Regular Expressions in Agda

Alexandre Agular, Bassel Mannaa

2009-05-15

1 Introduction

The goal of this project is to write an implementation of Regular Expressions in Agda. The program makes a provably correct decision of whether a string x belongs to the language expresses by a regular expression e. This is done by simulating a DFA in a very efficient way. On the side we programmatically prove some properties of a language represented by a regular expression.

1.1 What is Agda?

Agda is a dependently typed functional programming language based on intuitionistic type theory, a theory developed by Per Martin-löf.

The correspondence between a proof and a program (Curry-Howard correspondence) is a phenomenon in constructive/intuitionistic logic, by its own definition the only way to prove the existence of a mathematical object is to be able to construct. Hence, the proof serve as a procedure.

Agda allows to write program that are provably correct by the compiler itself. This means that every Agda program has to terminate and be crash free. Hence Agda is not a Turing-complete language.

It is also called a proof assistant since it can be used only as an interactive proof checker instead of a programming language.

1.2 What are Regular Expression?

Regular Expression are finite expressions that are used to define regular language. They are used today mainly to search for strings that respect a given finite pattern in a text . They can represent all and only regular languages, namely all languages that can be represented as a deterministic finite automaton. Regular expressions are constructed inductively with the following constructors :

- $\emptyset$: represents the empty language ;
- ϵ : represents the language containing only the empty string ;
- a : represents the language containing only the symbol a, that is $L(a) = a$;
- e^* : Kleene closure of a regular expression e, the language obtained by concatenating strings from e zero or more times ;

- $e_1 + e_2$: The union of the two languages represented by e_1 and e_2, $L(e_1 + e_2) = L(e_1) \cup L(e_2)$;
- $e_1 e_2$: The language obtained by concatenation of a string from e_1 with a string from e_2, $L(e_1 e_2) = L(e_1)L(e_2)$;

] '

2 Theory and Background

2.1 preliminary definitions

One possible definition of a Regular Language is the following :
Definition A language is regular if and only if it can be described by a formal regular expression.

The above definition is not unique, Regular Expressions are just one of many equally powerfull representations of Regular Languages. Among these, and one that will be of interest here is Deterministic Finite Automata. It is easy to show that these two representations are equivalent [1], that is a language is accepted by some DFA iff it can be described by a regular expression.

Definition A DFA is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where :

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

Letting Σ^* be the set of all strings formed by concatenating symbols from Σ 0 or more times, we additionally define $\hat{\delta} : Q \times \Sigma^* \to Q$ as follows:

- $\hat{\delta}(q, \epsilon) = q$ where ϵ denotes the empty string.
- $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ for all $x \in \Sigma^*$ and $a \in \Sigma$

We say that a string w is accepted by an automaton $A = (Q, \Sigma, \delta, q_0, F)$ if and only if $\hat{\delta}(q_0, w) \in F$

2.2 Myhill-Nerode Theorem

Myhill-Nerode theorem provide a representation free characterization of regular languages. The theorem states that a language $L \subseteq \Sigma^*$ is regular if and only if L is the union of some equivalence classes of a right invariant equivalence relation of finite index, moreover, the theory identifies the equivalence relation of the smallest index to be the one defined in terms of abstract states [1].

2.2.1 abstract states

For any $x, y \in \Sigma^*$ and $L \subseteq \Sigma^*$ we say that $x \equiv_L y$ if and only if $\forall z \in \Sigma^*.\ xz \in L \leftrightarrow yz \in L$.
We define $L/x = \{w \mid xw \in L\}$ to be the derivative of L to x. And we note that according to the above $x \equiv_L y \leftrightarrow L/x = L/y$ we call L/x an abstract state.

