CSE 331

Equality

Kevin Zatloukal

Administrivia

- **Section tomorrow is final exam practice**
  - **will focus on problems not included in midterm**

    mutable ADTs and writing loops given invariant

- **Will email some reading tonight**
  - **problems reference an ADT definition**
  - **time in section is too short to read and do problems**

    final exam is 1 hour and 50 minutes, so there will be time to read then

Equality

Equity of User-Defined Types

- For any type, useful to know which are "the same"

- TypeScript "`===`" is not useful on records:

```
{a: 1} === {a: 1}  // false!
```

  - as in Java, this is "reference equality"
  - tells you if they refer to the same object in memory

- `deepStrictEquals` would work here
  - checks that the records have the same fields and values
  - but that also is not perfect…

Recall: Queue With Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = concat(this.front, rev(this.back))
  readonly front: List<number>;
  readonly back: List<number>;
```

– two ways of representing the same abstract state:

```
{front: cons(1, cons(2, nil)), back: nil}  // = 1, 2
{front: nil, back: cons(2, cons(1, nil))}  // = 1, 2
```

– these should be considered equal!

Equality

- **Often useful / necessary to define your own `equal`**
  - **check if references point to records that are "the same"**

- **Sensible definition should act like "=" in math:**

  1. $\text{equal}(a, a) = T$ for any $a : A$ — reflexive

  2. $\text{equal}(a, b) = \text{equal}(b, a)$ for any $a, b : A$ — symmetric

  3. if $\text{equal}(a, b)$ and $\text{equal}(b, c)$, then $\text{equal}(a, c)$ for any … — transitive

  - **(311 alert: this is an "equivalence relation")**
  - **Java has two more rules for equals (see Java docs)**

Example: Duration

- **Define** Duration **representing an amount of time**

$$\textbf{type}\ \text{Duration} = \{\text{min} : \mathbb{Z}, \text{sec} : \mathbb{Z}\}\ \ \text{with}\ \ 0 \le \text{sec} < 60$$

  - **second part is an invariant**

- **Can define equality on** Duration **this way:**

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \ \ := \ \ (\text{m} = \text{n}) \text{ and } (\text{s} = \text{t})$$

  - **true iff these are the same amount of time**
    (wouldn't be true without the invariant)

Example: Duration

equal({min: m, sec: s}, {min: n, sec: t}) := (m = n) and (s = t)

- **Does this have the required properties?**
  - reflexive

equal({min: m, sec: s}, {min: m, sec: s})
= (m = m) and (s = s) **def of** equal
= T and T
= T

proof by calculation

  - symmetric

equal({min: m, sec: s}, {min: n, sec: t})
= (m = n) and (s = t) **def of** equal
= (n = m) and (t = s)
= equal({min: n, sec: t}, {min: m, sec: s}) **def of** equal

Example: Duration

equal({min: m, sec: s}, {min: n, sec: t}) := (m = n) and (s = t)

- Does this have the required properties?
  - reflexive yes
  - symmetric yes
  - transitive **also yes** (but a little long for a slide)

- Good evidence that this is a reasonable definition

Example: List Equality

- **Can define equality on List type this way:**

equal(nil, nil)	:=	T	
equal(nil, cons(b, R))	:=	F	
equal(cons(a, L), nil)	:=	F	
equal(cons(a, L), cons(b, R))	:=	F	if $a \neq b$
equal(cons(a, L), cons(b, R))	:=	equal(L, R)	if $a = b$

- **Checks that the values in the list are all the same**
  - **this is a definition, so we can only check it on examples…**

equal([1] → [2] , [1] → [2]) = equal([2] , [2])

= equal(nil, nil)

= T

Example: List Equality

- **Can define equality on** List **type this way:**

equal(nil, nil)	:=	T	
equal(nil, cons(b, R))	:=	F	
equal(cons(a, L), nil)	:=	F	
equal(cons(a, L), cons(b, R))	:=	F	if $a \neq b$
equal(cons(a, L), cons(b, R))	:=	equal(L, R)	if $a = b$

- **Checks that the values in the list are all the same**
  - **this is a definition, so we can only check it on examples…**

$$\text{equal}(\boxed{1} \rightarrow \boxed{2}, \boxed{1} \rightarrow \boxed{3}) = \text{equal}(\boxed{2}, \boxed{3})$$

$$= \text{F}$$

Example: List Equality

- **Can define equality on** List **type this way:**

$$\begin{aligned}
&\text{equal}(\text{nil}, \text{nil}) &&:= \quad \text{T} \\
&\text{equal}(\text{nil}, \text{cons}(\text{b}, \text{R})) &&:= \quad \text{F} \\
&\text{equal}(\text{cons}(\text{a}, \text{L}), \text{nil}) &&:= \quad \text{F} \\
&\text{equal}(\text{cons}(\text{a}, \text{L}), \text{cons}(\text{b}, \text{R})) &&:= \quad \text{F} &&\text{if a} \neq \text{b} \\
&\text{equal}(\text{cons}(\text{a}, \text{L}), \text{cons}(\text{b}, \text{R})) &&:= \quad \text{equal}(\text{L}, \text{R}) &&\text{if a} = \text{b}
\end{aligned}$$

- **Has all three required properties**
  - **how would we prove this?** induction

Recall: Subtypes of Concrete Types

- We initially defined types as sets
- In math, a **subtype** can be thought of as a **subset**
  - e.g., the even integers are a subtype of $\mathbb{Z}$
  - e.g., the numbers {1, 2, 3, 4, 5, 6} are a subtype of $\mathbb{Z}$
  - likewise, a **superset** would be a **supertype**
- Any even integer "is an" integer
  - "is a" is often (but not always) good intuition for subtypes

Recall: Subtypes of Abstract Types

- Subtypes are **substitutable** for supertype
  - this is the "Liskov substitution principle"
  - due to Barbra Liskov

- For ADTs, we use this as our definition of subtype

- When is ADT B substitutable for A?

  1. B must provide all the methods of A

     If A has a method "f", then B must have a method called "f"

  2. B's corresponding method spec must be stronger than A's

     must accept all the inputs that A's does

     must also promise everything in A's postcondition

Example: Duration Again

