A MECHANIZED PROOF OF KLEENE'S THEOREM IN WHY3

André Duarte Teixeira Trindade

Advisers:
António Maria Lobo Alarcão Ravara
Mário José Parreira Pereira

FEBRUARY 2021

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

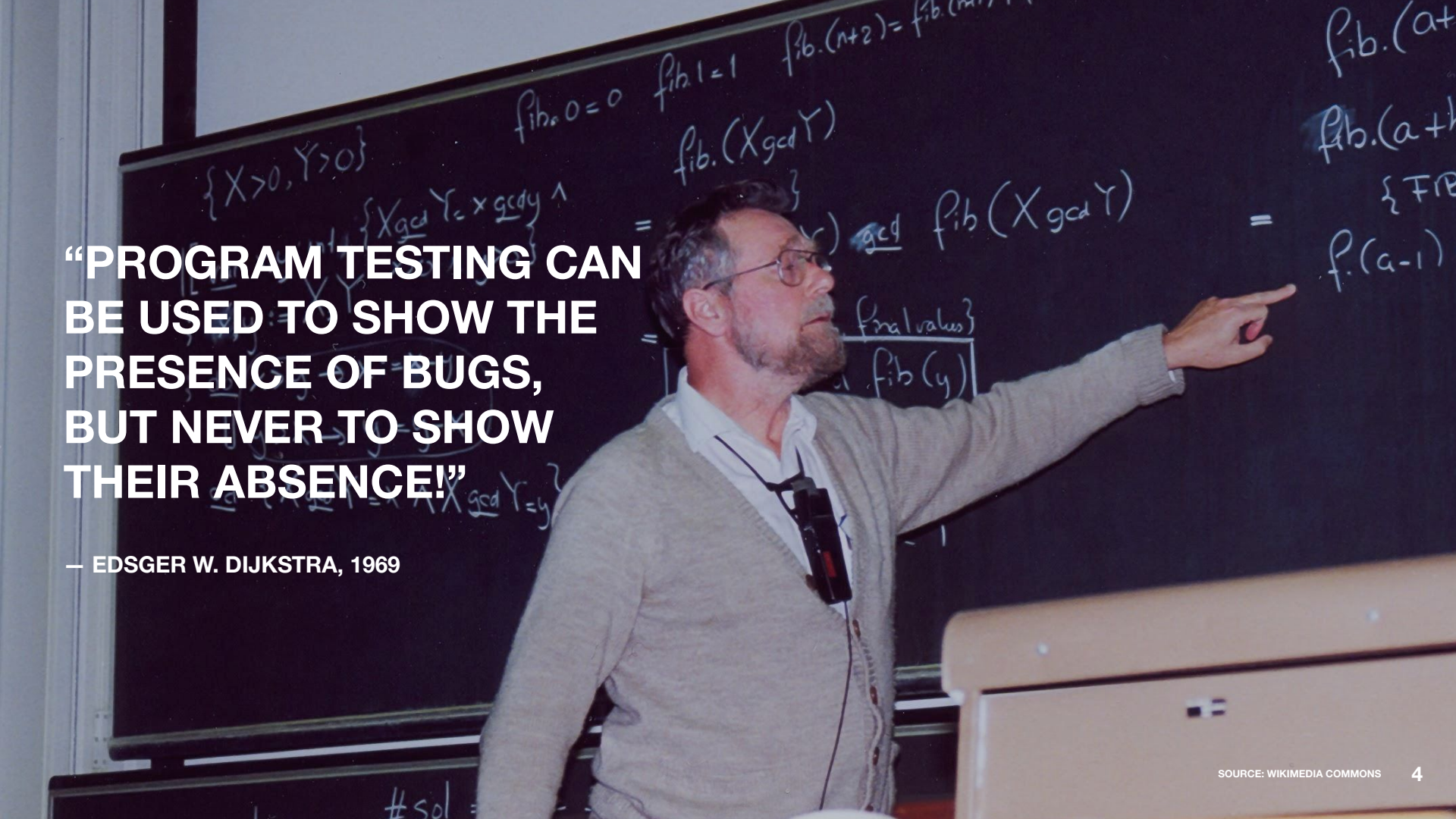
7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

Algorithms everywhere.

… But things can go wrong!

SOURCES: SPACE10, X, THINKSTOCK, VISA, VISION-BOX



SOURCE: WIKIMEDIA COMMONS

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

Program correction is difficult.

Most programmers are not capable of specifying programs;

The distance between specification and object poses a big challenge;

Difficult to know if the specification is correct and complete;

Specification is a delicate and complex process.

Certifying software leads to fewer problems when debugging during test and integration phases — potentially saving time, money and resources.

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

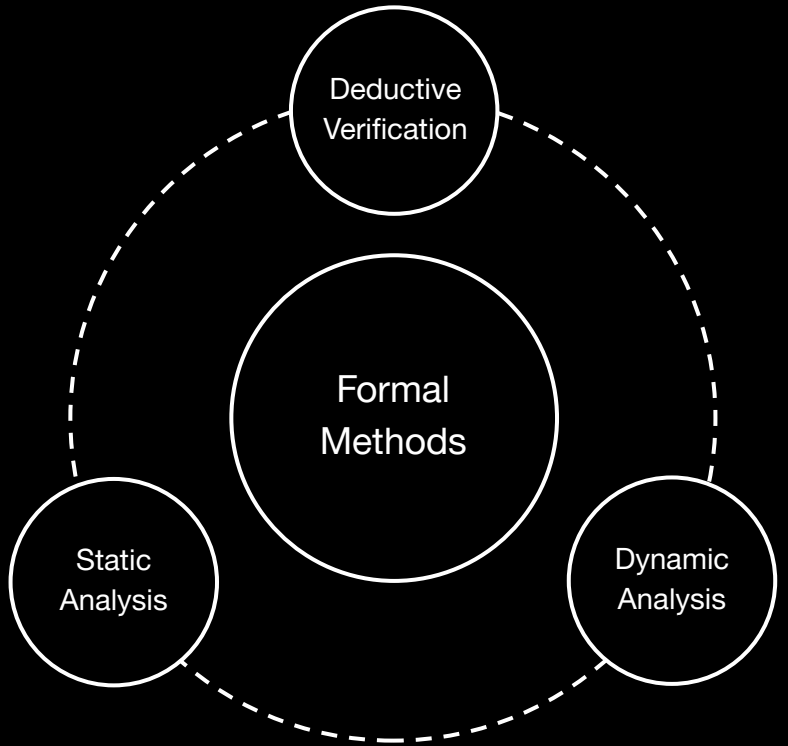
5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

Formal methods model complex systems as mathematical entities to verify their properties and ensure their correct behaviour.



Formal methods model complex systems as mathematical entities to verify their properties and ensure their correct behaviour.

Deductive verification

Program correction is achieved by specifying the source code through logical assertions — proving them correct using theorem provers.

Dynamic analysis

Examines the programs's properties by reasoning on its execution — detecting violations and vulnerabilities, and providing a better understanding of its behaviour.

Static analysis

Examines the source code directly, deriving properties for the program as whole — without actually executing it.

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

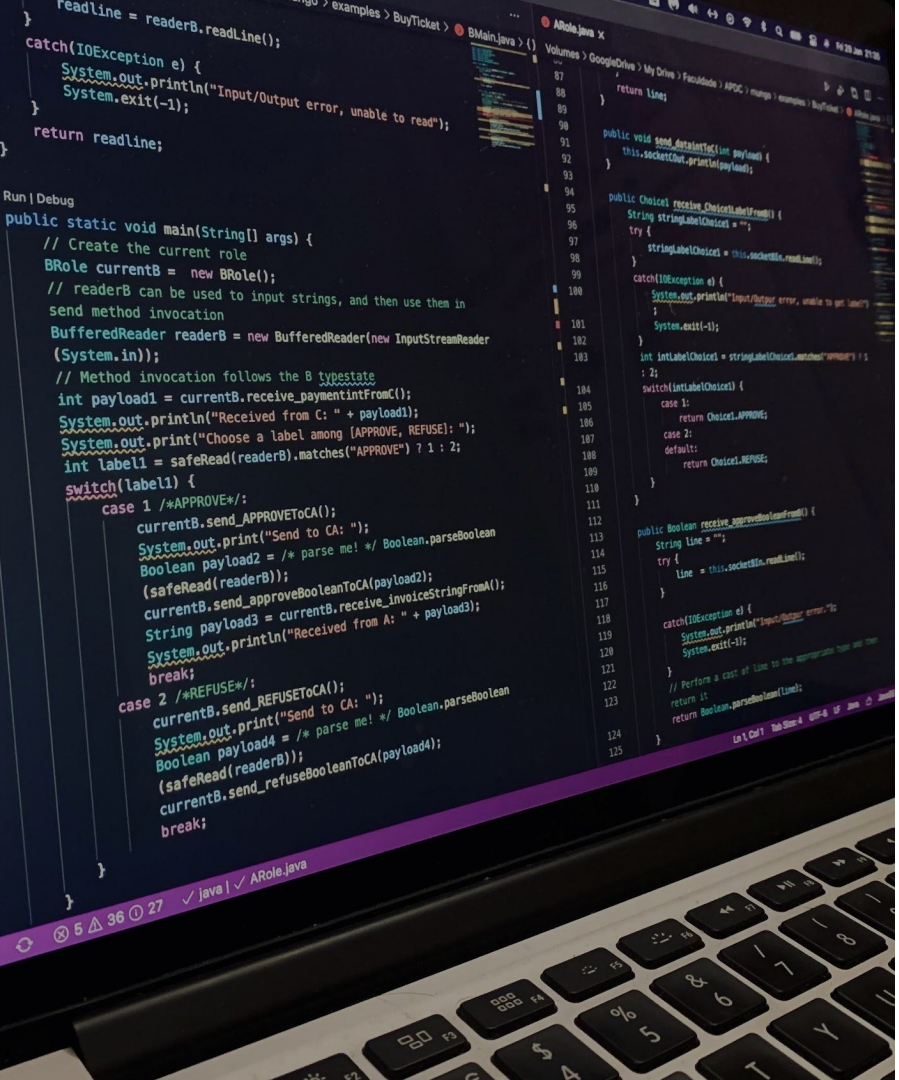
4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK



