CSE 331

Arrays

Kevin Zatloukal

Reminder

- Grades are a lot less important than before
  - companies care much more about interviews
  - grad schools care much more about recommendations
- Understanding the material is very important
  - needed for future classes and on the job
- HW4 Q5 is closest so far to an interview question
  - working on paper is practice for a whiteboard

    (using a computer would help practice this)

"Bottom Up" Loops on the Natural Numbers

func f(0) := ...
f(n+1) := ... f(n) ... for any n : ℕ

- Can be implemented with a loop like this

```
function f(n: number): number {
  let i: number = 0;
  let s: number = "…";     // = f(0)
  {{ Inv: s = f(i) }}
  while (i != n) {
    s = "… f(i) …"[f(i) ↦ s]     // = f(i+1)
    i = i + 1;
  }
  return s;
}
```

Processing Lists with Loops

- Hard to process lists with loops
  - only have easy access to the last element added

    natural processing would start from the other end
  - usually end up with the result in the reverse order

"Top Down" Loops on Lists

func f(nil) := nil

f(cons(x, L)) := cons(g(x), f(L)) for any $x : \mathbb{Z}$ and L : List

- Can be implemented with a loop like this

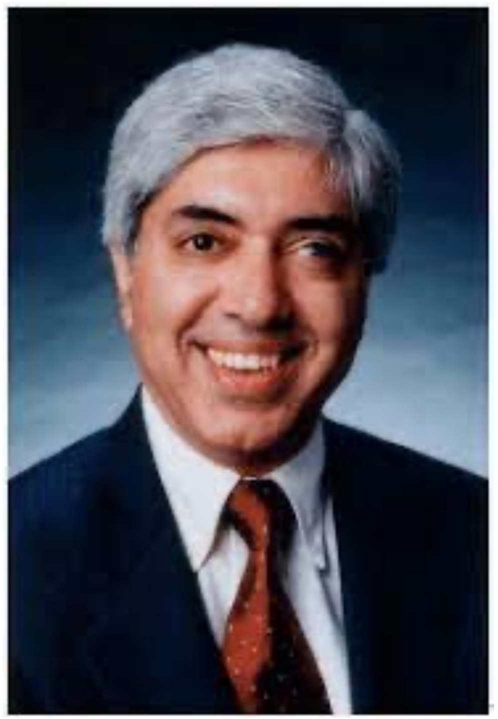
```
function f(L: List): List {
  let R: List = L;
  let S: List = nil;    // = f(nil)
  {{ Inv: f(L) = concat(rev(S), f(R)) }}
  while (R !== nil) {
    S = cons(g(R.hd), S);
    R = R.tl;
  }
  return rev(S);        // = f(L)
}
```

Processing Lists with Loops

- Hard to process lists with loops
  - only have easy access to the last element added
    - natural processing would start from the other end
  - usually end up with the result in the reverse order

- There is an easier way to fix this
  - switch data structures
  - use one that lets us access either end easily

"Lists are the original data structure for functional programming,
just as arrays are the original data structure of imperative programming"



Ravi Sethi

we will work with lists in HW2+ and arrays HW6+

Array Accesses

- **Easily access both $A[0]$ and $A[n\text{-}1]$, where $n = A.\text{length}$**
  - **bottom-up loops are now easy**

- **"With great power, comes great responsibility"**

- **Whenever we write "$A[j]$", we must check $0 \leq j < n$**
  - **new possibilities for bugs**
    - with list, we only need to worry about nil and non-nil
    - once we know L is non-nil, we know L.hd exists
  - **TypeScript will not help us with this!**
    - type checker does catch "could be nil" bugs, but not this

Array Concatenation

- Define the operation "$+\!\!+$" as array concatenation
  – makes clear the arguments are arrays, not numbers

- The following properties hold for any arrays A, B, C

$$A +\!\!+ [] = A = [] +\!\!+ A$$ ("identity")

$$A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$$ ("associativity")

  – we will use these facts *without* explanation in calculations
  – second line says parentheses *don't matter*, so
    we will write $A +\!\!+ B +\!\!+ C$ and not say where the $(..)$ go

