

The Verification of Conversion Algorithms between Finite Automata (An Extended Abstract)

Dongchen Jiang

State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences,
`jiangdongchen@ict.ac.cn`

1 Introduction

In this paper, the specifications of three conversion algorithm between finite automata have been verified in Isabelle/HOL [5]. The main purpose of this work is to propose the verified specifications of these algorithms for code generation, so that the functional correctness of the codes is guaranteed by verification. Thanks to the Isabelle/HOL Collection Framework (ICF), which can automatically produce executable codes from certain specifications [3], I only need to focus on the data structure selection for formalization and the functional correctness proof for specifications.

Three conversion algorithms in [2] are selected for verification: the algorithm *N2D* that converts a normal NFA into a normal DFA, the algorithm *Ne2N* that converts an NFA with ε into a normal NFA, and the Hopcroft's algorithm *Mini* that converts a normal DFA to its minimisation. All algorithms are formalized in Isabelle/HOL, and their functional correctness is proved interactively.

Lammich and Tuerk propose a refinement framework which replaces operations on abstract datatypes by corresponding operations on concrete datatypes in Isabelle/HOL [4]. And they verify an implementation of Hopcroft's algorithm by using this framework. Being different from their work, this paper uses a more direct way to verify these specifications. All finite sets are formalized by distinct lists, which is closer to daily programming. Compared with Lammich and Tuerk's work, these specifications look complex, but they avoid the proofs about the refinement framework.

2 Preliminary

Normally, a finite automata is defined as a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of a finite set Q of states, an alphabet Σ , a transition function $\delta : Q \times \Sigma \rightarrow Q$, an initial state q_0 and a set F of accepting states.

In this paper, a finite automata is also formalized by the 5-tuple. But being different from the normal definition, all finite sets are formalized by distinct lists.

This is because lists are familiar to programmers and they can be transformed into arrays directly.

$$\begin{aligned}
& \text{definition } FA :: "(a, s)fa \Rightarrow bool" \\
& \text{where } "FA(Q, \Sigma, \delta, q_0, F) = (distinct\ Q \wedge distinct\ \Sigma \wedge distinct\ \delta \\
& \quad \wedge\ q_0 \in set\ Q \wedge set\ F \subseteq set\ Q \\
& \quad \wedge (\forall d \in set\ \delta. Fst\ d \in set\ Q \wedge Snd\ d \in set\ \Sigma \wedge Trd\ d \in set\ Q))"
\end{aligned} \tag{1}$$

In above formalization, the transition function δ is also formalized by a list of triples. If $(s_1, a, s_2) \in set\ \delta$, s_1 and s_2 are two states in Q and a is a character of Σ . In this paper, we also call (s_1, a, s_2) a transition triple, and δ is the list that contains all transition triples of an finite automata.

For a nondeterministic finite automaton (NFA), its definition is the same as the definition of FA . But for a deterministic finite automaton (DFA), the following property about the transition triple list should be also satisfied:

$$\forall s \in set\ Q. \forall a \in set\ \Sigma. \exists! s' \in set\ Q. (s, a, s') \in set\ \delta. \tag{2}$$

Here, $\exists! s' \in set\ Q$ means that there exists a unique s' in $set\ Q$ satisfies the followed property.

3 Functional Correctness

There are three main theorems for the functional correctness of each conversion algorithm: firstly, it should be proved that all iterations in an algorithm terminate; secondly, the output of an conversion algorithm should be an automata with the expected type; thirdly, the input and the output of an algorithm should recognize the same language.

3.1 Termination

Termination is an important property of a correct algorithm. The termination proof of an algorithm guarantees that all loops in such algorithm terminate. In our formalization, loops are implemented by recursive functions or *while_option*. Isabelle/HOL can check the termination of most recursive functions automatically [1], which saves a lot of work. But for nondeterministic iterations by *while_option*, it is necessary to prove the iterations terminate interactively.

In algorithms *N2D* and *Ne2N*, *while_option* is only used for calculating reachable states in different implementations. After analyzing the process of states collection, a worklist based framework *Extend_option* is proposed to uniform this formalizations. A specific measure function is constructed accordingly to show that the general termination condition for *Extend_option*, i.e. if the set of all reachable states is finite, then *Extend_option* terminates. As this paper only focuses on finite automata, the termination theorems of *N2D* and *Ne2N* are proved with ease by this framework.

In algorithm *Mini*, the only nondeterministic loop is used for partitioning the state set, which is implemented by *while_option* directly. As the state set itself is finite, the times of partitioning should be no more than the cardinality of the state set. Therefore, the algorithm for partitioning must terminate.

3.2 Being an Automata with the Expected Type

To prove the output of a conversion algorithm is an expected automata, every property in (1) needs to be proved. Properties, such as *distinct Q*, *distinct Σ*, *distinct F*, $q_0 \in \text{set } Q$ and $\text{set } F \subseteq \text{set } Q$, can be proved with ease according to the specification. The main effort is made to prove the properties about δ , i.e. the last property in (1) and the property in (2).

Given a character a and a state set S of an NFA nfa , reach_a is the function that calculates the set of one-step reachable states from S with a . If S is one state of $N2D\ nfa$, then $\text{reach}_a\ S\ a$ is also a state of $N2D\ nfa$ and $(S, a, \text{reach}_a\ S\ a)$ is a transition triple of $N2D\ nfa$. As reach_a is a function, $\text{reach}_a\ S\ a$ is unique. Therefore, the list of transition triples, which is constructed by reach_a , satisfies the related properties in (1) and (2), i.e. $N2D\ nfa$ is a DFA.