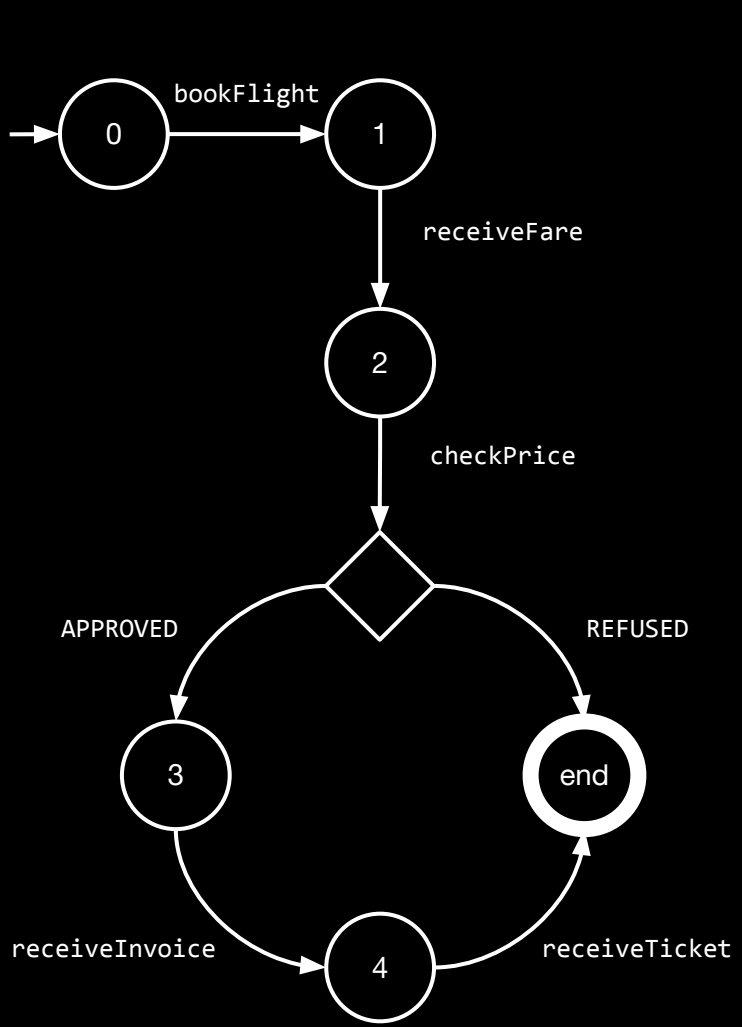
In the context of Behavioural Types

MUNGO

Java front-end tool to associate behavioural specifications of objects with Java classes;

Verifies if objects are used accordingly to their typestates.

```
typestate ClientProtocol {
    State0 = {
        void bookFlight(String): State1
    }
    State1 = {
        int receiveFare(): State2
    }
    State2 = {
        Code checkPrice(int):
            <APPROVED: State3,
             REFUSED: end>
    }
    State3 = {
        String receiveInvoice(): State4
    }
    State4 = {
        String receiveTicket(): end
    }
}
```



CONTRIBUTION

Typestates to Automata and back: a tool

Grammar for Mungo typestates;

Equivalent automaton model;

Algorithms to translate automata into typestates and *vice versa*;

Development of an online tool.

Typestates to Automata and back: a tool

André Trindade
NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
adt.trindade@campus.fct.unl.pt

João Mota
NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
jd.mota@campus.fct.unl.pt

António Ravara
NOVA School of Science and Technology
NOVA University Lisbon
Lisbon, Portugal
aravara@fct.unl.pt

Development of software is an iterative process. Graphical tools to represent the relevant entities and processes can be helpful. In particular, automata capture well the intended execution flow of applications, and are thus behind many formal approaches, namely behavioral types.

Typestate-oriented programming allow us to model and validate the intended protocol of applications, not only providing a top-down approach to the development of software, but also coping well with compositional development. Moreover, it provides important static guarantees like protocol fidelity and some forms of progress.

Mungo is a front-end tool for Java that associates a typestate describing the valid orders of method calls to each class, and statically checks that the code of all classes follows the prescribed order of method calls.

To assist programming with Mungo, as typestates are textual descriptions that are terms of an elaborate grammar, we developed a tool that bidirectionally converts typestates into an adequate form of automata, providing on one direction a visualization of the underlying protocol specified by the typestate, and on the reverse direction a way to get a syntactically correct typestate from the more intuitive automata representation.

1 Introduction

Detecting software errors and vulnerabilities is becoming increasingly important in a world where the demand for code development is soaring, often leading to incomplete specifications. Building tools to help achieve this goal is crucial as testing and manual revisions have proven to be insufficient to guarantee software correctness.

Indeed, a carefully designed test suite may detect the presence of many bugs, but it does not guarantee their absence [6]. A widespread practice is the use of programming languages with type systems [2]. These ensure that programs do not present errors for executing invalid operations, but the type of detected errors is quite limited. Day-to-day programmers have to deal with problems that arise from not checking the correct usage of an object. For example, a type system would prevent one from using an undefined method, but not from trying to write on a file prior to opening it.

Stateful objects are non-uniform [9], i.e., their methods' availability depends on their internal state. Behavioral types [8] are notions of types for programming languages representing the possible behavior of an entity, such as an automaton or a state machine. These notions allow us to declare behavioral specifications capturing the availability of methods. For example, a file should first be opened (once),

J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.):
13th Interaction and Concurrency Experience (ICE 2020).
EPTCS 324, 2020, pp. 25–42, doi:10.4204/EPTCS.324.4

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

GOALS

Apply the deductive approach to the verification of the algorithms—

Testing the viability of the approach by developing a proof of equivalence of regular expressions and finite automata.

CONTRIBUTIONS

1. Proof of equivalence of regular expressions and finite automata;
2. Formalisation of the conversion algorithm in pure functional code;
3. Development of a specification for the code;
4. Mechanical proof of correctness using the Why3 framework.

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

2.5.3 Epsilon-Closures

We shall proceed to give formal definitions of an extended transition function for ϵ-NFA's, which leads to the definition of acceptance of strings and languages by these automata, and eventually lets us explain why ϵ-NFA's can be simulated by DFA's. However, we first need to learn a central definition, called the *ϵ-closure* of a state. Informally, we ϵ-close a state q by following all transitions out of q that are labeled ϵ. However, when we get to other states by following ϵ, we follow the ϵ-transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ϵ. Formally, we define the ϵ-closure ECLOSE(q) recursively, as follows:

BASIS: State q is in ECLOSE(q).

INDUCTION: If state p is in ECLOSE(q), and there is a transition from state p to state r labeled ϵ, then r is in ECLOSE(q). More precisely, if δ is the transition function of the ϵ-NFA involved, and p is in ECLOSE(q), then ECLOSE(q) also contains all the states in $\delta(p, \epsilon)$.

2.5.4 Extended Transitions and Languages for ϵ-NFA's

The ϵ-closure allows us to explain easily what the transitions of an ϵ-NFA look like when given a sequence of (non-ϵ) inputs. From there, we can define what it means for an ϵ-NFA to accept its input.

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w. The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) =$ ECLOSE(q). That is, if the label of the path is ϵ, then we can follow only ϵ-labeled arcs extending from state q; that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of Σ; it cannot be ϵ, which is not in Σ. We compute $\hat{\delta}(q, w)$ as follows:

3.2.3 Converting Regular Expressions to Automata

We shall now complete the plan of Fig. 3.1 by showing that every language L that is $L(R)$ for some regular expression R, is also $L(E)$ for some ϵ-NFA E. The proof is a structural induction on the expression R. We start by showing how to construct automata for the basis expressions: single symbols, ϵ, and $\emptyset$. We then show how to combine these automata into larger automata that accept the union, concatenation, or closure of the language accepted by smaller automata.

All of the automata we construct are ϵ-NFA's with a single accepting state.

BASIS: There are three parts to the basis, shown in Fig. 3.16. In part (a) we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$, since the only path from the start state to an accepting state is labeled ϵ. Part (b) shows the construction for $\emptyset$. Clearly there are no paths from start state to accepting state, so $\emptyset$ is the language of this automaton. Finally, part (c) gives the automaton for a regular expression $\mathbf{a}$. The language of this automaton evidently consists of the one string a, which is also $L(\mathbf{a})$. It is easy to check that these automata all satisfy conditions (1), (2), and (3) of the inductive hypothesis.

