Outline

2019年11月29日 16:33

- Specs
- Testing - Exceptions
- ADTs (interfaces/classes)
    - Rep invariant
    - Abstraction Function
        - Mutability
    - Subtypes (informed by specs)
- Equality (AFs)
- Recursion & recursive types
- Higher-order functions
    - A function can be the argument of another function
- Grammars and parsing
- Concurrency
    - Parallelism (easy)
    - Shared data (hard)
        - Mutability

Data types

2019年10月8日 14:06

High quality software is correct, comprehensible and changeable
Definition: A type is a set of values and the operations that are permitted on the values.
Primitive types: int, double, float, String…
Cannot be extended.
Objects: Integer, Boolean can be extended
User-defined types(rely on primitive types/other user-defined types): Date.
In Java, Date is a class, and instances of it are objects.
Developer: develops types
User: uses the type to build applications

Access control:

- Private: enables defensive programming, modifications from outside is not allowed
  - Keep internal things private, easier to understand
- Public: freely accessible to other parts of the program. Won't be ready for change
- Final: the variable should never be reassigned, does not mean immutable
  - Elements of a final mutable objects (e.g. lists) can be changed
  - It's an immutable reference rather than immutable object
- Static: can be called without a particular object or instance, it keeps the last value it is assigned, regardless of the called object
- Protected: accessible to subclasses
- Instance: must be called on a particular object or instance

Type checking :

- Dynamic checking: verification of correctness at runtime

Stack and heap:

- Stack
  - Stack is a last in first out data type
  - If a stack is empty, the program terminates
  - All functions, variables has a pointer(address) on stack
  - All variables of a function are on the stack when executed. After the execution, they are removed
- Heap:
  - Smaller address than stack
  - No size restriction
  - Must use malloc() or calloc() to allocate memory
  - Must be deleted manually after use

Note: obj1==obj2 checks their address

Specifications

2019年10月8日 14:14

Definition: A specification is a description of what an artifact does.
Definition: A functional specification describe the functionality that is expected

- Preconditions: the restrictions on the use and input of the method/function
  - @param (requires)
- Postconditions: the intended output/behavior of the method/function based on the preconditions
  - @return(effects)
  - Throwing an exception potentially changes the postconditions
- Frame condition ("modifies"): identifies which (input) objects may be modified
  - Used for mutations
  - Included in @param
- If precondition holds when the method is called, then the postcondition must hold when the method completes

Tests:

- Black box tests: only knows the specifications
- White/glass box tests: knows the specifications as well as codes

Specifications allows restrict the domain and range of a method

Dimensions for comparing specifications:

- Deterministic: only a single output for a given input
- Declarative: spec characterizes what the output should be
  - Operational: give a series of steps that the method performs
  - Declarative: just the output and how it relates to the initial state(preferable)
- Strong: the spec has a small set of legal implementations

Stronger/weaker specifications:

- Spec A is stronger than or equal to spec B if:
  - A's precondition is weaker than or equal to B's
  - A's postcondition is stronger than or equal to B's
  - In this case, implementation that satisfies A can satisfy B
- E.g.
  - Statement1: $10<x<20$ is stronger than statement2: $x>10$
  - If set as preconditions, spec A is weaker than spec B
  - If set as postconditions, spec A is stronger than spec B
- The set of strong specs is a subset of the set of weak specs
- If spec A is neither stronger nor weaker than spec B, they might overlap or disjoint

Stronger specs allow more possible inputs

Good specifications:

- Coherent: should not have lots of different cases
- Informative:
- Strong enough: (throwing exceptions often weaken the spec)
- Weak enough: should have enough details
- Use abstract types where possible: e.g. Map instead of HashMap

To find bugs, it is better to fail fast
For binary search, we often need a precondition that the array is sorted

Decision to use a precondition depends on

- the cost of the check,
- Scope of the method

Weak specification (preconditions) gives more efficient implementation

Testing

2019年10月8日 17:31

Validation:

- Testing: Running the program on selected inputs and checking the results
- Code review: having others to proofread the code
- Formal reasoning(verification): formal proof

Principle of testing: make it fail

Ways of testing:

- Exhaustive testing: infeasible - too many test cases
- Haphazard testing: just try and see if it works - less likely to find bugs
- Random or statistical testing: does not work well for software
- Test suits: a set of test cases that cover different aspects of a program's behavior
    - Small enough to run quickly
    - Big enough to validate the program
    - Divide input space into subdomains
        - Consider boundary conditions
        - We can choose from
            - Full cartesian product, i.e. all possible combination of subdomains
            - Cover each part, i.e. each part is covered but not every combination

Black/White box testing

- Black box testing: choosing test cases only from the specification
    - It cannot violate the specification
    - Based on preconditionsWhite box testing: choosing test cases with knowledge of how the function is implemented

Coverage

- Statement coverage: is every statement run by some test case
- Branch coverage: if and while, both true and false tested (desirable)
- Path coverage: every possible combination of branches covered (infeasible)

Automated and regression testing

- Junit help build automated test suites
- Regression testing: rerun tests when code is modified

Code review

- Style standards
- Avoid duplication
- Comments
    - Specification document assumptions
- Fail fast
- Avoid magic numbers: constant numbers need to be explained
    - E.g. in Date, use JANUARY instead of 1
- One purpose for each variable
    - Do not reuse
- Use good names
- Do not use global variables
- Coherent methods: one method does only one thing
- Methods should return results, not print them
- Use whitespace to help the reader

