15-414: Bug Catching: Automated Program Verification

Lecture Notes on Dynamic Logic

Frank Pfenning

Carnegie Mellon University
Lecture 6
February 3, 2022

1 Introduction

We start the lecture by writing the regular expression matcher we specified in the last lecture 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.

Then we continue on our path towards formalization of various necessary components for program reasoning: a language of arithmetic expressions e, a language of formulas P, and a language is simple while programs α. We gave a mathematical semantics of each relative to an assignment ω of integers to variables. The meaning of an expression $\omega[\![e]\!] \in \mathbb{Z}$ is an integer, a formula can be true $\omega \models P$ (or false $\omega \not\models P$), and the meaning of a program is a relation $\omega[\![\alpha]\!]\nu$ between prestates ω and poststates ν. At this point we can reason about programs *semantically*, that is, mathematically reason about the meaning of programs.

None of this gives us a *logic* for reasoning about programs: all reasoning is reduced back to general mathematics. It is therefore difficult to mechanize and automate since general mathematics is difficult to mechanize and automate.

The goal of today's lecture is to develop a *logic* for reasoning about programs called *dynamic logic*. In this lecture we take yet again a *semantic* approach, that is, we specify when a formula that expresses properties of programs is true. But we also write *axioms* to capture in syntax how we can reason in dynamic logic.

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

- Test axioms that are intended as specifications
- Implement and verify an algorithm for matching regular expressions
- Interpret the meaning of formulas in dynamic logic (DL)
- Determine if simple formulas are true in a given state
- Determine if simple formulas are valid
- Validate DL axioms against the semantics of DL programs
- Design semantics and axioms for simple language extensions

2 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.

2.1 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

```

```
16 axiom mem_plus : forall w r1 r2.
17   mem w (Plus r1 r2) <-> mem w r1 \/ mem w r2
18
19 axiom mem_star1 : forall w r.
20   mem w (Star r) <-> w = Nil \/ exists a w1 w2. w = Cons a w1 ++ w2
21                                 /\ mem (Cons a w1) r /\ mem w2 (Star r
                                       )
22
23 (* ...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:

```
1 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.

```
1 goal plus_zero : forall w r.
2   mem w (Plus Zero r) <-> mem w r
3
4 goal times_assoc : forall w r1 r2 r3.
5   mem w (Times r1 (Times r2 r3)) <-> mem w (Times (Times r1 r2) r3)
6
7 (* fails to prove, even though true *)
8 (*
9 goal star_star : forall w r.
10   mem w (Star (Star r)) <-> mem w (Star r)
11 *)
```

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.

2.2 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 $\mathsf{nullable}\, r$ 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 $\mathsf{nullable}\, r$ 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.

2.3 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.

2.4 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 $\mathsf{nullable}\, r$, 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.

```
1  let rec nullable (r:regexp) : bool =
2  variant { r }
3  ensures { result <-> mem Nil r }
4  match r with
5  | Char _a     -> false
6  | One         -> true
7  | Times r1 r2 -> nullable r1 && nullable r2
8  | Zero        -> false
9  | Plus r1 r2  -> nullable r1 || nullable r2
10 | Star _r     -> true
11 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.

2.5 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:

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

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.

3 The Key Idea: Boxes and Diamonds

Now we proceed to developing dynamic logic (DL), a logic to reason about programs in our little while-loop language. The key idea of dynamic logic is to add two new kinds of formulas.

$$\text{Formulas} \quad P, Q \quad ::= \quad \ldots \mid [\alpha]P \mid \langle\alpha\rangle P$$

It is surprisingly easy to define the meaning of these new formulas. $[\alpha]P$ means that in *every* poststate reachable by α, the formula P is true. And $\langle\alpha\rangle P$ means that in *some* poststate reachable by α, the formula P is true.