2.2.2 Derivation of Regular Expression

If we have a way to construct the abstract states directly from a regular expression (i.e. calculate the derivation of a regular expression to some symbol in the alphabet), then we can construct a DFA directly from a regular expression in which the abstract states would serve as the set Q of the DFA states. better yet, if the purpose is to test membership of a string x in a language $L(E)$ where E is the regular expression at hand; we don't need to build the whole automaton, we only need the abstract states (derivatives) that occur along the path from the start state to the final state with transitions on symbols of x. For exmple the DFA in figure 1, represents the regular expression $(aa + bb) * cd$, the only states we need to construct are those on the path cd from start state, that is, only s, c, d.
Luckily, there is a set of rules with which we can correctly calculate the derivative of a regular expression; those rules are due to Brzozowski [6]. The rules guarantee that $L(E)/a = L(E/a)$, for some $a \in \Sigma$. The rules are defined by structural induction on the regular expression :

- $\emptyset/a = \emptyset$
- $\epsilon/a = \emptyset$
- $b/a = \begin{cases} \epsilon & if\ \ b = a \\ \emptyset & \text{Otherwise} \end{cases}$
- $e^*/a = (e/a)e^*$
- $(e_1 + e_2)/a = (e_1/a) + (e_2/a)$
- $(e_1e_2)/a = \begin{cases} (e_1/a)e_2 & if\ \ \epsilon \notin e_1 \\ (e_1/a)e_2 + e_2/a & \text{Otherwise} \end{cases}$

A proof of the correctness of the above rules can be found in the

Figure 1: Minimal DFA for $e = (aa + bb)^*cd$. The highlighted path is the derivation path of e/cd

2.3 The Algorithm

In its simplest form the algorithm recursively derivates the regular expression to the symbols in the string. The derivation stops whenever either the derivative is empty $e/a = \emptyset$ which means there is no path to an accepting state on the arc labeled with a out of state e or when the string has no more symbols $x = \epsilon$, at the latter case we need to check if the state we are at now is an accepting state we do this by checking if the current state has epsilon, meaning if the regular expression representing the state accepts ϵ. Leaving out the details of checking if the regexp is empty or if it accepts ϵ, the algorithm would look like this

accept :	[char] $\rightarrow$	RegExp $\rightarrow$	Bool	
accept	[ ]	e		= hasEpsilon e
accept	$a :: x$	e	| empty e	= false
			| otherwise	= accept x (e/a)

2.3.1 Time Complexity

From Myhill-Nerode Theorem we know the the number of abstract states (derivatives) of a given regular expression e is finite. We have a finite number of symbols in the alphabet, so we have a pair of derivative e', symbol a that have the worst time complexity fo computation of e'/a. Let's denote this worst bound T, we

note that T depends only on the regular expression e and the alphabet Σ. We can bound the worst case time complexity of the algorithm for a given string x to be $O(T|x|)$.
The above analysis doesn't guarantee that the algorithm is linear. Since T might be exponential in terms of the length of the input regular expression. The fact is, derivation sometimes causes structural expansion to the regular expression (this is the reason for the simplification *simp* we add in the code), however, we could not reach a tight analysis of the limit of this structural expansion with derivation. Hence, no guarantee on the asymptotic complexity. However, the algorithm performs really well in practice.

3 Agda Implementation

The goal of the project is to try and implement the theoretical algorithm presented above in Agda. This in order to be able to prove that a given string w is in a language $L(E)$. In this section, we present some relevant part of our implementation to explain how the proof of $w \in L(e)$ is built.

3.1 The type of regular expressions

We first define a type for regular expression :

$$
\begin{aligned}
\text{data RegExp} &: \text{Set where} \\
\emptyset &: \text{RegExp} \\
\epsilon &: \text{RegExp} \\
\bullet &: \text{RegExp} \\
\text{symb} &: \text{carrier} \to \text{RegExp} \\
\_\star &: \mathit{RegExp} \to \text{RegExp} \\
\_|\_ &: \text{RegExp} \to \text{RegExp} \to \text{RegExp} \\
\_ \odot \_ &: \text{RegExp} \to \text{RegExp} \to \text{RegExp}
\end{aligned}
$$

The two things to note here are that *carrier* represents the set of symbols Σ. We add a new constructor, $\bullet$ that is not present in the classical set of constructor for regular expression. It represents any symbol. It can be proved that it doesn't affect any known theorem about regular expressions since it can be build using the other constructors.

