an alphabet which is the same with that of the NFA, and its states are all subsets of Q, where $\{q_0\}$ is the initial state. The transition function of the DFA maps a state S (a subset of Q) and a character a to the set $\{q' \mid (q, a, q') \in \mathit{set}\ \delta \wedge q \in S\}$, which means a subset of Q is a state of the DFA if all of its elements can be reached by a-transitions from some state of the DFA. Thus, not all subsets of Q are reachable from the initial state $\{q_0\}$. A state S of the DFA is an accept state if and only if $S \cap \mathit{set}\ F \neq \emptyset$.

Normally, the powerset construction can be implemented by the conversion algorithm $N2D$ in Algorithm B1:

Algorithm B1 $N2D$

```
Require: NFA $nfa = (Q, \Sigma, \delta, q_0, F)$;
Ensure: DFA $dfa = (\mathcal{Q}, \Sigma, \Delta, S_0, \mathcal{F})$;
 1: $\Delta, \mathcal{F} \Leftarrow []$;
 2: $S_0 \Leftarrow \{q_0\}$;
 3: $\mathcal{Q}, W \Leftarrow [S_0]$;
 4: while $W \neq []$ do
 5:     pick $S$ from $W$;
 6:     if $S \cap set\ F \neq \emptyset$ then
 7:         add $S$ to $\mathcal{F}$;
 8:     end if
 9:     for $a \in set\ \Sigma$ do
10:         $S' \Leftarrow \{q' \mid (q, a, q') \in set\ \delta, q \in S\}$;
11:         if $S' \notin set\ \mathcal{Q}$ then
12:             add $S'$ to $W$;
13:             add $S'$ to $\mathcal{Q}$;
14:         end if
15:         add $(S, a, S')$ to $\Delta$;
16:     end for
17: end while
```

$N2D$ is a worklist-based algorithm where the worklist W and the intermediate result $\mathcal{Q}$ are both lists of the states of the DFA and initialed with the initial state S_0. During each iteration step, the algorithm picks one state S from W and calculates all one-step reachable states from S. If a one-step reachable state is not in the intermediate result $\mathcal{Q}$, then it will be added to both $\mathcal{Q}$ and W; otherwise, the process continues. The iteration continues until W becomes empty.

For the conversion algorithm $N2D$, the key of implementation lies in the formalization of the operation (in Line 10 of Algorithm B1) that collects all a-reachable states of the NFA from a set of states according to δ[1]. And we construct the recursive function $CharReach_D$ to implement the operation, i.e.

$$\begin{aligned}&CharReach_D\ []\ S\ a = \emptyset,\\&CharReach_D\ (d\#de)\ S\ a = (\text{if } (Sta\ d \in S \wedge Cha\ d = a)\\&\qquad \text{then } insert\ (End\ d)\ (CharReach_D\ de\ S\ a) \text{ else } CharReach_D\ de\ S\ a).\end{aligned}$$

With $CharReach_D$, the collection of all states in the output DFA can be achieved step by step. However, to collect a state with a given character at a time will cause nested loop, which complicates the formalization and the verification. To simplify the specification and verification, we construct a recursive function $SigmaReach_D$ that collects all the one-step reachable states from S with all the possible characters, i.e.

$$SigmaReach_D\ nfa\ S = remdups\ (map\ (CharReach_D\ \delta\ S)\ \Sigma).$$

In this specification, nfa is a short notation for the input NFA $(Q, \Sigma, \delta, q_0, F)$, map is the standard list function such that $map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$, and $remdups$ is the function to remove all the redundant elements of a given list.

With all the above functions, the specification which calculates all the reachable states of the output DFA can be formalized by the framework $collect\_option$ with $(SigmaReach_D\ nfa)$ as its specific expansion function, i.e.

$$N2Ds\_option\ nfa = collect\_option\ (SigmaReach_D\ nfa)\ [\{q_0\}].$$

For any $S \subseteq set\ Q$, $(SigmaReach_D\ nfa\ S)$ is a distinct list of subsets of $set\ Q$, i.e. $set\ (SigmaReach_D\ fa\ S)$ is a subset of the powerset of $set\ Q$. As the powerset of $set\ Q$ is finite and the initial list $[\{q_0\}]$ is distinct, $N2Ds\_option$ terminates according to the termination theorem of $collect\_option$. Thus, we can calculate all the states of the output automaton in this specification. Then the state list $\mathcal{Q}$ can be acquired by removing the $Some$ constructor from the result of $N2Ds\_option\ nfa$, i.e.

$$\mathcal{Q} = \text{case } N2Ds\_option\ nfa \text{ of } None \Rightarrow []\ |\ Some\ wr \Rightarrow snd\ wr.$$