Exceptions

2019年10月8日 19:41

Types:

- Checked exceptions:
  - To signal special results
  - Checked by the compiler
  - They must be handled, or it will be a compiler error
  - Use try-catch-final or throws
    - Final will always run
  - All throwables and all of their subclasses except for Runtime Exception and Error, are checked
- Unchecked exceptions:
  - To signal bugs
  - Not to be handled
  - Will compile but will crash
  - E.g.: Runtime Exception, Error and their subclasses

Static error is caught before the program runs
Dynamic error is thrown during runtime

Throwable type:

- Class of objects that can be thrown or caught
- Exception
  - Runtime exception
- Error(unrecoverable and not caught)
  - Don't subclass it

Cost of throwing exceptions

- Performance (extends running time)
- A pain to use
  - Probably need to create a new class for the exception
  - Must use try-catch for a checked exception

Mutability

2019年10月8日 19:41

We can model mutability by finite state machines
Mutation: a reference variable points to changes
Immutable: once created, they always represent the same value

- All primitive types: String, Int, double, BigInteger
- Reduce bugs
- New objects created when changing, old ones be garbage-collected

Mutable: have methods that change the value of the objects

- String Builder, lists

Defensive copying: return a copy of a mutable object, without changing the old one
Immutability can be more efficient than mutability, because immutable types never need to be defensively copied.

Representation exposure: code outside the class can modify the representation directly. We must avoid this

Immutable Wrapper: Collections.unmodifiableList() takes a list and disable the mutators

- It provides immutability only at runtime, not at compile time

Aliasing: having multiple names/references for the same mutable object is risky

Iteration: used in for loops of a mutable object

- Next(), hasNext()

State machines

- State
- Init (initial state)
- Event (operation that change states)
- Transition
- They might not be deterministic (transitions may be indeterministic)

State machine coverage:

- All actions (create, has next, next)
- All states (create, next)
- All transitions (create, has next, next, next, has next)

Instance variables:

- Constructor: create new objects of a type
- Mutator: returns an element, also advances the iterator (alter the object)
    - E.g. add() in Lists, all set() methods
- Producers: create new objects from old objects of the same type
    - Concat() for strings
- Observers: takes input as an object of the abstract type and return a different type of object
    - Size() for lists

Total/Partial procedures

- Total procedure
    - Works for any input within the type checking restrictions of the language
- Partial procedure:
    - Works for some of the possible inputs
        - Not all intakes can be processed

- Has preconditions (requires clause)

Binary search: find index in an array, it is a partial procedure

Abstraction

2019年10月9日 16:19

Abstraction: omitting or hiding low-level details with a simpler, higher-level idea
Modularity: dividing a system into components or modules, each can be designed, implemented, tested, reasoned and reused separately
Encapsulation: building walls around a module so that the module is responsible for its own internal behavior. Other parts cannot damage it
Information hiding: hiding details of module's implementation from the rest of the system so that those details can be changed later without changing other parts
Separation of concerns: one feature is responsible for by a single module

An abstract data type is characterized by operations that can e performed on it
To design an abstract datatype, we need to choose a good set of operations and their behavior

- Better to have a few simple operations that can be combined in powerful ways
- Each operation should have a well-defined purpose and have a coherent behavior

A type may be generic (e.g. list, set, graph), or domain specific (street map), but it should not be a mixture of both

Representation independence:
A good abstract data type should be representation independent

- The use of an abstract type is independent of its representation
- Change in representation have no effect on code outside the abstract type
- E.g. List operations are independent of whether list is a linked or an array list

Interface

- Useful for expressing an abstract data type
- List of method signatures, implementation is in a different class
- Class A implements <interface>
- Advantages:
    - Interface specifies the contract for the client
    - Multiple different representations of the abstract data type can co-exist in the same program with different classes implementing the same interface
        - Note: abstract class is harder to have multiple representations
- E.g. Set is an interface, HashSet implements the interface
- Factor method: using a static method as a creator instead of a constructor
- Why using it?
    - Help compiler to catch implementation bugs and help human read specs
    - Allow performance trade-offs
        - We can have different performance characteristics but they are representation-independent
    - Flexibility in providing invariants:
        - HashSet has no ordering, but treeSet has
    - Optional methods: list mutators
    - Methods with intentionally underdetermined specifications
    - Multiple views of one class
        - A class can implement multiple interfaces
    - Trustworthy implements
        - Easy to make sure an implementation is correct if the interface is correct

Invariants:

- An invariant is a property of a program that is always true, for every possible runtime state of the program

- Good abstract data type preserves its own invariants
- E.g.: Immutability
- To establish invariants
    - Make the invariant true in the initial state and ensure all changes to the object keep the invariant true
        - creators/producers must establish invariant for the new objects
        - Mutators and observers preserve the invariant
        - Need to avoid representation exposure

Summary
A good ADT is simple, adequate, representation-independent and preserves its own invariants
Representation exposure threatens both representation independence and invariant preservation
Interfaces help formalize the idea of an abstract data type as a set of operations that must be supported by a type

RI and AF

2019年10月26日 8:27

Concrete Representation

- Definition: it shows how the abstract state of a class is represented within a Java object

