

Lecture 11 - February 10

Lab2 Solution Walkthrough, Model Checking

***Generalizing rounds, Function, Macro
Postconditions: getAllSuffixes
LTL Grammar: Top-Down Derivation***

Announcements/Reminders

- **ProgTest1** this Thursday during the lab session
 - + Please arrange your commute accordingly.
 - + Test will only be canceled if the university is closed.
- **Practice Test** questions and solutions released
- **Lab1** and **Lab2** solutions released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- TA contact information (on-demand for labs) on eClass

Lab2 Solution:

decideRPSGameResult

(Generalizing # of Rounds)

initial not-deterministic
choices for players
in the 1st round

non-deterministic
choices for player
in the next round

```
(*  
--algorithm decideRPSGameResult {  
variable  
  p1r1 \in {"R", "P", "S"},  
  p2r1 \in {"R", "P", "S"},  
  numOfRoundsWonByPlayer1 = 0,  
  numOfRoundsWonByPlayer2 = 0,  
  p1win = FALSE,  
  p2win = FALSE,  
  tie = FALSE,  
  i = 1,  
  rounds = 2;  
{  
  while(i <= rounds){  
    if (p1r1 = "R" /\ p2r1 = "P"){  
      numOfRoundsWonByPlayer2 := numOfRoundsWonByPlayer2 + 1;  
    };  
    else if (p1r1 = "R" /\ p2r1 = "S"){  
      numOfRoundsWonByPlayer1 := numOfRoundsWonByPlayer1 + 1;  
    };  
    ... /* Consider cases where p1r1 = "P" and p1r1 = "S" */  
  
    /* Get prepared for the next round.  
  
    choose_p1_next: with (x \in {"R", "P", "S"}) {  
      p1r1 := x;  
    };  
    choose_p2_next: with (x \in {"R", "P", "S"}) {  
      p2r1 := x;  
    };  
    i := i + 1;  
  };  
  
  if (numOfRoundsWonByPlayer1 = numOfRoundsWonByPlayer2){  
    tie := TRUE;  
  };  
  else if (numOfRoundsWonByPlayer1 > numOfRoundsWonByPlayer2){  
    p1win := TRUE;  
  };  
  else {  
    p2win := TRUE;  
  };  
};  
*)
```

Lab2 Solution:

decideRPSGameResult

(Postcondition)

```

2
CONSTANT rounds
(* --algorithm decideRPSGame {
  variables
    /* Define two total functions. */
    p1 \in [1..rounds] -> {"R", "P", "S"}],
    p2 \in [1..rounds] -> {"R", "P", "S"}],
    roundsWonByP1 = 0,
    roundsWonByP2 = 0,
    i = 1,
    p1win = 0,
    p2win = 0,
    tie = 0;
  define {
    NumberOfWins(player, opponent) == Cardinality({
      j \in 1..rounds :
        ✓ player[j] = "R" ∧ opponent[j] = "S"
        ✓ player[j] = "P" ∧ opponent[j] = "R"
        ✓ player[j] = "S" ∧ opponent[j] = "P"
    })
  };
}

```

Handwritten notes:

- total function* (pointing to the function definitions)
- range* (pointing to the range [1..rounds])
- pl is total function* (pointing to p1)
- what players play in all rounds* (pointing to the array access)
- pl = [1 → "R", 2 → "P"]* (pointing to p1)
- p2 = [1 → "S", 2 → "R"]* (pointing to p2)
- possible ways pl (player) can win* (pointing to the win conditions)
- set comprehension* (pointing to the set notation)
- pl won* (pointing to the result of the function)
- none of the disjunct is satisfied* (pointing to the win conditions)
- * pl[i] did not win function application* (pointing to the function call)

```

{
  while (i <= rounds) {
    if (p1[i] = "R") {
      if (p2[i] = "P") {
        roundsWonByP2 := roundsWonByP2 + 1;
      } else if (p2[i] = "S") {
        roundsWonByP1 := roundsWonByP1 + 1;
      };
    }
    /* Consider p1[i] = "P" or p1[i] = "S"
    ...
    i := i + 1;
  };

  /* Set p1win, p2win, tie w.r.t. roundsWonByP1, roundsWonByP2.
  ...

  /* Assert consistency among variables.
  assert
    /\ roundsWonByP1 > roundsWonByP2 => (
      /\ p1win = 1
      /\ p2win = 0
      /\ tie = 0)
  /* Also consider _ < _ and _ = _
  ...

  /* Verify the correctness of win decisions.
  assert NumberOfWins(p1, p2) = roundsWonByP1;
  assert NumberOfWins(p2, p1) = roundsWonByP2;
} *)

```

Handwritten notes:

- final value of prog. variable* (pointing to the final state of the program)
- calling the macro* (pointing to the NumberOfWins macro call)