3.1.1 Proofs in Agda

In Agda, dependent types can be used to prove some properties over specific elements of set. The idea is the following, suppose that you have a type A dependent on a type B. The type A define a special property of the elements of B. If we can construct the type $A\,b$, this mean that the property A is true for the element b. Put it another way, the fact that the type $A\,b$ is not empty means that the property is true and the proof of the property is the constructor that has been used to get the element of $A\,b$. see [3]

3.2 The type has-ϵ_

The type has-ϵ_ contains the proofs that some regular expression contains ϵ. Since RegExp is a recursive data type, this proof will be a proof by induction on the structure of the regular expression. Here is the code :

$$
\begin{aligned}
&\text{data has-}\epsilon : \text{RegExp} \to \text{Set where} \qquad \text{(has-e datatype)}\\
&\quad \epsilon\text{-}has\text{-}\epsilon : has\text{-}\epsilon\ \epsilon\\
&\quad |^{l}\text{-}has\text{-}\epsilon : \forall\{e_1\ e_2\} \to has\text{-}\epsilon\ e_1 \to has\text{-}\epsilon\ (e_1|e_2)\\
&\quad |^{r}\text{-}has\text{-}\epsilon : \forall\{e_1\ e_2\} \to has\text{-}\epsilon\ e_2 \to \text{has-}\epsilon\ (e_1|e_2)\\
&\quad \odot\text{-}has\text{-}\epsilon : \forall\{e_1\ e_2\} \to has\text{-}\epsilon\ e_1 \to has\text{-}\epsilon\ e_2 \to\ has\text{-}\epsilon\ (e_1 \odot e_2)\\
&\quad \star\text{-}has\text{-}\epsilon : \forall\{e\} \to has\text{-}\epsilon\ e^{\star}
\end{aligned}
$$

The idea is that one should enumerate all the cases in which the property hold. meaning, one should have a constructor that proves/constructs that some regular expression has ϵ if it has. Now the problem is that this type only prove if the property is true. To prove that this property is false what we do is a proof by contradiction namely, a function from the type has-ϵ e to a contradiction. The contradiction in Agda is represented by an empty set/absurdity (a set with no constructors, based on the analogy between falsehood of a statement and emptiness of a set) :

$$\text{data } \bot : \text{Set where}$$

And the negation is a function that constructs the signature of any proof by contradiction :

$$
\begin{aligned}
\neg\_ &: \text{Set} \to \text{Set}\\
\neg\ \text{P} &= \text{P} \to \bot
\end{aligned}
$$

We present now an example of such proof by contradiction. More examples can be found in the code. The example we present is the proof that the regular expression $e_1 \odot e_2$ does not contain ϵ.

First we define a type for the product of two sets :

$$
\begin{aligned}
&\text{data}\_ \times \_\ (\text{A B} : \text{Set}) : \text{Set where}\\
&\qquad\qquad \text{both} : \text{A} \to \text{B} \to \text{A} \times \text{B}
\end{aligned}
$$

And we define a small proof :

$$
\begin{aligned}
&has\text{-}\epsilon\text{-}both : \forall\{e f\} \to has\text{-}\epsilon\ (e \odot f) \to (has\text{-}\epsilon\ e) \times (has\text{-}\epsilon\ f)\\
&has\text{-}\epsilon\text{-}both\ (\odot\text{-}has\text{-}\epsilon\ p_1\ p_2) = \text{both}\ p_1\ p_2
\end{aligned}
$$

Here we are not concerned about building the set product type itself, as was the case above (has-e datatype) when building the datatype *has-*ϵ, rather it's the correctness of the implication that is of interest. Functions in Agda can be seen as the implication operator in logic. By writing this function, we write a proof of the implication stated in the signature of the function. The type checker guarantee the validity of the proof by checking that the returned object's type matches the defined dependent type in the signature [2].