```
// Represents an amount of time measured in seconds
class Duration {

  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  readonly min: number;
  readonly sec: number;

  equal = (d: Duration): boolean => {
    return this.min === d.min && this.sec === d.sec;
  };

  …
```

– **defines `Duration` as an ADT instead**

  `getMinutes` and `getSeconds` methods not shown

  `equal` still makes sense, just as before

Example: NanoDuration

- **Suppose a subclass also measures nanoseconds**

```
class NanoDuration extends Duration {
  // min: number (inherited)
  // sec: number (inherited)
  readonly nano: number;

  …
```

- **How should we define `equal`?**
  - **remember that it takes an argument of type `Duration`**

    we cannot accept fewer arguments

Example: NanoDuration

```
class NanoDuration extends Duration {

  // min: number (inherited)
  // sec: number (inherited)
  readonly nano: number;

  equal = (d: Duration): boolean => {
    if (d instanceof NanoDuration) {
      return this.min === d.min &&
             this.sec === d.sec &&
             this.nano === d.nano;
    } else {
      return false;
    }
  };
```

Must take `Duration` argument to be a subtype

No! It lacks symmetry

– does this have the three required properties?

Example: NanoDuration

```
const d = new Duration(2, 10);
const n = new NanoDuration(2, 10, 300);

console.log(n.equal(d));   // false
console.log(d.equal(n));   // true!
```

- `NanoDuration` **is only equal to other** `NanoDuration`**s**
- `Duration` **can be equal to a** `NanoDuration`
  if they have the same minutes and seconds

Example: NanoDuration

```
class NanoDuration extends Duration {
  // min (inherited)
  // sec (inherited)
  readonly nano: number;

  equal = (d: Duration): boolean => {
    if (d instanceof NanoDuration) {
      return this.min === d.min &&
             this.sec === d.sec &&
             this.nano === d.nano;
    } else {
      return this.min == d.min && this.sec == d.sec;
    }
  };
```

No! It lacks transitivity

– fixes symmetry! all good now?

Example: NanoDuration

```
const n1 = new NanoDuration(2, 10, 300);
const d = new Duration(2, 10);
const n2 = new NanoDuration(2, 10, 400);

console.log(n1.equal(d));    // true
console.log(d.equal(n2));    // true
console.log(n1.equal(n2));   // false!
```

– **transitivity requires `n1` to equal `n2` (but it doesn't)**

Subclasses and Equals Don't Always Mix

- No good solution to this problem!
  - inherent tension between subtyping and equality
    - subtyping wants subclasses to behave the same
    - equality wants to treat them differently (using extra information)
- This is a general problem for "binary operations"
  - equality is just one example
- Real issue may be that `NanoDuration` isn't a subtype
  - subclass does not mean subtype
  - (would have seen this if we documented the ADT properly)

Example: NanoDuration Again

- Suppose a subclass also measures nanoseconds

```
// Represents an amount of time in nanoseconds
class NanoDuration extends Duration {

  // RI: 0 <= sec < 60 and 0 <= nano < 10000
  // AF: obj = 60,000,000 * this.min +
  //           1,000,000 * this.sec +
  //           this.nano
  readonly nano: number;

}
```

- Abstract states of the two types are different
  - time in seconds vs nanoseconds
  - abstract states of subtypes would need to be subtypes

Constructors

Public Constructors

- Most Java classes have public constructors
  - e.g., create an `ArrayList` with "`new ArrayList<String>()`"

- For our ADTs, we didn't do this
  - class was hidden (not exported)
  - we exported a "factory function" that used the constructor

    e.g., `makeSortedNumberSet`
  - this was not accidental…

- Constructors have undesirable properties
  - surprisingly error-prone
  - several important limitations

Recall: Tight Coupling (Example 3)

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    int time = job.getTime();  // just one call
    total += time;
    addToLists(job.getName(), time);
  }
}
```

RI is not true in method call!