Lab2 Solution: getAllSuffixes_v3 (1)

```

----- MODULE getAllSuffixes_v3 -----
EXTENDS Integers, Sequences, TLC
CONSTANT input
ASSUME Len(input) > 0
(*
--algorithm getAllSuffixes_v3 {
  variable result = input, postfixSoFar = <<>>, i = Len(input) - 1;
  {
    postfixSoFar := <<input[Len(input)]>>;
    result[Len(input)] := postfixSoFar;
    while (i > 0) {
      postfixSoFar := <<input[i]>> \o postfixSoFar;
      result[i] := postfixSoFar;
      i := i - 1;
    }
    assert \A j \in 1..Len(input): Len(result[j]) = Len(input) - j + 1;
    assert \A j \in 1..Len(result): (\A k \in 1..Len(result[j]): result[j][k] = input[j - 1 + k]);
  }
*)

```

index
input: [23, 46, 69]

result:
 [[23, 46, 69],
 [46, 69],
 [69]]

index
 1
 2
 3

Concatenation

$len(r[j]) = len(input) + 1 - j$

$j + len(r[j]) = len(input) + 1$

<u>j</u>	<u>len(result[j])</u>	<u>len(input)</u>
1	3	3
2	2	3
3	1	3

Postcondition

- (1) Understand the quantification expression on some concrete example.
- (2) Practice writing from scratch.

Lab2 Solution: getAllSuffixes_v3 (2)

$$\begin{array}{c} j \\ \hline 1 \\ 2 \\ 3 \end{array} \quad \begin{array}{c} \text{result}[j] \\ \hline [23, 46, 69] \\ [46, 69] \\ [69] \end{array} \quad \begin{array}{c} k \\ \hline 1 \\ 2 \\ \dots \end{array} \quad \begin{array}{c} 2 \\ 2 \\ \dots \end{array} \rightarrow$$

----- MODULE getAllSuffixes_v3 -----

EXTENDS Integers, Sequences, TLC

CONSTANT input

ASSUME Len(input) > 0

(*

--algorithm getAllSuffixes_v3 {

variable result = input, postfixSoFar = <<>>, i = Len(input) - 1;

{

postfixSoFar := <<input[Len(input)]>>;

result[Len(input)] := postfixSoFar;

while (i > 0) {

postfixSoFar := <<input[i]>> \o postfixSoFar;

result[i] := postfixSoFar;

i := i - 1;

};

assert \A j \in 1..Len(input): Len(result[j]) = Len(input) - j + 1;

assert \A j \in 1..Len(result): (\A k \in 1..Len(result[j]): result[j][k] = input[j - 1 + k]);

}

}

*)

input: [23, 46, 69]

result:

[[23, 46, 69],

[46, 69],

[69]]

$j + k = 2 + 1 = 3$

$j + k = 2 + 2 = 4$

$j + k - 1$

seg of int.

another seg

??

e.g. result[1][2] → 46.

$$\begin{array}{c} j \\ \hline 1 \\ 2 \\ 3 \end{array}$$

$$\begin{array}{c} \text{result}[j] \\ \hline [23, 46, 69] \\ [46, 69] \\ [69] \end{array}$$

$$\begin{array}{c} \text{input} \\ \hline [23, 46, 69] \\ [23, 46, 69] \\ [23, 46, 69] \end{array}$$

$$\begin{array}{c} k \\ \hline 1 \\ 2 \\ 3 \end{array}$$

$$\begin{array}{c} * \\ \hline \text{result}[2][1] = \text{input}[2] \\ \text{result}[2][2] = \text{input}[3] \end{array}$$

LTL Syntax: Context-Free Grammar

propositional
logic

$\phi ::=$	$\top$	[true]
	$\perp$	[false]
	p	[propositional atom]
	$(\neg \phi)$	[logical negation]
	$(\phi \wedge \phi)$	[logical conjunction]
	$(\phi \vee \phi)$	[logical disjunction]
	$(\phi \Rightarrow \phi)$	[logical implication]
	$(X \phi)$	[next state]
	$(F \phi)$	[some Future state]
	$(G \phi)$	[all future states (Globally)]
	$(\phi U \phi)$	[Until]
	$(\phi W \phi)$	[Weak-until]
	$(\phi R \phi)$	[Release]

recursive cases

implicitly use
 $\forall$ and/or $\exists$

LTL

LTL

propositional
logic

X, F, G, U, W, R

✓ $\phi ::= \boxed{T} \textcircled{4}$

start symbol

- $\perp$
- $p \textcircled{2}$
- $(\neg \phi) \textcircled{1}$
- $(\phi \wedge \phi)$
- $(\phi \vee \phi)$
- $(\phi \Rightarrow \phi)$
- $(X \phi) \textcircled{3}$
- $(F \phi)$
- $(G \phi)$
- $(\phi U \phi)$
- $(\phi W \phi)$
- $(\phi R \phi)$

- [true]
- [false]
- [propositional atom]
- [logical negation]
- [logical conjunction]
- [logical disjunction]
- [logical implication]
- [ne**X**t state]
- [some **F**uture state]
- [all future states (**G**lobally)]
- [**U**ntil]
- [**W**eak-until]
- [**R**elease]

(top-down) derivation:

$\boxed{p \wedge X T}$

syntactically correct

$$\begin{aligned}
 \phi &= \phi \wedge \phi \\
 &= p \wedge \phi \\
 &= p \wedge X \phi \\
 &= p \wedge X T
 \end{aligned}$$