15-414: Bug Catching: Automated Program Verification

Lecture Notes on Modeling Semantics in Why3

Carson Piehl, Aarya Saraf, Nicole Zhang*

Carnegie Mellon University
Lecture 6
September 11, 2025

1 Introduction

On Tuesday we began our journey towards understanding the theory of program correctness and its verification. We introduced a small imperative language, and specified its semantics in terms of how the initial state of a program relates to its final state(s). To formalize the semantics, we essentially introduced a set of axioms that encode our understanding of the effects that a program will have on its state as it executes.

To gain more familiarity with these ideas, and relate them to what we have learned about specification in Why3, in this lecture we will formalize the semantics of regular expressions in a similar way. We will them implement and verify a regular expression matcher using Brzozowski derivatives [Brz64]. It is an verification exercise for relatively complex program, and we will also practice testing our axioms. Just like testing code, this is generally good practice because incorrect specifications may render verification meaningless. There is another purpose to practice regular expressions: they form a structure called a *Kleene algebra* [Kle56], a structure that is echoed in our language of programs. We will make that connection more precise in a future lecture.

Learning goals. After this lecture, you should be able to:

- Specify the semantics of regular expressions using axioms
- Formalize these semantics in Why3

*Based on notes written by Matt Fredrikson and Frank Pfenning

- Test the specification of these semantcs
- Implement and verify an algorithm for matching regular expressions

2 Specifying the Meaning of Regular Expressions

For simplicity, we use integers to represent the basic type of characters. A *word* is just a list of characters.

```
module RegExp
  use int.Int
  use list.List
  use list.Append

  type char = int
  type word = list char
  ...
end
```

Regular expressions r over characters a are usually defined in the following BNF notation

$$r \quad ::= \quad a \mid 1 \mid r_1 \cdot r_2 \mid 0 \mid r_1 + r_2 \mid r^*$$

In WhyML, the following type definition precisely expresses this grammar.

```
type regexp = Char char              (* single character *)
            | One                    (* empty string *)
            | Times regexp regexp    (* concatenation *)
            | Zero                   (* empty set *)
            | Plus regexp regexp     (* union *)
            | Star regexp            (* repetition *)
```

2.1 The Language Generated by a Regular Expression

A regular expression defines a *language*, $\mathcal{L}(r)$ which is a set of words over the alphabet of characters. Rather than explicitly using sets, we define a predicate

```
predicate mem (w : word) (r : regexp)
```

such that mem w r is true iff $w \in \mathcal{L}(r)$. The key step is now to translate the mathematical definition of $\mathcal{L}(r)$ into *axioms* describing the properties of mem.

Characters. Mathematically, we define $\mathcal{L}(a) = \{a\}$. Axiomatically, it would be correct but too weak to simply state

```
axiom mem_char : forall a. mem (Cons a Nil) (Char a)    (* too weak! *)
```

It only expresses that $a \in \mathcal{L}(a)$, or, in other words, $\{a\} \subseteq \mathcal{L}(a)$. To express the equality we should state

```
axiom mem_char : forall w a. mem w (Char a) <-> w = Cons a Nil
```

Empty word. We define $\mathcal{L}(1) = \{\varepsilon\}$, where ε represents the empty word. As an axiom:

```
axiom mem_one : forall w. mem w One <-> w = Nil
```

Concatenation. We define $\mathcal{L}(r_1 \cdot r_2) = \{w_1\ w_2 \mid w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2)\}$. To obtain a suitable axiom we need to say that a word $w \in \mathcal{L}(r_1 \cdot r_2)$ iff w can be decomposed into $w_1\ w_2$ such that $w_1 \in \mathcal{L}(r_1)$ and $w_2 \in \mathcal{L}(r_2)$. This requires an existential quantifier.

```
axiom mem_times : forall w r1 r2.
  mem w (Times r1 r2)
  <-> exists w1 w2. w = w1 ++ w2 /\ mem w1 r1 /\ mem w2 r2
```

Here we use list concatenation ++ from the `list.Append` module.