Here, it should be noted that from the perspective of the input NFA, $CharReach_D\ \delta\ S\ a$ is used to collect all the a-reachable states from S according to δ. But for the output DFA, $(CharReach_D\ \delta)$ is the transition function of the DFA, i.e. for a state S of the DFA and a character a, $CharReach_D\ \delta\ S\ a$ is the state of the DFA that is transited from S with a. Thus, the transition triple list of the DFA can be constructed by using Σ, $\mathcal{Q}$ and $CharReach_D$ as follows:

$$\Delta = delta\_triple\ (CharReach_D\ \delta)\ \mathcal{Q}\ \Sigma.$$

Here, $delta\_triple$ is a function that converts a transition function into its corresponding transition triple list according to the given alphabet and state list. (For more information, please refer to the specifications on http://pan.baidu.com/s/1nu8FGbn.)

After obtaining the state list $\mathcal{Q}$ of the DFA, the list of accept states can be obtained by the filter function in Isabelle/HOL, i.e. selecting the states from $\mathcal{Q}$ which intersects $set\ F$:

$$\mathcal{F} = [x \leftarrow \mathcal{Q}.\ x \cap set\ F \neq \emptyset].$$

Appendix B.1.2 *Correctness Proofs*

To prove $N2D$ is a correct specification of the conversion algorithm from NFA to DFA, we need to show that 1) the output of $N2D$ satisfies all the properties of DFA and 2) the input and the output automata recognize the same language.

Firstly, let us prove the output of $N2D$ is a DFA, i.e.

Lemma 1. $NFA\ nfa \Longrightarrow DFA\ (N2D\ nfa)$

If we assume nfa is an NFA, it is necessary to prove that $(\mathcal{Q}, \Sigma, \Delta, S_0, \mathcal{F})$ satisfies all the properties of $Finite\_Automata$ and Property (A1).

As all the distinction related properties, such as $distinct\ \mathcal{Q}$, $distinct\ \Sigma$, $distinct\ \Delta$ and $distinct\ \mathcal{F}$, can be proved with ease, and the properties $S_0 \in set\ \mathcal{Q}$ and $set\ \mathcal{F} \subseteq set\ \mathcal{Q}$ can be obtained automatically, we should focus more on the properties related to the transition triple list.

According to the construction of Δ, each transition triple in Δ has a form of $(S, a, CharReach\ \delta\ S\ a)$, where $S \in \mathcal{Q}$ and $a \in \Sigma$. As $set\ \mathcal{Q}$ is closed under the function $(CharReach_D\ \delta)$, $CharReach_D\ \delta\ S\ a \in \mathcal{Q}$ holds. Thus,

$$\forall (S, a, S') \in set\ \Delta.\ S \in set\ \mathcal{Q} \wedge a \in set\ \Sigma \wedge S' \in set\ \mathcal{Q}.$$

And for each $S \in \mathcal{Q}$ and each $a \in \Sigma$, $delta\_triple$ collects all the triples that have a form of $(S, a, CharReach\ \delta\ S\ a)$ to Δ. As $CharReach_D\ \delta$ is a function, $CharReach_D\ \delta\ S\ a$ is unique. Therefore, $(S, a, CharReach\ \delta\ S\ a)$ is also unique in Δ, and we have

1) In this paper, an a-reachable state of a state s is a state s' such that (s, a, s') constitutes a transition triple of the corresponding automaton.

$$\forall S \in set\ \mathcal{Q}.\ \forall a \in set\ \Sigma.\ \exists! S' \in set\ \mathcal{Q}.\ (S, a, S') \in set\ \Delta.$$

To sum up, $N2D\ nfa$ is a $Finite\_Automata$ that satisfies Property (A1), i.e. Lemma 1 holds.

Furthermore, if nfa is an NFA, we can prove that the input nfa and the output $N2D\ nfa$ recognize the same language, i.e.

Lemma 2. $NFA\ nfa \Longrightarrow accept\ nfa\ w \longleftrightarrow accept\ (N2D\ nfa)\ w$.

In general, to prove that an automaton accepts a word, the most direct way is to construct a state list and prove that the list satisfies all the properties of *accept*. And if the automaton is a DFA, the state list can be obtained by running the DFA on the input word. In this paper, a recursive function *run* is constructed to simulate the running of a DFA starting from a given state on a given word:

$$\begin{aligned}&run\ []\ dfa\ q = [q],\\ &run\ (x\#xs)\ dfa\ q = q\#(run\ xs\ dfa\ (delta\_fun\ \delta\ q\ x)).\end{aligned}$$

In this specification, $delta\_fun$ is a function that converts a transition triple list into its corresponding transition function. If $dfa = (Q, \Sigma, \delta, q_0, F)$ is a DFA, it can be proved that $run\ w\ dfa\ q_0$ satisfies all the properties of *accept* except $(run\ w\ dfa\ q_0)!size\ w \in set\ F$. Therefore, proving $accept\ dfa\ w$ equals to proving $(run\ w\ dfa\ q_0)!size\ w \in set\ F$ for DFA.

