Refinable Functions

Object-oriented Function Refinement for Pragmatic Language-based Modularity

Hiun Kim
Sejong University hiun@divtag.sejong.edu
NAVER Corporation hiun.kim@navercorp.com

1. Motivation

To provide universal way to implement language-based code modularity

Modularity: **disciplined way** to manage **source code**
Benefit: **easy/maintainable/evolutionary** development
Criteria: code locality, reusability, (un)pluggability

Major Impl. Mechanism : **Language Extensions**
ex) create new JVM-based language
AspectJ, DeltaJ, software product line languages

2. Modularity Implementation Issues

Difficulties on making language extensions for modern script languages

IoT App. Machine Learning App.
Extending **Python languages** requires to **modify Python interpreter in C/C++**

Web Applications
Extending **JavaScript languages** is difficult, since its runtime environment is depending on client

Domain Specific Languages
Many **DSL does not provides standard API** to modify compiler according to **feasibility issues**

3. Refinable Functions Goal

Object-oriented Programming as **modularity mechanism for function** by framming ***Function as a Classes***

Function as a Classes

RefinableFunction
behaviorStore : (Function|*Self*)[]
constructor(—*Self*[]): this *Self*.add(Function|*Self*): this *Self*.before(Function|*Self*): this *Self*.after(Function|*Self*): this *Self*.update(Function|*Self*): this *Self*.map(Function(Any):Function): this *Self*.delete(): this assign(Object): this exec(Any): this catch(Function): this new(): this defineMethod(String, Function): this

Function Inheritance for Commonality Reuse

$$rf\_root := new\ rf(sub1,\ sub2,\ sub3)$$
$$rf\_child := rf\_root.new()$$
$$rf\_child\_child := rf\_child.new()$$
$$rf\_child\_child.exec(args).catch(handler)$$

Function Inheritance for Variability Localization

$$rf\_child.sub2.update(new\_sub2)$$
$$rf\_child.sub3.after(other\_rf)$$
$$rf\_child\_child.sub3.delete()$$
$$rf\_child\_child.assign(traits\_rf)$$

4. Refinable Functions as Aspects

Refinable Functions as an Advice Function

Commonalities of **Observer Design Pattern**

```
var ObserverAspect = new Self.utils.aspect({
  subjectObservers: new WeakMap(),
  getObservers: (subject) =>
    {searchWeakMap(subject, this.subjectObservers)},

   subjectChange: new Self().add(inputCheck)
   .add(function (subject, observer) {
     return this.updateObserver(subject, observer)
    }).export.before()
});
```

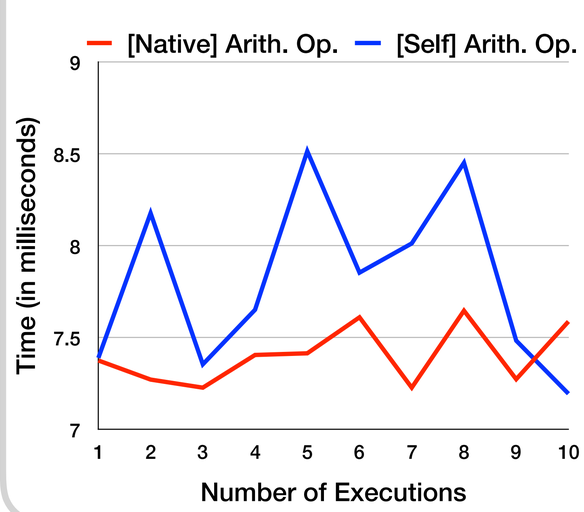
Commonalities of **Color Observer**

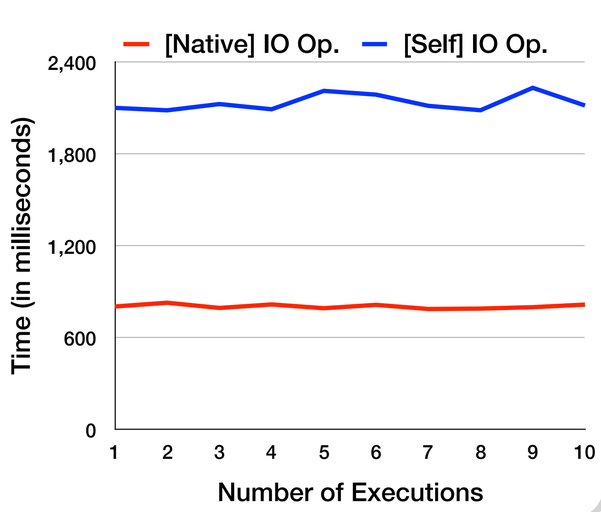
```
var ColorObserver = new Self.utils.aspect({
  subjectChange: function subjectChange(subjectChange) {
    return { setColor: subjectChange,
             setLine: subjectChange } },
  updateObserver: (subject) =>
   { console.log('color updated at ' + subject.location); }
});
```

Aspect Inheritance & Class Weaving

```
ColorObserver.extends(ObserverAspect);
ColorObserver.compose(PaintAppClass);
```

5. Performance Evaluation





6. Conclusion

- Pragmatic approach to implement modularity in widly spreaded OOP technique.
- Adds reflectivity to traditional function composition techniques like currying.
- Practical Refinable Functions implementation in JavaScript: https://hiun.org/self