(1)

Indiana University-Purdue University Indianapolis

Department of Computer and Information Science

CSCI 56500 – Programming Languages

Fall 2014

Qualifier Examination

Date: December 16, 2014

Time: 10.00 am – 12.00 pm

100 Points

85/100

Name: Baichuan Zhang

(Baichuan Zhang)

IMPORTANT NOTE

This test consists of seven questions. Please read the questions *carefully*. Good Luck!!!

---

1. Use the Owicki-Gries theory to prove or disprove the partial correctness of the following program with respect to the given pre- and post-conditions. Also indicate the *auxiliary variables* used in the proof. – 15 Points.

```
{X = y}
        L1          L2          L3
   X = X * a || X = X * b || X = X * c

{(X = y * a * b * c)}
```

Solution: since it has three parall programs, => auxiliary variables, should be L_1, L_2, L_3 ✓

use Owicki-Gries theory to prove: (construct its structure):

$\{X=y, L_1=1, L_2=1, L_3=1\}$ or $\{X=y*b, L_1=1, L_2=2, L_3=1\}$ or $\{X=y*c, L_1=1, L_2=1, L_3=2\}$ or $\{X=y*b*c, L_1=1, L_2=2, L_3=2\}$

-2

$(true \;\{\!|\; (X = X*a,\ L_1 = 2))$ ✓

$\{X=y*a, L_1=2, L_2=1, L_3=1\}$ or $\{X=y*a*b, L_1=2, L_2=2, L_3=1\}$ or $\{X=y*a*c, L_1=2, L_2=1, L_3=2\}$ or $\{X=y*a*b*c, L_1=2, L_2=2, L_3=2\}$

good ||

$\{X=y, L_1=1, L_2=1, L_3=1\}$ or $\{X=y*a, L_1=2, L_2=1, L_3=1\}$ or $\{X=y*c, L_1=1, L_2=1, L_3=2\}$ or $\{X=y*a*c, L_1=2, L_2=1, L_3=2\}$

$(true \;\{\!|\; (X=X*b, L_2=2))$ ✓

$\{X=y*b, L_1=1, L_2=2, L_3=1\}$ or $\{X=y*a*b, L_1=2, L_2=2, L_3=1\}$ or $\{X=y*b*c, L_1=1, L_2=2, L_3=2\}$ or

|| 2

$\{X=y*a*c*b, L_1=2, L_2=2, L_3=2\}$

$\{X=y, L_1=1, L_2=1, L_3=1\}$ or $\{X=y*a, L_1=2, L_2=1, L_3=1\}$ or $\{X=y*b, L_1=1, L_2=2, L_3=1\}$ or $\{X=y*a*b, L_1=2, L_2=2, L_3=1\}$

$(true \;\{\!|\; (X=X*c, L_3=2))$

$\{X=y*c, L_1=1, L_2=1, L_3=2\}$ or $\{X=y*a*c, L_1=2, L_2=1, L_3=2\}$ or $\{X=y*b*c, L_1=1, L_2=2, L_3=2\}$ or (Next page)

$\{X = Y^a * b * C,\ L_1 = 2,\ L_2 = 2,\ L_3 = 2\}$

for auxiliary variables. When $L_1 = 2$, $L_2 = 2$, $L_3 = 2$, means the three parallel programs all terminate.

So the final result of the program is:

$\{X = Y^a * b * C,\ L_1 = 2,\ L_2 = 2,\ L_3 = 2\}$

So remove the auxiliary variables.

$\{X = Y^a * b * C\}$

→ You had to formally invoke the OX G theorem to indicate $P_1' \to P$, $Q' \to Q$, $P_P' \to P_P$.

So the partial correctness of the program in the question is proved.

2. Show how to implement Conditional Critical Regions using Semaphores. (Hint: You will need an indivisible *updown* primitive that *ups* one semaphore and *downs* another semaphore. You will also need to use another primitive called *upall* which performs *up* until there are not more threads blocked on the semaphore). – 15 Points.

Solution: Conditional Critical regions use the region for protecting the shared variables, and wait statement for checking the condition, if condition not satisfied, then block the thread from entering into the critical region.

Similarly we can embed the ups and down between the shared variables which replace the region statement in conditional critical regions. for upall, if more than one thread ~~is waiting for entering into~~ finished execution in the critical section, then we up its value of all thread using upall operation.

Let's give example of bounded buffering using semaphore.

Constant size = 10.

variable: buffer array of (0.. size-1)

Incount, Outcount.