INDUCTION: The three parts of the induction are shown in Fig. 3.17. We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expression; that is, the languages of these subexpressions are also the languages of ϵ-NFA's with a single accepting state. The four cases are:

1. The expression is $R + S$ for some smaller expressions R and S. Then the automaton of Fig. 3.17(a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for R or the automaton for S. We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R)$ or $L(S)$, respectively. Once we reach the accepting state of the automaton for R or S, we can follow one of the ϵ-arcs to the accepting state of the new automaton. Thus, the language of the automaton in Fig. 3.17(a) is $L(R) \cup L(S)$.
2. The expression is RS for some smaller expressions R and S. The automaton for the concatenation is shown in Fig. 3.17(b). Note that the start state of the first automaton becomes the start state of the whole, and the accepting state of the second automaton becomes the accepting state of the whole. The idea is that the only paths from start to accepting state go first through the automaton for R, where it must follow a path labeled by a string in $L(R)$, and then through the automaton for S, where it follows a path labeled by a string in $L(S)$. Thus, the paths in the automaton of Fig. 3.17(b) are all and only those labeled by strings in $L(R)L(S)$.
3. The expression is R^* for some smaller expression R. Then we use the automaton of Fig. 3.17(c). That automaton allows us to go either:
   (a) Directly from the start state to the accepting state along a path labeled ϵ. That path lets us accept ϵ, which is in $L(R^*)$ no matter what expression R is.
   (b) To the start state of the automaton for R, through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$, and so on, thus covering all strings in $L(R^*)$ except perhaps ϵ, which was covered by the direct arc to the accepting state mentioned in (3a).
4. The expression is (R) for some smaller expression R. The automaton for R also serves as the automaton for (R), since the parentheses do not change the language defined by the expression.

$$\text{ECLOSE} \subseteq S \to 2^S$$

$$\text{ECLOSE} \triangleq \left\{ (s, Q) \,\middle|\, Q = \{s\} \cup \{\, r \mid \forall q.\ q \in \delta(s, \varepsilon) \ \wedge\ q \neq s \ \wedge\ r \in \text{ECLOSE}(q) \,\} \right\}$$

$$\delta^* \subseteq S \times \Sigma^* \to 2^S$$

$$\delta^*(s, \varepsilon) = \text{ECLOSE}(s)$$

$$\delta^*(s, ua) = \bigcup_{q \in \bigcup_{r \in \delta^*(s,u)} \delta(r,a)} \text{ECLOSE}(q)$$

$$= \{\, p \mid \forall r, q.\ r \in \delta^*(s, u) \ \wedge\ q \in \delta(r, a) \ \wedge\ p \in \text{ECLOSE}(q) \,\}$$

$$\texttt{Compile} \subseteq \text{RegExp} \to \varepsilon - NFA_\Sigma$$

$$\texttt{Compile}(\emptyset) = \langle\, \{s_0\},\ \emptyset,\ s_0,\ \emptyset,\ \emptyset \,\rangle$$

$$\texttt{Compile}(\varepsilon) = \langle\, \{s_0, s_1\},\ \emptyset,\ s_0,\ \{(s_0, \varepsilon, s_1)\},\ \{s_1\} \,\rangle$$

$$\texttt{Compile}(a) = \langle\, \{s_0, s_1\},\ \{a\},\ s_0,\ \{(s_0, a, s_1)\},\ \{s_1\} \,\rangle$$

$$\texttt{Compile}(EG) = \langle\, S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ s_{0_E},\ \delta_{EG},\ F_G \,\rangle \text{, with}$$
$$\delta_{EG} = \delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G \text{, and}$$
$$S_E \cap S_G = \emptyset$$

$$\texttt{Compile}(E + G) = \langle\, \{i\} \cup S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ i,\ \delta_{E+G},\ F_E \cup F_G \,\rangle \text{, with}$$
$$\delta_{E+G} = \{(i, \varepsilon, s_{0_E}), (i, \varepsilon, s_{0_G})\} \cup \delta_E \cup \delta_G \text{, and}$$
$$i \notin (S_E \cup S_G) \ \wedge\ S_E \cap S_G = \emptyset$$

$$\texttt{Compile}(E^*) = \langle\, \{i\} \cup S_E,\ \Sigma_E,\ i,\ \delta_{E^*},\ \{i\} \,\rangle \text{, with}$$
$$\delta_{E^*} = \{(i, \varepsilon, s_{0_E})\} \cup \delta_E \cup \{(f, \varepsilon, i) \mid f \in F_E\} \text{, and}$$
$$i \notin S_E.$$

2.5.4 Extended Transitions and Languages for ϵ-NFA's

The ϵ-closure allows us to explain easily what the transitions of an ϵ-NFA look like when given a sequence of (non-ϵ) inputs. From there, we can define what it means for an ϵ-NFA to accept its input.

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w. The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$. That is, if the label of the path is ϵ, then we can follow only ϵ-labeled arcs extending from state q; that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of Σ; it cannot be ϵ, which is not in Σ. We compute $\hat{\delta}(q, w)$ as follows:

$$\delta^* \subseteq S \times \Sigma^* \to 2^S$$

$$\delta^*(s, \varepsilon) = \text{ECLOSE}(s)$$

$$\delta^*(s, ua) = \bigcup_{q \in \bigcup_{r \in \delta^*(s,u)} \delta(r,a)} \text{ECLOSE}(q)$$

$$= \{\ p \mid \forall r, q.\ r \in \delta^*(s, u)\ \wedge\ q \in \delta(r, a)\ \wedge\ p \in \text{ECLOSE}(q)\ \}$$

```
let rec ghost function delta_ext
(s: state) (w: word) (a: automaton) : fset state
=
   match w with
   | Nil -> eclose s a
   | Cons x Nil -> if eq x eps then eclose s a
                   else let rs = eclose s a in
                        let qs = fold_delta rs x a in
                            fold_eclose qs a
   | Cons x xs ->  let rs = delta_ext s xs a in
                   let qs = fold_delta rs x a in
                       fold_eclose qs a
   end
```

PROOF

FOR ANY REGULAR EXPRESSION, THE AUTOMATON RESULTING FROM THE CONVERSION OF THAT REGULAR EXPRESSION DESCRIBES THE SAME LANGUAGE.

$$\forall R \in RegExp_{\Sigma}.\ L(R) = L(\texttt{Compile}(R))$$

The proof is done by structural induction on the given regular expression—

$R = \emptyset$ $\quad$ $R = \varepsilon$ $\quad$ $R = a$

$R = EG$ $\quad$ $R = E + G$ $\quad$ $R = E^*$

Case $R = \emptyset$.

By definition, $L(\emptyset) = \emptyset$.
Let $A = \texttt{Compile}(\emptyset) = \langle \{s_0\}, \emptyset, s_0, \emptyset, \emptyset$

We want to show that $L(A) = \emptyset$.

Let $F = \texttt{finalStates}(A) = \emptyset$.
Hence, $L(A) = \{ w \in \emptyset^* \mid (\delta^*(s_0, w) \cap \emptyset)$

Case $R = \varepsilon$.

By definition, $L(\varepsilon) = \{\varepsilon\}$.
Let $A = \texttt{Compile}(\varepsilon) = \langle \{s_0, s_1\}, \emptyset, s_0, \{($

We want to show that $L(A) = \{\varepsilon\}$.

Note that, by definition, $\emptyset^* = \{\varepsilon\}$ and $\{\varepsilon$
Let $F = \texttt{finalStates}(A) = \{s_1\}$.
We have,

$$\begin{aligned}
\delta^*(s_0, \varepsilon) &= \texttt{ECLOSE}(s_0) \\
&= \{s_0\} \cup \{ r \mid \forall q.\ q \in \delta(s \\
&= \{s_0\} \cup \{ r \mid \forall q.\ q \in \{s_1\} \\
&= \{s_0\} \cup \texttt{ECLOSE}(s_1) \\
&= \{s_0\} \cup \{s_1\} \cup \{ r \mid \forall q.\ q \\
&= \{s_0\} \cup \{s_1\} \cup \emptyset \\
&= \{s_0, s_1\}
\end{aligned}$$

Hence,

$$\begin{aligned}
L(A) &= \{ w \in \Sigma^* \mid \\
&= \{ w \in \{\varepsilon\}^* \\
&= \{ w \in \{\varepsilon\} \mid \\
&= \{ w \in \{\varepsilon\} \mid \\
&= \{\varepsilon\}
\end{aligned}$$

Case $R = a$.

By definition, $L(a) = \{a\}$.
Let $A = \texttt{Compile}(a) = \langle \{s_0, s_1\}, \{a\}, s_0,$

We want to show that $L(A) = \{a\}$.

Note that, by definition, $\{a\}^* = \{a^n \mid n \in$
Furthermore, $\forall w \in \Sigma^*.\ w = \varepsilon w = w\varepsilon$.
Let $F = \texttt{finalStates}(A) = \{s_1\}$.