Now we can write a proof by contradiction for the fact that $e \odot f$ does not have ϵ if either e or f doesn't, here we present the case when e doesn't have ϵ :

$$\epsilon\text{-}proof\text{-}\odot^l : \forall\{e\ f\} \to \neg\ (has\text{-}\epsilon\ e) \to \neg\ (has\text{-}\epsilon\ (e \odot f))$$
$$\epsilon\text{-}proof\text{-}\odot^l \text{ npe pef with } has\text{-}\epsilon\text{-}both \text{ pef}$$
$$\mid \text{both pe pf} = \text{npe pe}$$

Note here that according to the definition of $\neg$, this function's signature is equivalent to :

$$\epsilon\text{-}proof\text{-}\odot^l : \forall\{e\ f\} \to \neg\ (has\text{-}\epsilon\ e) \to has\text{-}\epsilon\ (e \odot f) \to \bot$$

Which means that this function take as argument an *implication* from has-ϵ e to $\bot$ and a hypothetical element of (starts an assumption block) has-ϵ $(e \odot f)$ and return a contradiction. The parallel with natural deduction here is striking : what we do is simply use *modus ponens* :

$$\dfrac{\dfrac{\dfrac{}{has\text{-}\epsilon(e \odot f)}}{has\text{-}\epsilon\ e \times has\text{-}\epsilon\ f} \qquad \dfrac{}{has\text{-}\epsilon\ e \Rightarrow \bot}}{\bot}$$

Using the small proofs described above.

Having done this, the next part is to write a function that given a regular expression returns a proof of wether this regular expression has ϵ or not. To do this we use a data type that can contain these two types :

$$\text{data Dec(P : Set) : Set where}$$
$$\text{yes : (p : P)} \to \text{Dec P}$$
$$\text{no : (p : } \neg \text{ P)} \to \text{Dec P}$$

This type can be constructed with either *yes* with a proof that the property is true or no with a proof that it is false. Now we can define our function to construct of proof of wether a regular expression has ϵ or not :

$$has\text{-}\epsilon? : (e : \text{RegExp}) \to \text{Dec}(has\text{-}\epsilon\ e)$$

We don't present this code here, it is available in the source code and does not contain any new proof technique. What it does is simply pattern match on the regular expression structure and build a proof for each case which is equivalent in nature to what we would do in natural deduction.

3.3 Proof of $x \in L(e)$

To finish this section, we present some of the data types we use to build a proof that some string is in some language represented by a regular expression.

First we build a type $x \in_\smile [\![e]\!]$ that can be constructed only if x is in $L(e)$. We define as follows :

$$\text{data\_}\in_\smile[\![\_]\!] : \text{List carrier} \to \text{RegExp} \to \text{Set where}$$
$$\text{base} : \forall\{e\} \to has\text{-}\epsilon\ e \to []\in_\smile [\![e]\!]$$
$$\text{step} : \forall\{a\ as\ es\ e'\ e\} \to e' = e/a \to es\ simp\ e' \to \neg\ (is\text{-}\emptyset\ es) \to as\in_\smile [\![es]\!] \to a :: as \in_\smile [\![e]\!]$$

Here the *base* constructor states that the only way to construct an element of type $[] \in_\smile [\![e]\!]$ is to provide a proof of has-ϵ e. The other constructor, *step*, require more proofs :

- $e' = e/a$ which is a proof that e' is the derivation of e by a ;
- es simp e' which is a proof that es is the simplification of e' (used for efficiency purpose) ;
- $\neg$ (is-$\emptyset$ es) which is a proof that es is not the empty language (which means that at least one string of $L(e)$ starts with a) ;
- $as \in_\smile [\![es]\!]$ which is a proof that the rest of the string, as, is in the language of the derivation (using the simplified version of it for efficiency purpose).

This constructor returns $a :: as \in_\smile [\![e]\!]$ if all the proof described before can be provided. Each of these proofs is available in the source code.

Here is the signature of the *accepts* function that build this proof :

$$\text{accepts} : (re : \text{RegExp}) \to (as : \text{List carrier}) \to \text{Maybe}(as \in_\smile [\![re]\!])$$

The *Maybe* type is equivalent to the one in Haskell : it returns either *just something* or *nothing*. It is important to note here that we did not use the type *Dec* like for has-ϵ. The is due to some limitations of regular expression. Indeed it has been proventhat it is impossible to write a general function that could check if two regular expressions represent the same language without transforming first the regular expression to another representation of regular languages. Consequently we can prove that some regular expression is not the derivation of another and we can not prove either that some strings are not part of a language.