Now, let us prove Lemma 2 and assume nfa is an NFA. As $(N2D\ nfa)$ is a DFA according to Lemma 1, we *run* the DFA $(N2D\ nfa)$ on the word w from the initial state S_0 and obtain the state list $run\ w\ (N2D\ nfa)\ S_0$ for proving its acceptance.

If we assume $accept\ nfa\ w$, a state list $slist_n$ of nfa which satisfies all the properties of *accept* will be obtained. It can be proved that each state in $slist_n$ is one element of the corresponding state in $run\ w\ (N2D\ nfa)\ S_0$, i.e.

$$\forall i \leqslant size\ w.\ (slist_n!i) \in (run\ w\ (N2D\ nfa)\ S_0)!i.$$

Thus, $(run\ w\ (N2D\ nfa)\ S_0)!size\ w \cap set\ F \neq \emptyset$, which means $accept\ (N2D\ nfa)\ w$.

If we assume $accept\ (N2D\ nfa)\ w$ and prove $accept\ nfa\ w$, a state list $slist_d$ of $(N2D\ nfa)$ which satisfies all the properties of *accept* can be obtained. As each state of $(N2D\ nfa)$ is a nonempty state set of nfa, we can pick some state of nfa from the elements of $slist_d$ and combine them as a state list of nfa with the same order. Then it can be proved that this constructed list of nfa satisfies all the properties of *accept*, i.e. $accept\ nfa\ w$.

In this way, Lemma 2 can be proved. Together with Lemma 1, we prove that $N2D$ is a correct algorithm that converts an NFA into a DFA without changing the recognized language.

Appendix B.2 Removing the ε-Transitions of an ε-NFA

NFA with ε-transitions is a further generalization of NFA. This kind of automata replaces the transition function with a function that allows the empty word ε as a possible input. The transitions which consume ε are called ε-transitions. As ε is not a character in any alphabet, it is necessary to define the datatype $\alpha\ eps$ to adjoin a new element *Eps* to a type α:

$$\alpha\ eps = Eps \mid Char\ \alpha.$$

Here, *Eps* is the name of the type constructor for ε, and *Char* is the name of the type constructor for normal characters. The datatype $\alpha\ eps$ is used to model the basic input (characters or ε) that can lead to state transitions of an ε-NFA. Hereby, the datatype of ε-NFA becomes $(\alpha\ eps, s)fa$, and the alphabet becomes $\Sigma \cup \{\varepsilon\}$, which is denoted by Σ_ε.

Appendix B.2.1 *Algorithm Formalization*

In this paper, we convert an ε-NFA $(Q, \Sigma_\varepsilon, \delta, q_0, F)$ into an NFA $(Q', \Sigma, \delta', q_0, F')$ though two steps: firstly, to calculate all the ε-reachable states from any given state and construct all the possible transitions of the output NFA; secondly, to collect all the reachable states of the output NFA according to the newly obtained transitions.

For the first step, the algorithm *FiniteEpsReach* in Algorithm B3 is proposed to calculate all the ε-reachable states from a given state.

This algorithm is a worklist-based algorithm. After the initialization of the worklist W and the state list $Q_{q-\varepsilon}$ with $[q]$, the algorithm repeatedly picks a state q' from W and collects the states that can be transited from q' with an ε-transition to $Q_{q-\varepsilon}$. This process continues until W becomes empty.

This algorithm is implemented by $collect\_option$ with an expansion function *EpsReach*, i.e.

$$FiniteEpsReach\_option\ nfae\ q = collect\_option\ (EpsReach\ \delta)\ [q].$$

In this specification, *EpsReach* is the expansion function that collects all the directly connected states from q with an ε-transition, which is recursively defined as follows:

$$\begin{aligned}&EpsReach\ []\ q = [],\\ &EpsReach\ (d\#ds)\ q = (if\ Sta\ d = q \wedge Cha\ d = eps \wedge End\ d \notin set\ (EpsReach\ ds\ q)\\ &\qquad then\ (End\ d\#EpsReach\ ds\ q)\ else\ (EpsReach\ ds\ q)).\end{aligned}$$

It can be proved that for an arbitrary $q \in set\ Q$ $EpsReach\ \delta\ q$ is a distinct state list, i.e. $set\ Q$ is closed under $(EpsReach\ \delta)$. According to the termination theorem of $collect\_option$, $FiniteEpsReach\_option$ terminates. Then *FiniteEpsReach* can be constructed by removing the *Some* constructor, i.e.