Empty set. We define $\mathcal{L}(0) = \{\,\}$. For consistent style we define

```
axiom mem_zero : forall w. mem w Zero <-> false
```

but we could have said equivalently $\forall w.$not (mem w Zero)

Union. We define $\mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$. In axiomatic form:

```
axiom mem_plus : forall w r1 r2.
  mem w (Plus r1 r2) <-> mem w r1 \/ mem w r2
```

Repetition. We can defined inductively that $\mathcal{L}(r^*) = \mathcal{L}(\varepsilon + r \cdot r^*)$. Expanding it, we would get

```
  axiom mem_star0 : forall w r.
  mem w (Star r)
  <-> w = Nil \/ exists w1 w2. w = w1 ++ w2
                            /\ mem w1 r /\ mem w2 (Star r)
```

A difficulty here appears to be the fact that if $w_1 = \varepsilon$ then $w_2 = w$ and the question if $w \in r^*$ comes again down to $w \in r^*$. While there is nothing wrong with that in an inductive definition, the automated provers supporting Why3 seem to have some problems of using it effectively. But we can observe that there is really no point of using this property when $w_1 = \varepsilon$ since it does not add to the set $\mathcal{L}(r^*)$. So we can restrict the axiom to non-empty words matching r without affecting $\mathcal{L}(r^*)$.

```
  axiom mem_star1 : forall w r.
  mem w (Star r)
  <-> w = Nil \/ exists a w1 w2. w = Cons a w1 ++ w2
                    /\ mem (Cons a w1) r /\ mem w2 (Star r)
```

This axiom has the helpful property that when the regular expression r^* recurs on the right-hand side, the string w_2 is shorter than w on the left-hand side. So progress is being made in more than one way: when we read the clauses of the definition of mem w r (expressed via our axioms) from left to right, either the regular expression becomes smaller, or the regular expression stays the same but then the word becomes shorter.

Alternatively, we could have used the solution from while loops and specified that $w \in r^*$ iff there exists an $n \geq 0$ such that $w \in r^n$ and then define the n-fold iteration r^r by induction on n. We used the first solution in part to demonstrate a different way to make the inductive nature of definitions explicit.

3 Regular Expression Matching

The goal of this section will be to implement a verified matcher for regular expressions. We use the very elegant algorithm using Brzozowski derivatives [Brz64] which has more recently been reexamined from the practical perspective by Owens, Reppy, and Turon [ORT09]. We do not consider the translation to finite-state automata or the efficiency improvements by Owens et al., just the basic algorithm.

Besides the intrinsic elegance of the algorithm, the main purpose of this exercise is to exemplify effective logical specification for relatively complex types such as regular expressions.

3.1 Testing the Specification

Before we get into the algorithm, let's test the specification of the meaning of regular expressions as sets of words over a given alphabet. For simplicity, we represented the alphabet as just integers, and words as list of integers. We only give two sample axioms here (see regexp.mlw for the complete list.

```
type char = int
type word = list int

type regexp = Char char
            | One
            | Times regexp regexp
            | Zero
            | Plus regexp regexp
            | Star regexp

predicate mem (w:word) (r:regexp)

axiom mem_char : forall w a.
  mem w (Char a) <-> w = Cons a Nil

axiom mem_plus : forall w r1 r2.
  mem w (Plus r1 r2) <-> mem w r1 \/ mem w r2

axiom mem_star1 : forall w r.
  mem w (Star r) <-> w = Nil \/ exists a w1 w2. w = Cons a w1 ++ w2
                                /\ mem (Cons a w1) r /\ mem w2 (Star r
                                    )

(* ...more axioms.. *)
```

A first way to test is if Why3 can *prove* that $w \in \mathcal{L}(r)$ for some specific w and r. Such proofs are generally difficult, since the logical specification doesn't imply an particular algorithm for regular expression matching, so we want to pick small examples. Here is one that the system can prove:

```
goal test1 : mem (Cons 0 (Cons 1 Nil)) (Star (Plus (Char 0) (Char 1)))
```

It was reassuring that when we made an error and forget the Star, the proof attempt failed as it should.

