15-414: Bug Catching: Automated Program Verification

Lecture Notes on Semantics

Frank Pfenning

Carnegie Mellon University

Lecture 5
February 1, 2022

1 Introduction

This lecture begins Part II: *From Informal to Formal Reasoning*. We need to formalize the various objects of study in the course so far: (1) a programming language, (2) a logic for reasoning about programs, and (3) a proof that the reasoning is correct in the sense that it is consistent with the meaning of programs. Once these aspects have been nailed down, formally, we have a basis for implementing them. We also understand them more thoroughly which means we will be able to use verification tools more effectively and even extend them to cover new computational phenomena.

There are many choices and tradeoffs for such a study, such as the extent of the features in the programming language, the expressive power of the logic, and the pragmatics of using the language and its logic for verification. A natural first idea would be to use WhyML, but it is too complex for us to study in the kind detail we wish to in this course. We conclude that the language should be small, but have the essential features that help us understand WhyML. Within that context, we should also decide whether to focus on functional or imperative aspects of WhyML. We have the opportunity to study functional programming and type theory in a number of other courses in the curriculum including 15-312 *Foundations of Programming Languages* and 15-417 *Constructive Logic*. In this course were therefore tak a different perspective and study a *small imperative programming language*.

Even in the context of reasoning about imperative programs there are different traditions and approaches. Historically, there is *Hoare logic* that studies tripes $P\{\alpha\}Q$ consisting of a precondition P, a program α, and a postcondition Q. This has been generalized to handle heap-allocated objects in *separation logic* and shared-memory concurrency in *concurrent separation logic*. We follow a different trajectory in choosing *dynamic logic* inspired by traditional *modal logic*. It has also been generalized in multiple ways, including *differential dynamic logic* which supports reasoning about hybrid discrete and continuous evolving systems. Differential dynamic logic is at the core of 15-424 *Logical Foundations of Cyberphysical Systems*.

In outline, we will introduce a small imperative language and a language of formulas and then define the meaning ("*semantics*") of both programs and formulas. This will answer the questions *How do programs execute?* and *When are formulas true?* We will ignore aspects of *concrete syntax* and work with *abstract syntax*, being unconcerned with how to parse or type-check programs. You can learn more about those aspects of programming languages in 15-312 (mentioned above) and 15-411 *Compiler Design*.

Toward the end of the lecture we go through an exercise of specifying the meaning of regular expressions, using Why3 as a tool. At the beginning of the next lecture we will actually implement and verify a regular expression matcher.

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

- Simulate the dynamics of simple while programs
- Determine if programs are semantically equivalent
- Define the meaning of imperative language constructs
- Reason semantically about arithmetic formulas
- Specify semantics relationally

2 Straight-Line Programs

We now present our small imperative programming language in stages. The development is inherently open-ended in the sense that we will introduce more constructs as our study goes on.

For the sake of simplicity we assume that all variables range of the integers $\mathbb{Z}$. We have a simple language of *arithmetic expressions* e, with the usual conventions that we do not detail here. We use a, b, c for integer constants and x to stand for variables.

$$\text{Arithmetic Expressions} \quad e \quad ::= \quad c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \ldots$$

Since expressions contains variables, their meaning is determined with respect to a *state* that assigns integers to variables. We use ω, μ, ν to range over states and assume that they are defined on all variables. We write $\omega(x) = c$ if ω maps x to c. Then the value of expression e in state ω is written as

$$\omega[\![e]\!] = c$$

and is easily defined based on the structure of e. For example:

$$\begin{array}{lcl} \omega[\![c]\!] & = & c \\ \omega[\![x]\!] & = & \omega(x) \\ \omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!] \\ \omega[\![e_1 - e_2]\!] & = & \omega[\![e_1]\!] - \omega[\![e_2]\!] \\ \ldots \end{array}$$

The last two equations may look somewhat odd—we have to keep in mind that '$+$' and '$-$' on the left-hand side are pieces of syntax that form expressions while '$+$' and '$-$' on the right-hand side are the mathematical operations on integers. Other operations are defined analogously.

