Implementing Language-based Modularity using Object-oriented Refinement of Function

Hiun Kim

Department of Computer Science and Engineering
College of Electronics and Information Engineering
Sejong University
Seoul, South Korea
hiun@divtag.sejong.edu

Abstract

Due to the rising use of script languages, adapting advanced modularity techniques is important for achieving scalable engineering using script languages. The language-based technique like Aspect-oriented Programming is important since it implements modularity directly at the level of source code using language primitives. However, many existing modularity implementations are based on compiler extensions which are hard to applicable for script languages because it operates on identical compile and execution environment that is depended on clients. To implementing modularity in arbitrary runtimes, This paper introduces language feature *Refinable Functions*, class-based function which gradually reuses commonalities and localizes variabilities of software to implement behavioral and structural abstraction of modularity by inheritance and refinement of function, as a result refinable functions enables to refine/inherit/subtype function similarly how we refine/inherit/subtype of classes. refinable functions are purely based on OO feature which does not require runtime constraint but succeeds primitive syntactic support from OOP. We make applications of Refinable Functions to shows 1)efficiency of using refinable functions as a modularity technique for code reuse and localization, 2)a formal model of refinable function with JavaScript implementation and discussions on language and performance issues. This paper shows the application of OO concept as an optimal medium of a pragmatic language-based modularity method.

Keywords - Modularity, Object-oriented Programming, Software Product Line

Contents

List of Figures

List of Tables

1 Introduction

Modularity plays a key role in modern software engineering due to its importance on code reuse and localization which is crucial for handling the scale of modern software in many aspects. One of the notable work of modularity is Aspect-oriented Programming[1], which is language-based modularity technique by localizing scattered and tangled code-cross cutting concerns in software widely adapted in practical usages[2, 3]. Also, Feature-oriented Programming an software construction technique in the notion of a feature in a composable modular unit of code[4]. The goal of the implementation of both techniques is varied but properly localize scattered concerns as well as provides a disciplined way of composing software shares same motivation and stand on top of the study of modularity.

1.1 Traditional Modularity Implementations

The language approach to modularity implementations, however the target language of research of major implementation are mostly converged to Java by its industry-strength usage and the well-defined method for extending compiler using language extension provided by Java Virtual Machine.[5, 6, 7].

The language extension enables the addition of new language primitives such as custom operator with its operational mechanism however this approach requires to compile the application in the designated compiler. In most compiler-based languages, this compiler dependency problem is not a significant issue because the compiler generates a binary instruction or intermediate language like Java Byte Code to making application suitable for general runtime environment, the mapping of compilation and execution environments are low. The concrete example includes, AspectJ[5] which is widely used an aspect-oriented extension for Java works based on its bytecode weaving. Also, Variability management techniques for feature-oriented programming like Jak[6], Delta-oriented Programming[7] is based on Java language extension feature.

However, the great demands of interpreter-based language, the compiler-based language modularity approach cannot be applicable because interpreter-based language enforces development environment and operating environment to be the same this especially problematic for interpreter language JavaScript, the only languages that are usable in client-side web programming and large-scale application developments on enterprise applications and intelligent systems.

1.2 Implementation Criteria for Script Languages

In order to support modularity for script language, implementing modularity should support variance of runtime in expressive language primitives in the feasible language used practically in the industry.

Runtime Variance Supports. The runtime of interpreter languages are varied, since it uses the identical system for build and executes the program. Also, the runtime system is controllable in an environment like web client. As productivity comes to main issues for the software engineering, building enterprise application using interpreter languages are common using frameworks[8, 9, 10] from JavaScript[11], Python[12] or Ruby[13]. Also, recently machine learning applications built on top of Python[14, 15] or for JavaScript[16, 17] also adds the size of the application that script language could build implementing modularity technique without modifying or constrain runtime environment is highly regarded to improve practical relevences[4]. To support runtime variance, refinable functions takes modularity implementation using commodity technique, object-oriented programming to improve compiler compatibility.

Language Primitives Supports. The traditional language-based modularity is implementable with design patterns, even parameters[4], however, amplyfing expressiveness within the context of host language have limitations, for example, a library-based aspect-oriented mechanism implementation has many structuring cost even for the small composition of cross-cutting concerns[library citations?]. Another significant issue of the classic mechanism is refinability. Once, functions or patterns is constructed, it works as a black box, thus the concept of refinement cannot be applicable for reuse. To support modularity operation with language primitives, refinable functions takes object-oriented programming and its syntax to ensure familiarity and high expressiveness to model modularization and make refinements.

Practical Industries Supports. Many languages support runtime independent, language primitives syntax for implementing modularity. Notably Swift supports Method Swizzling for Function composition for cross-cutting concerns or Haskell support function composition pattern like Pointfree that can be applied to implement language-based modularity but many of these languages are use limitedly due to domain integration, compiler support, ecosystems leak and performance issues including languages for academic usages, the practicality of using that languages are limited,

To resolve this runtime dependency problem for modularity while keeping the expressiveness and reflectivity that programming model as previous research.

Apparently, the way to solve this problem is, implement language-based modularity in common and classical language feature, such as parameter and subroutine or design pattern. However, these classical techniques are good localization bit not it hurts comprehension by the overuse imperative and low-level operation to provides same-level of expressiveness as previous techniques offers.

1.3 Bringing Object-orientation for Function

This paper attempt to add reflectivity and refinement for a function to implementing code modularity by localization and reuse. The data structure of the object is incrementally reused and refined by inheritance, but the function is simply overridden for an update in the classical refinement in object-oriented software. However, reuse and refine the *body* of the function at the granularity of function(not an object) is important for implementing aspect and feature-oriented software development. This because major variabilities are involved in data as well as the behavior of a function, but in contrast, the data type can be extensible with the refinement, but there is no formal model to extends function with refinement.

This paper attempts to frame *function as a class* to enable modular, reflective composition and refinement of function to implementing compiler and runtime independent language-based modularity. The modularity of function is achieved by refine/inherit/subtype function as just we refine/inherit/subtype classes, and the resulting class-based function, *Refinable Function* can be composed to normal as class similarity as aspect weaving to achieve aspect-oriented and feature-oriented programming.

