VSComp: The Verified Software Competition

Peter Müller (ETH Zurich) and N. Shankar (SRI)

Computer Science Laboratory
SRI International
Menlo Park, CA

August 18, 2010

Robin and Amir

Introducing VSComp

- The competition is between teams of up to three people, armed with one or more verification tools.
- We describe five verification exercises in this presentation. All require total correctness proofs.
- Anyone can participate, and student teams are encouraged.
- An analysis of the results will be announced at the Tools & Experiments workshop on Thursday.
- Valentin Wüstholz helped select and prepare the problems.
- Gary Leavens, Peter Müller, and Shankar are the judges
- These slides are available on `http://www.macs.hw.ac.uk/vstte10/comp.pdf`.

Ground Rules

- Teams can select a name for themselves and nominate a leader.
- Teams can only ask questions and receive answers from us in public.
- You have two hours from 1600 to 1800 hours to work on these problems.
- Complete or partial solutions (code, specification, proof) should be emailed to `peter.mueller@inf.ethz.ch`.
- Each email should consist of a solution to a one of the problems and should be labeled VSComp-*teamname*-*problem*#.

The Solution

- In solving the problems, you have to formalize the specification, write the program, construct and verify the proof with the aid of a verification tool.
- You don't have to worry about numeric overflows or underflows or resource bounds, but other kinds of uncaught exceptions must be shown to be absent.
- It would be helpful if you could execute your code on the test cases.
- Solutions must be reproducible without change on the verification tools that you have used.
- We also need the complete transcript of the verification so that we can examine the steps and observe the run times. Any other documentation will be helpful.

Evaluation of Submissions

- At the end of the competition, we will forward the solutions to all the participants.
- Each team can send us their evaluation of the submissions made by others.
- The judges (Peter, Shankar, and Gary) will evaluate both the submissions and the comments from the teams in making the final evaluation.
- Our evaluation of results is subjective (completeness, elegance, automation).
- We will not be ranking the submissions, but presenting our overall assessment of the candidate solutions for each problem.

Problem Format

- Description: What does the program compute?
- Properties to prove: Informal statement of properties. (You have to formalize these in your own terms, and show termination of all the functions used.)
- Pseudocode: A candidate program that is a rough guide, but you can verify a different program with the same behavior.
- Test Cases: Examples to illustrate how the program should work.

Problem 1: Sum and Maximum

- Description: Given an N-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array.
- Properties: Given that $N \geq 0$ and $\texttt{a[i]} \geq 0$ for $0 \leq \texttt{i} < \texttt{N}$, prove the post-condition that $\texttt{sum} \leq \texttt{N} * \texttt{max}$.
- Pseudocode:

```
int sum, max = 0;
int i;
for (i=0; i<N; i++){
  if (max < a[i]){
    max = a[i];
  }
  sum += a[i];
}
```

- Test Case: With the array $9, 5, 0, 2, 7, 3, 2, 1, 10, 6$, N is 10, max is 10, and the sum is 40.

Problem 2: Inverting an Injection

- Description: Invert an injective array A on N elements in the subrange from 0 to $N-1$, i.e., the output array B must be such that $B[A[i]] = i$ for $0 \leq i < N$.
- You can assume that A is surjective.
- Properties: Show that the resulting array is also injective. For bonus points, you can demonstrate other properties, e.g., that `A` and `B` are inverses.
- Pseudocode:

```
int A[];
 for (i=0; i<N;  i++){
      B[A[i]] = i
    }
```

- Test: If `A` is $9, 3, 8, 2, 7, 4, 0, 1, 5, 6$, then output `B` should be $6, 7, 3, 1, 5, 8, 9, 4, 2, 0$.

Problem 3: Searching a Linked List

- Probem: Given a linked list representation of a list of integers, find the index of the first element that is equal to 0.
- Properties: You have to show that the program returns an index *i* equal to the length of the list if there is no such element. Otherwise, *i*'th element of the list must be equal to 0, and all the preceding elements must be non-zero.
- Pseudocode: You may use linked list representation given with Problem 5.

```
jj = ll;
int i = 0;
while (jj != null && jj.head != 0){
  jj = jj.next;
  i++;
}
return i;
```

Problem 4: N-Queens

- Problem: Write a program to place N queens on an $N \times N$ chess board so that no queen can capture another one with a legal move.
- The algorithm returns a placement if there is a solution, and an empty board, otherwise. You can represent the empty board with a flag or a null pointer.
- A placement is given by a `board` which is an N-element array where the i'th element is j, when the queen is placed in the j'th row for the i'th column.
- Properties: The post-condition should establish that when the algorithm returns a placement, it is legal, and if it returns an empty board, there is no solution.
- Thus, with $N = 2$, the result should be empty, whereas with $N = 4$, there should be a legal placement.