$$FiniteEpsReach\ nfae\ q = \text{case}\ FiniteEpsReach\_option\ nfae\ q\ \text{of}\ None\ \Rightarrow []\ \mid Some\ wr \Rightarrow snd\ wr.$$

Algorithm B2 *FiniteEpsReach*

```
Require: ε-NFA $nfae = (Q, \Sigma_\varepsilon, \delta, q_0, F)$ and $q \in set\ Q$;
Ensure: a state list $Q_{q-\varepsilon}$;
 1: $Q_{q-\varepsilon}, W \Leftarrow [q]$;
 2: while $W \neq []$ do
 3:     pick $q'$ from $W$;
 4:     for $(q', \varepsilon, q'') \in \delta$ do
 5:         if $q'' \notin set\ Q_{q-\varepsilon}$ then
 6:             add $q''$ to $W$;
 7:             add $q''$ to $Q_{q-\varepsilon}$;
 8:         end if
 9:     end for
10: end while
```

Based on *FiniteEpsReach*, we can construct all the possible transitions of the output NFA. This process is implemented by combining finite ε-transitions with one character-transition. In this paper, we call two states of an ε-NFA ε-a-reachable if they are connected by a list that begins with finite εs and ends with character a. We use *EpsCharReach* to calculate all the ε-a-reachable states of an ε-NFA from state q, i.e.

$$EpsCharReach\ nfae\ q\ a\ =\ maps\ (CharReach_F\ \delta\ a)\ (FiniteEpsReach\ nfae\ q).$$

In this specification, $CharReach_F$ is a function similar to $CharReach_D$ in $N2D$ algorithm. For state $q \in Q$ and character $a \in \Sigma$, $CharReach_N\ \delta\ a\ q$ is a list of all the direct a-reachable states from q. And we use *maps* to apply $(CharReach_N\ \delta)$ to the list of all the ε-reachable states from q and obtain a list of all the ε-a-reachable states from q.

Actually, for state q and character a, $EpsCharReach\ nfae\ q\ a$ provides all the possible states that can be transited from q with a. Together with *FiniteEpsReach*, we can construct the conversion algorithm $Ne2N$ that converts an ε-NFA into an NFA as follows:

Algorithm B3 $Ne2N$

```
Require: ε-NFA $nfae = (Q, \Sigma_\varepsilon, \delta, q_0, F)$;
Ensure: NFA $nfa = (Q', \Sigma, \delta', q_0, F')$;
 1: $\delta', F' \Leftarrow []$;
 2: $Q', W \leftarrow [q_0]$;
 3: while $W \neq []$ do
 4:     pick $q$ from $W$;
 5:     if $set\ (FiniteEpsReach\ \delta\ q) \cap set\ F \neq \emptyset$ then
 6:         add $q$ to $F'$;
 7:     end if
 8:     for $a \in set\ \Sigma$ do
 9:         for $q' \in set(EpsCharReach\ nfae\ q\ a)$ do
10:             if $q' \notin set\ Q'$ then
11:                 add $q'$ to $W$;
12:                 add $q'$ to $Q'$;
13:             end if
14:             add $(q, a, q')$ to $\delta'$;
15:         end for
16:     end for
17: end while
```

In the algorithm $Ne2N$, to collect all the reachable states of the output NFA also need to use a worklist-based method. But different from $N2D$, it does not collect the normal one-step reachable states but all the ε-a-reachable states from q.

To simplify the implementation, we *map* function $(EpsCharReach\ nfae\ q)$ on the alphabet Σ, remove redundant elements, and obtain all the ε-α-reachable states from q with all possible characters by $SigmaReach_N$:

$$SigmaReach_N\ nfae\ q\ =\ remdups\ (map\ (EpsCharReach\ nfae\ q)\ \Sigma).$$

Taking $(SimgaReach_N\ nfae)$ as the specific expansion function, the function that calculates all the reachable states of the output NFA can be formalized by *collect_option* as follows:

$$Ne2Ns\_option\ nfae = collect\_option\ (SigmaReach_N\ nfae)\ [q_0].$$

It can be proved that if $nfae$ is an ε-NFA, the results of *FiniteEpsReach*, $CharReach_N$, *EpsCharReach* and $SigmaReach_N$ are all distinct, and all kinds of reachable states are still states of Q, i.e. *set* Q is closed under all these state collecting functions. According to Theorem 2, *Ne2Ns_option* also terminates. Thus Q' can be acquired by removing the type constructor *Some* from the result of *Ne2Ns_option*, i.e.

$$Q' = \text{case } Ne2Ns\_option\ nfae \text{ of } None \Rightarrow []\ |\ Some\ wr \Rightarrow snd\ wr.$$

