CS655: Final Exam 2000 Solutions

You have 90 minutes to complete the written part of the exam. Write out answers to all the questions on the answer sheets. Question values are percents of the total exam grade, the written part counts for 50% of the exam.

Operational Semantics and Concurrency

In Lecture 19, we defined a standard operational semantics for Bjarfk, a language with concurrency primitives shown below:

Program ::= *Instruction**	Program is a sequence of instructions Instructions are numbered from 0. Execution begins at instruction 0, and completes with the initial thread halts.
Instruction ::= *Loc* := *Expression*	*Loc* gets the value of *Expression*
| *Loc* := FORK *Expression*	*Loc* gets the value of the ThreadHandle returned by FORK; Starts a new thread at instruction numbered *Expression*.
| JOIN *Expression*	Waits until thread associated with ThreadHandle *Expression* completes.
| HALT	Stop thread execution.

Expression ::= *Literal* | *Expression* + *Expression* | *Expression* * *Expression*

The relevant parts of the operational semantics (identical to Lecture 19) are shown on the next page.

Ben Bitdiddle would like to extend Bjarfk to support SUSPEND and RESUME. To prevent mistakes, only a running thread may be suspended, and only a suspended thread may be resumed. The new language, Bsjarrfk adds the following instructions to Bjarfk:

Instruction ::= SUSPEND *Expression*
Temporarily stop execution of the thread identified by the ThreadHandle *Expression.* The suspended thread must not execute any more instructions until it is used as an operand of a RESUME instruction.
| RESUME *Expression*
Restart the thread identified by the ThreadHandle *Expression*. The thread must currently be suspended.

1. (15%) Describe how to modify the operational semantics to describe Bsjarrfk. You may modify the configurations of the virtual machine. Show the new transition rules for Assignment, SUSPEND, and RESUME. If you find the informal descriptions to be ambiguous, clarify any ambiguities in a sensible way in your rules.

Answer:

We can describe suspend by keeping additional information about each thread, indicating whether it is suspended or running. We change the configurations to be:

C = Instructions x Threads x RegisterFile
where Instructions: array[Instruction]
Threads: array[<ThreadHandle, PC, status ∈ { suspended, running } >]
RegisterFile: array[int]

The input function will set the status of the main thread to running:

Threads = [<0, 0, running>]

The final configurations not need to match a 3-tuple:

F = Instructions x Threads x RegisterFile
where <0, PC, running> ∈ Threads and Instructions[PC] = **HALT**

The transition rules need to be extended to check the thread is running:

Assignment:

<t, PC_t, running> ∈ Threads & Instructions[PC_t] = *Loc* := *Value*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile' >
where Threads = Threads – {<t, PC_t, running >} + {<t, PC_t + 1, running }
RegisterFile'[n] = RegisterFile[n] if n ≠ *Loc*
RegisterFile'[n] = value of *Value* if n = *Loc*

The rule for SUSPEND:

<t, PC_t, running> ∈ Threads & Instructions[PC_t] = SUSPEND *Value*
& <value, PC_S, running> ∈ Threads & value is the ThreadHandle value denoted by *Value*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile' >
where
if (value = t) // Note: PC_S must equal PC_t (it would be worth checking this)
Threads = Threads – {<t, PC_t, running >} + {<t, PC_t + 1, suspended }
else
Threads = Threads – {<t, PC_t, running >} + {<t, PC_t + 1, running }
– {<value, PC_S, running >} + {<value, PC_S, suspended> }

Note that we need to handle the case where a thread suspends itself specially. The most sensible solution is to suspend the thread with the PC advanced, as shown by this rule.

The rule for RESUME:

<t, PC_t, running> ∈ Threads & Instructions[PC_t] = RESUME *Value*
& <value, PC_S, suspended> ∈ Threads & value is the ThreadHandle value denoted by *Value*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile' >
where
Threads = Threads – {<t, PC_t, running >} + {<t, PC_t + 1, running }
– {<value, PC_S, suspended>} + {<value, PC_S, running> }

Operational Semantics of Bjarfk

C = Instructions x Threads x RegisterFile
where Instructions: array[Instruction]
Threads: array[<ThreadHandle, PC>]
RegisterFile: array[int]

Input Function: I: Program → C

C = Instructions x Threads x RegisterFile
where
For a Program with *n* instructions from 0 to *n - 1*:
Instructions[m] = Program[m] for m >= 0 && m < n
Instructions[m] = **ERROR** otherwise
RegisterFile[*n*] = 0 for all integers *n*
Threads = [<0, 0>]
The top thread (identified with ThreadHandle = 0) starts at PC = 0.