Specification Field

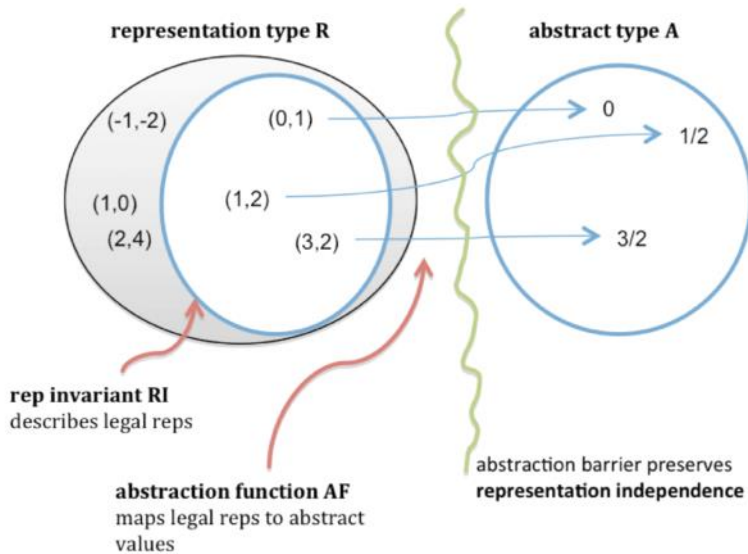
- Definition: An attribute of a datatype that a client may use and retrieve

Representation Invariant

- Definition: a condition that must be true over all valid concrete representation of a class. It defines the domain of the abstraction function. It is not limited to where specification includes an abstract invariant
- Purpose: to define valid and invalid internal states for this ADT object
- Makes it easy to catch bugs caused by a corrupted data structure
- E.g.:
    - To represent numbers from zero to nine, we need to restrict symbols from 0 to 9, these symbols are representation invariants
- Notation: $RI: R(rep) \rightarrow \text{boolean}$
    - $RI(r)$ is true if and only if $AF(r)$ is well-defined.
    - It tells us whether a given rep value is well-formed
    - RI can be thought of as a subset of rep values on which AF is defined
- We need to check rep invariant if the implementation asserts the rep invariant at run time
- There should be no null values in the representation invariant

Abstraction Function

- Definition: a function mapping from an object's concrete representation to the abstract value it represents
- Gives a way to clearly define the equality operation on an abstract data type
- E.g.:
    - The mapping from 4 to the number four is an abstraction function
- Every abstract value is mapped to by some representation value
    - E.g. String (Representation space) to Set of characters (Abstract value)
        - ""(rep)→ $\{ \}$(abstract), "abc","bac"(rep)→ $\{a, b, c\}$(abstract)
- Some abstract values are mapped to by more than one rep value
    - E.g. "abc","bac"(rep)→ $\{a, b, c\}$(abstract)
- Not all rep values are mapped
    - E.g. rep values containing duplicates
        - "abbc" has no abstract mapping value to a set of characters
- Notation $AF: R(rep) \rightarrow \mathrm{A(abstract)}$
    - R is an java object,
    - A exists only in our imagination, not in the computer
- It is surjective, not necessarily bijective(one-to-one), often partial



Note:

- With the same RI, we can still interpret the representation differently, with different AF
- RI and AF are recorded to help with implementing, testing, debugging, modifying and extending the type
- RI and AF should appear as internal comments in the class body
- Designing an abstract type means two things:
    - Choose the two space
        - abstract value space for the specification
        - Representation value space for the implementation
    - Decide what representation values to use and how to interpret them

Writing RI and AF

- Representation invariant
    - May mix concrete Java syntax with abstract mathematical syntax
    - May be simple English
    - Not appropriate to refer to the representation fields of another class directly
    - To choose representation invariant
        - Look for rep values on which the abstraction function has no meaning
            - E.g. index<0 or >51 for play cards
        - Look for rep values on which your methods would produce the wrong abstract value
            - E.g. duplicates in Strings → charSet
        - Look for constraint required by the data structures or algorithms
            - E.g. array must be sorted in order to use binary search
            - Tree cannot have cycles
            - Two tree nodes cannot have the same child
        - Look for fields that need to stay coordinated with each other
            - E.g. balance and sum of transaction should always be in sync
        - Look for rep values that would cause your code to throw unexpected exceptions
            - E.g. null pointers, division by zero, index out of range, class cast
        - Look for constraints imposed by the application domain(abstract values)
            - E.g. balance never went negative for any transactions
            - Abstract invariants
                - Refer only to properties of the abstract values
                - Should be documented for the client
                - Imply constraints that need to be included in the rep invariant
    - Checking RI at runtime
        - Using checkRep() function, right after the filed
            - Throw an exception if the rep invariant is violated.
            - Intersperse the constraints of rep invariant as exception messages
            - Use assertion

- Call at the start and end of every public method and at the end of every constructor
      - We can add a static boolean variable to enable and disable the checkRep
      - Making checkRep() public does not expose the representation
  - Do not write:
    - Facts that are impossible in the rep. (unnecessary)
      - E.g. value is less than or equal to MAX_VALUE, null
    - Rep invariant that
      - cannot be checked by just examining the concrete representation
- Abstraction functions
  - $AF: R \to \mathrm{A}$
  - Whether the type is mutable or immutable is a crucial property that should be mentioned in the class overview
  - Recursive defined function:
    - $AF(r) = [\square]$ if $r = null$
    - $[r.first]: AF(r.rest)$ if $r \,! = null$
    - [] and : are sequence construction syntax
  - We can write the right part only, if the abstraction function is not recursively defined
  - It's a clear and unambiguous communication with other human
  - Need to be simple and clear
    - Introduce new names where they are useful
    - Use spec fields
    - Provide examples
    - Introduce new functions where they are useful
    - Use plain language wherever it is unambiguous and clearer or more concise than formal syntax