-5

- You are on the right track, however, not fully correct.

```
procedure putbuffer(data: what)
  downs
    buffer, Incount, Outcount
    if (Incount - Outcount < size)
      buffer[Incount mod size] = what
      Incount := Incount + 1
  ups
    upall other waiting threads
```

```
Variable
    Mutex, waiting : semaphore := 1;
loop
    down Mutex;
    if condition then exit end;
    updown Mutex, waiting
end;
body;
up Mutex;
upall waiting;
```

← Where is updown used?

```
procedure getbuffer(data: answer)
  downs
    buffer, Incount, Outcount
```

if Incount − outcount > 0

answer := buffer[outcount mod size]

outcount := outcount + 1

wps

upall all other waiting threads.

after one specific thread finished execution in the critical section which is getbuffer() or putbuffer() in this example, then all other waiting threads performs up which is upall operations.

3. Explain the following type definitions.

```
type GenericBuffer =
   ∀ Element. ∃Buffer. GenericBufferWRT[Element][Buffer]

type GenericBufferWRT[Element][Buffer] =
   {
      emptyBuffer:Buffer → Bool,
      write:(Element, Buffer) → Buffer,
      read:Buffer → Element,
      size:Buffer → Integer
   }
```

Now define two distinct, but identical in their semantics, forms of a function that can accept an instance of the above type and all subtypes of the above type and returns an Integer. Briefly describe these two forms. – 15 Points.

Solution: ① the type above defines a generic buffer, using generic and existential quantifiers, that hides its representation. Generic means the element can be any type, (int, bool, string ...) ∃ means we hides the data structure that can represent the elements; such as array or list and so on.

emptyBuffer: means checking given buffer is empty or not.

write: means putting one new element into buffer.

read: from buffer, read an element.

Size: get the size (length) of the buffer. -2

② functions:

i: first we can use the subtype

value f_1 = all[T ≤ GenericBuffer] fun(X: T) (len(X)).

defined where?

basically the above function returns the length of T and use the inclusion from which is "≤" symbol.

ii: second function:

~~value f_2 = all[fun(len : ∀ Element, ∃ buffer, GenericBuffer[Element][Buffer] →~~

value f_2 = fun(len ∀ Element, ∃ buffer. GenericBufferWRT[Element][Buffer] → Int) fun(X: GenericBuffer) len(X).

why is this needed?

value f2 = fun(x: GenericBuffer) len(x)

f_2 uses existential and universal quantifiers to define the len function, and return the integer which is the length of buffer.

4. Does Val support explicit parallelism? Justify your answer. If it does support explicit parallelism then using its technique, write a code snippet (syntax similar to the one discussed in the class) that will start 100 concurrent streams of executions on two vectors; add each element of the first vector to the corresponding one of the second vector resulting a new vector and a scalar that will be generated by adding all the elements of the resultant vector. The resultant vector and the scalar should be returned as the outputs of the code snippet. The code also should check for validity of each addition. – 15 Points.

Solution: Yes, Val supports explicitly parallelism. Since Val has the functional programming property if the two tasks dont depend on each other, we can execute them independently.

Also Val has explicitly parallelism since in its language it has the construct the forall expression which has two basic operations, Construct and accumulate. Construct creates each parallel streams independently and put each result into an array. accumulate merges all the executed results using some kind of associative binary operator like +.

Code:

~~for all i in [0, 99]~~

vector vector1, vector2, Newvector

forall i in [0, 99]

do

a := Vector1[i]

b := vector2[i]

~~error: return error.~~

Construct: ~~if ok, then c := a+b, else error(c+b).~~

Syntax is almost correct!

~~Accumulate: if ok then~~

addition :c = a+b

Addition-check(addition): if ok then true else false.

Construct: if true then Newvector.append(c) else error.

accumulate + : if ~~op~~ then ~~newvector~~ else error.

accumulate +

if this is + then

then

Sum

How is this Calculated?

the above program will return the resulting vector Newvector and the summation of all elements in newvector which is stored in sum variable.

(10)

5. Briefly explain the cut operator (!) of Prolog. Use that operator to write a Prolog program that ascertains if a given element is a member of a given list or not when the list may contain that element more than once. In the case where an element occurs more than once in the list, the program will be asked to redo that goal and achieve the goal without any redundant effort. Explain how your program will achieve these requirements. – 15 Points.

Solution: ① Cut operator (!) basically prevents the backtracking of a particular query. Since if some subtask make the backtrack the database unecessary, then use the cut operator.

② the prolog program can be as follows:

```
Member( Element, [Element|List]) :- !, Member(Element, [Element|List]).
Member( Element, [_|List]) :- Member( Element, List).
```

- Is this really needed?

example, if invocates Member(2, [1, 2, 2, 3]) then the whole program will only return 2 and stop.

because in base case above, if we found an element is a member of a list, then because of ! operator, no backtracking is performed. So once we found the answer, we cannot call the Member function again.

```
Member( Element, [Element|List]) :- !, Member(Element, [Element|List]).
```

