CSE 331

Arrays

Kevin Zatloukal

Recall: Turning Recursion Into a Loop

- Saw templates for structural recursion on
  - natural numbers — straightforward
  - lists — harder
- Special case for tail recursion on
  - lists — straightforward

Processing Lists with Loops

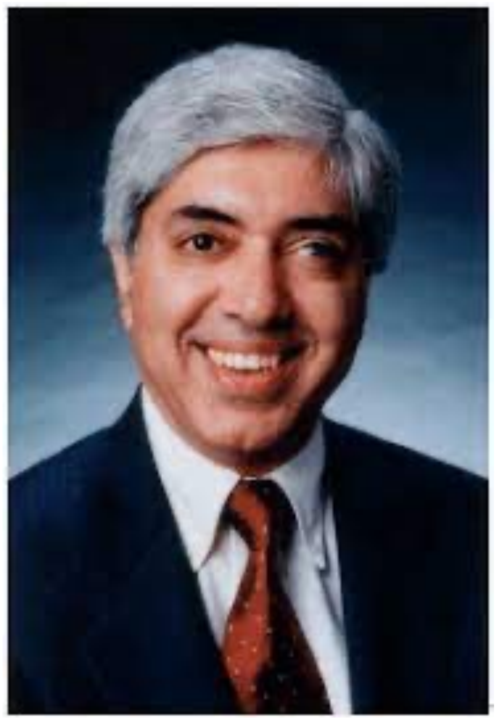
- **Hard to process lists with loops**
  - **only have easy access to the last element added**

    natural processing would start from the other end
  - **must reverse the list to work "bottom up"**

    that requires an additional O(n) space

- **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

Array Accesses

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

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

  — the Peter Parker Principle

- **Whenever we write "$A[j]$", we must check $0 \leq j < n$**
  - **new bug just dropped!**

    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 Literals

- **Write array values in math like this:**

  $$A := [1, 2, 3] \qquad\qquad (\text{with } A : \text{Array}_{\mathbb{Z}})$$

  - **the empty array is "[ ]"**

- **Array literal syntax is the same in TypeScript:**

```
const A: Array<number> = [1, 2, 3];
const B: number[] = [4, 5];
```

  - **can write** $\text{Array}_{\mathbb{Z}}$ **as** "`Array<number>`" **or** "`number[]`"

Array Concatenation

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

- The following properties hold for any arrays $\mathrm{A}, \mathrm{B}, \mathrm{C}$

$$\mathrm{A} \mathbin{+\!\!+} [] = \mathrm{A} = [] \mathbin{+\!\!+} \mathrm{A} \qquad \text{("identity")}$$

$$\mathrm{A} \mathbin{+\!\!+} (\mathrm{B} \mathbin{+\!\!+} \mathrm{C}) = (\mathrm{A} \mathbin{+\!\!+} \mathrm{B}) \mathbin{+\!\!+} \mathrm{C} \qquad \text{("associativity")}$$

  - we will use these facts *without* explanation in calculations
  - second line says parentheses *don't matter*, so
    we will write $\mathrm{A} \mathbin{+\!\!+} \mathrm{B} \mathbin{+\!\!+} \mathrm{C}$ 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 <u>not</u> 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

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

  - note that this includes $A[j]$

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

Subarrays

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

- **Define this formally as follows**

func $A[i\ ..\ j]$	$:= []$	**if** $j < i$
$A[i\ ..\ j]$	$:= A[i\ ..\ j\text{-}1] + [A[j]]$	**if** $i \leq j$

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

    $A[i\ ..\ j]$ is undefined if $i \leq j$ and $(i < 0$ or $n \leq j)$

  - **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

func A[i .. j]	:= []	**if** $j < i$
A[i .. j]	:= A[i .. j-1] ⧺ [A[j]]	**if** $0 \le i \le j <$ A.length
A[i .. j]	:= undefined	**if** $i \le j$ and ($i < 0$ or A.length $\le j$)

- **Some useful facts**

  A = A[0 .. n-1] (= [A[0], A[1], ..., A[n-1]])
  where n = A.length

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

  A[i .. j] = A[i .. k] ⧺ A[k+1 .. j]

  - **holds for any** $i, j, k : \mathbb{N}$ **satisfying** $i - 1 \le k \le j$ **(and** $0 \le i \le j < n$**)**
  - **we will use these *without* explanation**

TypeScript Arrays

- Translating math to TypeScript

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

    $0 \leq \mathrm{A[j]}$ for any $2 \leq \mathrm{j} \leq 10$
  - shorthand for 9 facts $(0 \leq \mathrm{A[2]}, \ldots, 0 \leq \mathrm{A[10]})$

Finding an Element in an Array

- Can search for an element in an array as follows

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

- Searches through the array in linear time
  - did the same on lists

- 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

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

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

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

    I will add one extra fact we also need

Sum of an Array

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

- **Loop implementation:**

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

could write "`j !== A.length`" but this is normal

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:**

```
let j: number = 0;
let s: number = 0;
{{ j = 0 and s = 0}}
{{ Inv: s = sum(A[0 .. j – 1]) and 0 ≤ 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 $A : \mathrm{Array}_{\mathbb{Z}}$

- **Loop implementation:**

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

$s = 0$
$= \mathrm{sum}([])$ def of sum
$= \mathrm{sum}(A[0 .. -1])$
$= \mathrm{sum}(A[0 .. j - 1])$ since $j = 0$

$j = 0$
$\leq A.\mathrm{length}$

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:**

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

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:**

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

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

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:**

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

$s = \text{sum}(A[0 .. j-2]) + A[j-1]$ **since** $s - A[j\text{-}1] = \text{sum}(..)$
$= \text{sum}(A[0 .. j-2] \mathbin{+\!\!+} [A[j-1]])$ **def of** sum
$= \text{sum}(A[0 .. j-1])$

Linear Search of an Array

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

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

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

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

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:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
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

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

- **Loop implementation:**

```
let j: number = 0;
{{ j = 0 }}
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
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 and 0 ≤ j ≤ A.length }}
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

$0 \le 0 = j$ and $j = 0 \le A.length$

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 and 0 ≤ j ≤ A.length }}
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

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

- **Loop implementation:**

```
let j: number = 0;
{{ Inv: contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
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;
```

$F = \text{contains}(A[0 .. j{-}1], x)$
$= \text{contains}(A[0 .. A.length - 1], x)$ **since** $j = \ldots$
$= \text{contains}(A, x)$

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

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

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 0 ≤ j < A.length }}

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

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

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

{{ contains(A[0 .. j−1], x) = F and 0 ≤ 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 0 ≤ j < A.length and A[j] ≠ x }}
  {{ contains(A[0 .. j], x) = F and 0 ≤ j+1 ≤ A.length }}
}
{{ contains(A[0 .. j], x) = F and 0 ≤ j+1 ≤ A.length }}
j = j + 1;
{{ contains(A[0 .. j–1], x) = F and 0 ≤ j ≤ A.length }}
```

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

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

```
}
```

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)