CS565: 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:

\[ \text{Program ::= Instruction}^* \]

Program is a sequence of instructions

\[ \text{Instruction ::= Loc := Expression} \]

Loc gets the value of Expression

\[ \text{Instruction ::= Loc := FORK Expression} \]

Loc gets the value of the ThreadHandle returned by FORK; Starts a new thread at instruction numbered Expression.

\[ \text{Instruction ::= JOIN Expression} \]

Waits until thread associated with ThreadHandle Expression completes.

\[ \text{Instruction ::= HALT} \]

Stop thread execution.

\[ \text{Expression ::= Literal | Expression + Expression | Expression * Expression} \]

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:

\[ \text{Instruction ::= SUSPEND Expression} \]

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.

\[ \text{Instruction ::= RESUME Expression} \]

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:

\[ C = \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \]

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:

\[ \mathcal{F} = \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \]

where <0, PC, running> ∈ Threads and Instructions[PC] = HALT

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

Assignment:
\[
<\text{t, PC}_t, \text{running}> \in \text{Threads} \land \text{Instructions}[\text{PC}_t] = \text{Loc} := \text{Value} \\
\text{< Instructions } \times \text{Threads } \times \text{RegisterFile } \Rightarrow \text{< Instructions } \times \text{Threads’ } \times \text{RegisterFile’ }> \\
\text{where Threads } = \text{Threads } - \{<\text{t, PC}_t, \text{running}>, <\text{t, PC}_t + 1, \text{running}>\} + \{<\text{t, PC}_t + 1, \text{suspended}>\} \\
\text{RegisterFile’}[n] = \text{RegisterFile}[n] \text{ if } n \neq \text{Loc} \\
\text{RegisterFile’}[n] = \text{value of Value if } n = \text{Loc}
\]

The rule for SUSPEND:
\[
<\text{t, PC}_t, \text{running} > \in \text{Threads} \land \text{Instructions}[\text{PC}_t] = \text{SUSPEND Value} \\
\land <\text{value, PC}_s, \text{suspended} > \in \text{Threads} \land \text{value is the ThreadHandle value denoted by Value} \\
\Rightarrow < \text{Instructions } \times \text{Threads } \times \text{RegisterFile } > \\
\text{where} \\
\text{if (value } = \text{t}) \quad \text{// Note: PC}_s \text{ must equal PC}_t \text{ (it would be worth checking this)} \\
\text{Threads } = \text{Threads } - \{<\text{t, PC}_t, \text{running}>, <\text{t, PC}_t + 1, \text{suspended}>\} \\
\text{else} \\
\text{Threads } = \text{Threads } - \{<\text{t, PC}_t, \text{running}>, <\text{t, PC}_t + 1, \text{running}>\} + \{<\text{value, PC}_s, \text{suspended}>\}
\]

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:
\[
<\text{t, PC}_t, \text{running} > \in \text{Threads} \land \text{Instructions}[\text{PC}_t] = \text{RESUME Value} \\
\land <\text{value, PC}_s, \text{suspended} > \in \text{Threads} \land \text{value is the ThreadHandle value denoted by Value} \\
\Rightarrow < \text{Instructions } \times \text{Threads } \times \text{RegisterFile } > \\
\text{where} \\
\text{Threads } = \text{Threads } - \{<\text{t, PC}_t, \text{running}>, <\text{t, PC}_t + 1, \text{running}>\} + \{<\text{value, PC}_s, \text{suspended}>\} + \{<\text{value, PC}_s, \text{running} >\}
\]
Operational Semantics of Bjarfk

\[ C = \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \]
where

\[ \text{Instructions: array[Instruction]} \]
\[ \text{Threads: array[<ThreadHandle, PC>]} \]
\[ \text{RegisterFile: array[int]} \]

Input Function: \( I: \text{Program} \rightarrow C \)

\[ C = \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \]
where

For a Program with \( n \) instructions from 0 to \( n - 1 \):

\[ \text{Instructions}[m] = \text{Program}[m] \text{ for } m \geq 0 \text{ and } m < n \]
\[ \text{Instructions}[m] = \text{ERROR} \text{ otherwise} \]
\[ \text{RegisterFile}[n] = 0 \text{ for all integers } n \]
\[ \text{Threads} = [ <0, 0> ] \]

The top thread (identified with ThreadHandle = 0) starts at PC = 0.