Array Concatenation Math

- Same properties hold for lists

$[] \mathbin{+\!\!+} A = A$ | $\text{concat}(\text{nil}, L) = L$

$A \mathbin{+\!\!+} [] = A$ | $\text{concat}(L, \text{nil}) = L$

$A \mathbin{+\!\!+} (B \mathbin{+\!\!+} C) = (A \mathbin{+\!\!+} B) \mathbin{+\!\!+} C$ | $\text{concat}(A, \text{concat}(B, C))$
$= \text{concat}(\text{concat}(A, B), C)$

  – we required explanation of these facts for lists
  – but we will **not** require explanation of these facts for arrays
    (trying to reason more quickly, now that we have more practice)

Defining Functions on Arrays

- Can still define functions recursively

func $\text{count}([], x)$	$:= 0$		for any $x : \mathbb{Z}$
$\text{count}(A \mathbin{+\!\!+} [y], x)$	$:= 1 + \text{count}(A, x)$	if $x = y$	for any $x : \mathbb{Z}$ and any $A : \text{Array}_{\mathbb{Z}}$
$\text{count}(A \mathbin{+\!\!+} [y], x)$	$:= \text{count}(A, x)$	if $x \neq y$	for any $x : \mathbb{Z}$ and any $A : \text{Array}_{\mathbb{Z}}$

  – could write patterns with "$[y] \mathbin{+\!\!+} A$" instead

Subarrays

- **Often useful to talk about part of an array (subarray)**
  - **define the following notation**

    $A[i .. j] = [\ A[i], A[i{+}1], \dots, A[j]\ ]$

  - **note that this includes** $A[j]$

    (some functions exclude the right end; we will include it)

Subarrays

$A[i .. j] = [\ A[i], A[i+1], ..., A[j]\ ]$

- **Define this formally as follows**

func $A[i .. j]$	$:= []$	if $j < i$
$A[i .. j]$	$:= A[i .. j\text{-}1] \mathbin{+\!\!\!+} [A[j]]$	if $i \le j$

  - **second case needs $0 \le i \le j < n$ for this to make sense**

    undefined if $i \le j$ and ($i < 0$ or $n \le j$)

  - **note that $A[0 .. \text{-}1] = []$ since $\text{-}1 < 0$**

    "Isn't -1 an array out of bounds error?"

    In code, yes — In math, no (the definition says this is an empty array)

Subarray Math

$$\textbf{func } A[i .. j] := [] \qquad \text{if } j < i$$
$$A[i .. j] := A[i .. j\text{-}1] \mathbin{+\!\!+} [A[j]] \qquad \text{if } 0 \le i \le j < A.\text{length}$$

- **Some useful facts**

$$A = A[0 .. n\text{-}1] \qquad (= [A[0], A[1], \ldots, A[n\text{-}1]])$$
$$\text{where } n = A.\text{length}$$

  - **the subarray from** 0 **to** $n - 1$ **is the entire array**

$$A[i .. j] = A[i .. k] \mathbin{+\!\!+} A[k\text{+}1 .. j]$$

  - **holds for any** $i, j, k : \mathbb{N}$ **satisfying** $0 \le i \le k \le j < n$
  - **we will use these *without* explanation**

TypeScript Arrays

- **Translating math to TypeScript**

Math	**TypeScript**
$A \mathbin{+\!\!+} B$	`A.concat(B)`
$A[i \mathinner{..} j]$	`A.slice(i, j+1)`

  - **JavaScript's `A.slice(i, j)` does not include $A[j]$, so we need to increase `j` by one**

- **Note: array out of bounds does not throw Error**
  - **returns `undefined`**

    (hope you like debugging!)

Facts About Arrays

- **"With great power, comes great responsibility"**

- **Since we can easily access any $A[j]$, may need to keep track of facts about it**
  - **may need facts about *every* element in the array**

    applies to preconditions, postconditions, and intermediate assertions

- **We can write facts about several elements at once:**
  - **this says that elements at indexes 2 .. 10 are non-negative**

    $0 \leq A[j]$ for any $2 \leq j \leq 10$

  - **shorthand for 9 facts** $(0 \leq A[2], \ldots, 0 \leq A[10])$

Finding an Element in an Array

- Can search for an element in an array as follows

func contains([], x)	:= F		for any …
contains($A \mathbin{+\!\!+} [y]$, x)	:= T	if $x = y$	for any …
contains($A \mathbin{+\!\!+} [y]$, x)	:= contains(A, x)	if $x \neq y$	for any …

