Information

In this document we will cover questions about the study of the advanced programming cours of Computer Engineering at IST.

Introduction

What is a computational system?: A computational system, given a domain, it's a system that reasons and act. It's represented using internal structures and processes.

A is program a computational system? No, a program is just a description of the system. But a running program it's actually a computational system.

What is a computational meta-system?: It's a system that can **reason** about other systems. Some examples are like debugger and profiler. Given a program, it can reason about it. The other system is called object system.

Definition of Reflection: It's the ability of a system to reason about **itself**.

Definition of reflective system: Given a system, it's a system that has itself as Object-system. Can manipulate his structure and behavior at runtime.

Definition of introspection: Given a program, it's the ability to reason about the structure and behavior of the program itself.

Definition of intercession: Given a program, it's the ability to change the structure and behavior of the program itself.

Definition of reification: The creation of an entity that represent an entity of the object system, inside the meta-system, where a meta-system is a system that can reason about other systems.

Example: Imagine having a program that can reason about itself, that's a reflective system. If we create an entity that represents a class of the program, that's reification.

```
class A {
    int x;
    void m() {
        System.out.println("Hello");
    }
}

// Reification
Class c = A.class;
```

N.B: Reification is a pre-condition for reflection.

Differences between structural and behavioral reification:

- Structural: Ability of a program to reify its structure, with reify we mean to create an entity that represents an entity of the object system.
- Behavioral: Ability of a program to reify its execution.

Note that behavioral reification is more complex than structural reification.

What are the problems of Reification:

- Compilation harder
- Execution slower
- Hard code

2 Reflection in Java

What are reference types: Reference types are only types that have a reference to an object. They are not primitive types.

- int -> Primitive type
- BigInteger -> Reference type

Motivation: Performances.

What are reified types: Each reference and primitive there is a unique instance of the class java.lang.Class that represents the type.

What are some important method of the Class class:

- getName(): Returns the name of the class
- getSuperclass(): Returns the superclass of the class
- isArray(): Returns true if the class is an array
- getMethods(): Returns an array of Method objects that represent the public methods of the class
- getDeclaredMethods(): Returns an array of Method objects that represent the methods of the class

What is multiple dispatch: It's a feature that some programming languages have. It's the ability to call a method based on the type of multiple arguments of the method. At runtime, the method is chosen based on the type of the arguments.

Java doesn't have multiple dispatch.

```
public class Vehicle {
}

public class Car extends Vehicle {
}

public class Bike extends Vehicle {
}

public void print_something(Vehicle v) {
    System.out.println("Vehicle");
}

public void print_something(Car c) {
    System.out.println("Car");
}

public void print_something(Bike b) {
    System.out.println("Bike");
}

List<Vehicle> vehicles = new ArrayList<>();
vehicles.add(new Car());
vehicles.add(new Bike());

for (Vehicle v : vehicles) {
    print_something(v);
}
```

In this case, java should choose the method to call based on the type of the object, but it doesn't have multiple dispatch, so it will always call the method that takes a Vehicle as an argument.

What does java uses to choose the method to call?: It's dynamic dispatch. Good for polymorphism but not for multiple dispatch.

What can be a solution for this problem?: TypeCast inside the method print. CHeck check the instance of the object and do a cast to the class.

What is double dispatch in Java?: It's a pattern in which you modify the object that calls the other object, and you pass the reference to the class. Then on the caleld object you implement a method for each class that can be passed as an argument.

This of course make easy adding a new called object, but adding a new calling object is hard because you have to modify all the called objects.

What is defined method call mechanism?: It's a mechanism in which you choose the righe method to call based on the type of the object that calls the method. It uses the `invoke` method to dynamically call the method, going up in

the hierarchy of the class until it finds the right method to call or not. Problems:

- only good for 1 parameter
- only good for public methods
- only with double dispatch
- class hierarchy but we have interfaes
- boxing and unboxing

What is the boxing and unboxing problem: When you pass a primitive type to a method that takes an object, the primitive type is boxed into an wrapper object. This is a problem because you don't know the behavior of the method that takes the object.

