Lecture 01/11/16

Lecturer: Xiaodi Wu

January 11th, 2016
Outer Loop

\(<P_2>\) \(x \leftarrow a; y \leftarrow b; z \leftarrow 1\)
\textbf{while} \(y > 0\) [\(C_2\)] \textbf{do}
\hline
\(\langle P_1 \rangle\): \((y > 0) \land (x^y = d)\).
\(I_1: (y > 0) \land (x^y = d)\).
\(\langle Q_1 \rangle\):
\((y > 0) \land (x^y = d) \land \text{odd}(y)\)
\(y \leftarrow y - 1; z \leftarrow z \times x\) [\(B_2\)]
\textbf{end while}
\textbf{return} \(z\).
\(<Q_2>\)

\[\text{Simplify with the inner loop invariant!}\]
Outer Loop

\[
\langle P_2 \rangle \times \leftarrow a; \quad y \leftarrow b; \quad z \leftarrow 1
\]

while \( y > 0 \) \( [C_2] \) do

\[
\langle P_1 \rangle: \quad (y > 0) \land (x^y = d).
\]

\[
l_1 : (y > 0) \land (x^y = d).
\]

\[
\langle Q_1 \rangle:
\]

\[
(y > 0) \land (x^y = d) \land \text{odd}(y)
\]

\[
y \leftarrow y - 1; \quad z \leftarrow z \times x \quad [B_2]
\]

end while

return \( z \).

\[
\langle Q_2 \rangle
\]

- Simplify with the inner loop invariant!
- Observe and define the new invariant!

Time complexity \( \Theta(\log b) \).
Outer Loop

\( \langle P_2 \rangle \) \( x \leftarrow a; y \leftarrow b; z \leftarrow 1 \)

**while** \( y > 0 \) \([C_2] \) **do**

\( \langle P_1 \rangle : (y > 0) \land (x^y = d) \).

\( l_1 : (y > 0) \land (x^y = d) \).

\( \langle Q_1 \rangle : \\
(y > 0) \land (x^y = d) \land \text{odd}(y) \)

\( y \leftarrow y - 1; z \leftarrow z \times x \) \([B_2] \)

**end while**

**return** \( z \).

\( \langle Q_2 \rangle \)

- Simplify with the inner loop invariant!
- Observe and define the new invariant!
- \( l_2 : (y \geq 0) \land (z \times x^y = d') \) for a fixed \( d' \).
Outer Loop

\[ \langle P_2 \rangle \ x \leftarrow a; \ y \leftarrow b; \ z \leftarrow 1 \]

while \( y > 0 \) \ [C_2] \ do

\[ \langle P_1 \rangle: \ (y > 0) \land (x^y = d). \]

\[ l_1 : (y > 0) \land (x^y = d). \]

\[ \langle Q_1 \rangle: \]

\( (y > 0) \land (x^y = d) \land \text{odd}(y) \)

\[ y \leftarrow y - 1; \ z \leftarrow z \times x \ [B_2] \]

end while

return \( z \).

\[ \langle Q_2 \rangle \]

- Simplify with the inner loop invariant!
- Observe and define the new invariant!
- \( l_2 : (y \geq 0) \land (z \times x^y = d') \) for a fixed \( d' \).
- \( P_2 \rightarrow z \times x^y = a^b \).
Outer Loop

\(\langle P_2 \rangle \ x \leftarrow a; y \leftarrow b; z \leftarrow 1\)

while \(y > 0 \ [C_2] \) do

\(\langle P_1 \rangle: \ (y > 0) \land (x^y = d).\)

\(I_1 : (y > 0) \land (x^y = d).\)

\(\langle Q_1 \rangle: \ (y > 0) \land (x^y = d) \land \text{odd}(y)\)

\(y \leftarrow y - 1; z \leftarrow z \times x \ [B_2] \)

end while

return \(z.\)

\(\langle Q_2 \rangle\)

- Simplify with the inner loop invariant!
- Observe and define the new invariant!
- \(l_2 : (y \geq 0) \land (z \times x^y = d')\) for a fixed \(d'.\)
- \(P_2 \rightarrow z \times x^y = a^b.\)
- \(Q_2 : y = 0; z \times x^y = a^b \rightarrow z = a^b.\)
Outer Loop

\[ \langle P_2 \rangle \] x ← a; y ← b; z ← 1

while \( y > 0 \) [\( [C_2] \) do

\[ \langle P_1 \rangle \]: \((y > 0) \land (x^y = d)\).

\[ l_1 \]: \((y > 0) \land (x^y = d)\).

\[ \langle Q_1 \rangle \]: \((y > 0) \land (x^y = d) \land \text{odd}(y)\)

\( y ← y - 1; z ← z \times x \) [\( [B_2] \)]

end while

return \( z \).

\[ \langle Q_2 \rangle \]

- Simplify with the inner loop invariant!
- Observe and define the new invariant!
- \( I_2 \): \((y \geq 0) \land (z \times x^y = d')\) for a fixed \( d'\).
- \( P_2 \rightarrow z \times x^y = a^b \).
- \( Q_2 \): \( y = 0; z \times x^y = a^b \rightarrow z = a^b \).
- Time complexity \( \Theta(\log b) \).
Remarks on Loop Invariants

A formal written proof (a sample of what you should do in homework) will be posted in the note.
Remarks on Loop Invariants

A formal written proof (a sample of what you should do in homework) will be posted in the note.