On criteria for evaluating the efficiency of Refinable Functions, we compare its concept and functionality with several examples that are complementary to cross-cutting modularization of aspect-oriented programming to feature composition and feature interaction problem of feature-oriented software product line method[4]. We also show that creational and structural design pattern for performing construction and refinement of refinable function programmatically.

The contribution of the Refinable function is not only provided novel implementation method of modularity using well-established object-oriented technology but also shows radical approach to using a function as an executable, ordered object that takes refinements. The result of this work allows applying a various object-oriented technique to function such as

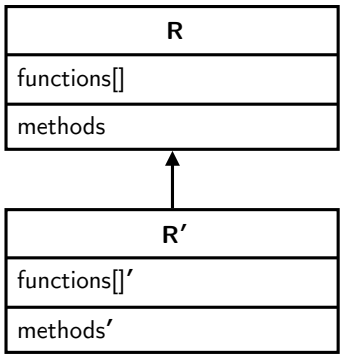


Figure 1: Function Classes and its Inheritance

design patterns, traits, and mixin compositions, which enables many possibilities not only on the context of modularity, metaprogramming but language designs.

As a result of conceptual and concrete code-level analysis, we found that Refinable functions can leverage OOP to implement language-based modularity without runtime dependency. As a result of conceptual and concrete code-level analysis, we found that Refinable Functions can implement advanced language-based modularity in commodity object-oriented feature that provided by many different programming languages while keeping the expressiveness and reflectivity that programming model as previous research.

1.4 Object-oriented Function Refinement

The goal of Refinable Functions implement optimal, and practical medium language-based modularity technique for script language among criteria of runtime variance, language primitives in industry practical languages.

To meet this motivation, Refinable Functions framed function as a class and apply syntax and mechanism of OOP to implement modularity of function, as a result, Since functions are treated as an object, the programmer can inherit/refine/subtype function just like we did for class.

Hierarchical modeling is fundamental idea of OOP, the reuse and localization of data and method are implemented by hierarchy through the inheritance and composition of classes. The principle of hierarchical modeling reuse and localization is already applied to data such as extending types or data structure. This paper presents Refinable Functions for reuse and localization of function by bringing the concept of classes inheritance and refinement to function to achieve general purpose modularity mechanism that is complementary for aspect and feature-oriented programming.

Function as a Class. Refinable Functions frames function can be constructed from classes which instantiate function object that contains an array of function and method for refinement. Figure 1 illustrates function class and its inheritance. The method extends and mutate behavior of refinable function by a adding updating and deleting member function or member refinable function using method like `R.add`, `R.before` or `R.assign`. The function class encapsulate functions array external method calls.

Function Inheritance. The inheritance is core mechanism for making hierarchy for code reuse and localization, As classes use inheritance as a subclassing mechanism, the function can be this function class can be inherited to localize more sophis-

Table 1: Comparison of Modularity and Software Product Line Properties to Refinable Functions

	Implementation Techniques		
Modularity Properties	Aspect-oriented Lang.	Feature-oriented Lang.	Refinable Functions
Code Reuse	Containment Hierarchy Composition		Function Inheritance
Func. Lv. Code Localization	Advice Type	Superimposition	Function Refinement
Class Lv. Code Localization	Containment Layer (e.g. Aspect Object)		
Composition Mechanism	Aspect Weaving	Class Refinement	
Code Container	Advice Function	Class or Modules	Object-oriented Function
Base Technology	Language Extension	Tools and Language Extension	OOP

[a] Function and Class level localization represent a medium of localizing variabilities of code at the granularity of function(method) and classes.

Software Product Line Prop.	Refinable Functions Prop.
Commonality Reuse	Function Inheritance
Variability Localization	Function Extension / Function Mutation
Composition Validity Checking	Function Subtyping Rule

ticated behavior by extends and mutate via the method. Using multi-level inheritance to gradually reuse commonality and localize variability can be possible. The inheritance mechanism is simple, make a duplicated copy of class using `R.new` method like `R'`. Currently, Function inheritance is studied in the context of functional programming initiated by a generalization of inheritance from an object into functions and more[18]. Function inheritance is studied in this context to provide extending behavior of existing function using another function[19]. The key difference is Refinable Function share same motivation but enables reflection and refinement to allows inherited function can be used to software engineering technique from only functional programming technique.

Function Subtyping. Upon to extends and mutate the function, checking Refinable function uses function subtyping rules for ensuring correctness of refinement. Function subtyping[20] is to create a subtype of function according to input and output type. The subtyped function should be contravariance(more general) to input for its superfunction and covariant(more specific) to the output type of superfunction. Since the execution of refinable function is sequential invocation of internal function array, the result of the previous function is the argument of current function, and the result of current function will be the input of next function, thus input subtyping rule for previous and current function and output subtyping rule for current and next function is important for typechecking of valid refinement.

Function Refinement. Application of extension and mutation of function, the behavior of a function can be refined, and the refinement enables to implement aspect-oriented and feature-oriented programming. For instance, cross-cutting concern localization is implemented by inheriting boilerplate of function with cross-cutting concerns and extends that function to contain distinguished behavior. For feature composition, same refinement mechanism by extension and mutation is applicable in function level and class level. Table 3 compares core components of language-based modularity for relatives.

1.5 Contributions of This Paper

The goal of this paper to present an object-oriented approach to implementing pragmatic language-based modularity. The premise of this paper is on the application of object-orientation to resolve language and environmental constraints to imple-

ment advanced language-based modularity and software product line mechanism that modern software platform with solid OOP technologies.

- **Concept of Refinable Functions.** This paper proposes concept of functions as classes, to promote reuse of common code and localization of code through object hierarchy, and shows the concept of function inheritance, function subtyping and function refinement to localize variabilities and inject commonalities gradually. To this end, refinable functions are generally usable language feature of in many favored language in the industry-strength language with language primitive feature without dependency in runtime.

- **Concrete Implementation of Refinable Functions.** We show referential design and implementation of Refinable function constructor in JavaScript, a widely used script language which no language extensions supported nor can not preserve runtime invariance since its original usage is on client-side scripting for web applications.