Reflection in LISP

What is LISP: It's a programming language that has a lot of features. It's a functional programming language.

What are dialects in LISP?: There are a lot of dialects in LISP, like Scheme, Common LISP, Clojure. They are all based on the LISP language and they have some differences.

Define function in LISP, Common LISP, Scheme and Emacs LISP:

- LISP: (defun name (args) body)
- Common LISP: (defun name (args) body)
- Scheme: (define (name args) body)
- Emacs LISP: (defun name (args) body)
- Clojure: (defn name [args] body)

What happens in LISP when you define a function?: You are binding that to a symbol. You are creating a mapping between the symbol and the function.

What is the difference between a function and a lambda function in LISP?: A function is a named function, a lambda function is an anonymous function.

Explain CAR, CDR: They are primitive functions in LISP. They are used to access the first element of a list and the rest of the list.

What is the difference between CAR and FIRST: They are the same, but CAR is used in LISP and FIRST is used in Scheme. Same for CDR and REST.

CAR and SETCAR: CAR is used to access the first element of a list, SETCAR is used to set the first element of a list.

What is tracing?: Tracing a function means printing at each call of the function the arguments and the result of the function. It's a `behavioral introspection`.

Code for tracing function:

```
(defun traced-lambda (name lambda-form)
        (cons 'lambda
                (cons (cadr lambda-form)
                        (cons (list 'princ
                                (cons 'list
                                        (cons (list 'quote name)
                                                (cadr lambda-form))))
                                (cons '(princ '->)
                                        (cddr lambda-form))))))
```

What is metaprogramming?: Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyse or transform other programs, and even modify itself while running.

Metaproramming simplification using `, , and ,@:

In LISP we can use those symbols to simplify our metaprogramming adventure. Like:

- `: The quote operator makes use able to not evaluate the expression that is passed

```
`(+ 1 2) -> (+ 1 2)
```

- ,: The comma operator is used with subexpressions inside a quoted expression to evaluate them

```
`(1 ,(+ 1 2)) -> (1 3)
```

- ,@: The comma-at operator is used to split so expand the result of a list

```
`(1 ,@(list 2 3 4)) -> (1 2 3 4)
```

How does these work in Julia?:

- :(expr) : Quote the expression and don't evaluate it
- $(subexpr): Evaluate the subexpression
- $(subexpr...): Expand the subexpression

What is memoization: Memoization is a technique used to store values of a function in a cache. It's good when we have repetitive calls to the same function with the same arguments, so we can store the result of the function in a cache and return it when the function is called again with the same arguments to save time and resources.

Fibonacci memoization:

```
(defun memoize (lambda-form)
                (setcar (cddr lambda-form)
                        `(let ((result ,(caddr lambda-form)))
                                (setcar (cddr ',lambda-form)
                                        `(if (eql ,',(caadr lambda-form)
                                                  ',,(caadr lambda-form))
                                        ',result
                                        ,(caddr ',lambda-form)))
                        result)))
```

Good and bad stuff:

- GOOD: Code modified itself
- BAD: Code is hard to read and debug

Javassist

How does java work?: The java code is converted into bytecode, then the bytecode is executed by the JVM. It's important to note that **after you create the bytecode it can't be changed**

What is Javassist: Javassist is a library that allows you to manipulate the bytecode of a class at runtime. It's a library that allows you to do metaprogramming in Java.

What are the 4 main steps of Javassist:

1. Reification: Create a representation of the class
2. Modification: do instrospection and intercession to alter the class definition (think about fibonacci memoized)
3. Translation: Translate the representation into bytecode
4. Reflection: Load the byteode of the new class.

What are the 2 main component that we used in javassist for meoization:
A result hashtable and a method that checks if the result is already in the hashtable.
The method is called in the same name of the method that we want to memoize, but with a postfix of $original. We are sure we don't have problems because we can't have a method with that name in Java, since$ is not allowed in method names.