Notice that, by definition of the automa
lead the initial state s_0 to the only final
the only words possible to form are seq

Lemma 2. $\forall n \in \mathbb{N}_0,\ n > 1,\ \delta^*(s_0, a^n) \notin F$

We have,

$$\begin{aligned}
\delta^*(s_0, \varepsilon) &= \texttt{ECLOSE}(s_0) \\
&= \{s_0\} \cup \{ r \mid \forall q.\ q \in \delta \\
&= \{s_0\}
\end{aligned}$$

$$\begin{aligned}
\delta^*(s_0, a) &= \{ p \mid \forall r, q.\ r \in \delta^*(s_0, \\
&= \{ p \mid \forall r, q.\ r \in \{s_0\} \wedge \\
&= \{ p \mid \forall r, q.\ r = s_0 \wedge \\
&= \{ p \mid \forall r, q.\ r = s_0 \wedge \\
&= \{ p \mid \forall r, q.\ r = s_0 \wedge \\
&= \texttt{ECLOSE}(s_1) \\
&= \{s_1\}
\end{aligned}$$

Hence,

$$\begin{aligned}
L(A) &= \{ w \in \Sigma^* \mid (\delta^*(s_0, w) \cap F) \neq \\
&= \{ w \in \{a\}^* \mid (\delta^*(s_0, w) \cap \{s_1\} \\
&\quad \text{by Lemma 2} \\
&= \{ w \in \{\varepsilon, a\} \mid (\delta^*(s_0, w) \cap \{s \\
&= \{ w \in \{\varepsilon\} \mid (\delta^*(s_0, \varepsilon) \cap \{s_1\}) \\
&= \{a\}
\end{aligned}$$

Case $R = EG$.

Let $E \in \text{RegExp}(\Sigma_E)$ and $G \in \text{RegExp}(\Sigma_$
By definition, $L(EG) = L(E) \cdot L(G)$

Let

$$\begin{aligned}
\texttt{Compile}(E) &= \langle S_E \\
\texttt{Compile}(G) &= \langle S_G
\end{aligned}$$

By induction hypothesis,

$$L(\texttt{Compile}(E)) = L(E)$$

Finally, let

$$\begin{aligned}
A &= \texttt{Compile}(EG) = \langle S_E \cup S \\
\delta_{EG} &= \delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F \\
S_E \cap S_G &= \emptyset
\end{aligned}$$

Remark. $\delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G =$

We want to show that $L(A) = L(EG)$.

Note that, by definition of the automat
s_{0_E}, to the accepting state(s) in G, are t
E, and then go through the automaton
δ_{EG}, where words starting in one of th
the start state of the automaton for G, s
be split in two words: one accepted by

Lemma 3.

$$\begin{aligned}
&\forall w \in \Sigma^*.\ (\delta^*_{EG}(s_{0_E}, w) \cap F_G) \neq \emptyset \\
&\quad \Longrightarrow \exists u \in \Sigma^*_E,\ \exists v \in \Sigma^*_G.\ w = uv \wedge
\end{aligned}$$

Once again, we point to the fact that t
tributing to the overall structure of th
know that, when we compute the ac
whenever we reach the (what used to b
tion to (what used to be) the initial sta
due to ECLOSE. Specifically, we have,

$$\begin{aligned}
\forall f \in F_E,\ \texttt{ECLOSE}(f) &= \{f\} \cup \{r \mid \forall q.\ q \in \\
&= \{f\} \cup \texttt{ECLOSE}(s_{0_G}) \\
&= \{f, s_{0_G}\} \cup \{r \mid \forall q
\end{aligned}$$

Hence,

$$\begin{aligned}
L(A) &= \{ w \in \Sigma^* \mid (\delta^*_{EG}(s_{0_E}, w) \cap F_G \\
&= \{ w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_{EG}(s_{0_E}, \\
&\quad \text{by Lemma 3} \\
&= \{ uv \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_E(s_{0_E}, \\
&= \{ u \in \Sigma^*_E \mid (\delta^*_E(s_{0_E}, u) \cap F_E) \\
&= L(E) \cdot L(G) \\
&\quad \text{by induction hypothesis} \\
&= L(EG)
\end{aligned}$$

Case $R = E + G$.

Let $E \in \text{RegExp}(\Sigma_E)$ and $G \in \text{RegExp}(\Sigma_G)$
By definition, $L(E + G) = L(E) \cup L(G)$

Let

$$\begin{aligned}
\texttt{Compile}(E) &= \langle S_E \\
\texttt{Compile}(G) &= \langle S_G
\end{aligned}$$

By induction hypothesis,

$$L(\texttt{Compile}(E)) = L(E)$$

Finally, let

$$\begin{aligned}
A &= \texttt{Compile}(E + G) = \langle \{i\} \cup S_E \\
i &\notin (S_E \cup S_G) \wedge S_E \cap S_G = \emptyset, \text{ and} \\
\delta_{E+G} &= \{(i, \varepsilon, s_{0_E}), (i, \varepsilon, s_{0_G})\} \cup \delta_E \\
F_{E+G} &= F_E \cup F_G
\end{aligned}$$

We want to show that $L(A) = L(E + G)$.

Remark. $\forall w \in \Sigma^*.\ w = \varepsilon w$.

We have,

$$\begin{aligned}
\texttt{ECLOSE}(i) &= \{i\} \cup \{ r \mid \forall q.\ q \in \delta_{E+G}(\\
&= \{i\} \cup \{ r \mid \forall q.\ q \in \{s_{0_E}, s_{0_G}\} \\
&= \{i\} \cup \texttt{ECLOSE}(s_{0_E}) \cup \texttt{ECL} \\
&= \{i, s_{0_E}, s_{0_G}\} \\
&\quad \cup \{ r \mid \forall q.\ q \in \delta_{E+G}(s \\
&\quad \cup \{ r \mid \forall q.\ q \in \delta_{E+G}
\end{aligned}$$

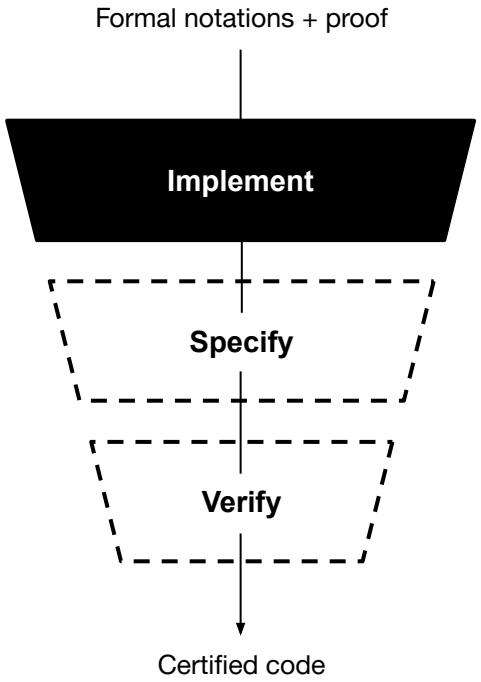
Note that, by definition of the automaton A, starting at the
can go to to the start state of either the automaton for E o
This is given by the above definition of the transition functio
state, i, leads to either s_{0_E} or s_{0_G}, the initial states for the
respectively, by ε. This behavior is, furthermore, shown in t
of ECLOSE(f). We can, thus, reach one of the accepting sta
which, by definition, is in $F_E \cup F_G$ — through a word w that, i
acceptance state of the automaton for E, otherwise ($w \in \Sigma^*_G$),
state of the automaton for G. This is also given by the de
besides the transitions we mentioned before, it comprises of

Lemma 4.

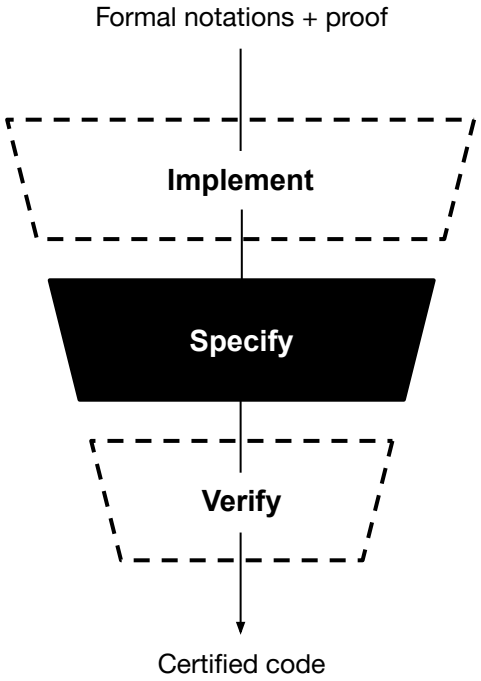
$$\begin{aligned}
&\forall w \in \Sigma^*.\ (\delta^*_{E+G}(i, w) \cap F_{E+G}) \neq \emptyset \\
&\quad \Longrightarrow \left(w \in \Sigma^*_E \wedge (\delta^*_E(s_{0_E}, w) \cap F_E) \neq \emptyset\right) \vee \left(w \in \Sigma^*_G \wedge (\delta^*_G \right.
\end{aligned}$$