- Searches through the array in linear time
  - could do the same on a list

- Can search more quickly if the list is sorted
  - precondition is $A[0] \leq A[1] \leq \ldots \leq A[n\text{-}1]$ (informal)
  - write this formally as

$$A[j] \leq A[j+1] \text{ for any } 0 \leq j \leq n - 2$$

Loops with Arrays

Sum of an Array

func $\text{sum}([])$ $:= 0$

$\text{sum}(A \mathbin{+\!\!+} [y]) := \text{sum}(A) + y$ for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Could translate this directly into a recursive function**
  - **that would be level 0**

- **Do this instead with a loop**
  - **use the "bottom up" template**
  - **start from** $[]$ **and work up to all of** A
  - **at any point, we have** $\text{sum}(A[0 \,..\, j\text{-}1])$ **for some index** j

    I will add one extra fact we also need

Sum of an Array

func sum([]) := 0
sum(A ⧺ [y]) := sum(A) + y   for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A) }}
return s;
```

Sum of an Array

func sum([]) := 0
sum(A ⧺ [y]) := sum(A) + y   for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ j = 0 and s = 0}}
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A) }}
return s;
```

Sum of an Array

func sum([]) := 0

sum(A ⧺ [y]) := sum(A) + y for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ j = 0 and s = 0 }}
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A) }}
return s;
```

s = 0
 = sum([]) **def of** sum
 = sum(A[0 .. -1])
 = sum(A[0 .. j – 1]) **since** j = 0

Sum of an Array

func sum([]) := 0

sum(A ⧺ [y]) := sum(A) + y   for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A[0 .. j – 1]) and j = A.length }}
{{ s = sum(A) }}
return s;
```

Sum of an Array

func $\text{sum}([]) := 0$

$\text{sum}(A \mathbin{+\!\!+} [y]) := \text{sum}(A) + y$ for any $y : \mathbb{Z}$ and $A : \text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  s = s + A[j];
  j = j + 1;
}
{{ s = sum(A[0 .. j – 1]) and j = A.length }}
{{ s = sum(A) }}
return s;
```

$s = \text{sum}(A[0 \,..\, j - 1])$
$= \text{sum}(A[0 \,..\, A.\text{length} - 1])$
$= \text{sum}(A)$

Sum of an Array

func sum([]) := 0
sum(A ⧺ [y]) := sum(A) + y   for any y : $\mathbb{Z}$ and A : $\text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
let j: number = 0;
let s: number = 0;
{{ Inv: s = sum(A[0 .. j – 1]) and j ≤ A.length }}
while (j !== A.length) {
  {{ s = sum(A[0 .. j – 1]) and j < A.length }}
  s = s + A[j];
  j = j + 1;
  {{ s = sum(A[0 .. j – 1]) and j ≤ A.length }}
}
{{ s = sum(A) }}
return s;
```

since j ≤ A.length
and j ≠ A.length

Sum of an Array

$$\textbf{func}\ \text{sum}([]) := 0$$
$$\text{sum}(A \mathbin{+\!\!+} [y]) := \text{sum}(A) + y \qquad \text{for any } y : \mathbb{Z} \text{ and } A : \text{Array}_{\mathbb{Z}}$$

- **Loop implementation:**

```
while (j !== A.length) {
  {{ s = sum(A[0 .. j – 1]) and j < A.length }}
  s = s + A[j];
  {{ s – A[j] = sum(A[0 .. j – 1]) and j < A.length }}
  j = j + 1;
  {{ s – A[j – 1] = sum(A[0 .. j – 2]) and j – 1 < A.length }}
  {{ s = sum(A[0 .. j – 1]) and j ≤ A.length }}
}
```

Sum of an Array

func sum([]) := 0
sum(A ⧺ [y]) := sum(A) + y   for any y : $\mathbb{Z}$ and A : $\text{Array}_{\mathbb{Z}}$