Steps to use javassist, code speaking: The steps are basically asking for a class and method to be modified. Then create a classPool, get the class, get the method. Call a method that modifies the method, then get the bytecode of the new class and load it.

What were the problem of our first implementation of the memoization using javassist?: The initial problem was that we could only handle 1 method per call. This was like this since we were asking for the specific class and specific method.

Note: we even had recompilation to be done manually. To solve this, actually we used something called **load time intercession**, using the method toClass() of the Javassist class.

How to do changes to more methods automatically?: It's easier to do using Annotations.

What annotations are?: Annotations are a form of metadata that provides data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate. They can survive annotation process. Can be processed at:

- compilation time
- load time
- runtime

How annotations solved our problem in Javassist: Since with annotations we can decide which class and methods to modify at runtime, *annotating* them makes automatic the retrive using the getDeclaredMethods() method and checking which annotations are present. If there is actually one annotation, we can proceed with the memoization.

What is a problem of this approach if we have more methods?: The problem is that, for memoization example, we have a unique hashtable for all the methods. This is a problem because we could have different methods with the same arguments, and we would have a problem with the cache. To solve this, is enough to give a specific name for the hashtable, using the name of the method as a key and adding it to the hashtable name.

What is javassist translator and why is usefull?: The javassist translator is a class that allows you to modify the bytecode of a class. It's useful because you can modify the bytecode of a class at runtime, and then load the new class. It has 2 main methods, start and onLoad.

What are the main methods for the translator:

- start: Start the classloader, useless
- onLoad: Called when a class is loaded

Undoable programs

What do Java Reflection API make possible?: The Java Reflection API makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time.

Explain what we used javassist for the undoable program:
We used javassist to select the methods inside our program and then inject inside them the undoable feature. This was done using the translator, that for each method, it would inspect and instrument it to save the name of the class, the name of the field and the value of it.

Does javassist create classes at runtime? It's suggested? Why?
Yes, javassist can create classes at runtime. It's dangerous because the javassist compiler is very fragile and may not support all the features of the Java compiler. It's better to use the javassist to modify existing classes.

Metacircular evaluators

What is an evaluator: An evaluator is a program that given an expression it computer its value.

What is a metacircular evaluator?: A metacircular evaluator is an interpreter for a language that is written in the same language. It's a way to implement an interpreter for a language using the language itself. For example our project was in Julia and we wrote an evaluator using Julia.

What are macros and macros expansions: Macros are is defined like a function but it doesn't compute a value, but another expression that will be computed later. This is called macro expansion.

Names in LISP: Names are not first class values, but they **denote** first class values. This means that you can't pass a name as an argument to a function, but you can pass the value that the name denotes.

Does this code compiles in LISP? Why?

```
(let ((x (+ 1 2)
         (y (* x 3)))))
```

No, and the reason is that the `x` is not defined when we use it to compute `y`. In LISP everything is evaluated before the scope with let is declared. In Julia, it works.

How does search for value insdie the environment works in our evaluator in lisp?: It's based on a recursive call searching for the binding between the value inside the environment. We can think of the environment as a list of bindings, where each binding is a pair of a name and a value.
(name1 value1) (name2 value2) (name3 value3) ...

What is REPL: REPL stands for Read-Eval-Print-Loop. It's a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result to the user.

How is LET handled in our evaluator: Every time we find a let, we create a new environment that is a list of bindings. We then evaluate the body of the let in the new environment.

What are initial bindings inside an environment? Why are they useful?: Initial bindings are the bindings that are present in the environment when it's created. They are useful because they allow us to have some predefined values in the environment, like:

- pi = 3.14
- e = 2.71
- golden_ratio = 1.61
- ...

What is variable shadowing: Variable shadowing is when a variable in an inner scope has the same name as a variable in an outer scope. The value of the variable in the inner scope shadows the value of the variable in the outer scope.

How are functions defined in our evaluator: We used flet to defined new functions. We then added the function to the environment as a binding between the name of the function and the function itself.