Programs are denoted by α and β and we start here with two simple constructs: *assignment* $x \leftarrow e$ and *sequential composition* $\alpha \,;\, \beta$.

$$\text{Programs} \quad \alpha, \beta \quad ::= \quad x \leftarrow e \mid \alpha \,;\, \beta \mid \ldots$$

The meaning of a program is a *relation* between the *prestate* and *poststate* of its execution. It is a relation instead of a function because we would like to accommodate nonterminating programs (no possible poststate) and also nondeterministic programs (multiple possible poststates). We write

$$\omega[\![\alpha]\!]\nu$$

if the meaning of the program α relates prestate ω to poststate ν.

We define the meaning of assignment $x \leftarrow e$ to evaluate e in the current state to c and then update the state to map x to c. In symbols:

$$\omega[\![x \leftarrow e]\!]\nu \quad \text{iff} \quad \nu = \omega[x \mapsto c] \text{ where } c = \omega[\![e]\!]$$

Here we use the notation $\omega[x \mapsto c]$ for the result of updating the state ω by mapping x to c (no matter what it was before).

The meaning of sequential composition $\alpha \,;\, \beta$ is to execute first α and then β from the resulting state. That is:

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

In other words, the relation denoted by $\alpha \,;\, \beta$ is the *composition* of the relations denoted by α and β.

As an example, let's compute

$$(\omega[x \mapsto a])[\![x \leftarrow x + 2]\!]\nu$$

and we find $\nu = (\omega[x \mapsto a])[x \mapsto a + 2] = \omega[x \mapsto a + 2]$ Slightly more complicated is

$$(\omega[x \mapsto a])[\![x \leftarrow x + 1 \,;\, x \leftarrow x + 1]\!]\nu$$

We determine that there is an intermediate state $\mu = \omega[x \mapsto a + 1]$ and a final state $\nu = \omega[x \mapsto a + 2]$.

So, both of these programs define the same relation between $\omega[x \mapsto a]$ and $\omega[x \mapsto a+2]$ Therefore we can state that these two programs are semantically equivalent

$$[\![x \leftarrow x + 2]\!] = [\![x \leftarrow x + 1 \ ; \ x \leftarrow x + 1]\!]$$

They have the same meaning because they have the same effects on the state. This, by the way, might fail to be true if the language were extended to allow shared memory concurrency because another process can intervene after the first assignment on the right, while the left atomically increments x by two. Lesson: we always have to be careful about the extent of the language when we reason about it, be it semantically (as here) or logically (as in the next lecture).

As another example we consider this strange way to swap the values between two variables x and y without an auxiliary variable. We would like to prove:

$$(\omega[x \mapsto a, y \mapsto b])[\![x \leftarrow x + y \ ; \ y \leftarrow x - y \ ; \ x \leftarrow x - y]\!](\omega[x \mapsto b, y \mapsto a])$$

For this we have to calculate the intermediate states. Those are

$$\omega[x \mapsto a + b, y \mapsto b]$$

after the first assignment and

$$\omega[x \mapsto a + b, y \mapsto a]$$

after the second assignment, after which we reach the desired poststate.

3 Conditionals

We now add conditionals $\mathsf{if}\, P\, \alpha\, \beta$ to our language, read as *"if P then α else β"*. A characteristic of the dynamic logic approach is that formulas P do double duty: on one hand they serve as conditions in if-then-else programs and (shortly) guards on while loops. On the other hand we also use them to *reason* about programs as shown in the next lectures.

$$\begin{array}{lrcl} \text{Programs} & \alpha, \beta & ::= & x \leftarrow e \mid \alpha \ ; \beta \mid \mathsf{if}\, P\, \alpha\, \beta \mid \ldots \\ \text{Formulas} & P, Q & ::= & e_1 = e_2 \mid e_1 \leq e_2 \mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \neg P \\ & & & \mid \forall x.\, P \mid \exists x.\, P \mid \ldots \end{array}$$

In concrete syntax we usually write $\top$ (top) as "true", $\bot$ (bottom) as "false", $\neg$ as "not" and we write out the quantifiers as "forall" and "exists".

