Semester Project Report

Rodrigo Bernardo
rodrigo.moreirabernardo@epfl.ch

January 2018

1 Introduction

The problem solved in this semester project was the development of a provably correct nondeterministic finite automaton (NFA) to deterministic finite automaton (DFA) conversion.

The motivation for the DFA to NFA transformation came as part of a project called *lexi*. *lexi* is a simple implementation of a lexer generator, written in Scala. It provides a small DSL that allows you to define tokens using regular expressions and compiles them to a DFA. The DFA outputs a sequence of tokens when given an input string.

The DFA is obtained in two steps: we start by compiling the regular expression to an NFA and then transform it to a DFA. In this project, we focus on the latter step.

Much of the difficulty came from the fact that the implementation for sets in *Stainless* is limited. For example, while it is possible to check element membership, there is no way to actually traverse the elements of the underlying structure. Also, it is not possible to check the cardinality of the set, as that functionality is broken (*Stainless* does not compile any program with calls to the *size* method).

With this it became obvious that a solution for this problem would need a new set implementation. However, with that, properties about sets that we usually take for granted, such as, e.g., reflexivity and transitivity of equality, or properties about the relative size of a set and one of its subsets, all need to be proven. Properties that otherwise *Stainless* would be able to easily verify, had we used the set implementation in the *Stainless* library.

In the end, the proof is complete apart from two quirks that we believe are shortcomings of *Stainless*.

2 Background Theory

We start by stating our definitions of DFA and NFA, as well as the conversion algorithm.

Definition 1.
A deterministic finite automaton (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite set (the states);*
- Σ *is a finite set (the alphabet);*
- δ *is a function defined over* $Qx\Sigma$ *with values in* Q *(the transition function);*
- q_0 *is an element of* Q *(the initial state);*
- *and* F *is a subset of* Q *(the final states).*

Definition 2.
We say that the DFA $A = (Q, \Sigma, \delta, q_0, F)$ *accepts the word* $w = w_1 \dots w_n$, $w_i \in \Sigma$, $1 \leq i \leq n$, *if there is a sequence of states* $r_0, \dots, r_n$, $r_i \in Q$, $0 \leq i \leq n$, *such that:*

- $r_0 = q_0$*;*
- $r_{i+1} = \delta(r_i, w_{i+1})$, $0 \leq i \leq n - 1$*;*
- *and* $r_n \in F$.

Definition 3.
The language recognized by the automaton $A = (Q, \Sigma, \delta, q_0, F)$ *is the set* $L(A) = \{w \in \Sigma^* : Aacceptsw\}$.

Definition 4.
We say that two automatons are equivalent if they recognize the same language.

Definition 5.
A nondeterministic finite automaton (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite set (the states);*
- Σ *is a finite set (the alphabet);*

- *δ is a function defined over $Qx(\Sigma \cup \{\epsilon\})$ with values in 2^Q (the transition function);*
- *q_0 is an element of Q (the initial state);*
- *and F is a subset of Q (the final states).*

Definition 6.
We say that the NFA $A = (Q, \Sigma, \delta, q_0, F)$ accepts the word $w = w_1 \dots w_n$, $w_i \in (\Sigma \cup \{\epsilon\})$, $1 \leq i \leq n$, if there is a sequence of states $r_0, \dots, r_n$, $r_i \in Q$, $0 \leq i \leq n$, such that:

- *$r_0 = q_0$;*
- *$r_{i+1} \in \delta(r_i, w_{i+1})$, $0 \leq i \leq n - 1$;*
- *and $r_n \in F$.*

It is straightforward to see that every NFA has an equivalent DFA. If we view nondeterminism through the perspective of parallel programming, we can imagine a nondeterministic computation to be a computation where at each step the machine creates threads corresponding to each of the possible next states. If we were to simulate a NFA, we would just to keep track of the current states of active threads, update them when a new symbol is read, and accept one of the threads if any of the threads is in a final state when we reach the end of the word. The idea is to create a DFA that runs this simulation algorithm.

3 Solution

3.1 Representation

3.1.1 Automata

The automata representation, as well as the conversion and the corresponding proof of correctness can be found in the file Automata.scala.

For NFAs, we do not explicitly represent the alphabet, as it is not relevant for the proof. In pratice, it is implicitly defined by the domain of the transition function. By the same reason, we do not restrict the domain of the transition function, but only its range. ϵ-transitions are represented by explicitly passing the value None() to the transition function. The set of final states is substituted by a predicate that decides final states.

ϵ-closure was the trickiest part of the project, and what ended up motivating the need to implement our own representation of sets. Without the need to prove termination, the proof would go just as well if we used lists instead.

It is also regarding ϵ-closure that one of our two assumptions is about. We claim (and is easibly verifiable by a human by inspection) that ϵ-closure is idempotent. However, *Stainless* can not verify this.

The DFA is much simpler than the NFA. We avoid representing the set of valid states and the alphabet of work explicitly, as they are not relevant for the proof. In theory, this structure would allow to represent non-finite deterministic state automata. The transition function is the same as with the NFA, except that we do not allow for ϵ-transitions.

3.1.2 Sets

The current implementation for sets allows for traversing and stores each element only once, by guaranteeing that the predicate *contains* is mantained as an invariant.

Other implementations were tried first. One was a wrapper around a *Stainless* list that mantained the invariant that a given list l was always equal $l.unique$, essentialy guaranteeing that each element is only stored once. This approach, while maybe obvious, lead to proof obligations that were difficult to prove. Another was using the *Stainless* implementation for maps, but maps in *Stainless* suffer from the same shortcoming as sets, where one is not able to traverse its keys.

The implementation is divided in two files: USet.scala and Set.scala. The former one is where the actual implementation and proofs are done. The latter is just a convenience wrapper that allows us to use sets without the need to explicitly require the set invariants as preconditions everywhere we use them.

Yet another approach was to eliminate sets altogether and try to work everything out with lists, with subsequences taking the place of subsets. However, due to problems regarding *Stainless* not supporting

3.1.3 Memoization

In the end, the DFA created effectively simulates the NFA and is not practical because at every transition we need to compute the ϵ-closure of some state in the original NFA. Because of this, we created a wrapper over the DFA representation that memoizes the transition function at every call. The new DFA also comes with a proof of equivalence to the original DFA.

The wrapper can be found in the file AutomatonCache.scala.

3.2 Proof

To prove equivalence we need to prove that the NFA and the generated DFA recognize the same words. We proved a stronger version of this. We proved that the state of the DFA after reading some string starting from some ϵ-closed state A is exactly the set of states the NFA could be in after reading the same string starting from some state in A.

3.3 Results and Discussion

In the end, the proof is complete apart from two quirks. First, *Stainless* cannot prove that ϵ-closure is idempotent, which is a necessary fact for the proof. Secondly, *Stainless* argues that the DFA adt invariant may be violated in the NFA to DFA conversion. However, the DFA class does not have any explicitly written preconditions, so we do not have any explanations for this.

Both the conversion algorithm and the statement of idempotence of ϵ-closure were treated with "@library" annotations, so that *Stainless* does not consider them in the verification summary. They are also obviously always terminating.