Final Configurations

F = Instructions x Threads x RegisterFile
where <0, PC> ∈ Threads and Instructions[PC] = **HALT**

Transition Rules

Assignment:

<t, PC_t> ∈ Threads & Instructions[PC_t] = *Loc* **:=** *Value*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile' >
where Threads = Threads – {<t, PC_t>} + {<t, PC_t + 1}
RegisterFile'[n] = RegisterFile[n] if n ≠ *Loc*
RegisterFile'[n] = value of *Value* if n = *Loc*

FORK:

<t, PC_t> ∈ Threads & Instructions[PC_t] = *Loc* := **FORK** *Literal*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile' >
where
Threads = Threads – {<t, PC_t>} + {<t, PC_t + 1} + { <nt, *Literal*> }
where <nt, *x*> ∉ Threads for all possible *x*.
RegisterFile'[n] = RegisterFile[n] if n ≠ *Loc*
RegisterFile'[n] = value of ThreadHandle nt if n = *Loc*

JOIN:

<t, PC_t> ∈ Threads & Instructions[PC_t] = **JOIN** *Value* & <v, PC_v> ∈ Threads
& Instructions[PC_v] = **HALT** & v = value of *Value*

< Instructions x Threads x RegisterFile > ⇒ < Instructions x Threads' x RegisterFile >
where Threads = Threads – {<t, PC_t>} + {<t, PC_t + 1}

Questions continue on next page

Reasoning about Datatypes

Consider the polygon type specified below. Note that we assume constructors are defined outside the datatype, so this is a complete specification for the polygon type.

polygon = **datatype is** equivalent

A polygon is with *n* sides is described by [$<l_0, a_0>$, $<l_1, a_1>$, …, $<l_{n-1}, a_{n-1}>$] where l_i is an integer giving the length of the i^{th} edge of the polygon and a_i is an integer giving the angle in degrees between the i^{th} edge and the $((i + 1) \bmod n)^{th}$ edge.

For example, the rectangle with edges length 5 and 6 could be represented by the polygon [<5, 90>, <6, 90>, <5, 90>, <6, 90>] or the equivalent polygon [<6, 90>, <5, 90>, <6, 90>, <5, 90>].

equivalent = **proc** (arg: polygon) returns (bool)

modifies nothing

ensures Returns true if and only if **this** and *arg* represent identical polygons.

Two polygons, p = [$<p_0, a_0>$, $<p_1, a_1>$, …, $<p_{n-1}, a_{n-1}>$] and

q = [$<q_0, b_0>$, $<q_1, b_1>$, …, $<q_{m-1}, b_{m-1}>$]

are equivalent if m = n and

$\exists$ k: $0 \le k < n$ such that $\forall$ i, $0 \le i < n$: $p_i = q_{(i + k) \bmod n} \wedge a_i = b_{(i + k) \bmod n}$.

Lucy Reasoner produced a Java implementation of **polygon** shown on the next page.

2. (5%) What is the rep invariant for **polygon**? Do not worry about any geometrical constraints on polygons, only what is needed for the abstraction function and code.

Answer:
I (r) ≡ r.edges.length = r.angles.length

The only thing we need in the rep invariant is that the size of the edges and angles arrays are equal. A correct answer could include geometrical constrains also (such as the number of edges being at least 3), but the question did not ask for this, and the correct geometric constraints are very complicated (e.g., knowing the edges don't cross based on their angles and lengths, and knowing the polygon is closed).

3. (20%) Sketch a proof that the result returned on line 18 satisfies the **ensures** clause. You may assume that Lucy has already proved the predicate Q on line 14 is true if the predicate P on line 8 is true. Your proof need not be at the low level of detail, but should explain how you prove the predicate P on line 8 is true and how you prove the result satisfies the postcondition.