- **Formal Model of Object-oriented Function Refinement.** We present a formal model of refinable function as language feature perspective. We show syntax and evaluation context of refinable function and shows operational semantics of state changes when performing function refinement including extension, mutation, function composition including export and other managerial functionality such as execution and method definition

- **Applications of Refinable Functions.** For proofing usability of refinable function, we show application and its analysis on the correspondence of aspect-oriented and feature-oriented languages. We show implementing both language mechanism using the refinable function. Also, an extension of this approach, we implemented creational and structural design pattern for a function to perform refinement in a programmatical manner.

1.6 Paper Organization

This paper is organized as follows, section 2 summarize independent motivating a previous research in different focuses; language-based modularity, function subtyping and function inheritance which is an essential concept that comprises the concept of object-oriented function refinement. Section 3 introduces refinable function in software engineering aspect to show high-level view structural and behavioral overview of refinable function and shows simple and compound example of hierarchical function refinement for a web application to shows what program built with refinable function looks likes. Section 4 elaborate refinable function in programming language aspect using structural operational semantics to shows the concept and underlying mechanism can be generally applicable to any language by formal definition. Section 5 shows applications of a refinable function to a software engineering existing problem like crosscutting modularization, feature-oriented programming and shows the novel concept of function composition using design pattern-based creating and structuring of refinable function. Section 6 describes design and implementation of *RefinableJS* a JavaScript-based refinable function constructor as a reference implementation. Section7 discusses the limitation of concept and JavaScript-specific limitation of refinable function implementation. Section 7 shows related works of refinable function among related fields include functional programming, language-based and architectural approach to implement modularity. Section 8 concludes the paper with some future research direction and contribution provided by this paper.

RefinableFunction	
behaviorStore: (Function|*Self*)[]	
constructor(|*Self*[]): this	assign(Object<Function|*Self*>): this
Self.add(Function|*Self*): this	exec(Any): this
Self.before(Function|*Self*): this	catch(Function|*Self*): this
Self.after(Function|*Self*): this	new(): this
Self.update(Function|*Self*): this	defineMethod(String, Function|*Self*): this
Self.prepend(Function|*Self*): this	asBefore(): Object
Self.map(Function(Any):Function): this	asAfter(): Object asAround(): Object
Self.around(Function(Any):Function): this	
Self.bind(Any): this	
Self.delete(): this	
Self.asEntry(): this	

Figure 2: Refinable Functions Class

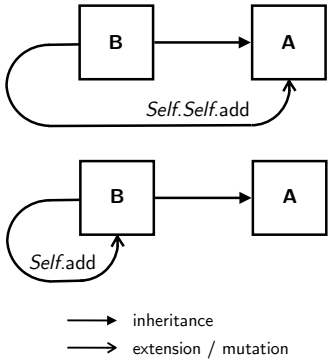


Figure 3: Referential Structure of Method

2 Refinable Functions

The goal of refinable functions is on provides implementation mechanism of language-based modularity based on OOP, thus script languages leverage advanced modularity feature without harming the expressiveness code of code. In this section, we introduce refinable function in architectural and software engineering aspect to shows the high-level model and concrete implementation, and next section provides a formal model of refinable function in language feature perspective.

As refinable function adapts OOP, it provides refinability on extension and mutation of function behavior via a method. This allows refining the function with primitive syntactic with compiler independent manner. Also, OOP is widely adapted by most of the major programming languages, industry adaptation of refinable function has no huddle.

The function as a classes concept allows to use the method of inheritance, and refinement technique like extends and mutates class into the function. The function variance check for function subtyping is used for checking the validity of refinement.

2.1 Functions as a Classes

Refinable Functions framed function as a refinable/inheritable classes. This allows bringing essential benefits of OOP to functions. Just like the reuse of data and method using inheritance and refinement of classes, Refinable Functions enables reuse of features and cross-cutting concerns of a function using inheritance and refinement of classes of function.

2.1.1 Structure and Construction

Refinable function constructor is a normal class that is consulted for storing functions as data in the array element, and series of the method the element in the data. For the element, refinable function supports native function but also accepts another refinable function. Fig.2 shows refinable function class. The class is comprised of data and method field. The data field contains, `behaviorStore`, whereas the method field contains set of a method for performing refinement. and list of a method for refinement.

2.1.2 Modification and Refinement

The mehtod is classified into two types, refinement and managerial method. Refinement method is a self-referential method of subfunction to extends and mutates member elements of refinable function. Refinement method is prefixed by *Self* in Figure 2, because these methods are not directly invocable instead, it is a method of designated subfunctions. As shown in the upper diagram on Figure 3, the *Self* prefixed method is modify *A* instead of *B* since is it invoked in object *B*. In contrast, the managerial method modifies directly for *B*. The refinement methods are dynamically created when the extension of refinable functions occur. The extension is performed by a method like `add` and these method takes both native function and refinable function, thus refinable function class is sort of recursive types for allowing this containment. Refinement method is invoked as a chain after refinable function for instance, when function g is member function of function f, then the refinement method *add* is invocable by calling *f.g.add*. The library performs multiple dispatches for the method at runtime to expose contained function name and its refinement method. The following code shows of construction and refinement. The `after`, `before` method plays similar role in advice type in aspect-orientrd languages;

2.1.3 Invocation Procedure

The refinement method is essentially manipulating `behaviorStore` function array for modifying its behavior. Most of the refinement method is receiving another type of refinable function, thus recursive composition and execution are possible. We will discuss each method in a later section. On the other hand, the primary role of management method is executed, handle errors refinable function. Also, it performs inheritance using *new* method and set user-defined refinement method by `defineMethod`.

2.2 Function Inheritance as a Commonality Reuse

The essence of inheritance is deriving an abstract concept from the concrete details[21], refinable functions allow to the programmer to build functions of a program like they are building classes. Inheritance of function is studied in the context of augmenting the operation of function without modifying its original source code in functional programming[19]. Refinable function extends this motivation but more ad-hoc approach hence, the more structured manner in a form of class which enables not only augmenting function from inheritance but also bring additive and shrinking refinement for the function containment. Fig. 5 shows an additive modification of refinable function for a simple web application that provides API for reading and writes texts and images.

