15-414: Bug Catching: Automated Program Verification

Lecture Notes on Ghosts

Frank Pfenning

Carnegie Mellon University
Lecture 4
February 11, 2021

1 Introduction

Complex data structures are often difficult to reason about. Compare, for example, a red/black tree implementation of a map to a simple association list. To control this complexity we have introduced the concept of a data structure invariant. In the red/black tree example, this would include the *ordering invariant* (keys in left subtrees are all smaller and keys in the right subtree are all larger than the key in a node), the *color invariant* (there are no two adjacent red nodes in the tree), and the *black height invariant* (the number of black nodes on any path from a leaf to the root is the same). These are all *internal invariants* in the sense that the implementation of the data structure must maintain them but the client should only care that red/black trees provide a correct and efficient implementation of a map from keys to values. In this case, we say the map provides a *model* of the intended behavior of the data structure. We would like the model to be *logical* and as high-level as possible to support reasoning by the client. Maps and sets are common models. In today's lecture we exemplify sets as a model of an ephemeral (mutable) implementation of bit vectors as arrays.

When implementing a data structure we have to maintain the correspondence between the low-level implementation and the high-level model. The purpose of the model is to *reason* about a data structure, but not to *compute* with it. So we would like to *erase* the code that maintains the model: actually computing it would negate all the advantages of the efficient implementation! This is the primary purpose of *ghosts*. They are pieces of code or data that exist solely for the purpose of verification and do not contribute to the outcome of the computation. This has to be checked by the verification engine. Ghost variables, or ghost fields of records, can only be used in other ghost computations. Otherwise, erasing them before the program is run would lead to

incorrect code. This condition is related to the fact that *executable contracts* in C0 can not have any externally observable effects: running the program with or without executable contracts should yield the same answer (as long as all contracts are satisfied, of course).

We start with a tiny example (Fibonacci, here we go again!) to illustrate the need for ghosts, and then use sets as a model for bit vectors. Toward the end of the lecture we illustrate more complex specifications by giving a verified implementation of a regular expression matcher using Brzozowski derivatives.

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

- Use ghosts in verification;
- Model data structures using ghosts;
- Express more complex specifications logically;
- Understand Brzozowski derivatives of regular expressions.

2 Ghost Variables

As a first example, consider the following alternative definition of a recursive Fibonacci function. Please take a few minutes to ponder this (correct!) definition and think about why it is correct and how you might verify it before you move on.

```
1  module Fib
2
3    use int.Int
4
5    function fib (n:int) : int
6    axiom fib0 : fib 0 = 0
7    axiom fib1 : fib 1 = 1
8    axiom fibn : forall n:int.
9      n >= 0 -> fib n + fib (n+1) = fib (n+2)
10
11   let rec fib_alt_rec (a : int) (b : int) (n : int) : int =
12   requires { 0 <= n }
13   variant { n }
14   if n = 0 then a else fib_alt_rec b (a+b) (n-1)
15
16   let fib_alt_top (n : int) : int =
17   requires { 0 <= n }
18   ensures { result = fib n } (* fails *)
19   fib_alt_rec 0 1 n
20
21 end
```

After staring at the function for a while it seems that it is correct. Initially, a and b are $\mathsf{fib}(0)$ and $\mathsf{fib}(1)$ and after i recursive calls, a will continue to hold $\mathsf{fib}(i)$. However, we cannot express this information because the number of iterations is not available to us. The only recourse we have is to add the integer i as a furhter argument to the function.

```
let rec fib_alt_rec (a:int) (b:int) (i:int) (n:int) : int =
requires { 0 <= n /\ 0 <= i }
requires { a = fib i /\ b = fib (i+1) }
ensures { ... }
variant { n }
if n = 0 then a else fib_alt_rec b (a+b) (i+1) (n-1)
```

We already added the requirement and i be nonnegative and a and b must be the values of $\mathsf{fib}(i)$ and $\mathsf{fib}(i+1)$.

What can we say about the return value? The result of the original call should be $\mathsf{fib}(n)$. On the first recursive call, it should *still* be $\mathsf{fib}(n)$, but for the original n! Meanwhile, we have decreased n, so as far as the postcondition is concerned it should now be $\mathsf{fib}(n+1)$. And so on. But how do we say this? Please think it through before reading on . . ..