For those familiar with *modal logic*: this harkens back to the meaning of $\Box P$ (P is true in every reachable world) and $\Diamond P$ (P is true in some reachable world). The difference here is that the reachable worlds are concretely determined by programs α and not just by a fixed reachability relation. Because of this strong analogy, we may pronounce $[\alpha]P$ as "*box* α P" and $\langle\alpha\rangle P$ as "*diamond* α P".

We define the meaning of the new constructs rigorously as a relation:

$$\begin{array}{ll} \omega \models [\alpha]P & \text{iff for every } \nu, \omega[\![\alpha]\!]\nu \text{ implies } \nu \models P \\ \omega \models \langle\alpha\rangle P & \text{iff there exists a } \nu \text{ such that } \omega[\![\alpha]\!]\nu \text{ and } \nu \models P \end{array}$$

Let's discover whether certain simple formulas are true or not. We may want to recall the definition of the meaning of programs $[\![\alpha]\!]$ in order to apply it to the following questions.

$$\omega \models [\mathsf{while}\,\mathsf{true}\,\alpha]P$$
$$\omega \models \langle \mathsf{while}\,\mathsf{true}\,\alpha\rangle P$$
$$\omega \models [?\mathsf{true}]P$$
$$\omega \models \langle ?\mathsf{true}\rangle P$$
$$\omega \models [\alpha]\mathsf{false}$$
$$\omega \models \langle\alpha\rangle\mathsf{false}$$

We suggest you try before moving on to the next page.

$\omega \models [\mathsf{while}\,\mathsf{true}\,\alpha]P$	always
$\omega \models \langle \mathsf{while}\,\mathsf{true}\,\alpha\rangle P$	never
$\omega \models [?\mathsf{true}]P$	iff $\omega \models P$
$\omega \models \langle ?\mathsf{true}\rangle P$	iff $\omega \models P$
$\omega \models [\alpha]\mathsf{false}$	iff α does not terminate (has no poststate)
$\omega \models \langle\alpha\rangle\mathsf{false}$	never

Even though truth depends on a state ω, there are many formulas whose truth does not depend on ω at all. In the example, the truth of the first, second, and last are independent of the ω. Formulas that are true in any state are called *valid*. In the examples above, only the first one is valid. We write

$$\models P$$

to express that P is valid. Here are some other examples:

$\omega[x \mapsto 3] \models [x \leftarrow x+1](x = 4)$	true
$\models [x \leftarrow 4](x = 4)$	valid
$\models x = 3 \rightarrow [x \leftarrow x+1](x = 4)$	valid

4 Determinism

We call a program *deterministic* if in any prestate it has at most one poststate. We call a *language* deterministic if every program in it is deterministic. The DL programming language we have shown so far is *deterministic* in this sense. One could prove this rigorously by induction over the structure of the program.

Expressed more mathematically, we say α is *deterministic* if for every ω, $\omega[\![\alpha]\!]\nu$ and $\omega[\![\alpha]\!]\nu'$ imply $\nu = \nu'$. A language is deterministic if every program in the language is deterministic.

Deterministic languages satisfy certain properties that are not true for languages in general. Here is one:

For a deterministic language, $\models \langle\alpha\rangle P \rightarrow [\alpha]P$ *for any program* α *and formula* P*.*

We have used here the notation $\models Q$ (omitting the state ω) to express that Q is valid. This property can easily be proved by appeal to the meaning of formulas and the definition of determinism.

In a deterministic language we say that $\langle\alpha\rangle P$ establishes *total correctness* (the program α has to satisfy terminate and satisfy the postcondition P), while $[\alpha]P$ establishes *partial correctness* (if α has to satisfy P, but only if it terminates).

In Why3, we can establish *total correctness* by specifying *variant* contracts that ensure termination and *partial correctness* by using the *diverges* contracts to allow nontermination.

In a deterministic language, to establish total correctness $\langle\alpha\rangle P$ we can first prove partial correctness $[\alpha]P$ and then separately prove termination.

In the remainder of today's lecture we focus on *partial correctness* and therefore the $[\alpha]P$ modality.