As mentioned above, for any $q \in set\ Q'$ and $a \in set\ \Sigma$, $EpsCharReach\ nfae\ q\ a$ provides all the possible states of the output that can be transited from q with a. Therefore, we can use the following function to construct a list of all the transition triples that have a form of $(q, a, *)$:

$$delta\_triples\ nfae\ (q, a)\ =\ map\ (\lambda s.\ (q, a, s))\ (EpsCharReach\ nfae\ q\ a).$$

If we *maps* this function on the pair list $Q' \times \Sigma$, the list δ' of all transition triples can be obtained by following equation:

$$\delta' = remdups\ (maps\ (delta\_triples\ nfae)\ (product\ Q'\ \Sigma)).$$

According to algorithm $Ne2N$, a state is in $set\ F'$ if and only if one of its ε-reachable states belongs to F. Then we can construct the final state list as follows:

$$F' = [x \leftarrow Q'.\ set\ (FiniteEpsReach\ nfae\ x) \cap set\ F \neq \emptyset].$$

Appendix B.2.2 *Correctness Proofs*

To prove $Ne2N$ is a correct conversion algorithm from ε-NFA to NFA, we first need to prove Lemma 3.

Lemma 3. $NFA\ nfae \Longrightarrow NFA\ (Ne2N\ nfae)$.

We assume that $nfae$ is an ε-NFA and let $Ne2N\ nfae = (Q', \Sigma, \delta', q_0, F')$. Most properties of *Finite_Automata*, such as *distinct* Q', *distinct* Σ, *distinct* F', *distinct* δ', $q_0 \in set\ Q'$ and $set\ F' \subseteq set\ Q'$, can be proved with ease. Then we need to prove the properties related with δ, i.e.

$$\forall (q, a, q') \in set\ \delta'.\ q \in set\ Q' \wedge a \in set\ \Sigma \wedge q' \in set\ Q'.$$

According to the specification of $Ne2N$, every element of δ' can be constructed by *EpsCharReach*, i.e. for each $(q, a, q') \in set\ \delta'$, $q \in Q'$, $a \in \Sigma$ and $q' \in set\ (EpsCharReach\ nfae\ s\ a)$. As $set\ Q$ is closed under $(EpsCharReach\ nfae)$, i.e. $set\ (EpsCharReach\ nfae\ s\ a) \subseteq set\ Q'$, the above δ'-related property holds, and $Ne2N\ nfae$ is an NFA.

Next, we are to show that the input and output automata recognize the same language.

Although the definition of *accept* has been provided in Section **??**, it is improper to formalize the word acceptance of an ε-NFA, because the actual alphabet of ε-NFA has a datatype of α *eps list* which is different from that a normal NFA. If we use *accept* to formalize both acceptance of input and output automata, the words would have different datatypes. Thus, to uniform the datatype of the input word, it is necessary to redefine the acceptance of a word for an ε-NFA.

We first propose the definition of *eps_word*, which is a special word with a type of α *eps list*:

$$eps\_word\ w = (\forall i < size\ w - 1.\ w!i = eps) \wedge w!(size\ w - 1) \neq \varepsilon \wedge w \neq [])$$

An *eps_word* is a special α *eps* word which is used to describe the relationship between a α *eps* word and its corresponding characters: an *eps_word* corresponds to the character of its last element. Then we can pay more attention to the character instead of the finite εs before it. Based on the concept of *eps_word*, the word acceptance of an ε-NFA can be redefined as follows:

$$\begin{aligned}accept_{\varepsilon}\ nfae\ w = \ & (\exists sl\ wl.\ size\ w = size\ wl \\ & \wedge size\ sl = size\ wl + 1 \\ & \wedge(\forall i < size\ wl.\ reachable\ nfae\ (sl!i)\ (sl!Suc\ i)\ (wl!i)) \\ & \wedge(\forall i < size\ wl.\ eps\_word\ (wl!i) \wedge last\ (wl!i) = Char\ (w!i)) \\ & \wedge sl!0 = q_0 \\ & \wedge set(FiniteEpsReach\ nfae\ (last\ sl)) \cap F \neq \emptyset).\end{aligned}$$

Besides the construction of a state list, it is still necessary to find a list of *eps_words* to show that an ε-NFA accepts a word. The last elements of all these *eps_words* constitute the accepted word with the given order. As we do not have to care which states has been transited with ε, what is significant is that every two adjacent states of the state list are reachable with their corresponding *eps_word* and the last state should be transited to some final state with finite ε-transitions.

With the definition of $accept_{\varepsilon}$, Lemma 4 is provided to show the input and output automata recognize the same language.

Lemma 4. $NFA\ nfae \Longrightarrow accept_{\varepsilon}\ nfae\ w \longleftrightarrow accept\ (Ne2N\ nfae)\ w$.