In order to define the meaning of the conditional, we first need to define the meaning of the formulas, in mathematical terms. Because variables (and therefore quantifiers) range just over integers, the language of formulas we are concerned with is that of *integer arithmetic*. We define their meaning relative to an assignment ω of values to variables

$$\omega \models P \qquad \textit{P is true in state } \omega$$

It is defined on the structure of P.

$$\begin{array}{ll}
\omega \models \top & \text{always} \\
\omega \models \bot & \text{never} \\
\omega \models e_1 = e_2 & \text{iff } \omega[\![e_1]\!] = \omega[\![e_2]\!] \\
\omega \models e_1 \leq e_2 & \text{iff } \omega[\![e_1]\!] \leq \omega[\![e_2]\!] \\
\omega \models P \wedge Q & \text{iff } \omega \models P \text{ and } \omega \models Q \\
\omega \models P \vee Q & \text{iff } \omega \models P \text{ or } \omega \models Q \\
\omega \models \neg P & \text{iff } \omega \not\models P \\
\omega \models P \to Q & \text{iff whenever } \omega \models P \text{ then also } \omega \models Q \\
\omega \models \forall x.\, P & \text{iff } \omega[x \mapsto a] \models P \text{ for all } a \in \mathbb{Z} \\
\omega \models \exists x.\, P & \text{iff } \omega[x \mapsto a] \models P \text{ for some } a \in \mathbb{Z}
\end{array}$$

Because quantified integer arithmetic is undecidable, this definition is not effective in the sense that we cannot use it directly to determine whether a give formula is true. This is a problem if we want to actually execute our programs containing conditionals. So we usually restrict the formulas that can appear in conditionals to be quantifier-free, in which case it is easy to determine whether they are true or false.

With this out of the way, we can now define the meaning of the conditional by cases on the truth of P.

$$\begin{array}{lll}
\omega[\![\mathsf{if}\, P\, \alpha\, \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\nu \text{ when } \omega \models P \\
 & & \omega[\![\beta]\!]\nu \text{ when } \omega \not\models P
\end{array}$$

4 While Loops

The abstract syntax for while loops is $\mathsf{while}\, P\, \alpha$ which should somehow be the same as $\mathsf{if}\, P\, (\alpha \; ; \; \mathsf{while}\, P\, \alpha)\, \mathsf{skip}$, where skip is a program that has no effect. Although it is perfectly possible to make this work as a so-called *inductive definition*, it has the issue that $\mathsf{while}\, P\, \alpha$ appears on both sides. So we break it down by "guessing" the number of iterations of the loop, using an auxiliary relation $[\![\mathsf{while}\, P\, \alpha]\!]^n$ indexed by an $n \geq 0$. If $n = 0$ we must exit the loop so P should be false, and if $n > 0$ we should go around the loop once, followed by $n - 1$ more iterations.

$$\begin{array}{lll}
\omega[\![\mathsf{while}\, P\, \alpha]\!]\nu & \text{iff} & \text{there exists an } n \geq 0 \text{ such that } \omega[\![\mathsf{while}\, P\, \alpha]\!]^n\nu \\
\\
\omega[\![\mathsf{while}\, P\, \alpha]\!]^0\nu & \text{iff} & \omega \not\models P \text{ and } \omega = \nu \\
\omega[\![\mathsf{while}\, P\, \alpha]\!]^{n+1}\nu & \text{iff} & \omega \models P \text{ and there exists a } \mu \text{ such that } \omega[\![\alpha]\!]\mu \text{ and } \mu[\![\mathsf{while}\, P\, \alpha]\!]^n\nu
\end{array}$$

We can appeal to this definition to compute the meaning of a few simple programs. Actually, we will look at whole families of programs because it doesn't matter what some of the components are. For example, any program $\mathsf{while}\ \mathsf{false}\ \alpha$ will behave the same, regardless of α. Instead of looking up the answer immediately, we suggest solving these yourself first with careful reference to the definitions.

$$\begin{array}{l}
\omega[\![\mathsf{while}\, \mathsf{true}\, \alpha]\!]\nu \\
\omega[\![\mathsf{while}\, \mathsf{false}\, \alpha]\!]\nu \\
\omega[\![x \leftarrow x]\!]\nu
\end{array}$$

We calculate

$$\begin{array}{ll} \omega[\![\mathsf{while\ true}\ \alpha]\!]\nu & \text{never} \\ \omega[\![\mathsf{while\ false}\ \alpha]\!]\nu & \text{iff } \nu = \omega \\ \omega[\![x \leftarrow x]\!]\nu & \text{iff } \nu = \omega \end{array}$$

We see, for example, that

$$[\![\mathsf{while\ false}\ \alpha]\!] = [\![x \leftarrow x]\!]$$

where the equality here denotes an equality between two relations. Further examples in the next section.

