VSTTE 2012 Verification Competition

Jean-Christophe Filliâtre, Andrei Paskevich, Aaron Stump

8–10 November 2012

For every problem, except Problem 2, we give an informal description and a suggested implementation in pseudocode. It is up to the participants to come up with a suitable specification (loop invariants, data structure invariants, pre- and postconditions, etc.). In Problem 2, on the contrary, we provide a formal specification and let the participants devise a conforming implementation.

In the problem statements, a type `int` is used to denote integers. Participants are free to interpret this type either as a type of mathematical, unbounded integers, or as a type of machine, bounded integers. In the latter case, participants may have to strengthen specifications to be able to prove the absence of overflows. We assume arrays to be indexed from 0.

1 Two-Way Sort (*50 points*)

We want to sort an array of Boolean values (assuming `false` < `true`) using only swaps.

Implementation.

```
swap(a: array of boolean, i: int, j: int) :=
  t <- a[i];
  a[i] <- a[j];
  a[j] <- t

two_way_sort(a: array of boolean) :=
  i <- 0;
  j <- length(a) - 1;
  while i <= j do
    if not a[i] then
      i <- i+1
    elseif a[j] then
      j <- j-1
    else
      swap(a, i, j);
      i <- i+1;
      j <- j-1
    endif
  endwhile
```

Verification Tasks.

1. *Safety.* Verify that every array access is made within bounds. *5 points.*
2. *Termination.* Prove that function `two_way_sort` always terminates. *5 points.*
3. *Behavior.* Verify that after execution of function `two_way_sort`, the following properties hold.
   (a) Array `a` is sorted in increasing order. *20 points.*
   (b) Array `a` is a permutation of its initial contents. *20 points.*

2 Combinators (*100 points*)

The Turing-complete language of S and K combinators is sometimes used in compilation of functional programming languages. For this problem, you will write a simple interpreter for combinators, and prove several properties about this interpreter. The syntax of combinators is defined by

$$\textit{terms} \qquad t \ ::= \ \mathsf{S} \mid \mathsf{K} \mid (t\ t)$$

We will use a call-by-value (CBV) interpreter, based on this definition of contexts:

$$\begin{array}{lrcl} \textit{CBV contexts} & C & ::= & \square \mid (C\ t) \mid (v\ C) \\ \textit{values} & v & ::= & \mathsf{K} \mid \mathsf{S} \mid (\mathsf{K}\ v) \mid (\mathsf{S}\ v) \mid ((\mathsf{S}\ v)\ v) \end{array}$$

The expression $C[t]$ denotes the term that we obtain by replacing $\square$ with t in context C. It is recursively defined as follows:

$$\begin{array}{rcl} \square[t] & = & t \\ (C\ t_1)[t] & = & (C[t]\ t_1) \\ (v\ C)[t] & = & (v\ C[t]) \end{array}$$

The single-step reduction relation $\to$ can then be defined this way:

$$\begin{array}{lcl} C[((\mathsf{K}\ v_1)\ v_2)] & \to & C[v_1] \\ C[(((\mathsf{S}\ v_1)\ v_2)\ v_3)] & \to & C[((v_1\ v_3)\ (v_2\ v_3))] \end{array}$$

The reduction relation is the reflexive transitive closure $\to^*$ of the single-step reduction relation. We will also write $t \not\to$ if there is no t' such that $t \to t'$. For example, $\mathsf{K} \not\to$.

Implementation Task.

1. Define a data type for representing combinator terms and implement a function `reduction` which, when given a combinator term t as input, returns a term t' such that $t \to^* t'$ and $t' \not\to$, or loops if there is no such term. *10 points.*

Verification Tasks.