Let $nfae$ be an ε-NFA. We first assume $accept_{\varepsilon}\ nfae\ w$, and then a state list sl and a list wl of *eps_words* can be obtained. The state list sl is just the state list to prove $accept\ (Ne2N\ nfae)\ w$. It can be proved that if two states are reachable with an *esp_word* in $nfae$, then there is a transition triple of $Ne2N\ nfae$ that connects the two states and is labeled by the last character of the *esp_word*. Therefore, $accept\ (Ne2N\ nfae)\ w$ holds.

Next, we assume $accept\ (Ne2N\ nfae)\ w$ first and get the state list and the transition triple for each character of w. By applying induction on the word w (a list), it can be proved that there is a list of *eps_words* such that the last element of each *eps_word* corresponds to the relevant character of w and any two adjacent states are reachable with the *eps_word*. Thus, $accept_{\varepsilon}\ nfae\ w$ also holds.

Therefore, $nfae$ and $(Ne2N\ nfa)$ share the same state list and recognize the same language.

Appendix B.3 DFA minimization

For each DFA, there exists a unique minimal automaton (a DFA with a minimum number of states) that recognizes the same language. The minimized DFA ensures minimal computational cost for language recognition. In this paper, we adopt the Hopcroft's Algorithm for DFA minimization. The algorithm first partitions the states of the input DFA into blocks, where the states in one block have the same transition behaviors Then the algorithm takes each block as a state of the minimized DFA and constructs other components of the minimized DFA accordingly.

Appendix B.3.1 *Algorithm Formalization*

Normally, a partition of a set S is a list of nonempty subsets of S which are called blocks. The union of these blocks is S and any two different blocks do not intersect. Thus, the partition of a state list can be formalized as follows:

$$\begin{aligned}\mathit{partition}\ \mathit{slist}\ P =\ &(\mathit{Sup}(\mathit{set}\ P) = \mathit{set}\ \mathit{slist}\\ &\wedge (\forall i < \mathit{size}\ P.\ P!i \neq \emptyset)\\ &\wedge (\forall i < \mathit{size}\ P.\ \forall j < \mathit{size}\ P.\ i \neq j \rightarrow (P!i) \cap (P!j) = \emptyset).\end{aligned}$$

In this specification, each block carries a datatype of $\alpha\ \mathit{set}$, and we use list P of blocks to express the partition of a state list.

For a DFA $(Q, \Sigma, \delta, q_0, F)$, its minimized DFA $(P, \Sigma, \delta_m, q_{m0}, F_m)$ is constructed by the conversion algortihm *Mini*. And the algorithm *Mini_State* is used to calculate all the states of the minimized DFA:

Algorithm B4 *Mini_State*

```
Require: DFA $dfa = (Q, \Sigma, \delta, q_0, F)$;
Ensure: A partition $P$ of $Q$;
1: if $set\ F = \emptyset$ or $set\ Q - set\ F = \emptyset$ then
2:     $P \leftarrow [set\ Q]$;
3: else
4:     $P \leftarrow [set\ F, set\ Q - set\ F]$;
5:     while $P$ is unstable do
6:         find $B, B' \in set\ P$, $a \in set\ \Sigma$ s.t. $(a, B')$ splits $B$;
7:         replace $B$ with $\{q \in B \mid (delta\_fun\ \delta)\ q\ a \in B'\}$ and $\{q \in B \mid (delta\_fun\ \delta)\ q\ a \notin B'\}$;
8:     end while
9: end if
```

Unlike the algorithms $D2N$ and $Ne2N$ which use *collect_option* for states collection, *Mini_State* calculates the state list by partitioning Q into different blocks: if $\mathit{set}\ Q = \emptyset$ or $\mathit{set}\ Q - \mathit{set}\ F = \emptyset$, the partition P only contains $\mathit{set}\ Q$; otherwise, the algorithm initials P with $[\mathit{set}\ F, \mathit{set}\ Q - \mathit{set}\ F]$ and does partitions repeatedly according to the stability of P.

A partition P is unstable if there are blocks B, B' in P, a character a in Σ and two states q_1, q_2 in B such that q_1 is transited to a state in B' with a while q_2 is transited to a state out of B' with the same a. If P is unstable, the states of B will show different transition behaviors with a, and we call (B, a) an unstable pair of P and call (B', a) splits B. Then, the specification of *unstable* is defined as follows:

$$\begin{aligned}\mathit{unstable}\ \mathit{dfa}\ P =\ &(\exists a \in \mathit{set}\ \Sigma.\ \exists i < \mathit{size}\ P.\ \exists j < \mathit{size}\ P.\ \exists s_1 \in \mathit{set}\ Q.\ \exists s_2 \in \mathit{set}\ Q.\\ &s_1 \in (P!i) \wedge s_2 \in (P!i) \wedge\\ &s_1 \neq s_2 \wedge (\mathit{delta\_fun}\ \delta\ s_1\ a) \in (P!j) \wedge (\mathit{delta\_fun}\ \delta\ s_2\ a) \notin (P!j)).\end{aligned}$$