Hence,

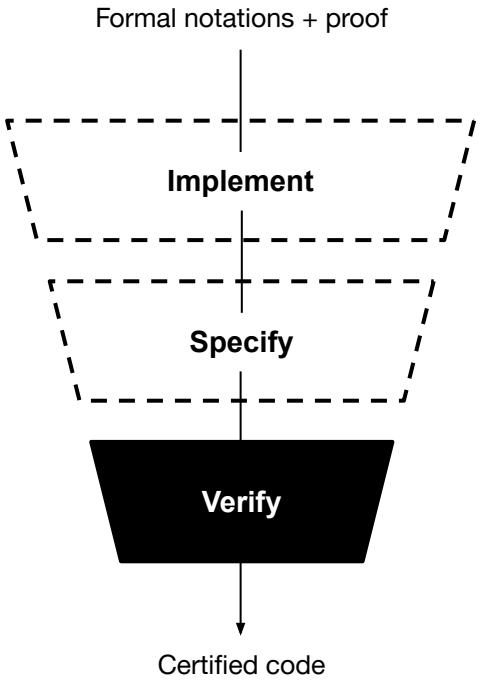
$$\begin{aligned}
L(A) &= \{ w \in \Sigma^* \mid (\delta^*_{E+G}(i, w) \cap F_{E+G}) \neq \emptyset \} \\
&= \{ w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_{E+G}(s_{0_E}, w) \cap (F_E \cup F_G)) \neq \emptyset \} \\
&\quad \text{by Lemma 4} \\
&= \{ w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_E(s_{0_E}, w) \cap F_E) \neq \emptyset \vee (\delta^*_G(s_{0_G} \\
&= \{ w \in \Sigma^*_E \mid (\delta^*_E(s_{0_E}, w) \cap F_E) \neq \emptyset \} \cup \{ w \in \Sigma^*_G \mid (\delta^*_G(\\
&= L(E) \cup L(G) \\
&\quad \text{by induction hypothesis} \\
&= L(E + G)
\end{aligned}$$



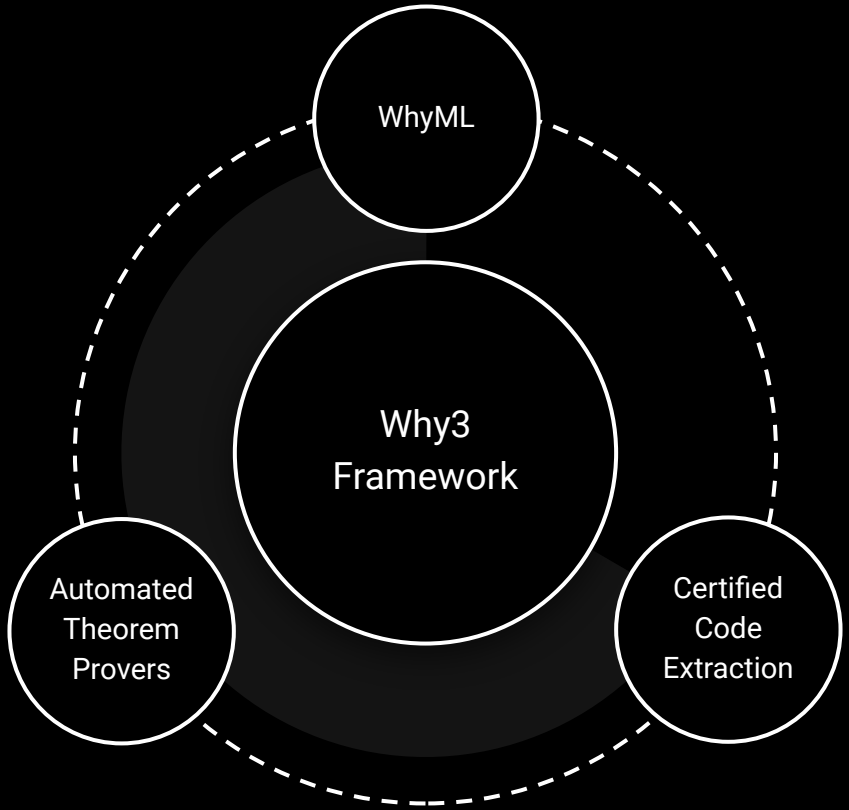

```
let rec ghost function delta_ext (s: state) (w: word) (a: automaton) : fset state
=
   match w with
   | Nil -> eclose s a
   | Cons x Nil -> if eq x eps then eclose s a
                   else let rs = eclose s a in
                        let qs = fold_delta rs x a in
                            fold_eclose qs a
   | Cons x xs ->  let rs = delta_ext s xs a in
                   let qs = fold_delta rs x a in
                       fold_eclose qs a
   end
```

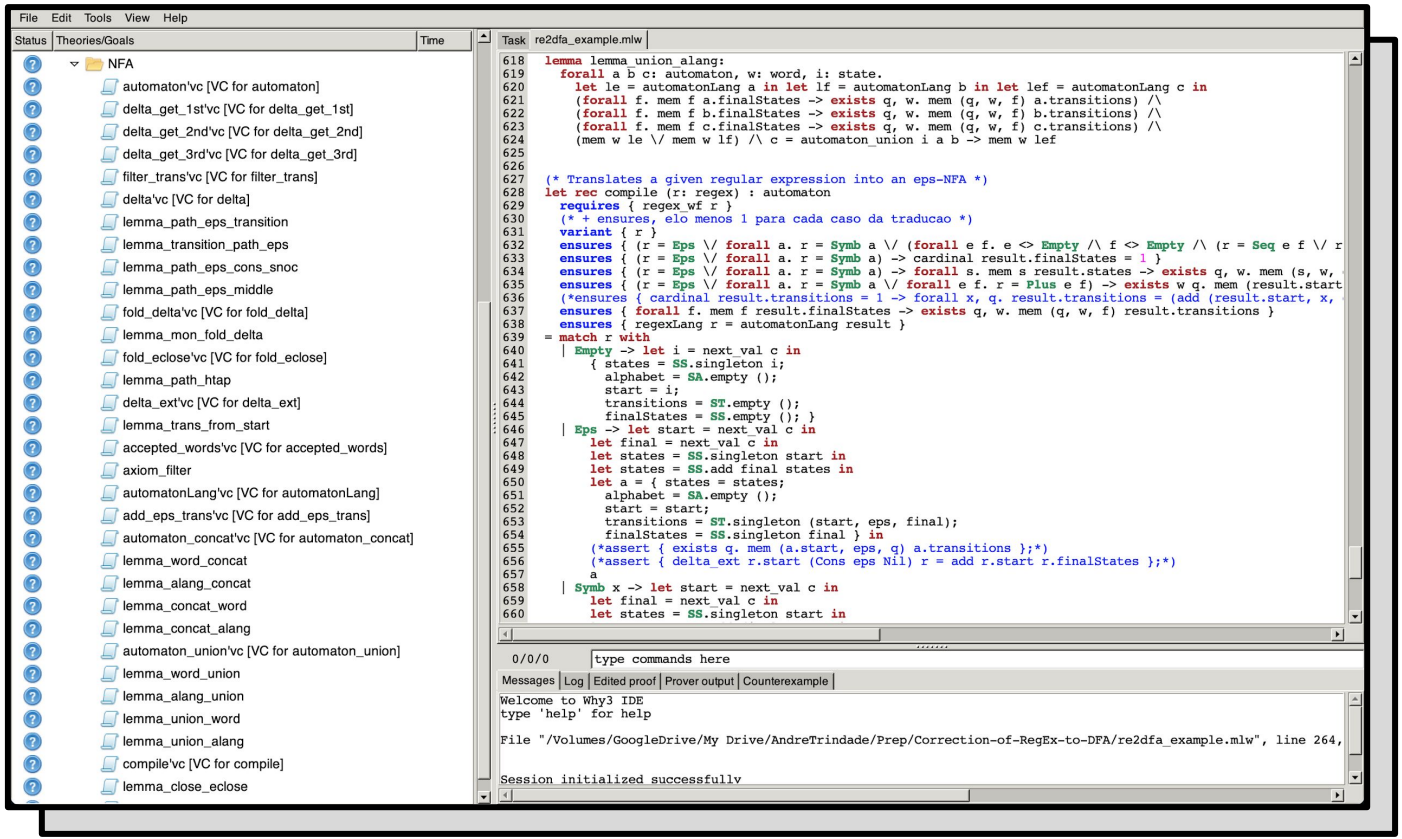



```
let rec ghost function delta_ext (s: state) (w: word) (a: automaton) : fset state
   variant  { Length.length w }
   ensures  { subset result a.states }
   ensures  { forall r. htap r w s a -> mem r result }
   ensures  { forall r. path s (reverse w) r a -> mem r result }
   ensures  { forall r. mem r result -> htap r w s a }
   ensures  { forall r. mem r result -> path s (reverse w) r a }
=
   match w with
   | Nil -> eclose s a
   | Cons x Nil -> if eq x eps then eclose s a
                   else let rs = eclose s a in
                        let qs = fold_delta rs x a in
                            fold_eclose qs a
   | Cons x xs ->  let rs = delta_ext s xs a in
                   let qs = fold_delta rs x a in
                       fold_eclose qs a
   end
```