How are functions called?: When a function is called, we search for the function in the environment, we evaluate all the arguments of the function, we extend the environment with the bindings between the arguments and their values,

and then we evaluate the body of the function in the new environment.

What are primitive operations and why can they be useful: Primitive operations are operations that are built-in in the evaluator. They are useful because they allow us to have some predefined operations that are always available, like:

- +, -, *, /, ...
  Moreover, we can say that something is primitive and handle it in a different way than the other operations. We can even create functions and set them as primitive operations. This has to be handled having a way to specify that a functions is primitive and in the function call check if the function is primitive or not. If so, we can handle it in a different way using the `apply` function. The `apply` function is a function that takes a function and a list of arguments and applies the function to the arguments.

Why this code goes inside an infinite loop?

```
>> (flet ((+ (x y) (* x y))
    (* (x y) (+ x y)))
(+ (* 1 2) 3))
...infinite loop
```

The problem is calling the symbol `+` inside the definition of the function `*`. This is because the `+` is redefined inside the `flet` and then the `*` is redefined inside the `flet`. This creates a loop because the `+` is redefined inside the `*` and the `*` is redefined inside the `+`.

This is because we are doing an extension of the global environment.

What is def?: Def is a definition that inject the name inside the environment.

Same goes for fdef

Problems of evaluating definitions: Evaluating definition modifies the current environment and even the order of evaluation is a little different.

Why did we change the global environment from lists to frames?:
Because right now there is no separation between the scopes declared using let and flet. Using frames, we can store each scope in a different frame and then search for the value in the frames to get the correct value.
Of course this changes a lot the code, because now we have to change:

- how to retrieve values from the environment
- how to augment the environment
- the update of variables

Why is global keyword useful?:

Imagine this:

```
>> (def counter 0)
0
>> (fdef incr ()
(def counter (+ counter 1)))
    (function () (def counter (+ counter 1)))
>> (incr)
1
>> (incr)
1 ;;What?
```

This happens because the counter defined inside the functions starts always from 0. This is because the counter is defined inside the function and it's not the same counter that is defined outside the function. To solve this, we can use the global keyword that tells the evaluator to use the counter defined outside the function.

What does the set keyword does in our evaluator: The set keyword is used to set the value of a variable in the environment. It's used to update the value of a variable in the environment even if it's outside the scope.

What is ordering of evaluation: Ordering of evaluation is the order in which the arguments of a function are evaluated. In our evaluator, the arguments are evaluated from left to right. In our evaluato we want to define the behavior using begin and end.

The implementation is based on identifying the begin and eval the arguments in the begin and return the last one.

What does it means to consider implicit begin?: It means that we consider that the arguments of a function are evaluated in a begin block. This means that the arguments are evaluated from left to right and the value of the last argument is returned. So, inside each handling of let and flet, we have to consider that the arguments are evaluated in a begin block.

What are high order functions?: High order functions are functions that take other functions as arguments or return functions as results. An example of very useful high order functions are:

- map: that applies a function to each element of a list
- filter: that filters a list based on a function
- reduce: that reduces a list to a single value using a function

What it's the problem with using multiple namespaces?
In our lisp evaluator, the problem was that we were not able to specify when something we were passing as an argument to some high order functions was a function or not. We tried to implement a fix for the problem that something in function position was a function or not, but it had a problem regarding the fact that everything in function position was being evaluated as a function, but that's not the case. Then we used **funcall** but the code was very complex.

Why are anonymous functions good in our evaluator?: They make easier to simplify our evaluator.

-fdef = def + lambda

What is the problem of using lambda in this case of let and flet?: Names in the functions were shadowing the names of the parameters.

What are dynamic scopes?: Dynamic scopes are used in a way like this:

- def: create binding in the current environment
- function call: create a set of bindings for the parameters that extend the curr env
- end of the call: delete the extended env
- name: is searched in the current env

What is the downards funarg problem?: The downward funarg problem refers to a situation in programming languages with dynamic scope where a function is passed as an argument to another function, and the free variables within that function may be inadvertently affected by variables in the calling context, leading to unexpected behavior.