Here, *unstable* is a DFA-related concept, and *delta_fun* is a function that converts the list δ of transition triples of dfa into the corresponding transition function. If a partition P of Q is *unstable*, an unstable pair can be found by using function *findUnstablePair*, which provides the first block and character that cause the unstability:

$$\mathit{findUnstablePair}\ \mathit{dfa}\ P\ =\ \mathit{hd}[x \leftarrow (\mathit{product}\ P\ \Sigma).\ \mathit{unstable\_element}\ \mathit{dfa}\ P\ x].$$

Then, the algorithm can *update* the partition P by *replace*ing the block B with the result of *split*ing, i.e.

$$\mathit{update}\ \mathit{dfa}\ P = \mathit{replace}\ P\ B\ (\mathit{split}\ (B, a)\ \delta\ P),$$

where $\mathit{split}\ (B, a)\ \delta\ P = [\{s | s \in B \wedge \mathit{delta\_fun}\ \delta\ s\ a \in B'\}, \{s | s \in B \wedge \mathit{delta\_fun}\ \delta\ s\ a \notin B'\}]$.

With all the above preparation, the non-deterministic loop in Algorithm *Mini_State* can be implemented by *while_option* directly:

$$\mathit{StatePartition}\ \mathit{dfa}\ P = \mathit{while\_option}\ (\mathit{unstable}\ \mathit{dfa})\ (\mathit{update}\ \mathit{dfa})\ P.$$

If P is *unstable*, *StatePartition* will find the unstable pair (B, a) and update the partition P by replacing B with $\mathit{split}\ (B, a)\ \delta\ P$. Then we define the measure function $f = \lambda i.\ (\mathit{size}\ Q) - (\mathit{size}\ P^{(i)})$ where i is the number of iterations. As $\mathit{size}\ (\mathit{split}\ (B, a)\ \delta\ P) = 2$ and $\mathit{size}\ P \leqslant \mathit{size}\ Q$, f strictly decreases when i increases. According to the termination theory of *while_option*, the specification *StartPartition* terminates and the state list of the minimized DFA can be obtained.

Appendix B.3.2 *Correctness Proofs*

In algorithm *Mini_State*, the transition triple list is not calculated. According to the definition of *unstable*, when *Mini_State* terminates, the states in the same block of P will have the same transition behaviors, i.e. for any $a \in \Sigma$ and $B \in \mathit{set}\ P$, the states of B must be transited to the states of some block of P with a-transitions. Thus, the transition function *mini_delta* of the minimized DFA can be constructed as follows:

$$\mathit{mini\_delta}\ \mathit{dfa} = \lambda\ S\ a.\ (\text{if } S \in \mathit{set}\ P \wedge a \in \mathit{set}\ \Sigma \text{ then } \mathit{hd}[B \leftarrow P.\ (\mathit{End}(\mathit{findTriple}\ \mathit{dfa}\ S\ a)) \in B] \text{ else } \emptyset).$$

In the specification of *mini_delta*, *findTriple* is used to find the transition triple (s, a, s') in δ such that $s \in S$, and *End* is used to get s' out of (s, a, s'). As dfa is a DFA, then s' is unique and there exists a unique block that contains s'.

Actually, $hd[B \leftarrow P.\ (End(findTriple\ dfa\ S\ a)) \in B]$ is the unique block of P that contains s'. Thus, $mini\_delta\ dfa$ is the transition function of the minimized DFA. Then we can use *delta_triple* to convert the transition function *mini_delta* into the transition triple list δ_m which satisfies all the relevant transition properties of a DFA.

If we construct $F' = [x \leftarrow P.\ x \cap set\ F \neq \emptyset]$, Lemma 5 can be proved.

Lemma 5. $DFA\ dfa \Longrightarrow DFA\ (Mini\ dfa)$.

Next, we show the input and output DFA accept the same words, i.e.

Lemma 6. $DFA\ dfa \Longrightarrow accept\ (Mini\ dfa)\ w \longleftrightarrow accept\ dfa\ w$.

As mentioned in the above, to judge whether a DFA accepts some word, the most direct way is to *run* the DFA on the given word and collect all the passed states. The list obtained by function *run* is the required list to prove the properties of *accept*. And showing a DFA *accepts* a word is equivalent to showing the last state of the list obtained by *run* is a final state of the DFA. Using this method, we can prove that dfa and $Mini\ dfa$ accept the same words, i.e. Lemma 6 holds.

Appendix B.4 Related Works and Comparison

Most studies on the formalization of conversion algorithms between finite automata are related with the recognition of regular languages. In 1997, Filliâtre first formalized finite automata in Coq [6]. A conversion algorithm from a RE to an ε-NFA was verified and all sets were formalized with lists for program extraction. Based on Filliâtre's work, a constructive proof for a partial derivative automata construction was proposed by Almeida, Moreira, Pereira and Sousa [1].