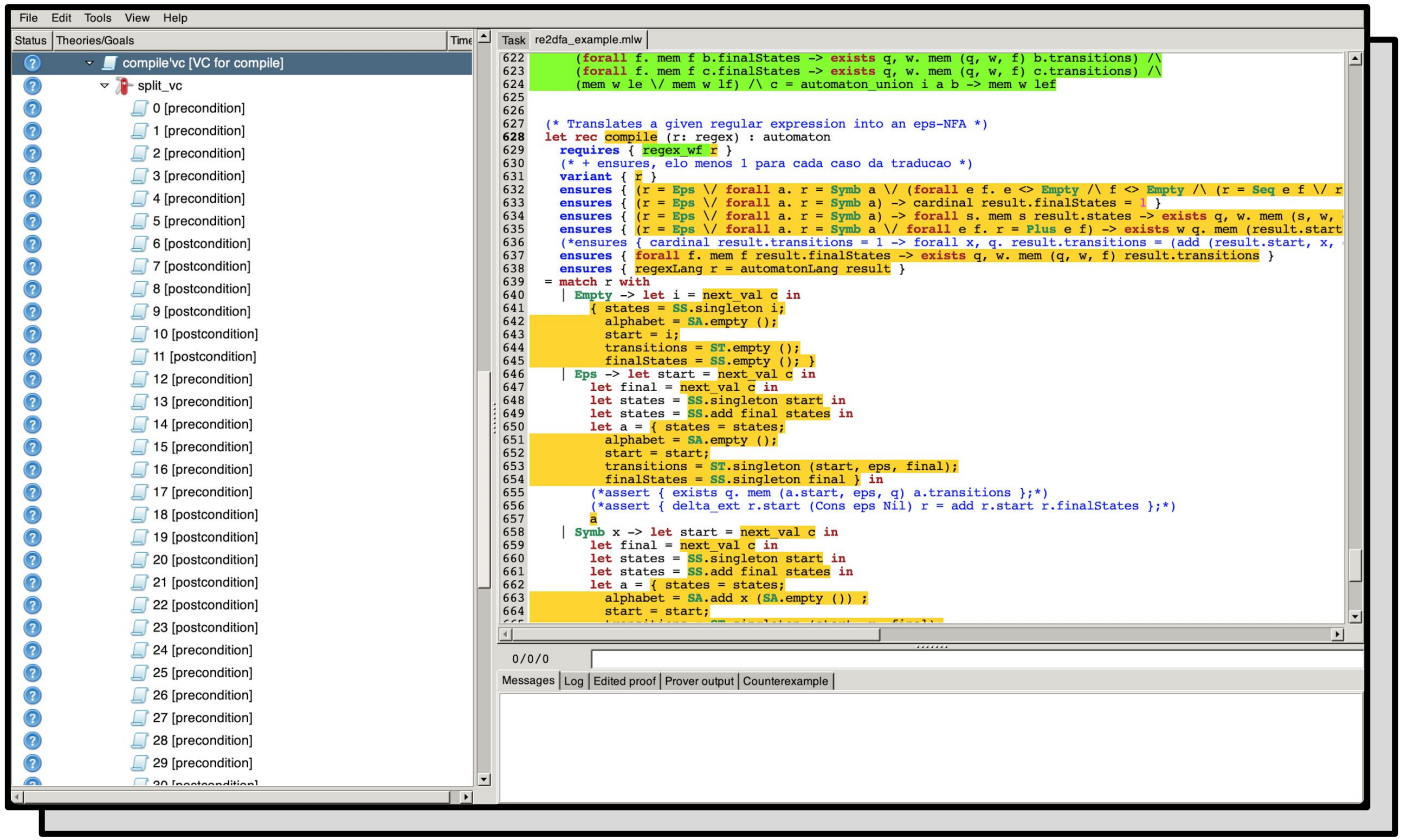



```
let rec ghost function delta_ext (s: state) (w: word) (a: automaton) : fset state
   variant  { Length.length w }
   ensures  { subset result a.states }
   ensures  { forall r. htap r w s a -> mem r result }
   ensures  { forall r. path s (reverse w) r a -> mem r result }
   ensures  { forall r. mem r result -> htap r w s a }
   ensures  { forall r. mem r result -> path s (reverse w) r a }
=
   match w with
   | Nil -> eclose s a
   | Cons x Nil -> if eq x eps then eclose s a
                   else let rs = eclose s a in
                        let qs = fold_delta rs x a in
                            fold_eclose qs a
   | Cons x xs ->  let rs = delta_ext s xs a in
                   let qs = fold_delta rs x a in
                       fold_eclose qs a
   end
```