Final Configurations

\[ f = \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \]
where \( <0, PC> \in \text{Threads} \) and \( \text{Instructions}[PC] = \text{HALT} \)

Transition Rules

Assignment:

\[ \langle t, PC_t \rangle \in \text{Threads} \& \text{Instructions}[PC_t] = \text{Loc} := \text{Value} \]

\[ \langle \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \rangle \Rightarrow \langle \text{Instructions} \times \text{Threads'} \times \text{RegisterFile'} \rangle \]
where

\[ \text{Threads'} = \text{Threads} - \{<t, PC_t>\} + \{<t, PC_t + 1}\}
\[ \text{RegisterFile'}[n] = \text{RegisterFile}[n] \text{ if } n \neq \text{Loc} \]
\[ \text{RegisterFile'}[n] = \text{value of Value} \text{ if } n = \text{Loc} \]

FORK:

\[ \langle t, PC_t \rangle \in \text{Threads} \& \text{Instructions}[PC_t] = \text{Loc} := \text{FORK Literal} \]

\[ \langle \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \rangle \Rightarrow \langle \text{Instructions} \times \text{Threads'} \times \text{RegisterFile'} \rangle \]
where

\[ \text{Threads'} = \text{Threads} - \{<t, PC_t>\} + \{<t, PC_t + 1}\} + \{<nt, \text{Literal}>\}
\[ \text{where } <nt, x> \notin \text{Threads} \text{ for all possible } x. \]
\[ \text{RegisterFile'}[n] = \text{RegisterFile}[n] \text{ if } n \neq \text{Loc} \]
\[ \text{RegisterFile'}[n] = \text{value of ThreadHandle nt if } n = \text{Loc} \]

JOIN:

\[ \langle t, PC_t \rangle \in \text{Threads} \& \text{Instructions}[PC_t] = \text{JOIN Value} \& <v, PC_v> \in \text{Threads} \]
\& \text{Instructions}[PC_v] = \text{HALT} \& v = \text{value of Value} \]

\[ \langle \text{Instructions} \times \text{Threads} \times \text{RegisterFile} \rangle \Rightarrow \langle \text{Instructions} \times \text{Threads'} \times \text{RegisterFile'} \rangle \]
where

\[ \text{Threads'} = \text{Threads} - \{<t, PC_t>\} + \{<t, PC_t + 1}\} \]

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.

\[
\text{polygon} = \text{datatype is equivalent}
\]

A polygon is with \( n \) sides is described by \([ <l_0, a_0>, <l_1, a_1>, \ldots, <l_{n-1}, a_{n-1}> ]\) 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 \((i + 1) \text{ mod } n\)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 \([ <6, 90>, <5, 90>, <6, 90>, <5, 90> ]\).

\[
equivalent = \text{proc (arg: polygon) returns (bool)}
\]

- \( \text{modifies nothing} \)
- \( \text{ensures} \) Returns true if and only if \( \text{this} \) and \( \text{arg} \) represent identical polygons.

Two polygons, \( p = [ <p_0, a_0>, <p_1, a_1>, \ldots, <p_{n-1}, a_{n-1}> ] \) and

\( q = [ <q_0, b_0>, <q_1, b_1>, \ldots, <q_{m-1}, b_{m-1}> ] \)

are equivalent if \( m = n \) and

\( \exists k: 0 \leq k < n \) such that \( \forall i, 0 \leq i < n: p_i = q_{(i + k) \text{ mod } n} \land a_i = b_{(i + k) \text{ mod } n} \).