Equality

2019年10月26日 14:52

Three ways to regard equality

- Using an abstraction function
  - $f: R \to \mathrm{A}$
  - a equals b if and only if $f(a) = f(b)$
- Use a relation:
  - An equivalence is a relation $E \subseteq \mathrm{T} \times \mathrm{T}$ that is
    - Reflexive: $t = t$
    - Symmetric: $s = t$ then $t = s$
    - Transitive: $x = y$ and $y = z$ then $x = z$
  - a equals b if and only if $E(a, b)$
- Using observation
  - Two objects are equal when they cannot be distinguished by observation
    - Every operation produces the same result for both objects

Reference equality

- "==" operator compares reference (referential equality)
- $a == b$ if a and b points to the same object

Object equality

- equals() operation compares object contents
- The default meaning is the same as referential equality
  - we need to override it to compare immutable data types
    - Use @override and instance of
  - Using overload may cause trouble when the method signature is not identical to Objects
- Has to be defined appropriately for every ADT

Object contract: the specification of the Object class
For the equals() method, the contract is

- Must define an equivalence relation
- Must be consistent in repeated calls
- For a non-null reference x, x.equals(null) should return false
- hashCode must produce the same result for two objects that are deemed equal by the equals() method
  - Note: $a.equals(b)$ means $a.hashCode(\square) == b.hashCode$
  - However $a.hashCode(\square) == b.hashCode$ does not imply $a.equals(b)$
  - We need to provide the same hashcode for two objects if they are equal
    - i.e. always override hashcode when override equals
  - Also, we want to reduce the probability that two objects that are not equal have the same hash code

Equality of mutable types

- Observational equality
  - When they cannot be distinguished by observation that does not change the state of the objects
    - Calling observers, producers, and creators behave the same
- Behavioral equality
  - When they cannot be distinguished by any observation, even state changes
    - Calling mutators behave the same.
- These two equality are identical for immutable types
- For mutable types, strict observational equality
  - Because mutation changes the hashCode and hash set does not realize the change
  - We should inherit equals() and hashCode() from Object for mutable objects

Final rule
- For immutable type:
    - Equals() should compare abstract values
        - i.e. provide behavioral equality
        - Abstraction function is the basis
    - hashCode() should map the abstract value to an integer
    - Override both equals() and hashCode()
- Mutable types
    - Equals() should compare references
        - i.e. provide behavioral equality
        - Reference equality is the only way to ensure consistency for equality in mutable data types
    - hashCode should map the reference into an integer
    - Do not override equals() and hashCode()

Autoboxing and equality
- The behavior that Java automatically converts between int and Integer is called autoboxing
    - They are not interchangeable
    - E.g. Integer x = new Integer(3); Integer y = new Integer(3);
        - We can only use equals() or cast to (int) to get equal

CompareTo() method
- It expresses ordering
- Return negative integer if this object is smaller than the object it is being compared to (obj)
- Return 0 if two objects are equal
    - compareTo should be consistent with equals()

Subtyping

2019年10月26日 15:49

A hierarchy of subtypes can be achieved using interfaces or subclasses
Subtyping is related to polymorphism
Polymorphism: dynamic method invocation

- The runtime or dynamic type of the object that determines the method that is executed
- E.g. Class A has $f(\square)\{return\ 10;\}$, class B extends Class A has $f(int\ x)\{return\ x;\}$
  - If we have an object B b = new B() and call $B.f()$, it will return 10
  - If we have an object A a = new A() and call $A.f(5)$, it will be an compiler error

Functional interfaces and delegation

- Functional interface is an interface that has exactly one method
  - Indicate with @FunctionalInterface
- Delegation: a software design pattern where an object relies on another object not for data but for functionality
  - Enables code reuse
  - We can use delegation to achieve code reuse and include additional functionality
    - By using a class composed of an object and delegates all the non-logging functionality to the object
  - Smaller interfaces with clear contracts(specifications)

Subtyping

- Use implements keyword
- E.g. ArrayList and LinkedList are subtypes of List interface
  - They differ in representation and support other methods that are nor part of the List interface
- If B is a subtype of A, then every object/value of type B is also an object/value of type A
- If B is a subtype of A, A and B need not have the same representation or the same abstraction function.

Subclass

- Use extends keyword
- It uses polymorphism
- Allows us to reuse the code in the superclass without having to reimplement a method
- We can use super(arguments) to call the constructor in the super class
- Inheritance

Subtyping and the Liskov Substitution Principle

- B is a subtype of A means every B object is also an A object
- Liskov Substitution Principle: subtypes must be substitutable for their super types
  - i.e. a subtype must fulfill the same contract as its super type
- B is a subtype of A if B's specification is at least as strong as A's specifaction
  - In programming languages, mutable square is not a subtype of mutable rectangle
    - Because square's specification is weaker than rectangle's
  - But immutable square is a subtype of immutable rectangle
- Class B cannot extend class A and then override some method to return a different type or throw new checked exceptions
- If declare a subtype to Java, it must be substitutable

Subclass vs. subtype

- Inherited superclass methods can break the subclass's rep invariant
- When a subclass overrides superclass methods, it may depend on how the superclass uses its own methods
  - E.g. overriding addAll in a subclass of arraylist may cause the count number to be

doubled
- When a class is subclassed, it must freeze its implementation forever, or all its subclasses must evolve with its implementation.
    - Or subclass may break the new rep invariant