4 Conclusion

The goal of the project was to implement Automata Theory in Agda. The main decision problem is to decide wether a given string is a member of a given language. Adapting the regular expression representation of a language, we use the Brzozowski derivation algorithm. Agda provides a good tool to tackle this decision problem and it's subproblems (emptiness of a language...etc) in a provably correct manner. Much of the effort was invested in using as less assumption (postulates) as possible, which turned out to be very time consuming.

However, we note a particular defficiency in our program. That is the inability to prove the non-membership of some string in a given language. Indeed, we prove membership by constructing an element of the dependent data type $x \in_\smile [\![E]\!]$ in doing this we have to be able to construct an element of the datatype $e' = e/a$ at each step of the induction, to do a complete pattern matching we need to include the cases when this construction fails (i.e. the $Nothing$ case). The fact that we couldn't construct an element of the datatype doesn't execulde the possibility that e' is a derviative of e to a since e' might be semanticaly equivalent to e/a but syntactically different. From our search it

doesn't seem possible to prove equality of two regexp by symbolic manipulation (rewriting). The only way we are aware of to do this is by constructing the correspoding minimal DFAs of both regexp in question, but since the main idea behind the algorithm is to avoid this particualr construction, it would render the whole program pointless.

During this project, we had an opportunity to gain knowledge on two theoretical subjects, namely *dependent type theory* and *program verification*. Invariants, post/pre conditions are naturally present in the signatures of dependent types and constructors in Agda.

One way of extending this project is by plugging more of the Automata Theory proofs, specially, proving some closure properties of regular languages. For example, closure under homomorphism, since homomorphic function can be applied directly to regular expressions.

References

[1] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Massachusetts, ISBN 0-201-029880-X, 1979.

[2] Ana Bove and Peter Dybjer, *Dependent Types at Work*. Chalmers University of Technology, Göteborg, Sweden, 2008.

[3] Bengt Nordström, Kent Petersson, Jan M. Smith, *Programming in Martin-Löf's Type Theory*. Department of Computing Sciences, University of Göteborg, Chalmers University, Göteborg, Sweden. Oxford University press, 1990.

[4] Ulf Norell, *Dependently typed programming in Agda*. Chalmers University of Technology, Göteborg, Sweden,

[5] U. Norell, *Towards a practical programming language based on dependent type theory. PhD thesis*, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 G¨oteborg, Sweden, September 2007.

[6] Janusz A. Brzozowski, *Derivatives of Regular Expressions*, Journal of the ACM (JACM), v.11 n.4, p.481-494, Oct. 1964, SE-412 96 G¨oteborg, Sweden, September 2007.

A Proof of Brzozowski Derivatives

Statement

$$\forall e \;\; x \in e/a \iff ax \in e \tag{1}$$

by induction on e

A.A Base case

A.A.1 $\emptyset$

($\Rightarrow$)
By derivation rule $\forall a \in \Sigma \quad \emptyset/a = \emptyset$, and we know that, $\forall x \in \Sigma^* \quad x \in \emptyset \equiv False$. hence,

$$x \in \emptyset/a \Longrightarrow ax \in \emptyset \tag{2}$$

($\Leftarrow$)
$\forall y \in \Sigma^* \quad y \in \emptyset \equiv False$, thus $ax \in \emptyset \equiv False$. so we get

$$ax \in \emptyset \Longrightarrow x \in \emptyset/a \tag{3}$$

by (2) and (3) we get

$$x \in \emptyset/a \Longleftrightarrow ax \in \emptyset \tag{4}$$

A.A.2 ε

($\Rightarrow$)
By derivation rule $\forall a \in \Sigma \quad \varepsilon/a = \emptyset$, and we know that, $\forall x \in \Sigma^* \quad x \in \emptyset \equiv False$. hence,

$$x \in \varepsilon/a \Longrightarrow ax \in \varepsilon \tag{5}$$

($\Leftarrow$)
since $|ax| > 0$ we know that $ax \in \varepsilon \equiv False$, so we get

$$ax \in \varepsilon \Longrightarrow x \in \varepsilon/a \tag{6}$$

by (5) and (6) we get

$$x \in \varepsilon/a \Longleftrightarrow ax \in \varepsilon \tag{7}$$

A.A.3 b

($\Rightarrow$)
if $b = a$ By derivation rule $b/a = \varepsilon$. if $x \in \varepsilon$ then $x = \epsilon$, hence, $ax = a$, and by definition $a \in a$.
If $b \neq a$, in this case by derivation rule $b/a = \emptyset$. $x \in \emptyset \equiv False$ hence,