RF_1 := new RF();
RF_1.add(f); $RF_1.f$.after(g); $RF_1.f$.before(e);
RF_2 := new $RF(RF_1)$;
$RF_2.f$.update(f_2);

Figure 4: Extension of Refinable Function

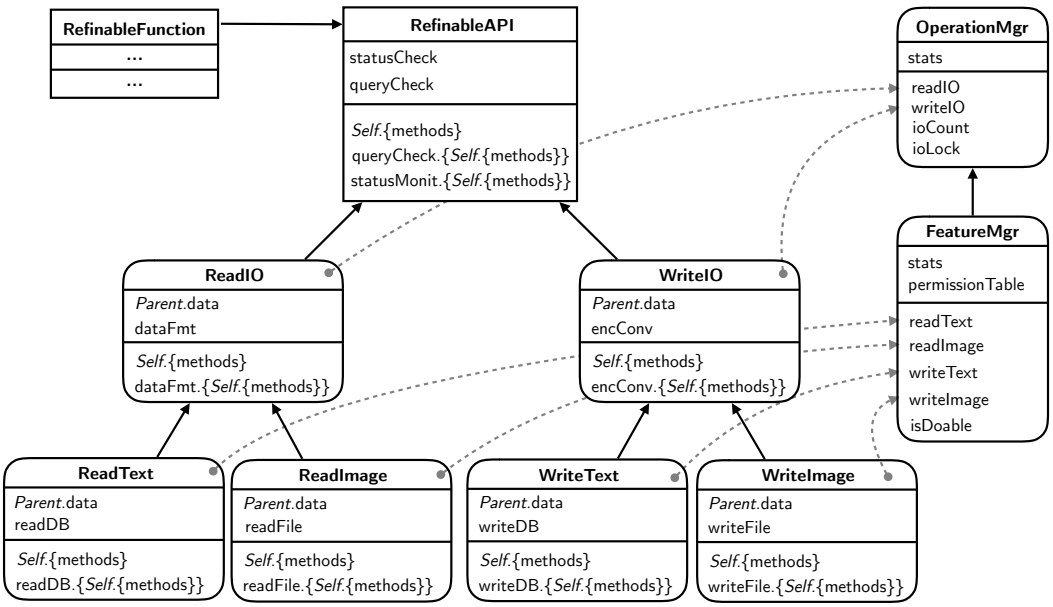


Figure 5: Simple Additive Modification of Refinable Functions Hierarchical Relationship

2.2.1 Extension of Function

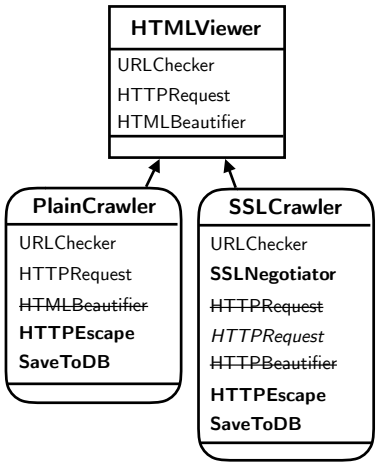
The way implementing reuse and localization with inheritance mechanism supported primitively on languages enables to implementing variability of natively in many programming langages. The figure 6 shows abstract example of function inheritance and class composition as used in figure 5. As function *RefinableAPI* is constructed by adding cross-cutting concerns, the listing 6.1 shows addition of function f and g to RF_1. Then made inheritance of function via `.new` method call in 6.2 which is identical part of inheritance in every inheritance in figure 5 as it keeps functions *Parent.data* shich is function f and g from parent refinable function RF_2.

2.2.2 Web Application using Functions Inheritance

Similar to domain analysis, the API server can be modeled using commanlity and variability analysis or derived model from feature structure tree[4]. The roots commonalities includes cross-cutting concerns like *statusCheck* and *queryCheck* into refinable function called *RefinableAPI*. The result of inheritance of *RefinableAPI* is done by duplicating its data and method. After inheritancce operation-level cross-cutting concern is applied for two instance of *RefinableAPI* which is *ReadIO* and *WriteIO*. The variability *dataFmt* and *encConv* is localized respect to the *ReadIO* and *WriteIO*.

2.2.3 Composition for Classes

Individual function can be usable as a method definition of classes, thus refinable function can be composed into classes at the right level of granularity of module contains. For example, the *ReadIO* and *WriteIO* is composed to *OperationMgr*, since it contain operation-level granularity. *OperationMgr* is a class that can be inherited for feature-specific Manager, lets extends *OperationMgr* by adding permission table for checking API issuer's priviledges. The inheritance of *FeatureMgr* allows




```
1  var HTMLViewer = new Refinable();
2  HTMLViewer.add(URLChecker)
3    .add(HTTPRequest).add(HTMLBeautifier);
4
5  var PlainCrawler = HTMLViewer.new();
6  PlainCrawler.HTMLBeautifier.delete()
7      .add(HTMLEscape).add(SaveToDB);
8
9  var SSLCrawler = HTMLViewer.new();
10
11 SSLCrawler.HTTPRequest.bind(credential);
12 SSLCrawler.HTTPRequest.before(SSLNegitoator)
       ;
13 SSLCrawler.HTMLBeautifier.delete()
14     .add(HTMLEscape).add(SaveToDB);
```

Figure 7: Function Refinement of `HTMLViewer` to `PlainCrawler` and `SSLCrawler`

the inheritance of refinable function as well. Such as *ReadIO* and *WriteIO* is transformed into feature-specific functions like *ReadText* or *WriteImages* by reusing operation-specific crosscutting concern and localizing feature-specific concern like *readDB* or *writeFile*.

Later, in 6.3 Aspect object $Aspect_1$ is created for use of RF_2 as a *method* in some arbitrary class. $Aspect_1$ is composed to $Class_1$ as a before advice function in 6.4. As right side 5, the refinable function `ReadIO` and `WriteIO` plays role of before advice, when *OperationMgr* is inherited to *FeatureMgr* by *.asBefore* method in 6.3. Aspect is composed to class, to follow the native method invocation mechanism, refinable function is wrapped into normal method function and additional refinement is not be made.

$$RF_1 \;:=\; \text{new } RF().\text{add}(f).\text{add}(g); \tag{6.1}$$
$$RF_2 \;:=\; RF_1.\text{new}(); \tag{6.2}$$
$$Aspect_1 \;:=\; \text{new } Aspect(\{\,\text{method: } RF_2.\text{asBefore}()\,\}); \tag{6.3}$$
$$Aspect_1.\text{compose}(Class_a); \tag{6.4}$$

Figure 6: Extension of Refinable Functions as an Advice Functions for Classes

2.3 Function Refinement as a Variability Localization

In the previous part, we studied function inheritance as reuse mechanism of modularity for the commonalities. In this section, we introduce function refinement which aims to take extends and mutates body of function by modifying its comprised element. Previously, including function inheritance, research programming is studied in the context of types[20], despite the input/output is good at the express compatibility of function but on the other hand, we foresee function body is another type of feature we can distinguish the behavior of functions.