Covariant and contravariant
- Covariant: return type is smaller
- Contravariant: argument type is larger

Violating Liskov Substitution Principle
- Mutability (subclassing)
    - If there is a mutable class extending an immutable class
        - Clients that depend on the superclass immutability may fail when given subclass values
    - Mutable counterparts of immutable classes should not be declared as subtypes
- Adding values in the subtype
    - E.g. BigInt extends BigNat
        - We cannot say that every big int is a big nat
        - Big int fails to make guarantees made by big nat
            - So the spec of big int is not at least as strong as the spec of big nat
                - Might be incomparable
- Specifications
    - Consider setSize() for square and rectangle
        - The method has more restrictions on the input for square, and the spec for square is weaker than rectangle
        - We should not overload a method in subclass

Wrapper pattern
- A wrapper modifies the behavior of another class without subclassing
- It decouples the wrapper from the specific class it wraps
    - E.g. countingList implements List, could wrap an arrayList, linkedList, or another countingList

Summary
- B is a subtype of A, if B satisfies A's specification(contract)
    - B should guarantee all the properties that A does (e.g. immutability)
    - B's methods should have the same or weaker preconditions and the same or stronger postconditions as those required by A. (i.e. stronger specs)
- When subclassing is necessary, design for it
    - Define a protected API for subclasses, in the same way you define a public API for clients
    - Document for subclass maintainers how you use methods
    - Do not expose representation
- Subclass breaks encapsulation, we must use subclass only for true subtype

Interfaces
- Every implementation of an interface must implement all methods in the interface
    - Does not directly allow us to reuse code
- Any class that defines all the required methods and follows the specification of an interface can be a subtype of the interface
- Easy to implement to interfaces
- Can build type structures that are not strictly hierarchical

Abstract Class
- A class that can only be subclassed, cannot be instantiated
- Can provide some instance methods, when interfaces cannot
- Only subclass of abstract class can implement the type defined by the abstract class
- Cannot extend to a new abstract class

- Abstract class need to be high up in the type hierarchy

Abstract skeletal implementation

- Defined by the interface
  - E.g. public interface A{}, public interface B extends A{}
- The skeletal implementation makes the type easier to implement
- Can be combined with the wrapper class pattern
- Need to break down the interface and decide which operations can serve as primitives in terms of which the other operations can be defined
  - Primitive operations will be left unimplemented in the skeleton

Recursion

2019年11月1日 18:13

Recursive process(object) has two properties:

- A set of base cases, that do not rely on recursion and indicate terminal steps
- A recursive step that reduces an instance of the process to other instances of the same process, in the direction of one of the base cases

The correctness of recursion can be proved by mathematical induction
Decomposition: good decompositions are simple, short, easy to understand, correct and changeable

Structure of recursive implementations:

- General approach:
  - Base case: the simplest, smallest instance of the problem
    - Usually empty objects or zero
  - Recursive step: decompose a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls and later combined
- Helper method
  - It is useful to require a stronger specification for the recursive steps to make the recursive decomposition simpler or more elegant (make a new method)

Reentrant code:

- Direct recursion (a method calls itself):
  - E.g. factorial(n-1) can be called when factorial(n) has not finished
- Mutual recursion(two or more functions call each other):
  - A calls B, which calls A again

Reasons for using recursion

- Problem/data is naturally recursive
- Immutability

Drawbacks: take more space in stack

Mistakes:

- Missing base case
- Recursive steps does not reduce to a smaller subproblem (does not converge)

Recursive data:

- File system: folders contain other folders which contain folders, finally files

Recursive datatypes:

- Immutable lists
- Tree
  - Binary tree

Recursive datatype definitions:

- It has
  - An abstract datatype on the left, defined by its representation(concrete data type) on the right
  - The representation consists of variants of the datatype separated by +
  - Each variant is a constructor with zero or more named(and typed) arguments
- Examples
  - ImList<E> = Empty + Cons(first: E, rest: ImList)
  - Tree<E> = Empty + Node(e: E, left: Tree, right: Tree)

Functions over recursive datatype

- Declared in the specification for the type

- Can handle recursive and unbounded structures
- Convenient to describe operations over the datatype
- One common pattern of implementing an operation over a recursive datatype is by
  - Declaring the operation in the abstract datatype interface
  - Implementing the operation in each concrete variant
  - This is called interpreter pattern
- E.g. size: ImList -> int
  - size(Empty) = 0
  - Size(Con(first: E, rest: ImList)) = 1 + size(rest)

Null vs. Empty

- Sentinel object:
  - using an object rather than a null reference to signal the base case of a data structure
  - You can call methods on it

Declared type vs. actual type

- Declared type:
  - At compile time, every variable has a declared type, stated in its declaration
  - Compiler uses it to deduce declared types for every expression in the program
- Actual type:
  - At runtime
  - E.g. new String() makes an object whose actual type is String

Lambdas and map-filter-reduce

2019年11月2日 9:05

Stream:

- Sequence of elements:
  - Stream provides an interface to a sequenced set of values or data
  - Do not store elements
- Source:
  - Collections, arrays, I/O
- Aggregate operations:
  - Filter, map, reduce, find, match, sorted…
- Pipelining:
  - Stream operations return a stream, operations are chained to form a larger pipeline
- Internal iteration

Lambda functions:

- They are higher order functions which take a function as an input
- Map: (E -> F) × Seq<E> -> Seq<F>
  - E.g. Arrays.asList(1,4,9,16).stream().map(Math::sqrt)
    - Here we call function sqrt as a value
  - Maps operates on a stream and produces a stream of the same type as input
    - The return stream has the same length as the input stream
- Filter: (E -> boolean) × Seq<E> -> Seq<E>
  - Elements that satisfy the boolean are kept, others are removed
  - Return a new list, does not modify its input
- Reduce: (F × E -> F) × Seq<E> × F -> F
  - Combines the elements of the sequence together
  - Takes list, initial value and return value (if the list is empty)
  - Can return a value of different type
    - E.g. input string, output sum of length of the strings

Benefits of abstracting out control

- Make code shorter and simpler
- Allow the programmer to focus on the heart of the computation
- Safe concurrency

First Class functions:

- Functions as values
- In Java, use function interface
  - Because the only first-class values are primitive values and object references

Grammars, lexers and parsers

2019年11月2日 9:31

Grammar:

- Defines a set of sentence where each sentence is a sequence of symbols called tokens (terminals)
- It is a compact representation
- A grammar is a set of productions where each production defines a nonterminal
  - Nonterminal ::= expression of tokens, non-terminals and operators
  - Three most important operators
    - Sequence x ::=y z (an x is a y followed by a z, i.e. y-z pair)
    - Repetition x ::= y* (an x is zero or more y)
    - Choice x ::= y | z (an x is a y or a z)
  - Other operators
    - Option x ::= y? (an x is a y or is the empty sentence)
    - +1 repetition x ::= y+ (an x is one or more y)
      - Equivalent to x ::= y y*
    - Character classes
      - x ::= [abc] is equivalent to x ::= 'a' | 'b' | 'c'
  - Grouping using parentheses
    - x ::= (y z | a b)* (an x is zero ore more y-z or a-b pairs)
- E.g. URLs
  - url ::= 'http://' [a-z] + '.' [a-z] + '/'
  - Nonterminal version:
    - url ::= 'http://' hostname (':' port)? '/' (production1)
    - hostname ::= word '.' hostname | word '.' word (production 2)
    - word ::= [a-z]+ (production3)
    - port ::= [0-9]+ (production4)

Regular expressions:

- Special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only tokens and operators on the right-hand side
- E.g. url ::= 'http://' [a-z] + '.' [a-z] + (':' [0-9]+)? '/' is a regular expression

Context free grammars

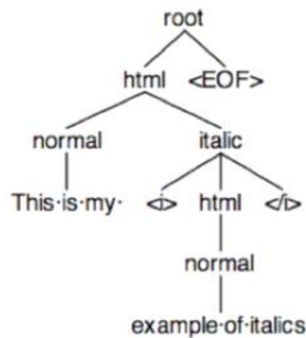
- A language that can be expressed with our system of grammars is called context-free
- Not all context-free languages are also regular
- Generally, any language with nested structure is context-free but not regular

Terminals and non terminals

- a terminal symbol is one that cannot be broken down further, e.g. a literal character or digit (but not necessarily as it depends on the grammar),
- a non-terminal symbol is a symbol that can be reduced further by the production rules

Lexing and parsing

- Lexical analysis (lexing): transforms the stream of characters into a stream of higher-level symbols (tokens), like words, or HTML tags.
  - Called tokenization
  - Low-level symbols -> higher-level symbols
  - An enum class is a useful way to define tokens
  - A lexer is like an iterator

- Parsing: takes the stream of tokens and interprets them
    - Checking the relationships between the tokens
    - Typically produces a parse tree
        - Shows how grammar productions can be expanded into a sentence that matches the token sequence
        - Root is the starting nonterminal of the grammar
        - Each node expands into one production of the grammar
        - 

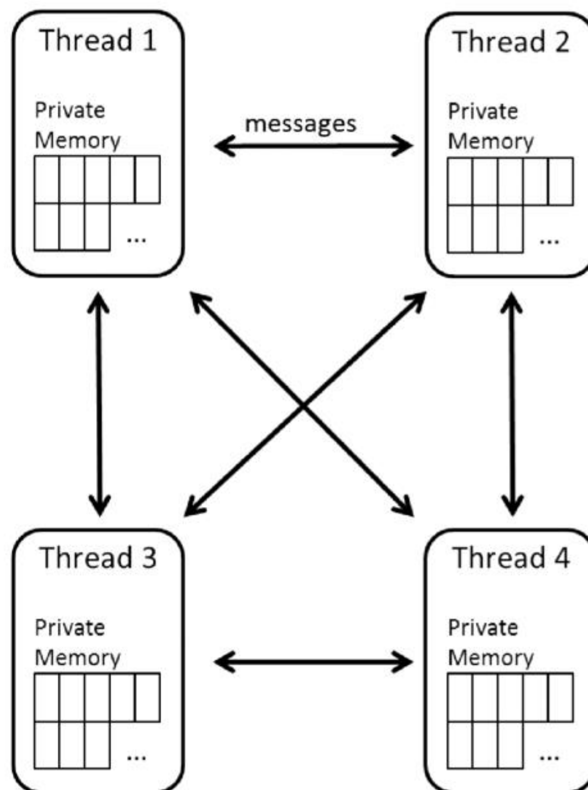
          Calling it produces a parse tree:

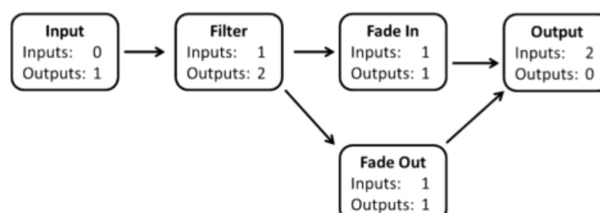
          ```
          ParseTree tree = parser.root();
          ```