- Q: How do I come with loop invariants for new codes?
Remarks on Loop Invariants

A formal written proof (a sample of what you should do in homework) will be posted in the note.

▶ **Q**: How do I come with loop invariants for new codes?
▶ **A**: Experience and careful analysis! Loop invariants could be very different!
Remarks on Loop Invariants

A formal written proof (a sample of what you should do in homework) will be posted in the note.

- **Q:** How do I come with loop invariants for new codes?
- **A:** Experience and careful analysis! Loop invariants could be very different!
- **Q:** What is the use of Loop Invariants?
Remarks on Loop Invariants

A formal written proof (a sample of what you should do in homework) will be posted in the note.

- Q: How do I come with loop invariants for new codes?
- A: Experience and careful analysis! Loop invariants could be very different!
- Q: What is the use of Loop Invariants?
- A: A framework to prove the correctness of your analysis!
Abstract Data Type (ADT)

Stack
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.
Abstract Data Type (ADT)

Stack
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Stack: operations

- push(o): insert object o at the top of the stack.
- pop(): remove and return the top of the stack.
Abstract Data Type (ADT)

Stack
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Stack: operations

- push(o): insert object o at the top of the stack.
- pop(): remove and return the top of the stack.
- size(): return the number of objects in the stack.
- isEmpty(): return a Boolean indicating if the stack is empty.
- top(): return the top of the stack.
Definition

The $n$th Fibonacci number $F(n)$ is defined recursively as

$$F(n) = F(n-1) + F(n-2)$$

for $n > 1$ with $F(0) = 0$, $F(1) = 1$. 

Algorithm

```plaintext
Fib(n)
if n > 1 then
  return Fib(n-1) + Fib(n-2)
else
  return n;
end if
```
Definition

The $n$th Fibonacci number $F(n)$ is defined recursively as

$$F(n) = F(n - 1) + F(n - 2) \text{ for } n > 1 \text{ with } F(0) = 0, F(1) = 1.$$  

Algorithm Fib($n$)

if $n > 1$ then
    return Fib($n-1$) + Fib($n-2$)
else
    return $n$;
end if
Definition
The $n$th Fibonacci number $F(n)$ is defined recursively as $F(n) = F(n-1) + F(n-2)$ for $n > 1$ with $F(0) = 0, F(1) = 1$.

Algorithm $Fib(n)$

if $n > 1$ then
    return $Fib(n-1) + Fib(n-2)$
else
    return $n$;
end if

Test run $Fib(4)$. 
Implementation with an $N$-element array $S$, with elements stored from $S[0]$ to $S[t]$, where $t$ is the index of the top element. **Note:** arrays start at index 0 and thus $t$ is initialized to -1.
Implementation with an $N$-element array $S$, with elements stored from $S[0]$ to $S[t]$, where $t$ is the index of the top element. **Note:** arrays start at index 0 and thus $t$ is initialized to -1.

- `size()`: return $t+1$;
- `isEmpty()`: return True if $t=-1$; else return False;
- `top()`: return $S[t]$;
Stack: Array-based Implementation

Algorithm push(o)
if size()=N then
    stack-full exception
end if

\[ t \leftarrow t + 1 \]

\[ S[t] \leftarrow o \]

Algorithm pop()
if isEmpty() then
    stack-empty exception
end if

\[ e \leftarrow S[t] \]

\[ S[t] \leftarrow null \]

\[ t \leftarrow t - 1 \]

return e.
Abstract Data Type (ADT)

Queue

A **queue** is a container of objects that are inserted and removed according to the **first-in first-out (FIFO)** principle. Enter at the **rear** and remove from the **front**.
Abstract Data Type (ADT)

Queue

A *queue* is a container of objects that are inserted and removed according to the **first-in first-out (FIFO)** principle. Enter at the **rear** and remove from the **front**.

Stack: operations

- enqueue(*o*): insert object *o* at the rear of the queue.
- dequeue(): remove and return the front of the queue.
Queue

A queue is a container of objects that are inserted and removed according to the **first-in first-out (FIFO)** principle. Enter at the **rear** and remove from the **front**.

Stack: operations

- `enqueue(o)`: insert object `o` at the rear of the queue.
- `dequeue()`: remove and return the front of the queue.
- `size()`: return the number of objects in the queue.
- `isEmpty()`: return a Boolean indicating if the queue is empty.
- `front()`: return the front of the queue.
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r − 1]$, where $f$, $r − 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue.
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue. 

Q: what if $r$ gets bigger than $N$?
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue. Q: what if $r$ gets bigger than $N$?

- size(): return $(N + (r - f)) \mod N.$
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue. Q: what if $r$ gets bigger than $N$?

- size(): return $(N + (r - f)) \mod N$.
- isEmpty(): return True if $r == f$; else return False;
- front(): return $S[f]$;