Function subtyping is essentially refining content function partially to implement variability without altering existing source code. To make a localizable form of function, we previously used function inheritance to make a new instance of the function.

```
1 var HTMLViewer = Refinable.new().apply([
2 { add: [URLChecker, HTTPRequest, HTMLBeautifier] } ]);
3
4 var SSLCrawler = HTMLViewer.new().apply([
5 { name: `HTTPRequest', bind: credential, before: SSLNegotiator }, { delete: `HTMLBeautifier' },
6 { add: [HTMLEscape, SaveToDB] } ]);
7
8 var PlainCrawler = HTMLViewer.new().apply([{ delete: `HTMLBeautifier' }, { add: [HTMLEscape, SaveToDB] } ]);
```

Figure 8: An alternative form of refinement using object.

2.3.1 Mutation of Function

Since Refinable Functions is essentially an object that contain array of function and list of method for refining array, providing simple. safe way to refining array is important. Refinable Functions adapted many essential functional compositon idea from Aspect-oriented Programming, such as `before`, `after`, `around` for augment behavior from relative position, and traits[22] from OOP via `assign` method call. `update` and `delete` method is newly added to perform update and deletion operation of sub-behavior.

On the other hands from refinement method, the managerial method in addition, there are non-refinement method exist for executing refinable function using `exec`, error handling using `catch`, inheritance using `new` and attach user-defined type of refinement using `.defineMethod`. The details of method API list is on Fig. 9

2.3.2 Web Applications using Function Refinement

The figure 7 shows function refinement of `HTMLViewer` as a `PlainCrawler` and `SSLCrawler`. As shown in the right side of Figure 7, it creates new instance of function to make two refinable function by calling `new` method in line 1 and 5. In order to create crawler function with HTMLViewer, the `PlainCrawler` removes HTMLBeautifier and adds concerns of HTTPEscape and SaveToDB by calling `delete` and `add` method in Line 6. Similar to PlainCrawler, it the `SSLCrawler` is derivated by adding SSLNegotiator before the HTTPRequest and perform same refinement procedure as PlainCrawler. The bind method in the HTTPRequest is same as parameter binding in many functional programming languages, using this binding the refinable function can separete crosscutting concerns as well as other scaffolding work to user.

2.3.3 Imperative and Declarative approach for Refinement

As we showed in 7 allows humane way of performing refinement of refinable function using Imperative manner. Imperative refinement is useful for building a small-sized application but when it comes to large-scale, the verification and control of refinement correspond to the requirement is hard because the method-based refinement is the human-centered approach. Refinable functions provide declarative form of refinement using pure data. The Fig. 8 shows equivalent form of refinement for 7. The apply method processes refinement based on the given array. As the function has execution flow, a direction which is the key difference to object, the refinement array represent the order of refinements to be applied. Here, the array has a list of object, in the object, the refinement method is used as key and the target of refinement is used as the value. For example, the line 1 and 2 shows extension, where the three element `URLChecker`, `HTTPRequest` and `HTMLBeautifier` to be added into newly created refinable function. To perform subfunction specific refinement, like line 5, the refinement array is consist of

Table 2: Standard Method in Refinable Function

Refinement Types	Method Name	Parameter: Result	Return Value
Extends	`add`	Function|*Self*: this	Append new subfunction
	`prepend`	Function|*Self*: this	Prepend new subfunction
	Self.`before`	Function|*Self*: this	Insert new subfunction before the target subfunction
	Self.`after`	Function|*Self*: this	Insert new subfunction after the target subfunction
Mutates	*Self*.`update`	Function|*Self*: this	Replace specified subfunction with new subfunction
	Self.`map`	(Function(Any):Function): this	Wrap target subfunction with new function
	Self.`around`	(Function(Any):Function): this	Alias of *Self*.`map`
	Self.`delete`	: this	Delete specified subfunction to array
	Self.`bind`	Any: this	Bind function arguments
	`assign`	Object<Function|*Self*>: this	Update or remove indivudual subfunction with traits
Exports	*Self*.`asEntry`	: this	Append new subfunction
	Self.`asBefore`	Object	Export refinable function as a before advice
	Self.`asAfter`	Object	Export refinable function as a after advice
	Self.`asAround`	Object	Export refinable function as a around advice
Invokes	*Self*.`exec`	Any: this	Invoke refinable function
	Self.`catch`	Function|*Self*: this	Catch invocation errors
defineMethod	`defineMethod`	String, Function|*Self*: this	Define custom refinement method
Inheritance	`new`	: this	Create new instance of refinable function

[a] The term *this*is denotes return current instance of refinable function itself for allow method chaning

three element where the first element is consist of application of binding `credential` and adds before advice SSLNegotiator to `HTMLRequest`. The refinement that does not take an argument can be formed without name explicitly, for example as the second element shows, mentioning refinement operation and put subfunction name into value is enough. when large refinement is made, the imperative approach may degrade readability, and hard to verify domain-specific constraints. A declarative approach to refinement is good for creating and handling large scale refinement in the relatively structured way.

2.4 Function Refinement for Methods and Classes

As OOP became dominant mechanism for promoting reuse in programming languages and software engineering, class-level reuse is important for many modularity techniques, thus in the last decades much class-level modularity technique is became researched notably Aspect-oriented Programming[1], the paradigm of feature-orientation is also largely focused in classes. We already showed the refinement of function that corresponds to advice function in the aspect, in this section, we extend advice function to implement advice method in AOP. Since the method of a class is essentially function with a shared variable, the refinable function can be usable as a method using along with built-in composition tools.