- **Loop implementation:**

```
while (j !== A.length) {
    {{ s = sum(A[0 .. j – 1]) and j < A.length }}
    s = s + A[j];
    {{ s – A[j] = sum(A[0 .. j – 1]) and j < A.length }}
    j = j + 1;
    {{ s – A[j – 1] = sum(A[0 .. j – 2]) and j – 1 < A.length }}
    {{ s = sum(A[0 .. j – 1]) and j ≤ A.length }}
}
```

s = sum(A[0 .. j – 2]) + A[j – 1]   **since** s – A[j-1] = sum(..)
= sum(A[0 .. j – 2] ⧺ [A[j-1]])   **def of** sum
= sum(A[0 .. j – 1])

Sum of an Array

$$\textbf{func}\ \mathrm{sum}([]) := 0$$
$$\mathrm{sum}(A \mathbin{+\!\!+} [y]) := \mathrm{sum}(A) + y \qquad \text{for any } y : \mathbb{Z} \text{ and } A : \mathrm{Array}_{\mathbb{Z}}$$

- **Loop implementation:**

```
while (j !== A.length) {
  {{ s = sum(A[0 .. j – 1]) and j < A.length }}
  s = s + A[j];
  {{ s – A[j] = sum(A[0 .. j – 1]) and j < A.length }}
  j = j + 1;
  {{ s – A[j – 1] = sum(A[0 .. j – 2]) and j – 1 < A.length }}
  {{ s = sum(A[0 .. j – 1]) and j ≤ A.length }}
}
```

$j \leq A.length$ **since** $j < A.length + 1$

Linear Search of an Array

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{contains}([], x) & := F & \\
\quad \mathrm{contains}(A \mathbin{+\!\!+} [y], x) & := T & \text{if } x = y \\
\quad \mathrm{contains}(A \mathbin{+\!\!+} [y], x) & := \mathrm{contains}(A, x) & \text{if } x \neq y
\end{array}
$$

- **Could translate this directly into a recursive function**
  - **that would be level 0**

- **Do this instead with a loop**
  - **use the "bottom up" template**
  - **start from** $[]$ **and work up to all of** A
  - **can stop immediately if we find x**

    contains returns true in that case
  - **otherwise, we have** $\mathrm{contains}(A[0 \,..\, j{-}1], x) = F$ **for some** j

Linear Search of an Array

func contains([], x) := F
  contains(A ⧺ [y], x) := T if x = y
  contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j–1], x) = F }}
while (j != A.length) {
  if (A[j] === x)
    {{ contains(A, x) = T }}
    return true;
  j = j + 1;
}
{{ contains(A, x) = F }}
return false;
```

Linear Search of an Array

$$
\begin{aligned}
&\textbf{func}\ \text{contains}([], x) && := \text{F} && \\
&\quad\ \text{contains}(A \mathbin{+\!\!+} [y], x) && := \text{T} && \text{if } x = y \\
&\quad\ \text{contains}(A \mathbin{+\!\!+} [y], x) && := \text{contains}(A, x) && \text{if } x \neq y
\end{aligned}
$$

- **Loop implementation:**

```
let j: number = 0;
{{ j = 0 }}
{{ Inv: contains(A[0 .. j–1], x) = F }}
while (j != A.length) {
  if (A[j] === x)
    return true;
  j = j + 1;
}
return false;
```

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if $x = y$
contains(A ⧺ [y], x) := contains(A, x) if $x \neq y$

- **Loop implementation:**

```
let j: number = 0;
{{ j = 0 }}
{{ Inv: contains(A[0 .. j−1], x) = F }}
while (j != A.length) {
  if (A[j] === x)
    return true;
  j = j + 1;
}
return false;
```

contains(A[0 .. j−1], x)
= contains(A[0 .. -1], x)   **since** j = 0
= contains([], x)
= F   **def of** contains

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if $x = y$
contains(A ⧺ [y], x) := contains(A, x) if $x \neq y$

- **Loop implementation:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j−1], x) = F }}
while (j != A.length) {
  if (A[j] === x)
    return true;
  j = j + 1;
}
{{ contains(A[0 .. j−1], x) = F and j = A.length }}
{{ contains(A, x) = F }}
return false;
```