Lucy Reasoner produced a Java implementation of \textit{polygon} shown on the next page.

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

\textbf{Answer:} \\
\( I (r) \equiv r.\text{edges}.\text{length} = r.\text{angles}.\text{length} \)

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 \textit{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 \textit{while} useful:

\[
P \Rightarrow \text{Inv}, \\
\text{Inv \{ Pred \} Inv}, \\
\text{Inv} \land \text{Pred} \{ \text{StatementList} \} \text{Inv}, \\
(\text{Inv} \land \neg \text{Pred}) \Rightarrow Q
\]

\[
P \{ \text{while (Pred) \{ StatementList \} } \} Q
\]
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:

\[
\text{result} \equiv m = n \quad \text{and} \quad \exists k: 0 \leq k < n \quad \text{such that} \quad \forall i, 0 \leq i < n: \quad p_i = q_{(i + k) \mod n} \land a_i = b_{(i + k) \mod n}.
\]

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:

\[
\text{result} \equiv \text{this.edges.length} = \text{arg.edges.length} \quad \text{and} \quad \exists k: 0 \leq k < \text{this.edges.length} \quad \text{such that} \quad \forall i, 0 \leq i < \text{this.edges.length}: \quad \text{this.edges}[i] = \text{arg.edges}[(i + k) \mod \text{this.edges.length}] \land \text{this.angles}[i] = \text{arg.angles}[(i+k) \mod \text{this.edges.length}]
\]

The abstraction function is

\[A(r) = [\langle p_0, a_0 \rangle, \langle p_1, a_1 \rangle, \ldots, \langle p_{n-1}, a_{n-1} \rangle] \quad \text{where} \quad n = r \text{.edges.length} \quad \text{and} \quad p_i = r \text{.edges}[i] \quad \text{and} \quad a_i = r \text{.angles}[i]
\]

So, using \(A(\text{this})\) and \(A(\text{arg})\) we can map the concrete property to:

\[
\text{result} \equiv m = n \quad \text{and} \quad \exists k: 0 \leq k < n \quad \text{such that} \quad \forall i, 0 \leq i < n: \quad p_i = q_{(i + k) \mod n} \land a_i = b_{(i + k) \mod n}
\]

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:

\[
\text{result} = \bigvee_{j = 0}^{k - 1} \left( (\text{result in previous iteration}) \land (\text{this.edges}[i] = \text{arg.edges}[(i + j) \mod \text{nedges}] \land \text{this.angles}[i] = \text{arg.angles}[(i+j) \mod \text{nedges}]) \right)
\]

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: \(\land k > 0\). (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 \(\text{nedges}\) once we have \(\text{this}\) and \(\text{arg}\) and ensure \(k \leq \text{nedges}\) (since we will need this for the post-condition).
So, the full invariant is:

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

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

Let’s use:

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

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

\[
P \Rightarrow \text{Inv}:
\]

\[
k = 0 \land \text{result} = \text{false} \Rightarrow \text{result} = k > 0 \land \ldots \\
k = 0 \Rightarrow k \leq \text{nedges} \text{ (since we know nedges = this.edges.length which cannot be negative.)} \\
The rest of the terms match exactly.
\]

\[
\text{Inv} \land \text{Pred} \{ \text{StatementList} \} \text{ Inv}:
\]

Lucy already proved P8 \implies Q14, so we need to show

\[
\text{Inv} \land k < \text{nedges} \{ \text{int} \ i = 0; \text{boolean} \ \text{isequiv} = \text{true}; \}
\]

P8: \text{nedges} = \text{this.edges.length} \land \text{nedges} = \text{arg.edges.length} \\
\wedge \text{nedges} = \text{this.angles.length} \land \text{nedges} = \text{arg.angles.length} \\
\wedge i = 0 \land \text{isequiv} \\
\wedge \text{k}_0 = k \land \text{nedges}_0 = \text{nedges} \land \text{result}_0 = \text{result} \land \text{this}_0 = \text{this} \land \text{arg}_0 = \text{arg}
\]