The figure 11 shows composition of refinable functions as a cross-cutting concern. The line 1 and 2 shows build of cross-cutting concern as a normal refinable function. Each functions `logger` and `errHdr` localize cross-cutting concrn `requestLogging` and `errHandler`, then both function is inherited to build `preProc` and `postProc` which localize anothe crosscutting concern `networkCheck` and `errResp` in a proper position. The line 4 and 5 shows construction of aspect object using aspect

Figure 9: Aspect Composition Rules and Types

Host Advice (A_h)	Target Advice (A_t)	Return Refinable Functions Type	Return Advice Type
before	before	$A_t \circ A_h \circ \square$	brefore
	after	$A_t \circ \square \circ A_h$	entry
	around	-	-
	entry	$A_{t1} \circ \square \circ A_h \circ A_{t2}$	entry
after	before	$A_t \circ \square \circ A_h$	entry
	after	$\square \circ A_h \circ A_t$	after
	around	-	-
	entry	$A_{t1} \circ A_h \circ \square \circ A_{t2}$	entry
around	before	$A_t \circ (\lambda.\ A_{h1} \circ \square \circ A_{h2})$	entry
	after	$(\lambda.\ A_{h1} \circ \square \circ A_{h2}) \circ A_t$	entry
	around	$A_{t1} \circ (\lambda.\ A_{h1} \circ \square \circ A_{h2}) \circ A_{t2}$	entry
	entry	$A_{t1} \circ (\lambda.\ A_{h1} \circ \square \circ A_{h2}) \circ A_{t2}$	entry
entry	before	$A_t \circ A_{h1} \circ \square \circ A_{h2}$	entry
	after	$A_{h1} \circ \square \circ A_{h2} \circ A_t$	entry
	around	$A_{t1} \circ A_{h1} \circ \square \circ A_{h2} \circ A_{t2}$	entry
	entry	$A_{t1} \circ A_{h1} \circ \square \circ A_{h2} \circ A_{t2}$	entry

[a] Composing to A_t that are typed to around advice are not possible by it absorbe refinement method on A_h

$$\frac{\tau_h = \tau_t \qquad RF_h \in \{\text{before}, \text{after}\}}{\tau_h \circ \tau_t \vdash \tau_h}$$

(a)

$$\frac{\tau_h = \tau_t \qquad RF_h \notin \{\text{before}, \text{after}\}}{\tau_h \circ \tau_t \vdash \tau_{entry}}$$

(b)

$$\frac{\tau_h \neq \tau_t}{\tau_h \circ \tau_t \vdash \tau_{entry}}$$

(c)

Figure 10: Advice Type Derivation Rules

```
1  var logger = new Refinable().add(requestLogging),  preProc = logger.new().prepend(networkCheck);
2  var errHdr = new Refinable().add(errHandler), postProc = errHdr.new().add(errResp);
3
4  var MonitAspect = new Refinable.utils.Aspect({ plain: preProc.asBefore(), ssl: preProc.asBefore() });
5  var ErrAspect = new Refinable.utils.Aspect({ plain: postProc.asAfter(), plain: postProc.asAfter() });
6
7  class Crawler {{plain: PlainCrawler, ssl: SSLCrawler}};
8  MonitAspect.compose(Crawler); ErrAspect.compose(Crawler);
9  var CrawlerA = new Crawler(websiteAUri);
10 var CrawlerC = new Crawler(websiteBUri);
```

Figure 11: Cross-cutting Concern Composition for `PlainCrawler` and `SSLCrawler`

constructor provided by `Refinable.utils.Aspect`, the form of aspect object is a normal object that contains refinable function in exportion format to define advice type. In this case, the `preProc` is exported `asBefore` where `postProc` is exported `asAfter`. The line 8 shows composition of aspect object into classes.

Support for class-based refinement is essential for using refinable functions as an aspect-orientation implementation.

As code example shown, refinable function supports 4 types of advice function. As typical AOP example, it supports before, after and around using `asBefore`, `asAfter` and `asAround` respectively. Futher more, To promotes maximum support of product line implementation, refinable function offer *entry* advice via `asEntry` method call which is special kinds of before advice except poiniting to be before advice of arbitrary subfunction. For consider refinable function R with subfunction a, b, c. By calling $R.b.asEntry$, the new arbitrary subfunction will be placed between existing subfunction b and c. The table 10 shows advice to advice composition rule aspect inheritance(subaspecting) in refinable function. Fig. 11 shows general derivation rule of aspect inheritance. As in Fig. 10 shows, the host advice A_h is augmented by A_t, as a result of inheritance, the refinable function type is extended where τ_h is type of aspect in the host aspect where $type_t$ is type of aspect in the target advice. The $\square$ shows subfunction composition point of refinable function where the $\circ$ symbol is means of actual composition. Most of resulting advice function type is entry except homogenous advice composition that are either before or after advice type that has aspect or direction for augmenting host advice A_h.

Class-level composition support for refinable function allows to implement refinable functions as a feature-oriented programming technique because AOP is a major way to implement FOP aspect-orientation is important for making Thus, support for class-level refinement as well as function level refinement is needed.

The support for class-level refinement require refined function need to be recomposed as an advice function, also the mechanism like aspect weaving is needed. The resulting function is plain and it is executable in the normal class environment. In our implementation the normal function wrap refinable function with containing some glue code for generating refinable function just in time we will discuss this issues later in the design and implementation section.

2.5 Typechecking of Function Refinement

Object-oriented function refinement presented by refinable function allows for manipulation of function body in an arbitrary way, therefore bringing typechecking for verifying the correctness of function refinement is required.

Essentially, refinable functions are executed by the sequential invocation of contained subfunction, the typechecking rule to determine whether the refinement operation is valid is simple.