5 Axioms

Now that we have a logic with a suitable semantics our next task is to develop some tools for reasoning *within* the logic. Let's think back to how we reasoned about regular expressions in Lecture 5. For each form of regular expression we wrote down an *axiom* specifying its meaning in a way the theorem provers could use. A critical idea in that case study was to make sure we break down the question if $w \in \mathcal{L}(r)$ to some $w' \in \mathcal{L}(r')$ where r' is a subexpression of r. This allows the theorem prover to break down questions about complicated regular expressions into simpler ones.

We'll follow the same strategy here: write down axioms for $[\alpha]P$ that help us break down the structure of the program α by logical reasoning without explicitly appealing to the semantics any more. Of course, the axioms themselves must be justified in terms of the underlying semantics: we don't want to conclude something that is not true!

Because they are *axioms* we need them to be *valid*, not just true in some particular state or even classes of states. We now go through the language constructs one by one, devising axioms.

5.1 Sequential Composition

Which axiom might describe $[\alpha\,;\,\beta]P$ in terms of $[\alpha]\_$ and $[\beta]\_$? Recall the meaning:

$$\omega[\![\alpha\,;\,\beta]\!]\nu \quad \text{iff} \quad \text{there exists } \mu \text{ such that } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\alpha]\!]\nu$$

Intuitively, P is true after α and β if $[\beta]P$ is true after α. So we propose the axiom

$$[\alpha\,;\,\beta]P \leftrightarrow [\alpha]([\beta]P)$$

In order to prove that this axiom is valid we can decompose it into two implications. We prove the first one of these.

We want to show that $\omega \models [\alpha\,;\,\beta]P \rightarrow [\alpha]([\beta]P)$
Assume $\omega \models [\alpha\,;\,\beta]P$ (1)
and show $\omega \models [\alpha]([\beta]P)$
By definition, this holds if for every μ, $\omega[\![\alpha]\!]\mu$ implies $\mu \models [\beta]P$
So assume $\omega[\![\alpha]\!]\mu$ for an arbitrary μ (2)
It remains to show that $\mu \models [\beta]P$
By definition, this holds if for every ν, $\mu[\![\beta]\!]\nu$ implies $\nu \models P$.
So assume $\mu[\![\beta]\!]\nu$ for an arbitrary ν (3)
It remains to show that $\nu \models P$ (*)
From (2) and (3) we conclude $\omega[\![\alpha\,;\,\beta]\!]\nu$ by definition of $[\![\alpha\,;\,\beta]\!]$ (4)
From (1) and (4) we conclude $\nu \models P$ by definition of $\omega \models [\alpha\,;\,\beta]P$
This conclusion is exactly what we needed to show (*)

The other direction works similarly, essentially just unfolding definitions and some shallow logical reasoning.

5.2 Tests

Recall that $\omega[\![?P]\!]\nu$ iff $\omega = \nu$ if $\omega \models P$.

By definition, then, $\omega \models [?P]Q$ iff for all ν with $\omega[\![?P]\!]\nu$ we have $\nu \models Q$. This requires that Q is true in ω if P is, and imposes no requirement if P is false. Therefore, the right axiom is

$$[?P]Q \leftrightarrow (P \rightarrow Q)$$

In the case of tests, let's also consider $\langle ?P\rangle Q$. Recall that $\omega \models \langle ?P\rangle Q$ iff there is exists a ν with $\omega[\![?P]\!]\nu$ such that $\nu \models Q$. But that can only be the case if both P and Q are true in ω. So:

$$\langle ?P\rangle Q \leftrightarrow (P \wedge Q)$$

5.3 Conditionals

Conditionals are straightforward given the intuition we have built up so far. $[\text{if } P\, \alpha\, \beta]Q$ should be true if P is true and $[\alpha]Q$, or P is false and $[\beta]Q$.

$$[\text{if } P\, \alpha\, \beta]Q \leftrightarrow (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q)$$

Assignments and while loops are trickier, so we postpone introducing axioms for them to the next lecture.

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.