DD2552 Seminar 7: Function Specification and Contracts

Karl Palmskog

KTH

Wednesday September 20, 2023

Course material

- PFPL Chapter 16 on parametricity properties of polymorphic functions (type variables)
- “Why3 – Where Programs Meet Provers” https://inria.hal.science/hal-00789533

Type variables

- lists and other “parameterized” data types are obiquitous
- we want to define functions an prove properties once and for all parameters
- this is usually called “parametric polymorphism” in contrast to “ad-hoc polymorphism”
- parametric polymorphism is typically via type variables
  - compare Java generics and C++ templates
- see PFPL chapter 16 for a more formal account

Parametric programming with lists

```
let rec f (p: 'a -> bool) (l:list 'a) : bool =
 match l with
 | Nil -> true
 | Cons x r -> p x && f p r
 end
```

- we can instantiate `'a` for any type
- correctness proofs can be done implicitly quantified over all `'a`

Companion lemmas vs. contracts

- reasoning about pure functions often done using companion lemmas
- lemmas organized separately from the code
- not clear which lemmas are important
- contracts: popular for imperative programs
  - specify requirements inline (“requires”)
  - specify guarantees inline (“ensures”)
  - specify what is modified inline (“modifies”)

Contracts for imperative programs

```
/*@ requires
  @   a != null;
  @ assigns
  @   a[..];
  @ ensures
  @   reversed{Old,Here}(a, 0, a.length/2, a.length - 1);
  @*/
public static void reverse(int[] a) {
  /*@ loop_invariant
    @   reversed{Pre,Here}(a, 0, i, a.length - 1) &&
    @    0 <= i <= a.length - i + 1;
    @ loop_variant a.length - 1 - i;
    @*/
  for (int i = 0; i <= a.length - 1 - i; i++) {
      swap(a, i, a.length - 1 - i);
  }
}
```

Contracts for/using functions?

- pure functions can be viewed restricted imperative programs
- so contract theory still applies
- not so popular in practice (but see Liquid Haskell)
- much more popular: pure functions **in contracts**

GCD function in WhyML

```
let rec function gcd (a b : int) : int =
  if a = 0 then 0
  else if b = 0 then 0
  else if a = b then a
  else if a < b then gcd a (b-a)
  else gcd (a-b) a
```

- what do we require of `a` and `b`?
- what is ensured about the `result`?
- what is the termination measure (`variant`)?

Auxiliary definitions for contracts

- in WhyML, we can define auxiliary predicates and functions only usable in contracts
- predicates are `bool`-valued functions that can use `forall`/`exists`

```
predicate divides (n m : int) =
  exists k. n * k = m
```

GCD function in WhyML with contracts

```
let rec gcd (a b : int) : int







=
if a = 0 then 0
else if b = 0 then 0
else if a = b then a
else if a < b then gcd a (b-a)
else gcd (a-b) a
```

GCD function in WhyML with contracts

```
let rec gcd (a b : int) : int
  requires { 0 <= a && 0 <= b }







=
if a = 0 then 0
else if b = 0 then 0
else if a = b then a
else if a < b then gcd a (b-a)
else gcd (a-b) a
```

GCD function in WhyML with contracts

```
let rec gcd (a b : int) : int
  requires { 0 <= a && 0 <= b }







  variant { a + b }
=
if a = 0 then 0
else if b = 0 then 0
else if a = b then a
else if a < b then gcd a (b-a)
else gcd (a-b) a
```

GCD function in WhyML with contracts

```
let rec gcd (a b : int) : int
  requires { 0 <= a && 0 <= b }
  ensures  { (0 < a && 0 < b) ->
     (divides result a && divides result b &&
  (forall k. (divides k a &&
        divides k b) -> k <= result))
  }
  variant { a + b }
=
if a = 0 then 0
else if b = 0 then 0
else if a = b then a
else if a < b then gcd a (b-a)
else gcd (a-b) a
```

List reversal in WhyML

```
let rec function reverse (l : list 'a) : list 'a =
  match l with
  | Nil -> Nil
  | Cons a l0 -> reverse l0 ++ Cons a Nil
  end

let rec function rev_aux (l1 l2: list 'a) : list 'a =
  match l1 with
  | Nil -> l2
  | Cons a l11 -> rev_aux l11 (Cons a l2)
  end

let rec function rev (l : list 'a) : list 'a
  ensures { result = reverse l }
  =
  rev_aux l Nil
```

Why use another function as a contract?