To verify the correctness of refinement, we used function subtyping rule for ensuring safety. The figure 12 shows subtyping rule for function. The rule is, the return type of function must be covariant(subtyped) for the parameter of succeeding function. In other words, the output of function must be specified type then its succeeding input type. At the parameter's perspective, the parameter must be contravariant for its argument returned from the preceding function.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \longrightarrow S_2 <: T_1 \longrightarrow T_2}$$

Figure 12: Rule for Function Subtyping

$$\frac{e_2 <: f_1 \qquad f_2 <: g_1}{e \circ f \circ g \vdash \top}$$

Figure 13: Rule for Function Extension and Mutation

$$\dfrac{\dfrac{o(e) \in \{String, Number\} \quad i(f) \in \{String\} \vee \{String, Number\}}{e \circ f \vdash \top} \qquad \dfrac{i(g) \in \{Number\} \quad o(f) \in \{Number, String\} \vee \{Number\}}{f \circ g \vdash \top}}{e \circ f \circ g \vdash \top}$$

$$\dfrac{\dfrac{o(e) \in \{String, Number\} \quad i(f) \in \{String\} \vee \{Boolean, Number\}}{e \circ f \vdash \bot} \qquad \dfrac{i(g) \in \{Number\} \quad o(f) \in \{Boolean, String\} \vee \{Number\}}{f \circ g \vdash \bot}}{e \circ f \circ g \vdash \bot}$$

Figure 14: Example of Refinement Typechecking for Function f for e and g.

Table 3: Types Notation

Name	Boolean Type	Number Type	Any Type	Null Type	Undefined Type	Union Type	Boolean Type
Notation	`Boolean`	`Number`	`'Any'`	`Null`	`Undefined`	`Object`	`Boolean`

2.5.1 Typingchecking Rule

Function subtyping[20] is to create subtype of function according to input and output type. The Figure 12 shows subtyping rule of function. Inorder to fulfill the subtyping relationship of function S with argument S_1 and output S_2 from function T, the input of subtyped function T_1 should be contravariance(more general) for its superfunction S_1. For the output S_2 should be covariant(more specific) to the output type of superfunction T_2. Since the execution of refinable functions is a sequential invocation of internal function array, the general-specific rule for function subtyping rule is applied to verifying refinement correctness. The result of preceding function e_2 is the argument of current function thus, e_2 should be subtype of f_1 which is input type of extended or mutated subfunction f. As for output of f, f_2 should be covariant for succeeding function g_1. When this conditions met, the resulting refinement for the composition of function e, g and f is valid.

2.5.2 Typing Refimenent

Our implementation follows the typing annotation from TypeScript[1] which is superset of JavaScript that adds optional static typing to the language, and Flow[2] A static typechecker for JavaScript. The table x shows list of type annotation. The type annotation is applied by `.setType` method. The following example shows typing for http request library `request`.

the figure 14 shows refinement typechecking for function f where it extens the refinable function comprised with e and g where i and o denotes input and output type and e g existing subfunction where f is new subfunction that placed between them. The refinement checking is placed in two part, a typechecking for preceeding function e and succuding function g. Only if both typechking is resolved(in $\top$) then the refinement(in this case, composition) of function e, f and g is resolved. For the typechecking of preceeding function, the right side of top tree shows valid refinement for return type of preceding function e typed as $String$ and $Function$, in this case, the input of f shoud be covariant, a subtype of $o(e)$ which is $String$ and $String, Number$, similarily, the same part on the bottom tree, the function refinement is not resolved by $i(e)$ is not a subtype

[1]TypeScript. JavaScript that scales. *https://www.typescriptlang.org*
[2]Flow. A Static Type Checker for JavaScript. *https://flow.org*

of $o(e)$ since it includes $Boolean$ instead of $String$. For the typechecking of succeeding function, refinable function, the left side of top tree shows valid typechecking for succedding function. The refinement of function $o(f)$ for $i(g)$ is valid as for $o(f)$ is contravariant, more exrended type for $i(o)$. However, the rightside of bottom tree does not resolved as it shares no common types from $i(g)$ for $o(f)$.

Implementing variance type checking is trivial, it just checks host type to guest type recursively if the type is an object. Primitive types are compared directly.

3 Syntax and Semantics

This section formulates syntax and semantics of refinable functions. The presented syntax and semantics are based on the JavaScript implementation *RefinableJS*. We formulate the calculus for refinable function $\lambda_r$for define operation refinable function concisely as an internal operation by small step operational semantics, which can hide language-specific issues to provides focus on essential points of internal operation of refinable functions.

3.1 Syntax of λ_r

The Fig. 15 shows syntax of λ_r. The syntax is classified as refinement operation types in table. 9. The expression e denotes refinable function. The [extends function] line shows 4 dirrerent kinds of function extension using e, as it shown the refinable function is usable in both host object of function as well as argument of another refinable function. Same mutation operation is applied to [mutates function] operation where refinable function is used as object and subject. The [exports function] shows syntax of usage on refinable function as an advice function, and the syntax of [invoke function], [dispatch function] represents syntax of managerial part of refinable function where takes function or refinable function as a callback handler. Each syntax transform internal states of refinable function which is described in Fig. 18. The $\lambda_r$is based on previous research on formalism for JavaScript[23] and language feature[24]. As the syntax is depended in the host language JavaScript, we use evaluation contexts[25] for the extended language $\lambda_r$in Fig. 16.

3.2 Runtime of λ_r

Fig. 17 describes runtime of λ_r. The runtime γ is defined by combination of state store ϕ and list of refinement methods m. The subfunction is either type of refinable function or native function. The instance of subfunction is is used for represent host function of refinement R, and guest function for refinement r for Host R. q and s is Preceding and Succeeding function for

$e \in Exp$	=	`constructor`(e)	
	|	e.`add`(e) | e.`before`(e) | e.`after`(e) | e.`prepend`(e)	[extends function]
	|	e.`map`(e) | e.`around`(e) | e.`bind`(e) | e.`delete`(e) | e.`assign`(e)	[mutate function]
	|	e.`asEntry`() | e.`asBefore`() | e.`asAfter`() | e.`asAround`(e)	[exports function]
	|	e.`exec`(e) | e.`catch`(e)	[invoke function]
	|	e.`defineMethod`(e)	[dispatch function]
	|	e.`new`()	[inherit function]

Figure 15: Syntax of λ_r

r. In function mutation part, we specify the target of mutation to subfunction t. The function store ϕ is comprised of a list of Subfunction. Finally, there is primitive types definition for input and output types respectively i and o; Each refinement step extends and mutates ϕ in order to change the behavior of the refinable function, and each step of refinement type checking for r to q and s is performed.