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if x = y
contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j–1], x) = F }}
while (j != A.length) {
  if (A[j] === x)
    return true;
  j = j + 1;
}
{{ contains(A[0 .. j–1], x) = F and j = A.length }}
{{ contains(A, x) = F }}
return false;
```

F = contains(A[0 .. j–1], x)
= contains(A[0 .. A.length – 1], x) **since** j = ...
= contains(A, x)

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if $x = y$
contains(A ⧺ [y], x) := contains(A, x) if $x \neq y$

- **Loop implementation:**

```
while (j != A.length) {
  {{contains(A[0 .. j–1], x) = F and j ≠ A.length }}
  if (A[j] === x)
    {{ contains(A, x) = T }}
    return true;
  j = j + 1;
  {{ contains(A[0 .. j–1], x) = F }}
}
return false;
```

Linear Search of an Array

func contains([], x) := F
  contains(A ⧺ [y], x) := T if x = y
  contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

```
{{contains(A[0 .. j–1], x) = F and j ≠ A.length }}
if (A[j] === x) {
  {{ contains(A, x) = T }}
  return true;
} else {
}
j = j + 1;
{{ contains(A[0 .. j–1], x) = F }}
```

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if $x = y$
contains(A ⧺ [y], x) := contains(A, x) if $x \neq y$

- **Loop implementation:**

{{contains(A[0 .. j–1], x) = F and j ≠ A.length }}
```
if (A[j] === x) {
```
{{contains(A[0 .. j–1], x) = F and j ≠ A.length and A[j] = x }}
{{ contains(A, x) = T }}
```
  return true;
} else {
```
…

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if x = y
contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

{{contains(A[0 .. j–1], x) = F and j ≠ A.length }}

```
if (A[j] === x) {
```

{{contains(A[0 .. j–1], x) = F and j ≠ A.length and A[j] = x }}

{{ contains(A, x) = T }}

```
  return true;
} else {
...
```

contains(A[0 .. j], x)
= contains(A[0 .. j-1] ⧺ [A[j]], x)
= T since A[j] = x

Can now prove by induction that contains(A, x) = T

Linear Search of an Array

func contains([], x) := F
contains(A ⧺ [y], x) := T if x = y
contains(A ⧺ [y], x) := contains(A, x) if x ≠ y

- **Loop implementation:**

{{contains(A[0 .. j−1], x) = F and j ≠ A.length }}

```
if (A[j] === x) {
  return true;
} else {
```

{{contains(A[0 .. j−1], x) = F and j ≠ A.length and A[j] ≠ x }}
{{ contains(A[0 .. j], x) = F }}

```
}
```

{{ contains(A[0 .. j], x) = F }}

```
j = j + 1;
```

{{ contains(A[0 .. j−1], x) = F }}

Linear Search of an Array

$$
\begin{aligned}
&\textbf{func}\ \text{contains}([], x) && := F && \\
&\quad \text{contains}(A \mathbin{+\!\!+} [y], x) && := T && \text{if } x = y \\
&\quad \text{contains}(A \mathbin{+\!\!+} [y], x) && := \text{contains}(A, x) && \text{if } x \neq y
\end{aligned}
$$

- **Loop implementation:**

```
{{contains(A[0 .. j–1], x) = F and j ≠ A.length }}
if (A[j] === x) {
  return true;
} else {
  {{contains(A[0 .. j–1], x) = F and j ≠ A.length and A[j] ≠ x }}
  {{ contains(A[0 .. j], x) = F }}
}
```

Linear Search of an Array

$$
\begin{aligned}
&\textbf{func}\ \text{contains}([], x) && := F && \\
&\quad \text{contains}(A +\!\!+ [y], x) && := T && \text{if } x = y \\
&\quad \text{contains}(A +\!\!+ [y], x) && := \text{contains}(A, x) && \text{if } x \neq y
\end{aligned}
$$

- **Loop implementation:**

{{contains(A[0 .. j–1], x) = F and j ≠ A.length }}

```
if (A[j] === x) {
  return true;
} else {
  {{contains(A[0 .. j–1], x) = F and j ≠ A.length and A[j] ≠ x }}
  {{ contains(A[0 .. j], x) = F }}
}
```

F = contains(A[0 .. j–1], x)
= contains(A[0 .. j –1] ⧺ [A[j]], x)   **def of** contains (**since** A[j] ≠ x)
= contains(A[0 .. j], x)