            - Italic nodes will create italic objects
            - Normal nodes will create text objects
            - Html nodes will create sequence objects
    - Parser generator
        - Takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar
            - Source code = a lexer + a parser
        - E.g. ANTLR
            - Token definitions (capitalized identifiers)
                - START_ITALIC: '<i>' ; END_ITALIC: '</i>';
                - Literal strings are quoted with single quotes
                - ~[<>] matches any character except < and >
            - Grammar rules
                - Nonterminal has to be lowercase:
                    - Root, html, normal, italic
                - Root is the entry point of the grammar
                    - The nonterminal that the whole input needs to match
                    - Don't have to call it root
                - EOF: a special token that means the end of the input
            - Do not edit the files generated by ANTLR
            - Regenerate the files whenever you edit the grammar file
- Outputs of the lexer are consumed by the parser
    - Separation of concerns

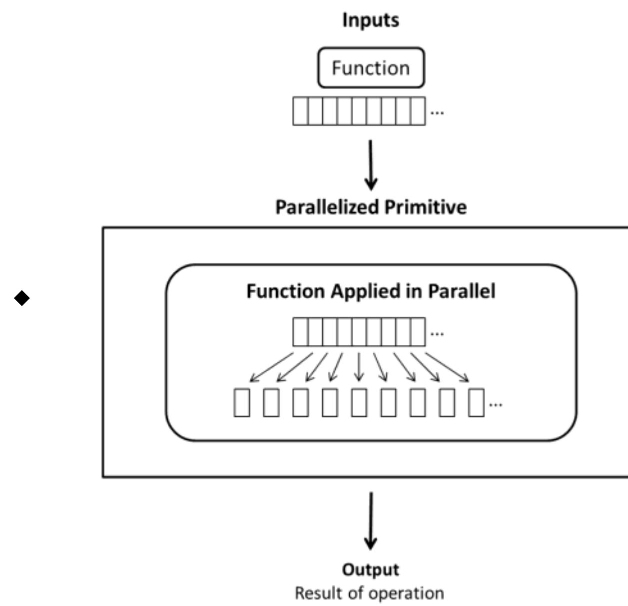
Parallel and concurrent programing

2019年11月30日 11:20

- Parallel programming: using additional computational resources to produce an answer faster
- Concurrent programming: correctly and efficiently controlling access by multiple threads to shared resources
- Programming model: explicit threads + shared memory
  - Thread: one thread can create one or more threads
    - Each thread has its own call stack and program counter, but all threads share one collection of static fields and objects
    - Thread t = new Thread(Something implement Runable)
    - Cannot predict the order of thread execution
      - Because they run at the same time
      - But their created time are always in the same order
  - Shared memory: Two or more threads can write and read fields of the same object
    - Alternatives:
      - Message-passing:
        - Explicit threads, do not share objects
        - To communicate between threads, message sends a copy of some data to its recipient
        - 

      - Dataflow:
        - Using graph and nodes
        - A node performs computation using inputs that arrive on its incoming edges, and sent to other nodes along their outgoing edges
        - 

      - Data parallelism

- Does not have explicit threads or nodes
- It has primitives for parallelism that involve applying the same operation to different pieces of data at the same time
- 

- Multithreaded programs are often nondeterministic

Network programming

2019年11月30日 11:58

- Network communication is inherently concurrent
- Client/server design pattern
  - Client initiates the communication by connecting to a server
  - sends requests to the server
  - disconnects after the server send the replies back
  - Server may handle connections from many clients concurrently
  - Clients may also connect to multiple servers at the same time
- IP address
  - Identify a network interface
  - IPv4 are 32-bit numbers written in four 8-bit parts
- Hostnames
  - Can be translated into IP addresses
  - A single hostname can map to different IP addresses at different times
  - Multiple hostnames can map to the same IP address
- Port numbers
  - Network interfaces have multiple ports identified by a 16-bit number from 1 to 65535
  - A server process binds to a particular port (listening)
    - Clients have to know which port number the server is listening on
    - E.g.
      - port 22 is for SSH
      - Port 25 is for email server
      - Port 80 is for web server
- Network sockets
  - A socket represents one end of the connection between client and server
    - Listening socket is used by a server process to wait for connections from remote clients
    - Connected socket can send and receive messages to and from the process on the other end of the connection
      - Identified by: local and remote IP address, port number
  - With sockets output of one process is the input of another process
- Buffers
  - An array in memory that holds the data until you read it
  - Used to send data in chunks
- Streams
  - Data going into or coming our of a socket is a stream of bytes
  - InputStream objects represent sources of data flowing into the program
    - FileInputStream: reading from a file on disk
    - System.in: take user input, input from a network socket
  - OutputStream objects represent data sinks, places we can write data to
    - FileOutputStream: save to file
    - System.out/PrintStream: output to a network socket
- Blocking:
  - A thread waits until an event occurs
  - If a method is a blocking method, then a call to that method can block, waiting until some event occurs before it returns to the caller
  - Bug: deadlock
    - Modules are waiting for each other to do something, so none of them can make any progress
  - ServerSocket.accpet() and in.readLine() are blocing
    - So need threads to handle multiple clients
- Wire protocols