1. Prove that if `reduction`(t) returns t', then $t \rightarrow^* t'$ and $t' \not\rightarrow$. *40 points.*
2. Prove that function `reduction` terminates on any term which does not contain S. *25 points.*
3. Consider the meta-language function *ks* defined by

   $$\begin{array}{lcl} ks\ 0 & = & \mathsf{K} \\ ks\ (n+1) & = & ((ks\ n)\ \mathsf{K}) \end{array}$$

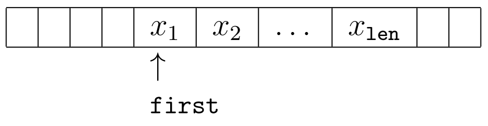
   Prove that `reduction` applied to the term $(ks\ n)$ returns K when n is even, and $(\mathsf{K}\ \mathsf{K})$ when n is odd. *25 points.*

3 Ring Buffer (*150 points*)

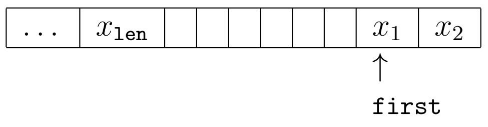
We want to implement a *queue* data structure using a ring buffer that can be described with the following type declaration:

```
type ring_buffer = record
  data : array of int;    // buffer contents
  size : int;             // buffer capacity
  first: int;             // queue head, if any
  len  : int;             // queue length
end
```

A ring buffer is an array `data` of fixed size `size` where we store the `len` queue elements starting from index `first`. There are two possible situations. If `first` + `len` $\leq$ `size`, then the queue elements are stored consecutively within `data`:



On the contrary, if `first` + `len` $>$ `size`, then the queue wraps over the end of `data` and continues from index 0:



A ring buffer is said to be *full* when `len` = `size` and *empty* when `len` = 0.

In this description, we fixed the type of queue elements to type `int`. However, participants are encouraged to implement and prove a generic data structure, using for example polymorphic types, type classes, generics, etc.

Implementation. You are supposed to implement the following operations:

`create`(n) takes a positive integer n as argument and returns a new ring buffer with size n and length 0 (no element).

```
create(n: int): ring_buffer :=
  return new ring_buffer(data = new array[n] of int;
                         size = n;
                         first = 0;
                         len = 0)
```

`clear`(b) takes a ring buffer b as argument and empties it.

```
clear(b: ring_buffer) :=
  b.len <- 0
```

`head`(b) takes a ring buffer b as argument and returns the first element in the queue.

```
head(b: ring_buffer): int :=
  return b.data[b.first]
```

You may either impose as a precondition that b is not empty or raise an exception when b is empty.

`push`(b, x) takes a ring buffer b and an element x as arguments, and adds x at the end of the queue.

```
push(b: ring_buffer, x: int) :=
  b.data[(b.first + b.len) mod b.size] <- x;
  b.len <- b.len + 1
```

You may either impose as a precondition that b is not full or raise an exception when b is full.

`pop`(b) takes a ring buffer b as argument, pops off its head (first element in the queue), and returns it.

```
pop(b: ring_buffer): int :=
  r <- b.data[b.first];
  b.first <- (b.first + 1) mod b.size;
  b.len <- b.len - 1;
  return r
```

You may either impose as a precondition that b is not empty or raise an exception when b is empty.

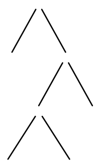
Verification Tasks.

1. *Safety.* Verify that every array access is made within bounds. *30 points.*
2. *Behavior.* Verify the correctness of your implementation w.r.t. the first-in first-out semantics of a queue. *100 points.*
3. *Harness.* The following test harness should be verified. *20 points.*

```
test (x: int, y: int, z: int) :=
  b <- create(2);
  push(b, x);
  push(b, y);
  h <- pop(b); assert h = x;
  push(b, z);
  h <- pop(b); assert h = y;
  h <- pop(b); assert h = z;
```

4 Tree Reconstruction (*150 points*)

An unlabeled binary tree can be uniquely characterized by the list of depths of its leaves. For example, the following tree:



corresponds to the list $1, 3, 3, 2$. On the contrary, there are list of positive integers that correspond to no binary tree. We want to recreate a binary tree given a list of leaf depths, or fail if there is no such tree.