- easier to understand than function being specified
- aid when proving: correct-by-construction / hand-in-hand
- in practice: helpful to proof automation machinery

```
lemma reverse_append:
  forall l1 l2: list 'a, x: 'a.
  (reverse (Cons x l1)) ++ l2 = (reverse l1) ++ (Cons x l2)

lemma reverse_cons:
  forall l: list 'a, x: 'a.
  reverse (Cons x l) = reverse l ++ Cons x Nil

lemma cons_reverse:
  forall l : list 'a, x: 'a.
   Cons x (reverse l) = reverse (l ++ Cons x Nil)

lemma reverse_reverse:
  forall l: list 'a. reverse (reverse l) = l
```

More auxiliary lemmas

```
lemma rev_aux_append_append_l:
 forall r [@induction] s t: list 'a.
    rev_aux (r ++ s) t = rev_aux s (rev_aux r t)

lemma rev_aux_append_r:
  forall r s t: list 'a.
    rev_aux r (s ++ t) = rev_aux (rev_aux s r) t

lemma rev_aux_append:
  forall r [@induction] s: list 'a.
   reverse r ++ s = rev_aux r s
```

More abstract data types and functions

- WhyML standard library provides `Fset` module for finite sets
- `Fset.mem` behaves as list containment
- `Elements.elements` creates a finite set from a list
- can we use `Fset.mem` in contracts easily?

```
let rec function contains (x : 'a) (l : list 'a) : bool
  ensures { result <-> Fset.mem x (elements l) }
  =
  Mem.mem eq x l
```

Type variables and equality tests

```
let rec function contains (eq : 'a -> 'a -> bool)
 (x : 'a) (l : list 'a) : bool
 ensures {
  result <->
  (exists y. Fset.mem y (elements l) && eq y x)
 }
 =
 Quant.mem eq x l
```

Add function

```
let function add (eq : 'a -> 'a -> bool)
 (l : list 'a) (x : 'a) : list 'a
 ensures {
  exists y. Fset.mem y (elements result) && eq x y
 }
 =
```

Add function

```
let function add (eq : 'a -> 'a -> bool)
 (l : list 'a) (x : 'a) : list 'a
 ensures {
  exists y. Fset.mem y (elements result) && eq x y
 }
 =
 if contains eq x l then l else (Cons x l)
```

Remove function

```
let function remove (eq : 'a -> 'a -> bool)
(l : list 'a) (x : 'a) : list 'a
```

Finite maps in WhyML

```
module FMap

type fmap 'k 'v

function add (k: 'k) (v: 'v) (m: fmap 'k 'v) : fmap 'k 'v

predicate mem (k: 'k) (m: fmap 'k 'v)

function remove (k: 'k) (m: fmap 'k 'v) : fmap 'k 'v

function find (k: 'k) (m: fmap 'k 'v) : 'v
```

Axioms

```
axiom add_contents_k:
 forall k v, m: fmap 'k 'v. (add k v m)[k] = v

axiom add_contents_other:
 forall k v, m: fmap 'k 'v, k1. mem k1 m -> k1 <> k ->
  (add k v m)[k1] = m[k1]

axiom find_def:
 forall k, m: fmap 'k 'v. mem k m -> find k m = m[k]

axiom remove_contents:
 forall k, m: fmap 'k 'v, k1. mem k1 m -> k1 <> k ->
  (remove k m)[k1] = m[k1]
```

Finite map implementations

- consider an implementation of maps as lists of tuples
- should users of this implementation have to reason about lists?
- if we use contracts based on `FMap`, they only need the axioms
- we implicitly have an **abstraction map** from lists to `fmaps`

```
let rec sqrt (r: real) (eps: real) (ghost n:int)
  (ghost eps0:real) : real
requires{ 0.0 <= r }
requires{ eps0 > 0.0 /\ Int.(>=) n 1 }
requires{ eps = (FromInt.from_int n) * eps0 }
variant{Int.(-)(Truncate.ceil (MinMax.max r 1.0 / eps0)) n
ensures{ result * result <= r
 < (result + eps) * (result + eps) }
= if r < eps && 1.0 < eps then 0.0 else
begin
assert { FromInt.from_int n * eps0 <= MinMax.max r 1.0 };
assert { 1.0 / eps0 > 0.0 };
assert { FromInt.from_int n * eps0 * (1.0 / eps0) <=
  MinMax.max r 1.0 * (1.0 / eps0)};
assert { FromInt.from_int n * eps0 / eps0 <=
  MinMax.max r 1.0 / eps0};
assert { FromInt.from_int n <= MinMax.max r 1.0 / eps0 };
let r' = sqrt r (2.0 * eps) (Int.(*) 2 n) eps0 in
if (r' + eps) * (r' + eps) <= r then r' + eps else r'
end
```