$$x \in b/a \Longrightarrow ax \in b \tag{8}$$

($\Leftarrow$)
since $|a| = 1$ we know that if $ax \in b$ then $x = \epsilon \wedge a = b$, hence, $b/a = \varepsilon$. hence, $x \in b/a$ we get

$$ax \in b \Longrightarrow x \in b/a \tag{9}$$

by (8) and (9) we get

$$x \in b/a \Longleftrightarrow ax \in b \tag{10}$$

A.B Step

We assum that for all Regex e_i of size less than k the statement (1). Using this assumption we prove the statement correct for a composite Regex.

A.B.1 $e_1 + e_2$

Given that the statement is valid for some e_1 and e_2 we want to prove

$$x \in (e_1 + e_2)/a \iff ax \in (e_1 + e_2) \tag{11}$$

($\Rightarrow$)
By derivation law, if $x \in (e_1 + e_2)/a$ then $x \in (e_1/a + e_2/a)$. and by definition of union of Regex. we conclude that $x \in e_1/a \vee x \in e_2/a$.
By induction hypothesis we get $ax \in e_1 \vee ax \in e_2$ and by definition of union this is equivalent to $ax \in (e_1 + e_2)$.

$$x \in (e_1 + e_2)/a \implies ax \in (e_1 + e_2) \tag{12}$$

($\Leftarrow$)
if $ax \in (e_1 + e_2)$ then by definition of union of Regex $ax \in e_1 \vee ax \in e_2$. By induction hypothesis we get $x \in e_1/a \vee x \in e_2/a$ thus $x \in (e_1/a + e_2/a)$.
by derivation definition $x \in (e_1 + e_2)/a$.

$$ax \in (e_1 + e_2) \implies x \in (e_1 + e_2)/a \tag{13}$$

by (12) and (13) we get

$$ax \in (e_1 + e_2) \iff x \in (e_1 + e_2)/a \tag{14}$$

A.B.2 e_1e_2

Assuming the induction hypothesis for e_1 and e_2 we will prove

$$x \in (e_1e_2)/a \iff ax \in e_1e_2 \tag{15}$$

Assume $\epsilon \notin e_1$ Lemma:

$$\forall x \;\; x \in e_1e_2 \iff \exists y, z \;\; x = yz \;\; \wedge y \in e_1 \;\; \wedge \;\; z \in e_2 \tag{16}$$

And by definition of concatenation $a(yz) = (ay)z$
($\Rightarrow$)
let $x = yz$

$x \in (e_1e_2)/a$	
$x \in (e_1/a)e_2$	(by definition of derviation)
$y \in e_1/a \wedge z \in e_2$	(by lemma (16))
$ay \in (e_1) \wedge z \in e_2$	(by induction hypothesis)
$(ay)z \in e_1e_2$	
$a(yz) \in (e_1e_2)$	(by definition of concatenation)
$ax \in (e_1e_2)$	(17)

($\Leftarrow$)
let $x = yz$

$ax \in (e_1 e_2)$
$ay \in e_1 \ \wedge \ z \in e_2$ (by Lemma (16) and since $\epsilon \notin e_1$)
$y \in e_1/a \wedge z \in e_2$ (by induction hypothesis)
$yz \in (e_1/a)e_2$
$yz \in (e_1 e_2)/a$ (by derviation rule for concatenation)
$x \in (e_1 e_2)/a$ (18)

by (17) and (18) we get

$$x \in (e_1 e_2)/a \Longleftrightarrow ax \in e_1 e_2 \quad \text{such that } \epsilon \notin e_1 \tag{19}$$

Assume $\epsilon \in e_1$ ($\Rightarrow$)