Similar to reach_a in $N2D$, the function Reach is proposed to calculate the third element of a transition triple in $Ne2N$. Given a state s and a character a of an NFA with ε , $\text{Reach}\ a\ s$ returns a list of all reachable states from s with an $\varepsilon - a$ list, where an $\varepsilon - a$ list is a list that begins with finitely many ε s and ends with the character a . By using the function map , the transition triple list can be constructed accordingly, and it satisfies the last property in (1).

In algorithm *Mini*, given a character a and one part S of the state set partition of a DFA dfa , the algorithm picks out one state $s \in S$ and finds out the transition triple (s, a, s') of dfa , where s' is unique for (s, a, s') . Then, the part S' that contains s' is used for constructing the transition triple (S, a, S') for *Mini* dfa . As S' is a part of the state set partition and s' is a state, S' is also unique. Therefore, the transition triple list of *Mini* dfa satisfies (2).

3.3 Accepting the Same Language

The purpose of these conversion algorithms is to simplify automata without changing the accepted languages. In this paper, a finite automata fa accepting a word w is defined as follows:

$$\begin{aligned}
 &\text{definition } \text{accept} :: "(a, 's)fa \Rightarrow (a)word \Rightarrow bool" \\
 &\text{where } "accept\ fa\ w = ((\exists slist. \text{size } slist = \text{size } w + 1 \\
 &\quad \wedge slist!0 = iFA\ fa \\
 &\quad \wedge (\forall i < \text{size } w. (slist!i, w!i, slist!Suc\ i) \in \text{set}(dFA\ fa)) \\
 &\quad \wedge slist!(\text{size } w) \in \text{set}(fFA\ fa))" \tag{3}
 \end{aligned}$$

To prove “ $\text{accept}\ fa\ w$ ”, it is necessary to show the existence of a state list which starts with the initial state and ends with an accepting state. Each pair of the adjacent states and the corresponding character of w should constitute a transition triple of fa .

To judge whether a DFA accepts w in $N2D$ and *Mini*, a function runD is proposed to simulate the state transitions while processing w . The simulation starts at the initial state, and does state transitions according to the transition triples and w . By collecting the states of transition, we obtain a state list which satisfies all properties in (3). By using this method, “ $\text{accept}\ nfa\ w \Rightarrow \text{accept}\ (N2D\ nfa)\ w$ ” and “ $\text{accept}\ (mini\ dfa)\ w \iff \text{accept}\ dfa\ w$ ” are proven.

To prove “ $\text{accept } (N2D \text{ nfa}) w \implies \text{accept nfa } w$ ”, it is necessary to obtain the transition state list of $N2D \text{ nfa}$ first. As a state of $N2D \text{ nfa}$ is a nonempty state set of nfa , we can pick one state from every element of the list and combine them as a state list with the same order. It can be proved that this list is the state list of nfa for proving “ $\text{accept nfa } w$ ”.

When processing an NFA with ε (nfae) accepts a word w in $Ne2N$, the definition of acceptance is a little different, but it is still equivalent to *accept*. Besides the existence of the transition state list, lists of ε s are added before each character of w . With this operation, w can be divided into several $\varepsilon - a$ lists. Every two adjacent transition states, which are *reachable* with the corresponding $\varepsilon - a$ list in nfae , are one-step reachable with a in $Ne2N \text{ nfae}$. Thus, nfae and $Ne2N \text{ nfa}$ share the same state list for accepting w .

4 Verification Effort

It takes about one year to formalize these specifications and prove the relevant theorems, which also includes all attempts to select proper data structures. The whole work includes all specifications of the three algorithms (239 lines), the lemmas, theorems and the relevant proofs about correctness (3035 lines), and also some assistant functions and definitions for proving (35 lines, e.g. the definitions of *accept* and *reachable* etc.). All specifications can be used for code generation by ICF. But it is unnecessary to generate executable codes from the assistant functions and definitions, and the only purpose for them is to make the correctness proofs easier.

Currently, an reorganizing work of the specifications and proofs is ongoing, and all specifications and theorems can be downloaded at <https://www.dropbox.com/sh/jqsjsx456tdkx1gl/6nWbEnV2Qx>.

Acknowledgements I would like to thank Tobias Nipkow for introducing me to the problem of verifying finite automata conversion algorithms, and I am indebted to him for the inspired discussion on this topic. I am also indebted to Jasmin Blanchette, Florian Haftmann, Alexander Krauss, Sascha Böhme and Lucas Bulwahn for the technique support and daily discussion in Munich.

References

1. Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding lexicographic orders for termination proofs in *isabelle/hol*. In *Theorem Proving in Higher Order Logics*, pages 38–53. Springer, 2007.
2. Javier Esparza. *Automata Theory: An Algorithmic Approach*. Unpublished, 2012. <https://www7.in.tum.de/~esparza/autoskript.pdf>.
3. Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming*, pages 103–117. Springer, 2010.
4. Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft’s algorithm. In *Interactive Theorem Proving*, pages 166–182. Springer Berlin Heidelberg, 2012.
5. Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.