To fix the downward funarg problem, one solution is to ensure that functions are evaluated in the correct environment, rather than relying on dynamic scoping. This can be achieved by using lexical scoping, where variables are resolved based on their lexical context rather than the dynamic calling context.

What is the upward funarg problem?:
The upward funarg problem, as described in the provided context, pertains to situations where a function returns another function, and the returned function contains free variables. These free variables may reference variables from the environment where the function was defined. However, when the returned function is called, these variables may no longer be bound or may be bound to different values, leading to unexpected behavior or errors

To fix the upward funarg problem, one potential solution is to ensure that the free variables in the returned function are bound at the time of the function's creation. This can be achieved by capturing the environment at the time of function creation, rather than relying on dynamic scoping.

Differenes between function definition and function creation:

- Function definition: when we are actually defining the function and binding it to a name. The function is created and stored in the environment.
- Function creation: when we are creating a function and returning it.

How to solve the problem of not finding functions inside functions:
You have to create the environment and then the function, and then you change the scope to the new one.

What's quoting used for?: Quoting an expressio in LISP is used to avoid the evaluation of the expression:
1+2 = 3
:(+ 1 2) = (+ 1 2)

What is a macro: A macro is a way to generate a new expression starting from an already existing one. It's a very powerful tool that can be used to generate code. The main expression is not evaluated and only the new generated expression is evaluated.

Difference between macro and function call:

- Function call: evaluates the arguments and computes the result

- Macro: generates a new expression that will be evaluated later

Why you should never repeat a parameter in the macro expansion:
Because the parameter will be evaluated twice, and this can lead to unexpected behavior.

How to solve name collision inside macro expansion:
You can use the gensym function that generates a new symbol that is unique.

What is a gensym function: It's a function that generates a new symbol that is unique, so it can be used to avoid name collision. In MACROS what happens using gensym the symbol is generated and then the symbol is replaced with the value.

What is hygiene: Hygiene is a property of a macro that ensures that the macro doesn't interfere with the variables in the code. It's very important to have hygiene in a macro because it can lead to unexpected behavior.

What is a quasiquote and why we use it: A quasiquote is a way to quote an expression but allowing the evaluation of some parts of the expression. It's very useful when we have to generate code that is similar to the code we are writing.

Why we removed the macro eval in the evaluator: Because we see macros as tagged function and the check is done during the evaluation of the function.

What does it mean and why do we reify the environment:
Reification is the process of creating an entity that represents an entity of the object system, inside the meta-system. In our case, we reify the environment to have a way to represent the environment in which the function is evaluated.

What is a continuation?:
A continuation is a way to represent the future of the copmutation. For example, if we have a function that is calling another function, the continuation is the future of the computation of the first function. It saves the state of the computation and allows us to continue the computation from that point.

Difference between direct style and continous passing style:

- Direct style: the function returns the result
- Continous passing style: the function returns the continuation

What is nondeterminism: Nondeterminism is a property of a program that can have more than one result. It's very useful when we have to generate all the possible results of a computation. In our evaluator we used the functions **amb** and **fails** to get the value and to get a new one.

Does CLOS support multiple dispatch?:
Yes, CLOS supports multiple dispatch. It's a very powerful feature that allows us to choose the method to call based on the runtime type of more than one of the arguments.

What is a generic function: A generic function is a function that can have more than one method. The method is chosen based on the runtime type of the arguments.

What is method combination: Method combination is a way to combine the results of the methods. We can have different ways to combine the results of the methods, like **and**, **or**, **append** and so on.

What's the generic function correct method finding?:
The generic function correct method finding is the process of finding the correct method to call based on the runtime type of the arguments. It's done in this way:

- select the applicable methods
- order the methods by precedence order
- combines the methods using the method combination and produces effective method

What's the class precedence list? Where is used?:
It's used in class inheritance. It's a list that contains the classes in the order in which they should be checked.