5 Tests

As the final language construct we consider the *test* or *guard* $?P$. Intuitively, it does nothing if P is true in the current state and "aborts" the computation if P is false. By "abort" we mean that there is no poststate, a semantics shared by a nonterminating while loop.

$$\omega[\![?P]\!]\nu \quad \text{iff } \omega \models P \text{ and } \omega = \nu$$

With this, we can define

$$\begin{array}{lcll} \mathsf{skip} & \triangleq & ?\mathsf{true} & \textit{does nothing} \\ \mathsf{abort} & \triangleq & ?\mathsf{false} & \textit{aborts} \end{array}$$

These are *notational definitions* in the sense that the new program on the right expands to the program on the left. If we want to compute the semantics of the new kind of program we would just expand the definition and then compute the semantics of the result.

Tests can be used to model preconditions. A program such as

$$?(n \geq 0)\ ;\ \alpha$$

tests the condition $n \geq 0$ and proceeds with α if it is true. Therefore we can assume this condition while reasoning about the effect of α. If the condition is false then the computation aborts, so the final states only reflect the initial states that satisfy the test.

As an example, consider (once again) the following program to compute the $\mathsf{fib}(n)$.

```
$?(n \geq 0)$ ;
$i \leftarrow 0$ ;
$a \leftarrow 0$ ;
$b \leftarrow 1$ ;
while $(i < n)$
  ( $b \leftarrow b + a$ ;
    $a \leftarrow b - a$ ;
    $i \leftarrow i + 1$ )
```

If we start in a state $\omega[n \mapsto c]$ then when we reach the while loop we have

$$\omega_0 = \omega[n \mapsto c, a \mapsto \mathsf{fib}(0), b \mapsto \mathsf{fib}(1), i \mapsto 0]$$

After iteration $k \leq c$, we have

$$\omega_k = \omega[n \mapsto c, a \mapsto \mathsf{fib}(k), b \mapsto \mathsf{fib}(k+1), i \mapsto k]$$

So the final state of the whole loop, which is also the final state of the program, has the form

$$\omega_c = \omega[n \mapsto c, a \mapsto \mathsf{fib}(c), b \mapsto \mathsf{fib}(c+1), i \mapsto c]$$

6 Side Note on Conditionals and Arithmetic

Goldbach's conjecture, proposed in 1742 and still open, states that every even natural number greater than 2 is the sum of two primes. We can actually express this quite easily in arithmetic.

$$\begin{array}{lcl}
\mathsf{prime}(p) & \triangleq & \neg\exists a.\, \exists b.\, a > 1 \wedge b > 1 \wedge a * b = p \\
\mathsf{even}(a) & \triangleq & \exists b.\, 2 * b = a \\
\mathsf{goldbach} & \triangleq & \forall n. n > 2 \wedge \mathsf{even}(p) \rightarrow \exists p.\, \exists q.\, \mathsf{prime}(p) \wedge \mathsf{prime}(q) \wedge p + q = n
\end{array}$$

Note that goldbach does not depend on any variables. According to our semantics it should therefore be either true or false. This in turns means that for

$$\omega[\![\mathsf{if}\ \mathsf{goldbach}\ (x \leftarrow 1)\ (x \leftarrow 0)]\!]\nu$$

we have $\nu(x) = 1$ if $\omega \models \mathsf{goldbach}$ and $\nu(x) = 0$ if $\omega \not\models \mathsf{goldbach}$. So you can appreciate the difficulty of trying to execute this program!

7 Using Why3 to Reason About Semantics

We can use Why3 to a certain extent to verify our reasoning about the semantics using the `assert` contract. For example, the swap program with its pre- and post-condition might be rendered as

```
let swap1 (a:int) (b:int) =
let ref x = a in
let ref y = b in
assert { x = a /\ y = b } ;
x <- x + y ;
y <- x - y ;
x <- x - y ;
assert { x = b /\ y = a }
```

We have to be careful to remember that the postcondition for a program, here formulated as `assert { x = b /\ y = a }` only checks these conditions but not other effects the code might have. For example, the following will also pass:

```
let swap2 (a:int) (b:int) =
let ref x = a in
let ref y = b in
let ref z = 0 in
assert { x = a /\ y = b } ;
z <- x ;
x <- y ;
y <- z ;
assert { x = b /\ y = a }
```

However, the two programs between the `assert` statements are *not* equivalent, because the second one will also set z to a while the first one will not.

