Exercises on Induction, Recursion, and Iteration

Induction on Natural Numbers

These exercises are intended to illustrate the trials and tribulations of induction, recursion, and iteration. The exercises in this section refer to the theory `induction.pvs`.

1. The factorial function is defined in the NASA PVS theory `ints@factorial` as follows:

   ```
   factorial(n): RECURSIVE posnat =
     IF n = 0 THEN 1
     ELSE n*factorial(n-1)
   ENDIDF
   MEASURE n
   ```

   Problem: Use induction to prove that the factorial of any number strictly greater than 1 is even. Lemma `factorial_even` specifies this statement in PVS. The predicate `even?` is defined in the PVS prelude library as follows.

   ```
   even?(i): bool = EXISTS j: i = j * 2
   ```

   Hint: First use `(induct "n")`. The base case is discharged by `(grind)`. For the inductive case, introduce the skolem constants, along with its type information, with the proof command `(skeep :preds?  t)`. Then, expand the definitions of `factorial` and `even?`. Be careful here, to avoid expanding all occurrences of `factorial` use the command `(expand "factorial"` *fnum*`)`, where *fnum* is a formula number. Next, you have to introduce an skolem constant for the existential formula in the antecedent, use for example `(skolem` *fnum* `"J")`, and to instantiate the existential variable in the consequent, use for example `(inst` *fnum* `"J*(ja+1)")`. The proof command `(assert)` finishes the proof.

2. **Problem:** Use induction to prove the following statement about the factorial function

   $$\forall n : n! \geq n.$$

   Lemma `factorial_ge` specifies this statement in PVS.

   Hint: First use `(induct "n")`. The base case is discharged easily. After expanding the right occurrence of `factorial`, assert that the factorial of `n` is greater than or equal to 1. This can be accomplished with the proof command `(case "factorial(n) >= 1")`. Multiply both sides of that inequality by `j+1` using the proof rule `mult-by` (see lecture on proving real number properties). Finally, use `(assert)`.

3. The two-variable Ackermann function can be defined as follows.

$$ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ ack(m-1,1) & \text{if } n = 0 \\ ack(m-1, ack(m, n-1)) & \text{otherwise.} \end{cases}$$

Problem: Prove the following statement about the Ackermann function

$$\forall m, n : ack(m,n) > m + n.$$

Lemma `ack_gt_m_n` specifies this statement in PVS.

Hint: Avoid induction, recursive judgments are your friends. Once you express the formula as a recursive judgement, the proof of `ack_gt_m_n` is just `(grind)`. The TCCs are discharged automatically using the Emacs command `M-x tcp`.

4. The exponent function is defined in the PVS prelude as follows.

```
expt(r, n): RECURSIVE real =
  IF n = 0 THEN 1
  ELSE r * expt(r, n-1)
  ENDIF
MEASURE n
```

The following is an imperative version of this function written in pseudo-code.

```
function expt_it(x:real,n:nat):nat {
  a := 1;
  // a = expt(x,0)
  for (i:=1; i <= n; i++) {
    // invariant: a = expt(x,i)
    a := a*x;
  }
  return a;
  // post: a = expt(x,n)
}
```

In PVS, using the for loop defined in `structures@for_iterate`, the function `expt_it` can be specified as follows.

```
expt_it(x:real,n:nat): real =
  for[real](1,n,1,LAMBDA(i:subrange(1,n),a:real):a*x)
```

Problem: Prove that the functions `expt_it` and `expt` coincide in all points `x` and `n`. Lemma `expt_it_sound` specifies this statement in PVS.

Hint: After expanding the definition of `expt_it` use lemma `for_induction[real]`. All universal variables in that lemma, but `inv`, are automatically instantiated using the proof command `(inst?` *fnum*`)`. The universal variable `inv` corresponds to the invariant of the loop and it is a predicate of the form

```
LAMBDA(i:upto(n),a:real):  ...
```

where `i` is the iteration number and `a` is the value of the accumulator at each iteration. Once you find the right invariant *inv* use the proof command `(inst` *fnum inv*`)`. The command `(grind)` finishes the proof.

5. The predicate `even?` can be inductively defined in PVS as follows.

```
even(n:nat): INDUCTIVE bool =
  n = 0 OR (n > 1 AND even(n - 2))
```

   Problem: Prove that for all natural number `n`, `even?(n)` holds if `even(n)` holds. Lemma `we_are_even` specifies this statement in PVS.

   Hint: Start the proof with `(rule-induct "even")` and then you are on your own.

Induction on Abstract Data Types

A data-type representing single variable polynomial expressions such as `(x+3)^2-5x` is defined in PVS. This data-type is provided with a function that evaluates a polynomial expression on a real value and a function that symbolically computes the derivative of a polynomial expression. The following lemmas have to be proved:

- The evaluation function is continuous.
- The evaluation function is differentiable.
- The function that computes the symbolic derivative of a polynomial expression is correct.

The following exercises refer to definitions that are provided in the theories `PolyExpr.pvs` and `poly_expr.pvs`.

1. Study the definitions in `PolyExpr.pvs` and `poly_expr.pvs`.

   Problem: Using those definitions write a statement that represents the following proposition: "The derivative of `(x+3)^2-5x` is equal to `2x+1`." Prove it.

   Hints:

- If `p1` and `p2` are PVS objects of type `PolyExpr`, what is the intended semantics of the statement "`p1` is equal to `p2`?"
- The proof command `decompose-equality` can be used to prove that two PVS functions are equal.

2. **Problem:** Prove the formula `eval_continuous` that states the fact that the evaluation function is continuous. This formula is expressed as a recursive judgment, which allows for an inductive proof without explicitly using induction.
   Hints:
   - The lemma `PolyExpr_inclusive`, which is part of the definition of the type `PolyExpr`, states that all elements of that type are built with either a constant, a variable, an addition, a subtraction, a multiplication, or a power constructor.
   - Note that the inductive hypothesis is hidden in the type of the quantified variable "v". To make this type explicit, use the command `typepred`, e.g., `(typepred "v(expr1(pexpr))")`.
   - The following lemmas in the NASA PVS Library state the continuity of the constant, identity, addition, subtraction, multiplication, and power functions, respectively: `const_cont`, `id_cont`, `add_cont`, `sub_cont`, `mult_cont`, and `pow_cont`.

3. **Problem:** Prove the recursive judgment `eval_differentiable` that states the fact that the evaluation function is differentiable.
   Hints:
   - The following lemmas in the NASA PVS Library state the differentiability of the constant, identity, addition, subtraction, and multiplication functions, respectively: `derivable_const_lam`, `derivable_id_lam`, `derivable_add_lam`, `derivable_sub_lam`, and `derivable_mult_lam`.
   - The differentiability of the power function has to be proved with the lemmas `comp_derivable_fun` and `derivable_pow_lam`.

4. **Homework:** Prove the lemma `eval_derivative` that states the correctness of the evaluation function. Use induction on the variable `pexpr`.

   Hints:
   - The following lemmas in the NASA PVS Library state the derivative of the constant, identity, addition, subtraction, and multiplication functions, respectively: `deriv_const_lam`, `deriv_id_lam`, `deriv_add_lam`, `deriv_sub_lam`, and `deriv_mult_lam`.
   - The derivative of the power function has to be proved with the lemmas `chain_rule[real,rea` and `deriv_pow_lam`.
   - The lemma `eta[real,real]` states the η-rule: For all f of type `[real->real]` and x of type `real`, $f = \lambda x.f(x)$.