You may find the partial correctness rule for **while** useful:

$$\frac{\begin{array}{l} P \Rightarrow Inv, \\ Inv \ \{ \ Pred \ \} \ Inv, \\ Inv \wedge Pred \ \{ \ StatementList \ \} \ Inv, \\ (Inv \ \& \ \neg Pred) \Rightarrow Q \end{array}}{P \ \{ \ \textbf{while} \ (Pred) \ \textbf{\{} \ StatementList \ \textbf{\}} \ \} \ Q}$$

Answer:

To show the result is correct, we need to construct a proof that the post-condition of the outer while loop (from line 5 – line 17) leaves result with a value that satisfies the **ensures** clause:

result ≡ m = n and
∃ k: 0 ≤ k< n such that ∀ i, 0 ≤ i< n: $p_{i} = q_{(i + k) \bmod n} \wedge a_{i} = b_{(i + k) \bmod n}$.

The ensures clause is at the level of abstract objects, so we will need to prove a similar property in terms of the concrete rep, and then use the abstraction function to map it to the ensures clause. The similar property we need is:

result ≡ **this**.edges.length = arg.edges.length and
∃ k: 0 ≤ k< this.edges.length such that
∀ i, 0 ≤ i< this.edges.length: this.edges[i] = args.edges[(i + k) mod this.edges.length]
∧ this.angles[i] = arg.angles[(i+k) mod this.edges.length]

The abstraction function is
A(r) = [< $<p_0, a_0>$, $<p_1, a_1>$, …, $<p_{n-1}, a_{n-1}>$] where
n = r.edges.length and p_i = r.edges[i] and a_i = r.angles[i]

So, using A(**this**) and A(arg) we can map the concrete property to:

result ≡ m = n and
∃ k: 0 ≤ k< n such that ∀ i, 0 ≤ i< n: $p_{i} = q_{(i + k) \bmod n} \wedge a_{i} = b_{(i + k) \bmod n}$

which matches the ensures clause exactly. (**Note**: it was not necessary to show the mapping between the concrete object and abstract ensures clause to get full credit for this question.)

Now, we need to show that the desired property is true for result returned on line 18. We need to construct an axiomatic proof that the post-condition of the outer loop implies the result property. For this, we need a loop invariant. Informally, the loop invariant should tell us that result is true if all the edges and angles matched correctly on a previous iteration of the loop. We can write this as:

$$\text{result} = \bigvee_{j=0}^{k-1} \left(\bigwedge_{i=0}^{nedges-1} (\text{this.edges}[i] = \text{arg.edges}[(i + j) \% nedges] \wedge \text{this.angles}[i] = \text{arg.angles}[(i + j) \% nedges]) \right)$$

We have taken the expression for isequiv in Q (line 4) for the inner predicate, and replaced k with j. We have a problem if k = 0, since the normal meaning of an or evaluation with no terms is **true**, but result should be **false**. So, we need to add a conjunct: ∧ k > 0. (One could argue strictly that two polygons with no edges should be view are equivalent, and the result should be true according to the ensures clause, so the code is incorrect if the polygons have zero edges; a better solution would be to add a requirement that the size of edges must be > 0.)

In addition, our invariant needs to preserve the values of nedges, this, arg and ensure k ≤ nedges (since we will need this for the post-condition).

So, the full invariant is:

$$\text{Inv} \equiv \text{result} = k > 0 \wedge \bigvee_{j=0}^{k-1} \left(\bigwedge_{i=0}^{nedges-1} (\text{this.edges}[i] = \text{arg.edges}[(i + j) \% nedges] \wedge \text{this.angles}[i] = \text{arg.angles}[(i + j) \% nedges]) \right)$$
$$\wedge nedges = \textbf{this}.\text{edges.length} \wedge nedges = \text{arg.edges.length}$$
$$\wedge nedges = \textbf{this}.\text{angles.length} \wedge nedges = \text{arg.angles.length}$$
$$\wedge \text{this}_0 = \text{this} \wedge \text{arg}_0 = \text{arg} \wedge k \leq nedges$$

To complete the proof, we need to show all the antecedents for the rule for while.

Let's use:

$P \equiv k = 0 \wedge nedges = \textbf{this}.\text{edges.length} \wedge nedges = \text{arg.edges.length}$
$\wedge nedges = \textbf{this}.\text{angles.length} \wedge nedges = \text{arg.angles.length}$
$\wedge \text{this}_0 = \text{this} \wedge \text{arg}_0 = \text{arg} \wedge \text{result} = \textbf{false}$

We get nedges = **this**.edges.length from the assignment on line 1.
We get nedges = arg.edges.length from line 2 – if this is false, we would never real the loop.
We get k = 0 from line 3, and result = **false** from line 4.
We get nedges = **this**.angles.length ∧ nedges = arg.angles.length from the rep invariant.
We $\text{this}_0 = \text{this} \wedge \text{arg}_0 = \text{arg}$ since nothing modifies either this or arg. (We can introduce $\text{this}_0 = \text{this} \wedge \text{arg}_0 = \text{arg}$ before the body of the method.)