We start with two abstract data types for trees and integer lists. A binary tree is either a leaf or a binary node. For the purpose of implementation, we provide two constructors:

```
type tree
Leaf(): tree
Node(l:tree, r:tree): tree
```

Regarding lists, we assume the following signature:

```
type list
is_empty(s: list): boolean  // check whether s is empty
head(s: list): int          // returns the head of the list
pop(s: list)                // pops the head of the list
```

Here, the list data structure is imperative: the pop function removes the head element of the list, if any. Functions is_empty and head do not modify the list. We accept solutions that use any other data structure for integer lists, including purely applicative ones (with appropriate changes to the code below).

Implementation. The following algorithm computes the binary tree or reports failure. You can implement `fail` either by raising an exception, by returning `null`, or in any other reasonable way.

```
build_rec(d: int, s: list): tree :=
  if is_empty(s) then fail; endif
  h <- head(s);
  if h < d then fail; endif
  if h = d then pop(s); return Leaf(); endif
  l <- build_rec(d+1, s);
  r <- build_rec(d+1, s);
  return Node(l, r)

build(s: list): tree :=
  t <- build_rec(0, s);
  if not is_empty(s) then fail; endif
  return t
```

Verification Tasks.

1. *Soundness.* Verify that whenever function `build` successfully returns a tree the depths of its leaves are exactly those passed in the argument list. *30 points.*
2. *Completeness.* Verify that whenever function `build` reports failure there is no tree that corresponds to the argument list. *60 points.*
3. *Termination.* Prove that function `build` always terminates. *30 points.*
4. *Harness.* The following test harness should be verified:
   - Verify that `build` applied to the list $1, 3, 3, 2$ returns the tree `Node(Leaf, Node(Node(Leaf, Leaf), Leaf))`. *10 points.*
   - Verify that `build` applied to the list $1, 3, 2, 2$ reports failure. *30 points.*

5 Breadth-First Search (150 points)

A directed graph is a set of vertices together with a set of arcs. If x and y are vertices, we note $x \to y$ when there is an arc from x to y and we call y a successor of x. A path of length n from a vertex x_0 to a vertex x_n is a sequence of vertices $x_0, \ldots, x_n$, possibly with repetitions, such that

$$x_0 \to x_1 \to x_2 \to \ldots \to x_{n-1} \to x_n.$$

The purpose of this problem is to verify an algorithm computing the length of the shortest path from one vertex to another using breadth-first search.

For the purpose of the implementation, we assume abstract data types for vertices and finite sets of vertices:

```
type vertex
type vertex_set
```

Then the graph is defined by a unique function `succ` returning the successors of a given vertex:

```
succ(v: vertex): vertex_set
```

Implementation. The following algorithm computes the length of the shortest path from `source` to `dest`, reports failure, or diverges. Variables `V`, `C`, and `N` are finite sets of vertices; `{}` stands for the empty set and `{`x`}` for the singleton set containing x. Variable `d` is an integer.

```
bfs(source: vertex, dest: vertex): int :=
  V <- {source};
  C <- {source};
  N <- {};
  d <- 0;
  while C is not empty do
    remove one vertex v from C;
    if v = dest then return d; endif
    for each w in succ(v) do
      if w is not in V then
        add w to V;
        add w to N;
      endif
    endfor
    if C is empty then
      C <- N;
      N <- {};
      d <- d+1;
    endif
  endwhile
  fail "no path"
```

Verification Tasks.

1. *Soundness.* Verify that whenever function `bfs` returns an integer n this is indeed the length of the shortest path from `source` to `dest`. *100 points.*

   A partial score of *50 points* is attributed if it is only proved that there exists a path of length n from `source` to `dest`.

2. *Completeness.* Verify that whenever function `bfs` reports failure there is no path from `source` to `dest`. *50 points.*