The keyword goal explicitly introduces a formula that Why3 has to prove. The fact that it has been proved is not exploited subsequently. That may be important because too many random facts about mem may pollute the search space in the verification of the functions we care about. When we need to introduce explicit lemmas because the provers cannot verify something directly, we use instead lemma $name : P$ which proves P and then assumes it for the remainder of the verification.

As mentioned in the introduction, regular expressions form a *Kleene algebra* that satisfies a number of laws. Here are three simple examples:

$0 + r$	$=$	r	0 is the unit of $+$
$r_1 \cdot (r_2 \cdot r_3)$	$=$	$(r_1 \cdot r_2) \cdot r_3$	concatenation is associative
$(r^*)^*$	$=$	r^*	iteration is idempotent

These are justified by equations between the sets denoted by the regular expressions on both sides. We can ask Why3 to prove them as a way of testing the axioms.

```
goal plus_zero : forall w r.
  mem w (Plus Zero r) <-> mem w r

goal times_assoc : forall w r1 r2 r3.
  mem w (Times r1 (Times r2 r3)) <-> mem w (Times (Times r1 r2) r3)

(* fails to prove, even though true *)
(*
goal star_star : forall w r.
  mem w (Star (Star r)) <-> mem w (Star r)
*)
```

It turns out that Why3 can prove the first two, but not the last. That's not necessarily a black mark: we would need to investigate further what the proof actually looks like, and whether we can guide Why3 to find it with appropriate lemmas.

3.2 Matching Regular Expressionsx with Derivatives

One basic problem for designing regular expression matcher is the definition of concatenation:

$$w \in \mathcal{L}(r_1 \cdot r_2) \quad \text{iff} \quad w = w_1\, w_2 \quad \text{with} \quad w_1 \in \mathcal{L}(r_1) \quad \text{and} \quad w_2 \in \mathcal{L}(r_2)$$

The question here is how to find the split of w into two subwords. In order to avoid this kind of guess we want to go through the word letter by letter from left to right. The

main function matching a word against a regular expression would be based on two auxiliary functions nullable r and $\partial_a r$ and the following definitions:

$$\begin{array}{lll} \varepsilon \in \mathcal{L}(r) & \text{iff} & \mathsf{nullable}\, r \\ a\, w \in \mathcal{L}(r) & \text{iff} & w \in \mathcal{L}(\partial_a r) \end{array}$$

If we can devise function nullable r and $\partial_a r$ then top-level matching function is easy to define since we terminate in the clause for the empty word ε and the word becomes smaller reading the second clause from left to right.

3.3 Writing the Matcher

Let's recall the key definitions and for now just *specify* the matcher and the auxiliary functions it uses.

```
let rec nullable (r:regexp) : bool =
ensures { result <-> mem Nil r }
...

let rec deriv (a:char) (r:regexp) : regexp =
ensures { forall w. mem (Cons a w) r <-> mem w result }
...

let rec re_match (w:word) (r:regexp) : bool =
ensures { mem w r <-> result }
...
```

Before writing deriv and nullable we can actually write an verify re_match, which should reassure us our general approach will eventually succeed.

```
let rec re_match (w:word) (r:regexp) : bool =
variant { w }
ensures { mem w r <-> result }
match w with
| Nil -> nullable r
| Cons a w' -> re_match w' (deriv a r)
end
```

For this verification we need a new form of the `variant` contract. It takes here not an integer quantity but a value of recursive type, namely w : word where word $=$ list int. Such a variant declaration for a function has to verify that all recursive calls will be on structurally smaller expressions of the given type. In the case of lists, it could be the tail, the tail of the tail, etc. This function can be verified since w' is the tail of w.