$x \in (e_1 e_2)/a$ (20)
$x \in (e_1/a)e_2 + e_2/a$ (by def of derivation)
$x \in (e_1/a)e_2 \vee x \in e_2/a$
by lemma (16) we get
$(\exists y, z \ \ x = yz \wedge y \in (e_1/a) \wedge z \in e_2) \vee x \in e_2/a$ (21)
by induction hypothesis we get
$(\exists y, z \ \ x = yz \wedge ay \in e_1 \wedge z \in e_2) \vee ax \in e_2$ (22)
by concatenation definition
$(\exists y, z \ \ x = yz \wedge ayz \in e_1 e_2) \vee ax \in e_2$ (23)
since $ax = ayz$
$(ax \in e_1 e_2) \vee ax \in e_2$ (24)
$ax \in e_1 e_2 + e_2$ (def of union)
$ax \in (e_1 + \epsilon)e_2$
but, $\epsilon \in e_1 \rightarrow (e_1 + \epsilon = e_1)$
$ax \in e_1 e_2$ (25)

($\Leftarrow$)

$ax \in e_1 e_2$

by lemma (16)

$\exists y, z \;\; ax = yz \wedge y \in e_1 \wedge z \in e_2$ (26)

LEM: $y = \epsilon \vee y \neq \epsilon$ (27)

case: $y = \epsilon$

$ax = z$, hence, $ax \in e_2$ (28)

by induction hypothesis $x \in e_2/a$ (29)

by $\vee$ introduction, $x \in e_2/a \vee x \in (e_1/a)e_2$ (30)

$x \in (e_1/a)e_2 + e_2/a$ (by definition of union)

case: $y \neq \epsilon$

since, $ax = yz$ hence, $y = ay'$ for some $y' \in \Sigma^*$ and $x = y'z$

we have, $ay' \in e_1 \wedge z \in e_2$ (31)

by induction hypothesis $y' \in (e_1/a) \wedge z \in e_2$ (32)

by concatenation $x = y'z \in (e_1/a)e_2$ (33)

by $\vee$ introduction $x \in (e_1/a)e_2 + e_2/a$ (34)

from (30) and (34) by $\vee$ elimination of (27) we get

$x \in (e_1/a)e_2 + e_2/a$ (35)

and that is exactly the defnition of the derivative when $\epsilon \in e_1$

$x \in e_1 e_2/a$ (36)

by (25) and (36) we get

$x \in (e_1 e_2)/a \iff ax \in e_1 e_2 \quad$ such that $\epsilon \in e_1$ (37)

by (19) and (37) (38)

$x \in (e_1 e_2)/a \iff ax \in e_1 e_2$ (39)

A.B.3 e^*

Assuming the induction hypothesis is valid for e we prove

$$x \in (e^*)/a \iff ax \in (e^*) \tag{40}$$

($\Rightarrow$)

$x \in e^*/a$ (41)

by dervation def for $*$

$x \in (e/a)e^*$ (42)

if $\epsilon \notin e$, by derivation rule for concatenation

$ax \in ee^*$ (43)

$ax \in \epsilon + ee^*$ (44)

$ax \in e^*$ (45)

if $\epsilon \in e$, from (42)

$x \in (e/a)e^* + e^*/a$ (46)

by derivation rule for concatenation

$ax \in ee^*$ (47)

$ax \in \epsilon + ee^*$ (48)

$ax \in e^*$ (49)

hence we get : $x \in (e^*)/a \Longrightarrow ax \in (e^*)$ (50)

($\Leftarrow$)

$ax \in e^*$ (51)

$ax \in (\epsilon + ee^*)$ (52)

since, $|ax| > 0$

$ax \in ee^*$ (53)

from (53), if $\epsilon \notin e$ by derivation of concatenation

$x \in (e/a)e^*$ (54)

but $(e/a)e^*$ is the definition of derivation for e^*

$x \in e^*/a$ (55)

from (53), if $\epsilon \in e$, by derivation of concatenation

$x \in (e/a)e^* + (e^*/a)$ (56)

but $(e/a)e^*$ is the definition of derivation for e^*

$x \in (e^*/a) + (e^*/a)$ (57)

$x \in e^*/a$ (58)

hence we get : $ax \in (e^*) \Longrightarrow x \in (e^*)/a$ (59)

by (50) and (59) we get

$$x \in (e^*)/a \Longleftrightarrow ax \in (e^*)$$