8 Summary

We summarize the various languages and definitions from this lecture, for reference.

$$\begin{array}{lll l}
\text{Expressions} & e & ::= & c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \ldots \\
\text{Formulas} & P, Q & ::= & e_1 = e_2 \mid e_1 \leq e_2 \mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \to Q \mid \neg P \\
 & & & \mid \forall x.\, P \mid \exists x.\, P \mid \ldots \\
\text{Programs} & \alpha, \beta & ::= & x \leftarrow e \mid \alpha \,;\, \beta \mid \mathsf{if}\ P\ \alpha\ \beta \mid \mathsf{while}\ P\ \alpha \mid ?P
\end{array}$$

$\omega[\![e]\!] = c \in \mathbb{Z}$ (The value of e in state ω is c)

$$\begin{array}{lcl}
\omega[\![c]\!] & = & c \\
\omega[\![x]\!] & = & \omega(x) \\
\omega[\![e_1 + e_2]\!] & = & \omega[\![e_1]\!] + \omega[\![e_2]\!] \\
\omega[\![e_1 - e_2]\!] & = & \omega[\![e_1]\!] - \omega[\![e_2]\!] \\
\omega[\![e_1 * e_2]\!] & = & \omega[\![e_1]\!] \times \omega[\![e_2]\!] \\
\ldots & &
\end{array}$$

$\omega \models P$ (Formula P is true in state ω)

$$\begin{array}{ll}
\omega \models \top & \text{always} \\
\omega \models \bot & \text{never} \\
\omega \models e_1 = e_2 & \text{iff } \omega[\![e_1]\!] = \omega[\![e_2]\!] \\
\omega \models e_1 \leq e_2 & \text{iff } \omega[\![e_1]\!] \leq \omega[\![e_2]\!] \\
\omega \models P \wedge Q & \text{iff } \omega \models P \text{ and } \omega \models Q \\
\omega \models P \vee Q & \text{iff } \omega \models P \text{ or } \omega \models Q \\
\omega \models \neg P & \text{iff } \omega \not\models P \\
\omega \models P \to Q & \text{iff whenever } \omega \models P \text{ then also } \omega \models Q \\
\omega \models \forall x.\, P & \text{iff } \omega[x \mapsto a] \models P \text{ for all } a \in \mathbb{Z} \\
\omega \models \exists x.\, P & \text{iff } \omega[x \mapsto a] \models P \text{ for some } a \in \mathbb{Z}
\end{array}$$

$\omega[\![\alpha]\!]\nu$ (Program α relates prestate ω to poststate ν)

$\omega[\![x \leftarrow e]\!]\nu$	iff $\nu = \omega[x \mapsto c]$ where $c = \omega[\![e]\!]$
$\omega[\![\alpha \, ; \beta]\!]\nu$	iff there is a μ such that $\omega[\![\alpha]\!]\mu$ and $\mu[\![\beta]\!]\nu$
$\omega[\![\text{if } P \, \alpha \, \beta]\!]\nu$	iff $\omega[\![\alpha]\!]\nu$ when $\omega \models P$ and $\omega[\![\beta]\!]\nu$ when $\omega \not\models P$
$\omega[\![\text{while } P \, \alpha]\!]\nu$	iff there exists an $n \geq 0$ such that $\omega[\![\text{while } P \, \alpha]\!]^n\nu$
$\omega[\![\text{while } P \, \alpha]\!]^0\nu$	iff $\omega \not\models P$ and $\omega = \nu$
$\omega[\![\text{while } P \, \alpha]\!]^{n+1}\nu$	iff $\omega \models P$ and there exists a μ such that $\omega[\![\alpha]\!]\mu$ and $\mu[\![\text{while } P \, \alpha]\!]^n\nu$
$\omega[\![?P]\!]\nu$	iff $\omega \models P$ and $\omega = \nu$

9 Specifying the Meaning of Regular Expressions

As an exercise in logical specifications using axioms (which is essentially what we did in this lecture), we use *regular expressions*. They are used, for example, in search engines, text editors, and compilers. At the beginning of the next lecture we will implement an elegant algorithm for regular expression matching proposed by Brzozowski [Brz64].

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 *)
```

9.1 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.

References

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