This is easy since the invariant (and no modifications of this or arg in the code) gives:

\[
\text{nedges} = \text{this.edges.length} \land \text{nedges} = \text{arg.edges.length} \\
\wedge \text{nedges} = \text{this.angles.length} \land \text{nedges} = \text{arg.angles.length} \\
\wedge \text{nedges}_0 = \text{nedges} \land \text{result}_0 = \text{result} \land \text{this}_0 = \text{this} \land \text{arg}_0 = \text{arg}
\]

and the two assignments give: i = 0 \land \text{isequiv}.
Next, we show:

Q14 = \( k_0 = k \land \text{nedges}_0 = \text{nedges} \land \text{result}_0 = \text{result} \land \text{this}_0 = \text{this} \land \text{arg}_0 = \text{arg} \)

\[
\text{nedges} - 1 \\
\land \text{isequiv} = \bigwedge_{i=0}^{\text{nedges} - 1} (\text{this}.\text{edges}[i] = \text{arg}.\text{edges}[(i + k) \mod \text{nedges}] \\
\land \text{this}.\text{angles}[i] = \text{arg}.\text{angles}[(i + k) \mod \text{nedges}])
\]

\{ \text{if (isequiv) result = true; k = k + 1; } \} \text{ 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 \( k \leq \text{nedges} \) term is satisfied because the predicate requires \( k < \text{nedges} \) before k is incremented.

\((\text{Inv} \land \neg \text{Pred}) \Rightarrow Q:\)

\[ k \leq \text{nedges} \land \neg (k < \text{nedges}) \Rightarrow k = \text{nedges}. \]

Substituting in Inv gives,

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

The get the result condition, we need to convert the outer \( \lor \) to

\[ \exists k: 0 \leq k < \text{this}.\text{edges}.\text{length} \text{ such that} \]

\[ \forall i, 0 \leq i < \text{this}.\text{edges}.\text{length}: \text{this}.\text{edges}[i] = \text{arg}.\text{edges}[(i + k) \mod \text{this}.\text{edges}.\text{length}] \\
\land \text{this}.\text{angles}[i] = \text{arg}.\text{angles}[(i + k) \mod \text{this}.\text{edges}.\text{length}] \]

This is true, since if the outer \( \lor \) is true, there must be some inner term that is true (corresponding to the \( \exists k \)); if the outer \( \lor \) is false, then no inner term is true, so no such \( k \) exists. The only discrepancy remaining is the \( \text{nedges} > 0 \) 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 \((t: \text{triangle})\) returns \((\text{bool})\)

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

\[ \exists k: 0 \leq k < 2 \text{ such that } \forall i, 0 \leq i < 2: p_i = q_{(i + k) \mod n}. \]

According to the behavioral notion of subtyping introduced by Liskov and Wing, would it be safe to introduce the type rule: \(\text{triangle} \subseteq \text{polygon}\)? 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 \(\text{polygon}\) (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 \text{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:

\[ [p_0, p_1, p_2] \Rightarrow [l_0, a_0>, l_1, a_1>, l_2, a_2> ] \]

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:

\[ \exists k: 0 \leq k < n \text{ such that } \forall i, 0 \leq i < n: p_i = q_{(i + k) \mod n}. \]

and add the clause \(\wedge a_i = b_{(i + k) \mod n}\) 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>, ..., <pᵣ₋₁, aᵣ₋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) {
        int nedges = this.edges.length;
        if (arg.edges.length != nedges) return false;

        int k = 0;
        boolean result = false;

        while (k < nedges) {
            int i = 0;
            boolean isequiv = true;

            // 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

            while (i < nedges) {
                if (this.edges[i] != arg.edges[(i + k) % nedges]) isequiv = false;
                if (this.angles[i] != arg.angles[(i + k) % nedges]) isequiv = false;
                i = i + 1;
            }

            // 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.)

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

        return result;
    }
}