- Protocol: set of messages that can be exchanged by two communicating parties
- Wire protocol: a set of messages represented as byte sequences
    - When designing:
        - Keep the number of different messages small, better to have a few commands and responses that can be combined
        - Each message should have a well-defined purpose and coherent behavior
        - The set of messages must be adequate for clients to make the requests they need to make and for servers to deliver the results

Fork-join Parallelism

2019年11月30日 13:09

- Join() method block the main thread until the helper thread is done
  - Need to catch InterruptedExcpetion to use Join method
- Fork-join parallelism
  - Using two for-loops, where the first one creates helper threads, and the second one waits for them all to terminate to encode the idea of a forAll-loop.
- Why not using one thread per processor
  - Different computers have different numbers of processors
  - Processors available to part of the code can change
  - Cannot predictably divide the work into approximately equal pieces
- Solution: divide the work into smaller pieces
  - Divide and conquer parallelism
- Reductions and maps
  - Reductions: take a collection of data items and reduce the information down to a single result
  - Maps: perform an operation on each input element independently, produces an array of outputs of the same length
- Work and Span
  - Define T_P to be the time a program/algorithm takes to run, if there are P processors available during its execution
  - T_1 is called the work, it is how long it takes to run on one processor,
    - Total of all the running time of all the pieces of the algorithm
    - The total does not depend on how the work is scheduled
  - T_∞ is called the span/critical path length/ computational depth
    - Not necessarily constant time
- Directed acyclic graph(DAG)
  - Nodes are pieces of work the program performs, each node will be a constant
    - Work is the number of nodes in the DAG
  - Edge represent that the source node must complete before the target node begins
    - Computational dependency along the edge
- A parallel reduction is basically described by two balanced binary trees whose size is proportional to the input data size
- Speedup and parallelism
  - Speedup = T_1/T_P is the speedup on P processors
    - Perfect speedup (speed up of 4 or more) is rare
    - Perfect linear speedup: doubling P cuts the running time in half
- Amdahl's law
  - $T_1 = S + (1 - S) = 1 \text{ and } T_P = S + (1 - S)/P$
  - $\textbf{Amdahl's Law: } T_1/T_P = 1/(S + (1 - S)/P).$
  - $T_1/T_\infty = 1/S$
  - As the number of processors grows, span is more important than work
    - Large amounts of parallelism can change algorithmic choices
  - Use the parallelism to solve new or bigger problems rather than solving the same problem faster

Shared Memory

2019年11月30日 14:58

One thread doing everything

- Parallelism
- Responsiveness
- Processor utilization
- Failure/performance isolation

Mutual exclusion:

- Allow only one thread to access any particular data at a time

Synchronization

- Atomically:
    - The act of checking there is no sign hanging and then hanging a sign has to be done all-at-once
    - Critical section: while the sign is hanging
- Synchronization primitives:
    - Atomic check-for-no-sign-and-hang-a-sign
- Locks: (mutual-exclusion lock/mutex)
    - Synchronized (expression) {statement}
        - Expression is evaluated, must produce a reference to an object
            - Every object is a lock that any thread can acquire or release
        - It acquires the lock
        - Then the statements are executed
        - When control leaves the statements, the lock is released
    - Used to avoid data races
- Data races
    - Multiple threads sharing the same mutable variable without coordinating what they are doing
    - Race condition is a mistake in the program such that whether the program behaves correctly or not depends on the order that the threads execute
        - Bad interleaving (higher-level race)
            - It depends entirely on what you are trying to do
        - Data race: simultaneous access error although nobody uses that term
            - One thread read an object field at the same moment that another thread writes the same field
            - One thread write an object field at the same moment that another thread also writes the same field
    - Wrong even thought they look right
        - We cannot reason the program in the presence of data races
        - Compiler optimization change the possible interleaving and ordering of codes
    - Avoiding data races
        - Use synchronization
        - Declare fields to be volatile so the code depends on the exact ordering of reads and writes to fields

Thread and concurrent safety

2019年12月1日 9:26

Four ways to make variable access safe in shared-memory concurrency

- Confinement: don't share the variable between threads
  - Need to check if it is an object reference (mutable/immutable)
- Immutability: make the shared data immutable
  - No mutator methods
  - All fields are private and final
  - No mutation in the representation
  - No representation exposure
- Thread-safe data types
  - A data type or static method is thread-safe if it behaves correctly when used from multiple threads, regardless of how those threads are executed and without additional coordination
  - E.g. wrapper/concurrent collections
- Synchronization

Thread safety arguments

- Immutability
- Avoid rep exposure

Safe concurrent programming

- Every memory location in the program should meet at least one of the following three criteria
  - Thread-local: only one thread ever accesses it
  - Immutable: it is only read, never written
  - Synchronized: locks are used to ensure there are no race conditions
    - Avoid data races
    - Use consistent locking
    - Start with coarse-grained locking and move to finer-grained locking only if contention is hurting performance
      - Coarse-grained: using fewer locks to guard more objects(entire array)
        - Introduces the possibility of deadlock
          - Forget to acquire a lock or acquire the wrong lock
        - Leads to threads waiting for other threads to release locks unnecessarily
      - Fine-grained: using more locks, each of which guards fewer memory locations
      - Locking granularity is a continuum, one direction is coarser, other is finer
      - Contention: the situation where threads are blocked waiting for each other
    - Make critical sections large enough for correctness but no larger. Do not perform I/O or expensive computations within critical sections
      - Smaller critical sections lead to less contention
    - Think in terms of what operations need to be atomic. Determine a locking strategy after you know what the critical sections are