Method Calls from Constructors

- Any method call from a constructor is dangerous!

- Almost always calling with RI false
  - usually, the RI does not hold until all fields are assigned
    typically, that is the last line of the constructor
  - hence, any methods are called with the RI still false

- Asking for trouble!
  - method needs to know that some parts of RI may be false
  - eventually, someone changing code will mess this up
  - better to avoid method calls in the constructor

Limitations of Constructors

- Constructor is called *after* the object is created
  - can't decide, in the constructor, not to create it

- Limitations of constructors

  1. Cannot return an existing object

  2. Cannot return a different class

  3. Does not have a name!

Singleton

- Factory functions can return an existing object
- Common case: there is only one instance!
  - factory function can avoid creating new objects each time
  - called the "singleton" design pattern
- Example from HW5…

Example: Singleton in HW5

```
// @returns ColorList containing all known colors
export const makeSimpleColorList = (): ColorList => {
  return new SimpleColorList(COLORS);
}
```

– every object returned is the same
– no need to make more than one

```
const simpleColorList = new SimpleColorList(COLORS);

// @returns ColorList containing all known colors
export const makeSimpleColorList = (): ColorList => {
  return simpleColorList;
}
```

Note: only allowed because `SimpleColorList` is **immutable**

Returning a Subtype

- Factory functions can return a subtype
  - declared to return `A` but returns subtype `B` instead
  - allowed since every `B` is an `A`

- Example:

```
// @returns an empty NumberSet that can be used to
//      store numbers between min and max (inclusive)
const makeNumberSet = (min: number, max: number): NumberSet => {
  if (0 <= min && max <= 100) {
    return makeArrayNumberSet();  // only supports small sets
  } else {
    return makeSortedNumberSet(); // use a tree instead
  }
}
```

Multiple Constructors

- Java classes allow multiple constructors

```
class HashMap {
  public HashMap() { … }  // initial capacity of 16
  public HashMap(int initialCapacity) { … }
}
```

- TypeScript classes do not, but
  you can fake it with *optional* arguments

```
class HashMap {
  constructor(initialCapacity?: number) { ... }
}
```

Constructors Have No Name

- Do not get to name constructors
  - in Java, same name as the class
  - in TypeScript, called "constructor"

- Names are useful

  1. Let you <u>distinguish</u> between different cases
     - use names to distinguish cases that otherwise look the same

  2. Let you <u>explain</u> what it does
     - the only thing you know the client will read!

Example: Distinguishing Constructors

- JavaScript's Array has multiple constructors

```
new Array()             // creates []

new Array(a1, …, aN) // creates [a1, …, aN]

new Array(2)            // creates [undefined, undefined]
```

  – what does "`new Array(a1)`" return when `a1` is a number?
  – how to make a 1-element array containing just `a1`

```
const A = new Array(1);
A[0] = a1;
```

  – don't have a name to distinguish these cases!

Example: Distinguishing Constructors

- Factory Functions have names
  - allow us to distinguish these cases

```
// @returns []
const makeEmptyArray = (): Array => { … };

// @returns A with A.length = len and
//     A[j] = undefined for any 0 <= j < len
const makeArray = (len: number): Array => { … };

// @returns [args[0], …, args[N-1]]
const makeArrayContaining = (...): Array => { … };
```

Example: Distinguishing Constructors

- Factory Functions have names
  - allow us to distinguish these cases

```
// @returns []
const makeEmptyArray = (): Array => { … };

// @returns A with A.length = len and
//     A[j] = undefined for any 0 <= j < len
const makeArray = (len: number): Array => { … };

// @returns A with A.length = len and
//     A[j] = val for any 0 <= j < len
const makeFilledArray =
    (len: number, val: number): Array => { … };
```

Be very, very careful…
Type checker won't notice if client mixes these up!

Argument Order Bugs

- Some famous bugs due to mixing up argument order!
- If you program long enough, you will see this one
  - … and just about every other bug

Carla Notarobot
@CarlaNotarobot

Junior dev: "I f[redacted]ed up bad, I'm so fired"

Senior dev: " I have 3 production outages named after me lol"

6:48 PM · Jan 12, 2022

Use Records to Force Call-By-Name

- Can use a record to make clients type names

```
// @returns A with A.length = len and
//      A[j] = val for any 0 <= j < len
const makeFilledArray =
    (desc: {len: number, value: number}): Array
```

  - takes one argument, not two
  - client writes "`makeFilledArray({len: 3, value: 0})`"

- Think about mistakes clients might make
  - be paranoid when debugging will be painful