The key insight here is the $i + n$ remains *invariant* on each recursive call: we decrease n and increase i. Initially, $i = 0$ so $i + n = n$, the needed answer. When the recursion terminates, we have $n = 0$ and i has counted up to the original n, so again $i + n$ is the correct answer. With this insight it is easy to complete and verify the code.

```
let rec fib_alt_rec (a:int) (b:int) (i:int) (n:int) : int =
requires { 0 <= n /\ 0 <= i }
requires { a = fib i /\ b = fib (i+1) }
ensures { result = fib (i+n) }
variant { n }
if n = 0 then a else fib_alt_rec b (a+b) (i+1) (n-1)

let fib_alt_top (n : int) : int =
requires { 0 <= n }
ensures { result = fib n } (* succeeds! *)
fib_alt_rec 0 1 0 n
```

The sad truth we have to confront now is that we have made our function (marginally) less efficient by adding another argument *just for the correctness proof*. But i plays no computational role; its sole purpose is to express the contracts for `fib_alt_rec`. This is where *ghosts* come to the rescue. we can declare the third argument to `fib_alt_rec` to be a *ghost argument* that can be safely erased before the program is executed.

```
let rec fib_alt_rec (a:int) (b:int) (ghost i:int) (n:int) : int =
requires { 0 <= n /\ 0 <= i }
requires { a = fib i /\ b = fib (i+1) }
ensures { result = fib (i+n) }
variant { n }
if n = 0 then a else fib_alt_rec b (a+b) (ghost i+1) (n-1)

let fib_alt_top (n:int) : int =
requires { 0 <= n }
ensures { result = fib n } (* succeeds! *)
fib_alt_rec 0 1 (ghost 0) n
```

We see three occurrences of the keyword `ghost`, and all of the expressions marked in this will be erased before executing the code. The verifier checks that this is indeed safe. Let's modify the first function to *illegally* use i in a computationally relevant context.

```
let rec fib_alt_rec (a:int) (b:int) (ghost i:int) (n:int) : int =
requires { 0 <= n /\ 0 <= i }
requires { a = fib i /\ b = fib (i+1) }
ensures { result = fib (i+n) }
variant { n }
if n = 0 then a+(i-i) else fib_alt_rec b (a+b) (ghost i+1) (n-1)
```

Even though the code is still correct ($i - i = 0$), it will now fail (as it should) even before any theorem prover is called.

```
% why3 prove fibalt.mlw
File "fibalt.mlw", line 10, characters 10-21:
Function fib_alt_rec must be explicitly marked ghost
%
```

This message explains that `fib_rec_alt` would be admissible, but only if the whole function were marked as a ghost. If so, the function could only be called in a ghost context, where the call itself would be erased.

The live code for this alternative implementation of fib can be found in fibalt.mlw.

3 Models

As an example of a model we use the standard set library to model a bit vector implementation of bounded finite sets. Here is an excerpt of the finite set module.

```
module Fset
  type fset 'a
  predicate mem (x: 'a) (s: fset 'a)
  predicate is_empty (s: fset 'a) = forall x: 'a. not (mem x s)
  constant empty: fset 'a
  function add (x: 'a) (s: fset 'a) : fset 'a
  axiom add_def: forall x: 'a, s: fset 'a, y: 'a.
      mem y (add x s) <-> (mem y s \/ y = x)
  function remove (x: 'a) (s: fset 'a) : fset 'a
  axiom remove_def: forall x: 'a, s: fset 'a, y: 'a.
      mem y (remove x s) <-> (mem y s /\ y <> x)
  ...
end
```

We start by defining the `Bset` module by defining a `bset` as a record consisting of an array `a` and a ghost field called `model` containing a finite set of integers. Because bsets are mutable (for example, we actually *change* a bset by adding an element to it), the `model` field must also be mutable.

```
type bset = { a : array bool ;
              mutable ghost model : Fset.fset int }
invariant {forall i. 0 <= i < a.length -> (a[i] <-> Fset.mem i model)}
by { a = Array.make 0 false ; model = Fset.empty }
```

The invariant states that element $a[i]$ of the array is true if and only if the number i is in the model set. We witness the existence of such a bset with the empty array and empty model.

To create an empty bset we need a bound on the elements we may add to the set, which will be the length of the array.

```
let empty_bset (bound : int) : bset =
requires { bound >= 0 }
ensures { Fset.is_empty result.model}
{ a = Array.make bound false ; model = Fset.empty }
```

The model is just the empty set. Note that the postcondition states that `empty_bset` models the empty finite set.

To add an element i to a bset we just set the corresponding array element to true (whether it was already true or not). This requires the precondition that the i is in the permissible range. Because this operation is destructive, modifying the given bset, the postcondition needs to state the model *after* the update is equal to the model *before* the

update, plus the element i. For this purpose we use again the `old` keyword to refer to the state of the model at the time the function is called.

```
let add_bset (i : int) (s : bset) : unit =
requires { 0 <= i < s.a.length }
ensures { s.model = Fset.add i (old s).model }
s.a[i] <- true ;
s.model <- Fset.add i s.model ;
()
```

This postcondition will allow the client to reason about the effects of it add operations. The remove operation is entirely analogous.

```
let remove_bset (i : int) (s : bset) : unit =
requires { 0 <= i < s.a.length }
ensures { s.model = Fset.remove i (old s).model }
s.a[i] <- false ;
s.model <- Fset.remove i s.model ;
()
```

We did not implement any more complex operations such as union or intersection, even though this would certainly be possible. You can find the live-code `Bset` module in the file bset.mlw.