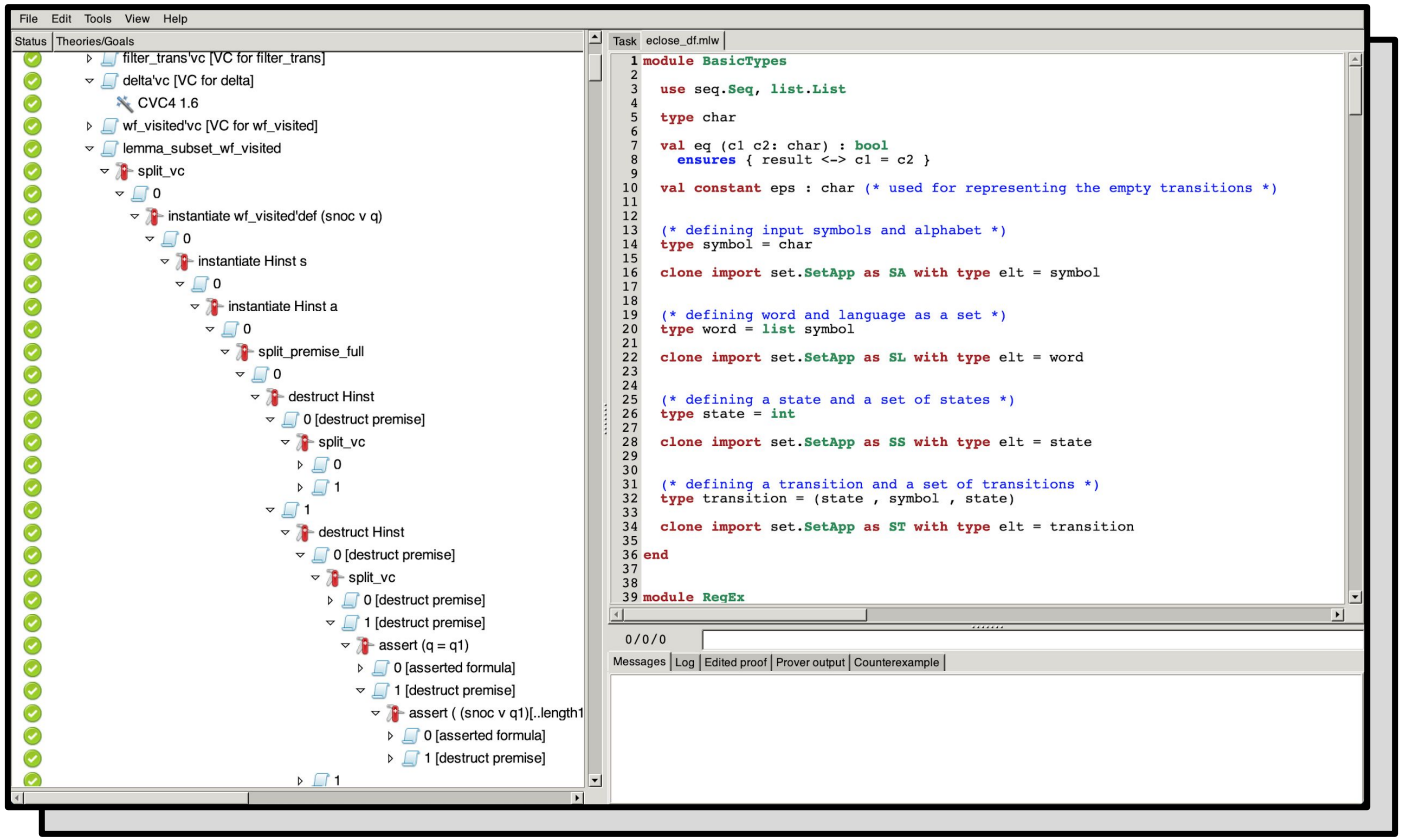

WhyML
Why3
Framework
Automated
Theorem
Provers
Certified
Code
Extraction


File Edit Tools View Help
Status Theories/Goals Time
NFA
automaton'vc [VC for automaton]
delta_get_1st'vc [VC for delta_get_1st]
delta_get_2nd'vc [VC for delta_get_2nd]
delta_get_3rd'vc [VC for delta_get_3rd]
filter_trans'vc [VC for filter_trans]
delta'vc [VC for delta]
lemma_path_eps_transition
lemma_transition_path_eps
lemma_path_eps_cons_snoc
lemma_path_eps_middle
fold_delta'vc [VC for fold_delta]
lemma_mon_fold_delta
fold_eclose'vc [VC for fold_eclose]
lemma_path_htap
delta_ext'vc [VC for delta_ext]
lemma_trans_from_start
accepted_words'vc [VC for accepted_words]
axiom_filter
automatonLang'vc [VC for automatonLang]
add_eps_trans'vc [VC for add_eps_trans]
automaton_concat'vc [VC for automaton_concat]
lemma_word_concat
lemma_alang_concat
lemma_concat_word
lemma_concat_alang
automaton_union'vc [VC for automaton_union]
lemma_word_union
lemma_alang_union
lemma_union_word
lemma_union_alang
compile'vc [VC for compile]
lemma_close_eclose
Task re2dfa_example.mlw
618 lemma lemma_union_alang:
619 forall a b c: automaton, w: word, i: state.
620 let le = automatonLang a in let lf = automatonLang b in let lef = automatonLang c in
621 (forall f. mem f a.finalStates -> exists q, w. mem (q, w, f) a.transitions) /\
622 (forall f. mem f b.finalStates -> exists q, w. mem (q, w, f) b.transitions) /\
623 (forall f. mem f c.finalStates -> exists q, w. mem (q, w, f) c.transitions) /\
624 (mem w le \/ mem w lf) /\ c = automaton_union i a b -> mem w lef
625
626
627 (* Translates a given regular expression into an eps-NFA *)
628 let rec compile (r: regex) : automaton
629 requires { regex_wf r }
630 (* + ensures, elo menos 1 para cada caso da traducao *)
631 variant { r }
632 ensures { (r = Eps \/ forall a. r = Symb a \/ (forall e f. e <> Empty /\ f <> Empty /\ (r = Seq e f \/ r
633 ensures { (r = Eps \/ forall a. r = Symb a) -> cardinal result.finalStates = 1 }
634 ensures { (r = Eps \/ forall a. r = Symb a) -> forall s. mem s result.states -> exists q, w. mem (s, w,
635 ensures { (r = Eps \/ forall a. r = Symb a \/ forall e f. r = Plus e f) -> exists w q. mem (result.start
636 (*ensures { cardinal result.transitions = 1 -> forall x, q. result.transitions = (add (result.start, x,
637 ensures { forall f. mem f result.finalStates -> exists q, w. mem (q, w, f) result.transitions }
638 ensures { regexLang r = automatonLang result }
639 = match r with
640 | Empty -> let i = next_val c in
641 { states = SS.singleton i;
642 alphabet = SA.empty ();
643 start = i;
644 transitions = ST.empty ();
645 finalStates = SS.empty (); }
646 | Eps -> let start = next_val c in
647 let final = next_val c in
648 let states = SS.singleton start in
649 let states = SS.add final states in
650 let a = { states = states;
651 alphabet = SA.empty ();
652 start = start;
653 transitions = ST.singleton (start, eps, final);
654 finalStates = SS.singleton final } in
655 (*assert { exists q. mem (a.start, eps, q) a.transitions };*)
656 (*assert { delta_ext r.start (Cons eps Nil) r = add r.start r.finalStates };*)
657 a
658 | Symb x -> let start = next_val c in
659 let final = next_val c in
660 let states = SS.singleton start in
0/0/0 type commands here
Messages Log Edited proof Prover output Counterexample
Welcome to Why3 IDE
type 'help' for help
File "/Volumes/GoogleDrive/My Drive/AndreTrindade/Prep/Correction-of-RegEx-to-DFA/re2dfa_example.mlw", line 264,
Session initialized successfully

HOW


File Edit Tools View Help
Status Theories/Goals Time
compile'vc [VC for compile]
split_vc
0 [precondition]
1 [precondition]
2 [precondition]
3 [precondition]
4 [precondition]
5 [precondition]
6 [postcondition]
7 [postcondition]
8 [postcondition]
9 [postcondition]
10 [postcondition]
11 [postcondition]
12 [precondition]
13 [precondition]
14 [precondition]
15 [precondition]
16 [precondition]
17 [precondition]
18 [postcondition]
19 [postcondition]
20 [postcondition]
21 [postcondition]
22 [postcondition]
23 [postcondition]
24 [precondition]
25 [precondition]
26 [precondition]
27 [precondition]
28 [precondition]
29 [precondition]
Task re2dfa_example.mlw
(forall f. mem f b.finalStates -> exists q, w. mem (q, w, f) b.transitions) /\
(forall f. mem f c.finalStates -> exists q, w. mem (q, w, f) c.transitions) /\
(mem w le \/ mem w lf) /\ c = automaton_union i a b -> mem w lef
(* Translates a given regular expression into an eps-NFA *)
let rec compile (r: regex) : automaton
requires { regex_wf r }
(* + ensures, elo menos 1 para cada caso da traducao *)
variant { r }
ensures { (r = Eps \/ forall a. r = Symb a \/ (forall e f. e <> Empty /\ f <> Empty /\ (r = Seq e f \/ r
ensures { (r = Eps \/ forall a. r = Symb a) -> cardinal result.finalStates = 1 }
ensures { (r = Eps \/ forall a. r = Symb a) -> forall s. mem s result.states -> exists q, w. mem (s, w,
ensures { (r = Eps \/ forall a. r = Symb a \/ forall e f. r = Plus e f) -> exists w q. mem (result.start
(*ensures { cardinal result.transitions = 1 -> forall x, q. result.transitions = (add (result.start, x,
ensures { forall f. mem f result.finalStates -> exists q, w. mem (q, w, f) result.transitions }
ensures { regexLang r = automatonLang result }
= match r with
| Empty -> let i = next_val c in
{ states = SS.singleton i;
alphabet = SA.empty ();
start = i;
transitions = ST.empty ();
finalStates = SS.empty (); }
| Eps -> let start = next_val c in
let final = next_val c in
let states = SS.singleton start in
let states = SS.add final states in
let a = { states = states;
alphabet = SA.empty ();
start = start;
transitions = ST.singleton (start, eps, final);
finalStates = SS.singleton final } in
(*assert { exists q. mem (a.start, eps, q) a.transitions };*)
(*assert { delta_ext r.start (Cons eps Nil) r = add r.start r.finalStates };*)
a
| Symb x -> let start = next_val c in
let final = next_val c in
let states = SS.singleton start in
let states = SS.add final states in
let a = { states = states;
alphabet = SA.add x (SA.empty ()) ;
start = start;
0/0/0
Messages Log Edited proof Prover output Counterexample

HOW


File Edit Tools View Help
Status Theories/Goals
filter_trans'vc [VC for filter_trans]
delta'vc [VC for delta]
CVC4 1.6
wf_visited'vc [VC for wf_visited]
lemma_subset_wf_visited
split_vc
0
instantiate wf_visited'def (snoc v q)
0
instantiate Hinst s
0
instantiate Hinst a
0
split_premise_full
0
destruct Hinst
0 [destruct premise]
split_vc
0
1
1
destruct Hinst
0 [destruct premise]
split_vc
0 [destruct premise]
1 [destruct premise]
assert (q = q1)
0 [asserted formula]
1 [destruct premise]
assert ((snoc v q1)[..length1
0 [asserted formula]
1 [destruct premise]
1
Task eclose_df.mlw
1 module BasicTypes
2
3 use seq.Seq, list.List
4
5 type char
6
7 val eq (c1 c2: char) : bool
8 ensures { result <-> c1 = c2 }
9
10 val constant eps : char (* used for representing the empty transitions *)
11
12
13 (* defining input symbols and alphabet *)
14 type symbol = char
15
16 clone import set.SetApp as SA with type elt = symbol
17
18
19 (* defining word and language as a set *)
20 type word = list symbol
21
22 clone import set.SetApp as SL with type elt = word
23
24
25 (* defining a state and a set of states *)
26 type state = int
27
28 clone import set.SetApp as SS with type elt = state
29
30
31 (* defining a transition and a set of transitions *)
32 type transition = (state , symbol , state)
33
34 clone import set.SetApp as ST with type elt = transition
35
36 end
37
38
39 module ReqEx
0 / 0 / 0
Messages Log Edited proof Prover output Counterexample

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

Program certification is an iterative process.

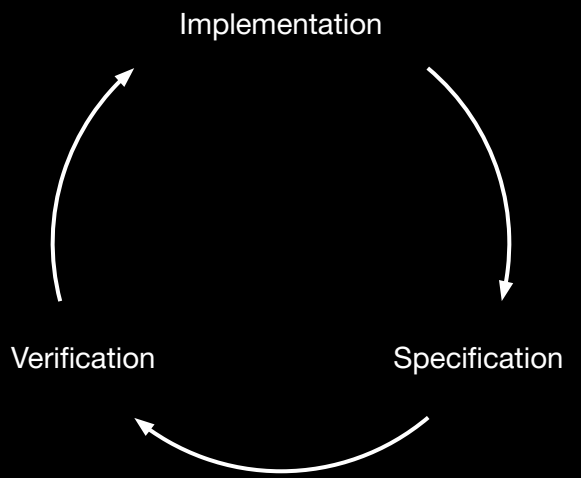
Runnable code *vs.* logical context—

Only runnable code needs to be implemented;

Functions exclusively used for specifying do not need to be implemented.