P ⇒ Inv:
$k = 0 \wedge \text{result} = \textbf{false} \Rightarrow \text{result} = k > 0 \wedge \ldots$
$k = 0 \Rightarrow k \leq nedges$ (since we know nedges = **this**.edges.length which cannot be negative.)
The rest of the terms match exactly.

Inv { *Pred* } Inv: trivially

Inv ∧ *Pred* { *StatementList* } Inv:
Lucy already proved P8 ⇒ Q14, so we need to show

Inv ∧ k < nedges { **int** i = 0; **boolean** isequiv = **true**; }
P8: nedges = **this**.edges.length ∧ nedges = arg.edges.length
∧ nedges = **this**.angles.length ∧ nedges = arg.angles.length
∧ i = 0 ∧ isequiv
$\wedge k_0 = k \wedge nedges_0 = nedges \wedge result_0 = result \wedge this_0 = this \wedge arg_0 = arg$

This is easy since the invariant (and no modifications of this or arg in the code) gives:
nedges = **this**.edges.length ∧ nedges = arg.edges.length
∧ nedges = **this**.angles.length ∧ nedges = arg.angles.length
$\wedge nedges_0 = nedges \wedge result_0 = result \wedge this_0 = this \wedge arg_0 = arg$

and the two assignments give: i = 0 ∧ isequiv.

Next, we show:
Q14 = $k_0 = k \wedge nedges_0 = nedges \wedge result_0 = result \wedge this_0 = this \wedge arg_0 = arg$

$$\wedge\ isequiv = \bigwedge_{i=0}^{nedges-1} (this.edges[i] = arg.edges[(i + k) \% nedges] \wedge this.angles[i] = arg.angles[(i + k) \% nedges])$$

{ if (isequiv) result = **true**; k = k + 1; } Inv

If isequiv is false, result does not change, and we increment k. Since we are or-ing with false, the invariant is preserved.

If isequiv is true, result becomes true, and we increment k. Since we are or-ing with true, the result clause in the invariant becomes true, and the invariant is preserved.

The k ≤ nedges term is satisfied because the predicate requires k < nedges before k is incremented.

(*Inv* & ¬*Pred*) ⇒ Q:

k ≤ nedges & ¬ (k < nedges) ⇒ k = nedges.

Substituting in Inv gives,

$$Inv \equiv result = nedges > 0 \wedge \bigvee_{j=0}^{nedges-1} \left(\bigwedge_{i=0}^{nedges-1} (this.edges[i] = arg.edges[(i + j) \% nedges] \wedge this.angles[i] = arg.angles[(i + j) \% nedges]) \right)$$
$$\wedge\ nedges = \textbf{this}.edges.length \wedge nedges = arg.edges.length$$
$$\wedge\ nedges = \textbf{this}.angles.length \wedge nedges = arg.angles.length$$
$$\wedge\ this_0 = this \wedge arg_0 = arg \wedge k \le nedges$$

The get the result condition, we need to convert the outer ∨ to

∃ k: 0 ≤ k< this.edges.length such that
∀ i, 0 ≤ i< this.edges.length: this.edges[i] = args.edges[(i + k) mod this.edges.length]
∧ this.angles[i] = arg.angles[(i+k) mod this.edges.length]

This is true, since if the outer ∨ is true, there must be some inner term that is true (corresponding to the ∃ k); if the outer ∨ is false, then no inner term is true, so no such k exists. The only discrepancy remaning is the nedges > 0 clause – we can either add a rep invariant to make this always true, or argue that in fact the implementation is incorrect because it returns the wrong result for a 0-edged polygon.

Note: This answer gives far more detail that was necessary to get full credit for this question. To get full credit, you needed to: come up with an invariant for the loop on line 5, explain how you use the rep invariant to prove P8, and explain how you use Q14 to show the loop preserves the invariant for the outer loop.