3.4 Deciding Nullability

We specified nullable with

$$\varepsilon \in \mathcal{L}(r) \quad \text{iff} \quad \mathsf{nullable}\, r$$

From this, its relatively straightforward to synthesize the defining equations for nullable r, depending on the regular expression r. A single character a or the empty set 0 obviously

do not generate the empty word. On the other hand, 1 and r^* do, by their definition. A concatenation $r_1 \cdot r_2$ generates the empty word if both r_1 and r_2 do, and a union $r_1 + r_2$ if either r_1 or r_2 do. This gives us the following definition, which clearly terminates because r decreases in each recursive call.

```
let rec nullable (r:regexp) : bool =
variant { r }
ensures { result <-> mem Nil r }
match r with
| Char _a     -> false
| One         -> true
| Times r1 r2 -> nullable r1 && nullable r2
| Zero        -> false
| Plus r1 r2  -> nullable r1 || nullable r2
| Star _r     -> true
end
```

And, indeed, this function is easily verified against the axioms for `mem`. We use an underscore '_' at the beginning of a variable that does not occur in its scope in order to prevent a spurious warning from the compiler.

3.5 Computing the Brzozowski Derivative

We specified the Brzozowski derivate of a regular expression r with respect to a character a, written as $\partial_a r$, with

$$a\,w \in \mathcal{L}(r) \quad \text{iff} \quad w \in \mathcal{L}(\partial_a r)$$

Remarkably, such a derivative exists: if a language is regular (that is, is generated by a regular expression), then the language of postfixes of any character a is again regular. Moreover, we can effectively compute $\partial_a r$.

As for `nullable`, we want to analyze the structure of the regular expression and see if we can find a way to compute the derivative. We start by defining the derivative in mathematical notation.

$$\begin{array}{lcll} \partial_a a & = & 1 & \\ \partial_a b & = & 0 & \text{for } a \neq b \\ \partial_a 1 & = & 0 & \\ \partial_a (r_1 \cdot r_2) & = & (\partial_a r_1) \cdot r_2 & \text{if not } \mathsf{nullable}(r_1) \end{array}$$

The last line is the most interesting. If r_1 does not generate the empty string, then the character a must be matched by r_1. The rest of the word is then matched by $\partial_a r_1$ followed by r_2. But what if r_1 is nullable? Then it is also possible that a is at the beginning of the word generated by r_2. So we continue:

$$\begin{array}{lcll} \partial_a (r_1 \cdot r_2) & = & (\partial_a r_1) \cdot r_2 + \partial_a r_2 & \text{if } \mathsf{nullable}(r_1) \\ \partial_a 0 & = & 0 & \\ \partial_a (r_1 + r_2) & = & (\partial_a r_1) + (\partial_a r_2) & \\ \partial_a (r^*) & = & (\partial_a r) \cdot r^* & \end{array}$$

The last line just says that for $a\,w \in \mathcal{L}(r^*)$ the first a has to be matched by a copy of r.

We now observe that in each case any appeal to ∂_a on the right-hand side is on a smaller regular expression. Translating this into WhyML is routine.

```
let rec deriv (a:char) (r:regexp) : regexp =
ensures { forall w. mem (Cons a w) r <-> mem w result }
variant { r }
match r with
| Char b      -> if a = b then One else Zero
| One         -> Zero
| Times r1 r2 -> if nullable r1
                 then Plus (Times (deriv a r1) r2) (deriv a r2)
                 else Times (deriv a r1) r2
| Zero        -> Zero
| Plus r1 r2  -> Plus (deriv a r1) (deriv a r2)
| Star r      -> Times (deriv a r) (Star r)
end
```

The complete live-code file with the verified regular expression matcher can be found in regexp.mlw.

References

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, volume 34 of *Annals of Mathematical Studies*, pages 3–42. De Gruyter, 1956.

[ORT09] Scott Owens, John H. Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.