However, the less assumptions, the better—

Implementing *all* functions avoids under-specifying.



$$\texttt{Compile} \subseteq \text{RegExp} \rightarrow \varepsilon - NFA_{\Sigma}$$

$$\begin{aligned}
\texttt{Compile}(\emptyset) &= \langle\ \{s_0\},\ \emptyset,\ s_0,\ \emptyset,\ \emptyset\ \rangle \\
\texttt{Compile}(\varepsilon) &= \langle\ \{s_0, s_1\},\ \emptyset,\ s_0,\ \{(s_0, \varepsilon, s_1)\},\ \{s_1\}\ \rangle \\
\texttt{Compile}(a) &= \langle\ \{s_0, s_1\},\ \{a\},\ s_0,\ \{(s_0, a, s_1)\},\ \{s_1\}\ \rangle \\
\texttt{Compile}(EG) &= \langle\ S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ s_{0_E},\ \delta_{EG},\ F_G\ \rangle \text{ , with} \\
&\quad \delta_{EG} = \delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G \text{ , and} \\
&\quad S_E \cap S_G = \emptyset \\
\texttt{Compile}(E + G) &= \langle\ \{i\} \cup S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ i,\ \delta_{E+G},\ F_E \cup F_G\ \rangle \text{, with} \\
&\quad \delta_{E+G} = \{(i, \varepsilon, s_{0_E}), (i, \varepsilon, s_{0_G})\} \cup \delta_E \cup \delta_G \text{ , and} \\
&\quad i \notin (S_E \cup S_G)\ \wedge\ S_E \cap S_G = \emptyset \\
\texttt{Compile}(E^*) &= \langle\ \{i\} \cup S_E,\ \Sigma_E,\ i,\ \delta_{E^*},\ \{i\}\ \rangle \text{, with} \\
&\quad \delta_{E^*} = \{(i, \varepsilon, s_{0_E})\} \cup \delta_E \cup \{(f, \varepsilon, i) \mid f \in F_E\} \text{ , and} \\
&\quad i \notin S_E.
\end{aligned}$$

$$\forall R \in RegExp.\ L(R) = L(\texttt{Compile}(R))$$

```
let rec compile (r: regex) : automaton
   ensures { regex_lang r = automaton_lang result }
```

```
let rec compile (r: regex) : automaton
   requires { regex_wf r }
   variant  { r }
   ensures  { (r = Eps \/ forall a. r = Symb a \/ (forall e f. e <> Empty /\ f <> Empty
              /\ (r = Seq e f \/ r = Plus e f)) ) -> forall s. mem s result.states -> useful s result }
   ensures  { (r = Eps \/ forall a. r = Symb a) -> cardinal result.final_states = 1 }
   ensures  { (r = Eps \/ forall a. r = Symb a) -> forall s. mem s result.states
              -> exists q, w. mem (s, w, q) result.transitions \/ mem (q, w, s) result.transitions }
   ensures  { (r = Eps \/ forall a. r = Symb a \/ forall e f. r = Plus e f)
              -> exists w q. mem (result.start, w, q) result.transitions }
   ensures  { forall f. mem f result.final_states -> exists q, w. mem (q, w, f) result.transitions }
   ensures  { regex_lang r = automaton_lang result }
=
   ...
```

```
let rec compile (r: regex) : automaton
   requires { regex_wf r }
   variant  { r }
   ensures  { (r = Eps \/ forall a. r = Symb a \/ (forall e f. e <> Empty /\ f <> Empty
              /\ (r = Seq e f \/ r = Plus e f)) ) -> forall s. mem s result.states -> useful s result }
   ensures  { (r = Eps \/ forall a. r = Symb a) -> cardinal result.final_states = 1 }
   ensures  { (r = Eps \/ forall a. r = Symb a) -> forall s. mem s result.states
              -> exists q, w. mem (s, w, q) result.transitions \/ mem (q, w, s) result.transitions }
   ensures  { (r = Eps \/ forall a. r = Symb a \/ forall e f. r = Plus e f)
              -> exists w q. mem (result.start, w, q) result.transitions }
   ensures  { forall f. mem f result.final_states -> exists q, w. mem (q, w, f) result.transitions }
   ensures  { regex_lang r = automaton_lang result }
=
   ...
```

```
let ghost function automaton_lang (a: automaton) : fset word
    ensures  { forall w. mem w result -> (a.final_states <> empty /\ exists f. mem f a.final_states
               /\ path a.start w f a /\ mem w (sigma_ext a.alphabet) ) }
```

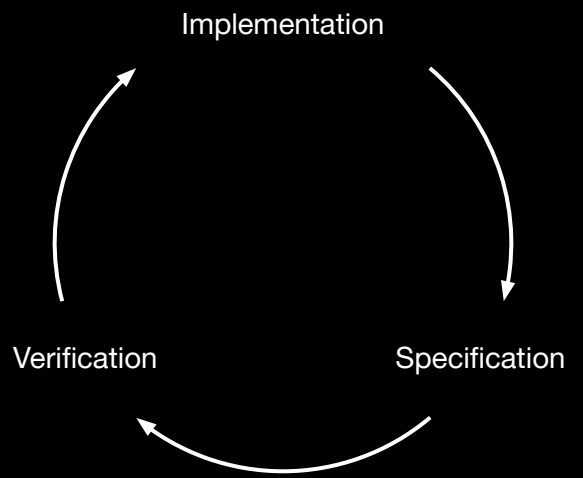
```
let ghost function automaton_lang (a: automaton) : fset word
   requires { forall f. mem f a.final_states -> exists q, x. mem (q, x, f) a.transitions }
   ensures  { forall w. mem w result -> forall x. LM.mem x w -> mem x (add eps a.alphabet) }
   ensures  { forall w. mem w result -> mem w (sigma_ext a.alphabet) }
   ensures  { forall w. mem w result -> w <> Nil }
   ensures  { is_empty a.final_states -> is_empty result }
   ensures  { result <> empty -> a.final_states <> empty }
   ensures  { (is_empty a.transitions /\ mem a.start a.final_states) -> result = singleton (Cons eps Nil) }
   ensures  { (is_empty a.transitions /\ not mem a.start a.final_states) -> result = empty }
   ensures  { mem a.start a.final_states -> mem (Cons eps Nil) result }
   ensures  { forall w. mem w result
               -> ( inter (delta_ext a.start (reverse w) a) a.final_states <> empty /\ mem w (sigma_ext a.alphabet) ) }
   ensures  { forall w. mem w result -> (a.final_states <> empty /\ exists f. mem f a.final_states
               /\ path a.start w f a /\ mem w (sigma_ext a.alphabet) ) }
   ensures  { forall w. (w <> Nil /\ mem w (sigma_ext a.alphabet)
               /\ inter (delta_ext a.start (reverse w) a) a.final_states <> empty) -> mem w result }
   ensures  { forall w f. (w <> Nil /\ mem w (sigma_ext a.alphabet) /\ mem f a.final_states
               /\ path a.start w f a) -> mem w result }
   ensures  { forall w. (w = Nil /\ inter (delta_ext a.start w a) a.final_states <> empty) -> mem (Cons eps Nil) result }
   ensures  { forall x f. mem f a.final_states /\ mem (a.start, x, f) a.transitions -> mem (Cons x Nil) result }
   ensures  { forall x f. mem f a.final_states /\ a.transitions = singleton (a.start, x, f)
               /\ not mem a.start a.final_states -> result = (singleton (Cons x Nil)) }
   ensures  { is_empty result \/ (exists w. mem w result /\ mem w (sigma_ext a.alphabet)
               /\ not is_empty (inter (delta_ext a.start (reverse w) a) a.final_states)) }
=
   filter (sigma_ext a.alphabet) (accepted_words a)
```

Program certification is an iterative process.

**“EVER TRIED. EVER FAILED.
NO MATTER.
TRY AGAIN. FAIL AGAIN. FAIL BETTER.”**

— SAMUEL BECKETT

Implementation, specification and verification inform each other.



We are smarter than we think,
we know more than we know.

The *pen-and-paper* proof still has informalities that need to be specified in the mechanised proof—

Concepts obvious to us, might not be obvious to the machine.

3 lemmas
mathematical proof

VS.

30 lemmas
mechanised proof

Limitations are not always limiting.

$$L \subseteq NFA_{\Sigma} \to 2^{\Sigma^*}$$
$$L(A) = \{w \in \Sigma^* \mid (\delta^*(s_0, w) \cap F) \neq \emptyset\}$$

```
let ghost function automaton_lang (a: automaton) : fset word
...
=
   filter (sigma_ext a.alphabet) (accepted_words a)


let ghost predicate accepted_words (a: automaton) (w: word)
   ensures  { result <-> exists f. mem f a.final_states
              /\ path a.start w f a }
=
   not (is_empty (inter (delta_ext a.start (reverse w) a)
a.final_states))


val function sigma_ext (a: fset symbol) : fset word
   ensures { forall w. mem w result -> w <> Nil }
   ensures { forall w. mem w result
             -> forall x. LM.mem x w -> mem x (add eps a) }
   ensures { forall w x. LM.mem x w /\ mem x (add eps a)
             /\ w <> Nil -> mem w result }
```

1 CONTEXT

2 PROBLEM

3 FORMAL METHODS

4 PREVIOUS CONTRIBUTIONS

5 GOALS + CONTRIBUTIONS

6 APPROACH

7 IMPORTANT LEARNINGS

8 CONCLUSIONS + FUTURE WORK

CONCLUSIONS

We contribute to the classic literature by rewriting known concepts and results more formally.

We prove the conversion from regular expression to finite automaton to be language-preserving—

Both in a *pen-and-paper* style and mechanically.

We show that Why3 allows for a mathematically rewarding mechanisation of results on Automata Theory—

Creating a base for mechanically-checked proofs of automata-related conversion algorithms.

FUTURE WORK

Introduce other automata-related concepts;

Proof of correction for the translation from finite automata to regular expressions;

Adapt the current proof to the proof of conversion between Mungo typestates and automata;

Develop a certified tool using the verified conversion algorithms.

A MECHANIZED PROOF OF KLEENE'S THEOREM IN WHY3

UNIVERSIDADE NOVA DE LISBOA

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

André Duarte Teixeira Trindade

2021