Legal Board

A legal board is defined by `IsConsistent`.

```
def IsConsistent(int[] board, int pos) : boolean {
    for (int q = 0; q < pos; q++) {
        if (!((board[q] != board[pos])
                && (board[q] - board[pos] != pos - q)
                && (board[pos] - board[q] != pos - q))) {
            return false;
        }
    }
    return true;
}
```

Search

The search for a consistent board position is defined recursively over the columns (pos) and scanning each position for a row value (i).

```
def Search(int pos, int[] board) : int[] {
    if (pos == board.length) {
        return board;
    }

    for (int i = 0; i < board.length; i++) {
        board[pos] = i;

        if (IsConsistent(board, pos)) {
            int[] s = Search(pos + 1, board);

            if (s != null) {
                return s;
            }
        }
    }
```

Problem 5: Amortized Queue

- An applicative queue with a good amortized complexity can be implemented using a linked list.
- The queue structure supports the operations (pseudocode to follow)
  1. `Enqueue(item: T)`: Place an element at the rear of the queue
  2. `Tail()`: Return the queue without the first element
  3. `Front()`: Return the first element of the queue.
- The queue is implemented as a record with two fields: **front** and **rear** which are linked lists so that the `Front` operation returns the first element in the list **front** and `Tail` returns a new queue with **front** as the tail of the original **front** list. The `Enqueue` operation teturns a new queue by inserting an element at the head of the list **rear**.

Properties

- You have to show that the implementation maintains the invariant that `queue.rear.length` $\leq$ `queue.front.length`.
- You also have to show that a client invoking these operations observes an abstract queue given by a sequence.

Pseudocode for Linked Lists

```
class LinkedList<T> {

   var head: T;

   var tail: LinkedList<T>;

   var length: int;


   /**
    * Constructs an empty linked list.
    */
   LinkedList() {
       tail = null;
       length = 0;
   }
```

Pseudocode for List Cons

```
/**
 * Returns a new linked list whose first element (head)
 * is "d" and whose tail is "this".
 */
def Cons(d: T) : LinkedList<T> {
    r = new LinkedList<T>;

    r.head = d;
    r.tail = this;
    r.length = length + 1;
    return r;
}
```

Pseudocode for List Concatenation

```
/**
 * Returns a new list that is the concatenation of this list and
 * the list "end".
 */
def Concat(end: LinkedList<T>) : LinkedList<T> {
    if (length == 0) {
        r = end;
    } else {
        var c = tail.Concat(end);
        r = c.Cons(head);
    }
}
```

Pseudocode for List Reverse

```
    /**
     * Returns a new list that is the reverse of this list.
     */
    def Reverse() : LinkedList<T> {
        var r;
        if (length == 0) {
            r := new LinkedList<T>;
        } else {
            r = tail.Reverse();

            var e = new LinkedList<T>;
            e = e.Cons(head);

            r = r.Concat(e);
        }
        return r;
    }
}
```

Pseudocode for Applicative Queue Constructor

```
class AmortizedQueue<T> {

    // The front of the queue.
    var front: LinkedList<T>;

    // The rear of the queue (stored in reversed order).
    var rear: LinkedList<T>;


    /**
     * Constructs an empty queue.
     */
    AmortizedQueue() {
        front = new LinkedList<T>;
        rear = new LinkedList<T>;
    }
    ⋮
    }
```

Pseudocode for Applicative Queue Constructor

```
/**
 * Constructs an new queue whose front is 'front' and whose rear
 * is 'rear'.
 *
 * 'front' and 'rear' should be non-null.
 */
AmortizedQueue(front: LinkedList<T>, rear: LinkedList<T>) {
    if (rear.length <= front.length) {
        this.front = front;
        this.rear = rear;
    } else {
        var f;
        f = rear.Reverse();
        this.front = front.Concat(f);

        this.rear = new LinkedList<T>;
    }
}
```

Pseudocode for Queue Operations

```
/**
 * Returns the first element of a non-empty queue.
 */
def Front() : T {
    return front.head;
}


/**
 * Returns a new queue that contains all elements of
 * this queue (non-empty) except for the first element.
 */
def Tail() : AmortizedQueue<T> {
    return new AmortizedQueue<T>(front.tail, rear);
}
```

Pseudocode for Queue Operations

```
/**
 * Returns a new queue that contains all elements of this queue
 * and an additional element "item" at the rear of the queue.
 */
def Enqueue(item: T) : AmortizedQueue<T> {
    var r;
    r = rear.Cons(item);
    return new AmortizedQueue<T>(front, r);
}
```