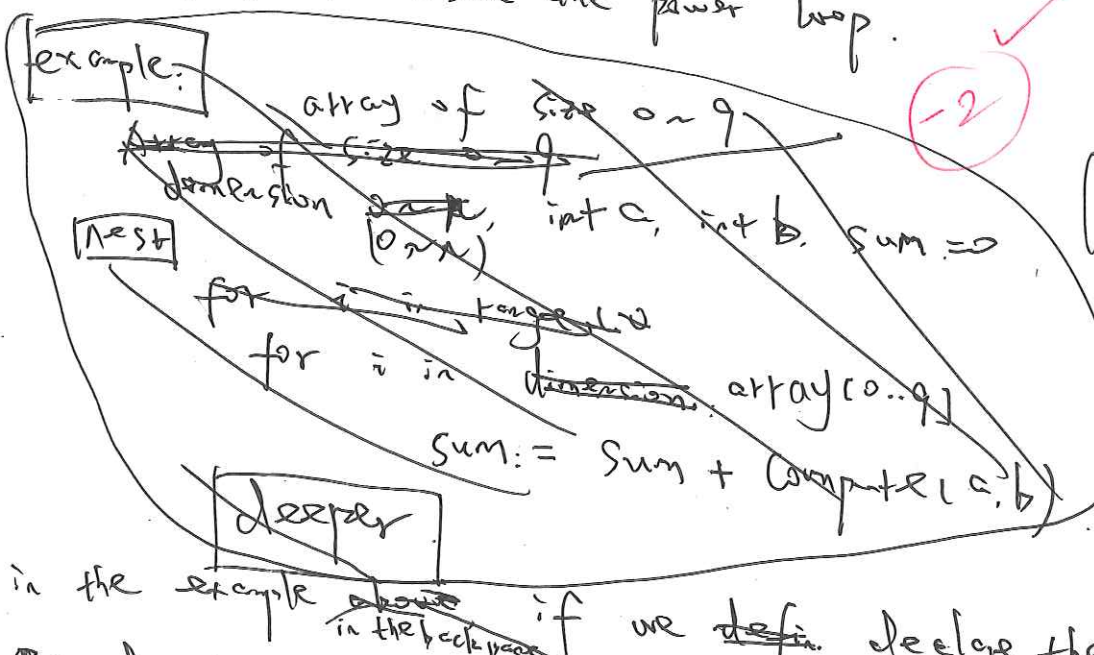
good

if true then no backtracking

6. Does it make sense to place declarations inside the replicated part or the body of a power loop? What will be ramifications of such declarations? – 10 Points.

Solution: basically, the idea of using power loop is repeating the part between rest and deeper multiple times. the number of execution depend on the # variables in the predefined data structure, like array.

if the variables declared inside the power loop depends on some other computation inside the power loop, then it doesn't make sense to put such declaration inside the power loop.



-2

example in back page

→ what would happen to scoping rules if such a declaration is allowed?

in the example in the back page, if we declare the variable sum inside the power loop. every time when executing for loop, the sum will be initialized to zero. But the current sum result depends on the previous sum result. So it doesn't make sense to put such variable declaration inside the power loop.

Exemple

```
begin:  n=3,  a=1,  b=1,  Sum=0
        dimension n
test:
        for i in range(0, dimension)
                Sum:= Sum + Compte(a, b)
        deeper
End
```

7. The first Church-Rosser theorem says:

If $e_0 \Leftrightarrow^* e_1$ then there exists an e_2 such that $e_0 \Rightarrow^* e_2$ and $e_1 \Rightarrow^* e_2$.

Based on this theorem, prove that:

No lambda expression can be converted to two distinct normal forms (ignoring the differences due to α-conversion). – 15 Points.

proof: proof by contradiction. ✓

Suppose there is a lambda expression e that can be converted to two distinct normal forms. Let's say e_{N_1}, e_{N_2},

So $\begin{cases} e \Leftrightarrow^* e_{N_1} \\ e \Leftrightarrow^* e_{N_2} \\ \text{and } e_{N_1} \neq e_{N_2} \end{cases}$

– You are heading in the right direction but your proof is not completely correct.

based on Church-Rosser theorem, -4

if $e \Leftrightarrow^* e_{N_1}$, then there exists an E_1, such that

$e \Rightarrow^* E_1$ and $e_{N_1} \Rightarrow^* E_1$ ①

if $e \Leftrightarrow^* e_{N_2}$, then there exists the E_1^*, such that

$e \Rightarrow^* E_1^*$ and $e_{N_2} \Rightarrow^* E_1^*$ ②

Are you referring to E_1 here or some other E_1^*?

– If E_1 then there how do you prove that ② same E_1 will be achieved from e_{N_1} & e_{N_2}?

– If different E_1^* then why would $e_{N_1} = e_{N_2}$ hold?

based on ① and ②,

$e_{N_1} = e_{N_2}$ which contradicts our condition

which is $e_{N_1} \neq e_{N_2}$

So the assumption is false and $e_{N_1} = e_{N_2}$

So No lambda expression can be converted to two distinct normal forms. ✓