To implement finite automata algorithms efficiently, Braibant and Pous constructed an equivalence decision procedure for RE with matrix, which involves a complete verification of converting a RE into a DFA [2]. Thought the matrix-based implementation is efficient, the equivalent transformations between automata and matrix do not come for free, and the whole work contains about 19000 lines of development. To simplify the development, Alexander and Tobias proposed an equivalence checker without the automata conversion by algebraic derivatives [12]. In 2012, Lammich and Tuerk introduced a stepwise refinement framework to achieve a clean separation between the abstract verification and the concrete implementation. They applied this framework to the Hopcroft's algorithm and achieved efficient implementation [11].

In addition to the developments that focus on efficiency and executability of algorithm implementation, some researchers are also interested in the simplification of the proofs. Using type theory and Ssreflect library in Coq, Doczkal et al presented a formal constructive automata theory with the datatype of finite set [4]. Lawrence provided similar formalization of the finite automata theory and the relevant conversion algorithms with hereditarily finite sets [14]. The use of the hereditarily finite sets makes the formalization and the proofs straightforward and easy, and reduces the whole development to around 9000 lines.

In fact, it does not matter which datatype is used for formalization, it always needs to convert the selected datatype into list to obtain executable codes from the formal specifications. Therefore, following Filliâtre's approach, we formalize three conversion algorithms with lists and verify their functional correctness by constructive proofs. The list-based specifications we formalize are more alike to those we implement in programs and can be used for code generation. Comparatively, the work covers 226 lines of specification formalization and around 2250 lines of constructive proofs. The development is shorter mainly due to our full use of the list-based properties for verification and no need to conduct conversion from other datatypes to list.

This work is part of the Computer Aided Verification of Automata project to formalize and verify important algorithmic parts of automata theory and model checking using the proof assistant Isabelle/HOL. All the three conversion algorithms come from the text book *Automata Theory: An Algorithmic Approach* [9]. We select these algorithms because they are important in practice, uncomplicated to construct, and easy to implement. The specifications in this paper come from the direct formalization of these algorithms. Now, all the specifications and proofs can be downloaded at http://pan.baidu.com/s/1nu8FGbn.

References

1 Almeida J B, Moreira N, Pereira D, et al. Partial derivative automata formalized in Coq. International Conference on Implementation and Application of Automata, Winnipeg, Canada, 2010, 59-68

2 Braibant T, Pous D. An Efficient Coq Tactic for Deciding Kleene Algebras. International Conference on Interactive Theorem Proving, Edinburgh, Scotland, 2010, 163-178

3 Brzozowski J. Derivatives of Regular Expressions. Journal of the Acm, 1964, 11(4): 481-494

4 Doczkal C, Kaiser J O, Smolka G. A Constructive Theory of Regular Languages in Coq. International Conference on Certified Programs and Proofs, Springer-Verlag, New York, USA, 2013, 82-97

5 Doczkal C, Smolka G. Two-Way Automtata in Coq. International Conference on Interactive Theorem Proving, Nancy, France, 2016, 151-166

6 Filliâtre J C. Finite auotmata theory in Coq - a constructive proof of Kleene's Theorem. 1997

7 Haftmann F, Nipkow T. Code Generation via Higher-Order Rewrite Systems. Functional and Logic Programming, International Symposium, Flops 2010, Sendai, Japan, 2010, 103-117

8 Hopcroft J E, Motwani R, Ullman J D. Automata theory, languages, and computation. Pearson Education, 2007

9 Javier Esparza. Automata Theory: An Algorithmic Approach. https://www7.in.tum.de/~esparza/autoskript.pdf. 2016

10 Lammich P, Lochbihler A. The Isabelle Collections Framework. International Conference on Interactive Theorem Proving, New Jersey, USA, 2010, 339-354

11 Lammich P, Tuerk T. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. International Conference on Interactive Theorem Proving, New Jersey, USA, 2012, 166-182

12 Krauss A, Nipkow T. Proof Pearl: Regular Expression Equivalence and Relation Algebra. Journal of Automated Reasoning, 2012, 49(1):1-12

13 Nipkow T, Paulson L C, Wenzel M. Isabelle/HOL – A proof assistant for higher-order logic. Lecture Notes in Computer Science, 2002, 2283

14 Paulson L C. A Formalisation of Finite Automata using Hereditarily Finite Sets. CADE-25 - International Conference on Automated Deduction, Berlin, Germany, 2015, 9195: 231-245

15 Rabin M O, Scott D. Finite automata and their decision problems. IBM Journal of Research and Development, 1959, 3(2): 114-125