4 Regular Expression Matching

The goal of this section will be to implement a verified matcher for regular expressions. In lecture, we managed to specify and prove the *finite* fragment (without the r^* operator). We use the very elegant algorithm proposed by Brzozowski [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.

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 *)
```

4.1 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 $\mathcal{L}(1) = \{\epsilon\}$, 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 $\mathcal{L}(r^*) = \mathcal{L}(\epsilon + r \cdot r^*)$. 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)
```

The difficulty here appears to be the fact that if $w_1 = \epsilon$ when $w_2 = w$ 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 $w_1 = \epsilon$ since it does not add to the set $\mathcal{L}(r^*)$. So we can restrict the axiom to non-empty words w_1 without affecting $\mathcal{L}(r^*)$.

```
  axiom mem_star0 : 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.

4.2 Specifying the Brzozowski Derivative

The Brzozowski derivate of a regular expression r with respect to a character a is written as $\partial_a r$. We would like to define it such that

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

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 $\partial_a r$. We can then define a regular expression matcher traversing the word left to right, computing the derivative at each step until the word is empty. Then it remains to decide if $\epsilon \in \mathcal{L}(r)$ which we call nullable(r), which can also be done effectively.

The outline of the rest of our program then becomes:

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

```

```
5  let rec deriv (a:char) (r:regexp) : regexp =
6  ensures { forall w. mem (Cons a w) r <-> mem w result }
7  ...
8
9  let rec re_match (w:word) (r:regexp) : bool =
10 ensures { result <-> mem w r }
11 variant { w }
12 match w with
13 | Cons a w' -> re_match w' (deriv a r)
14 | Nil -> nullable r
15 end
```

We have filled in the postconditions for `nullable` and `deriv` as well as `re_match`. You should study them carefully to make sure you understand they correctly render the mathematical definitions. We also have filled in the variant guaranteeing the termination of `re_match`. We see here a variant declaration for an inductive types such as `list 'a`. Such a declaration means that every recursive call will be on a *sublist* of the function argument. Here, this is obvious since w' is just the tail of w.

It remains to write and verify the `nullable` and `deriv` functions.

4.3 Nullable Regular Expressions

We have to write a function to determine if a regular expression would generate the empty word. This is actually quite straightforward. 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  ensures { result <-> mem Nil r }
3  variant { 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.

4.4 Computing the Brzozowski Derivative

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 have to keep the specification in

mind, so we repeat it here.

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

We start by defining the derivative in mathematical notation.

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

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 a'      -> if a = a' then One else Zero
| One          -> Zero
| Times r1 r2 -> let r1' = Times (deriv a r1) r2 in
                   if nullable r1 then Plus r1' (deriv a r2) else r1'
| Zero         -> Zero
| Plus r1 r2   -> Plus (deriv a r1) (deriv a r2)
| Star r       -> Times (deriv a r) (Star r)
end
```

The complete file can be found in regexp-all.mlw, the live-coded fragment without repetition is in regexp.mlw.

4.5 Mixed Verification

At this point, trying either of the most common provers (alt-ergo and CVC4) fails. But they fail on different subgoals in trying to verify the `deriv` function—both are able to prove `nullable re_match`.

```
% why3 prove -P alt-ergo regexp-all.mlw
regexp-all.mlw RegExp nullable'vc: Valid (0.05s, 546 steps)
```

```
regexp-all.mlw RegExp deriv'vc: Timeout (5.00s)
regexp-all.mlw RegExp re_match'vc: Valid (0.01s, 63 steps)
% why3 prove -P cvc4 regexp-all.mlw
regexp-all.mlw RegExp nullable'vc: Valid (0.44s, 86271 steps)
regexp-all.mlw RegExp deriv'vc: Timeout (5.00s, 357806 steps)
regexp-all.mlw RegExp re_match'vc: Valid (0.05s, 7930 steps)
%
```

Starting up the IDE and using strategy 'Auto level 2' fortunately breaks the verification condition into smaller subgoals, each of which can ultimately be proved by one or the other. Examining the session provides some details and statistics. We show only the final statistics.

```
== Statistics per prover: number of proofs, time (minimum/maximum/average) in seconds ==
  Alt-Ergo 2.3.1      :   6   0.01   9.00   1.73
  CVC4 1.7            :  16   0.03   0.46   0.07

%
```

References

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

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