4. (10%) Lucy specifies a new type, triangle as shown below:

triangle = **datatype is** equivalent
A triangle is described by [l_0, l_1, l_2] where l_i is an integer giving the length of the i^{th} side of the triangle. (Recall that the angles of a triangle are determined by the edges.)

equivalent = **proc** (t: triangle) returns (bool)
modifies nothing
ensures Returns true if and only if **this** and *t* represent identical triangles. Two triangles, p = [p_0, p_1, p_2] and q = [q_0, q_1, q_2] are equivalent if $\exists$ k: $0 \le k < 2$ such that $\forall$ i, $0 \le i < 2$: $p_{i} = q_{(i + k) \bmod n}$.

According to the behavioral notion of subtyping introduced by Liskov and Wing, would it be safe to introduce the type rule: triangle $\subseteq$ polygon? If so, produce a proof that it satisfies the subtype requirements; if not, show how it should be changed in order to satisfy them.

Answer:

No. It is not a subtype because the type of the parameter for the triangle equivalent method violates the contravariant requirement.

To fix the problem, we need to change the type of the parameter to **polygon** (or any supertype of polygon), and change the specification accordingly.

(That is all that you needed for a full credit answer.)

The other requirements of Liskov and Wing's notion of subtyping are:

- The precondition of the subtype methods must be implied by the precondition of the corresponding supertype method. This is trivially true since both preconditions are **true** (there is no requires clause).
- The postcondition of the subtype methods implies the postcondition of the supertype methods. This is harder since we need to convert the abstract triangle to an abstract polygon:

  $[p_0, p_1, p_2] \Rightarrow [\langle l_0, a_0 \rangle, \langle l_1, a_1 \rangle, \langle l_2, a_2 \rangle]$

  We can use $l_i = p_i$ and a_0, a_1, a_2 are the angles determined by the triangle. (Some of you revealed impressive memory of trigonometry actually calculating the angles, but this should have given you some sense that it probably wasn't the correct answer.)

  Then, we need to show that after mapping the abstract objects, the postcondition of triangle.equivalent implies the postcondition of polygon.equivalent. This follows since we replace 2 with n to get:

  $\exists$ k: $0 \le k < n$ such that $\forall$ i, $0 \le i < n$: $p_{i} = q_{(i + k) \bmod n}$.

  and add the clause $\wedge\ a_{i} = b_{(i + k) \bmod n}$ because the triangle sides determine the angles (so triangles with the same sides must also have the same angles.) (Partial credit was rewarded for **Yes** answers that explained this, but it was new necessary to explain any of this since the type property is enough to say it is not a subtype.)

```
class polygon {
    // The rep of a polygon is an array of edge lengths and a corresponding array of angles.
    // The abstraction function is:
    //   A (r) = [ <p0, a0>, <p1, a1>, …, <pn-1, a n-1> ] where
    //       n = r.edges.length and pi = r.edges[i] and ai = r.angles[i]

    private int [] edges;
    private int [] angles;

    public boolean equivalent (polygon arg) {
1       int nedges = this.edges.length;
2       if (arg.edges.length != nedges) return false;

3       int k = 0;
4       boolean result = false;

5       while (k < nedges) {
6           int i = 0;
7           boolean isequiv = true;
8           // P: nedges = this.edges.length ∧ nedges = arg.edges.length
                    ∧ nedges = this.angles.length ∧ nedges = arg.angles.length
                    ∧ i = 0 ∧ isequiv
                   ∧ k0 = k ∧ nedges0 = nedges ∧ result0 = result ∧ this0 = this ∧ arg0 = arg
9           while (i < nedges) {
10              if (this.edges[i] != arg.edges[(i + k) % nedges]) isequiv = false;
11              if (this.angles[i] != arg.angles[(i + k) % nedges]) isequiv = false;
12              i = i + 1;
13          }
14          // Q:       k0 = k ∧ nedges0 = nedges ∧ result0 = result ∧ this0 = this ∧ arg0 = arg

                                    nedges - 1
                        ∧ isequiv = ⋀  (this.edges[i] = arg.edges[(i + k) % nedges]
                                    i = 0    ∧ this.angles[i] = arg.angles[(i + k) % nedges])

                                (This means logical and of all the predicates for i = 0 to i =nedges – 1.)
15          if (isequiv) result = true;
16          k = k + 1;
17      }
18      return result;
19  }
}
```