3.3 Semantics of λ_r

We present the reduction semantics of λ_r, which based on the formal modeling work on JavaScript semantics[24, 23] with the evaulation riles shown in the Fig. 16 and Fig. 17. The semantics consists of two evaluation relation: The relation $\hookrightarrow$ represents expression-to-expression reductions, whereas the $\longrightarrow$ sign represents state-to-state transition.

[E-EXTENDS]. This rule handles extension of refinable function. Function extension is essential for additive modularization such as localizing more cross-cutting concerns which is a traditional mechanism for language-based modularity. The semantics represent only add method, but same semantics are applied for all extends function semantics in 15. The extension of function is adding new subfunction r into internal function array π. In order to accomplish push, the variance typechecking is performed to check the validity of composition. The expression $i(x)$ represent input type of function or refinable function x by method `setType` and the $o(x)$ is output for vice versa. In the extension rule, the input type of r should be a subtype, covariant to preceding function q, as well as output type of r, should be covariant to succeeding function s. Also, the function name used in an identifier in refinable function should be a conflict to an array of existing function π. If these conditions met, the function is inserted to π and refinement methods are created for r in object property m. In this end, the final state of m, π and R are all changed.

[E-MUTATES]. This rule handle mutation of refinable function. The goal of function mutation is to modify or shink existing module, by replacing updating subfunction or binding arguments. The semantics represent only `update` method for subfunction t, but same semantics are applied for all mutates function semantics in [fig syntax of λ_r]. The mutation of the function is essentially replaced and insert specified subfunction or bind pre-allocated argument. In order to make mutation of individual subfunction, typechecking to input and output type of new function r is to conform preceding and succeeding function is performed as same as extends, and the name of r replaced fixed to the name of t, this name checking is not performed. Finally, push r at the position of t to make a replacement.

[E-EXPORT]. Function exportation is important because it allows hiding indented composition mechanism to user side program, this it can enforce invalid usage of function, for example, some advice function or configuration in product line should be post- or pre-processed only, instead of imperatively following this rule, the writer of library can predefine its composition method. This rule transform function as an exportable format, that are compatible for composing to other refinable function using `compose` method in Fig. 9. Exportin function is essentially creating an identical copy of function with a predefined way of composition among before, after the refinable function or arbitrary position side the function using `asEntry`. However, exportation and composition of function are performed in different time, in order to defer the composition by additional invocation, it returns a curried function that contains composition mechanism with bounded composition type and refinable function. This process does not require to check the argument, but at the time of composition. To avoid manipulation from later refinement, export cuts the reference chain of m and π by creating hard copied set of each and return new instance $R'$with composition code wrapped function.

[E-COMPOSE]. the function composition is designated feature This rule compose a function with exported, advice function.

$$
\begin{array}{rcl}
E & = & \square \\
& | & E.\texttt{add}(e) \mid v.\texttt{add}(E) \mid E.\texttt{before}(e) \mid v.\texttt{before}(E) \mid E.\texttt{after}(e) \mid v.\texttt{after}(E) \mid E.\texttt{prepend}(e) \mid v.\texttt{prepend}(E) \\
& | & E.\texttt{map}(e) \mid v.\texttt{map}(E) \mid \mid E.\texttt{bind}(e) \mid v.\texttt{bind}(E) \mid E.\texttt{delete}(e) \mid v.\texttt{delete}(E) \mid E.\texttt{assign}(e) \mid v.\texttt{assign}(E) \\
& | & E.\texttt{asEntry}() \mid E.\texttt{asBefore}() \mid E.\texttt{asAfter}() \mid E.\texttt{asAround}() \\
& | & E.\texttt{exec}(e) \mid v.\texttt{exec}(E) \mid E.\texttt{catch}(e) \mid v.\texttt{catch}(E) \\
& | & E.\texttt{defineMethod}(e) \mid v.\texttt{defineMethod}(E) \\
& | & E.\texttt{new}()
\end{array}
$$

Figure 16: Evaluation context of λ_r

$$
\begin{array}{rcl}
\gamma \in RefinableFunctions & = & Addr \hookrightarrow (Store \times MethodList) \\
\rho \in Subfunctions & = & Addr \hookrightarrow RefinableFunction \mid Function \\
R \in Host & = & Addr \hookrightarrow (RefinableFunction \times Addr) \\
r \in Guest & = & Addr \hookrightarrow (Subfunctions \times Addr) \\
q \in Preceding & = & Addr \hookrightarrow (Subfunctions \times Addr) \\
s \in Succeeding & = & Addr \hookrightarrow (Subfunctions \times Addr) \\
t \in Targets & = & Addr \hookrightarrow (Subfunctions \times Addr) \\
m \in MethodList & = & Addr \hookrightarrow Object \\
\pi \in Store & = & (Subfunctions \times Addr) \\
i \in InputType & = & Types \\
o \in OutputType & = & Types \\
\psi \in Types & = & \{Boolean, Number, String, Object, Function\}
\end{array}
$$

Figure 17: Runtime of λ_r

$$\frac{i(r) \in o(q) \qquad o(r) \in i(s) \qquad r \notin \pi \qquad \pi' = \pi :::(r) \qquad m' = m::(r)}{< m,\ \pi,\ E[R.\texttt{add}(r)] > \ \longrightarrow\ < m',\ \pi',\ E[R'] >} \quad \text{[E-Extends]}$$

$$\frac{t \in \pi \qquad i(r) \in o(q) \qquad o(r) \in i(s) \qquad \pi' = \pi :::(r,t)}{< m,\ \pi,\ E[R.t.\texttt{update}(r)] > \ \longrightarrow\ < m,\ \pi',\ E[R'] >} \quad \text{[E-Mutates]}$$

$$\frac{t \in \pi \qquad m' = m \qquad \pi' = \pi}{< m,\ \pi,\ E[R.t.\texttt{asEntry}()] > \ \longrightarrow\ < m',\ \pi',\ E[\lambda.argR'] >} \quad \text{[E-Exports]}$$

$$\frac{i(r) \in o(q) \qquad o(r) \in i(s) \qquad \pi' = r.\pi :::(\pi) \qquad m' = r.m::(m)}{< m,\ \pi,\ E[r.\texttt{compose}(R)] > \ \longrightarrow\ < m',\ \pi',\ E[R'] >} \quad \text{[E-Compose]}$$

$$\frac{args \in i(R) \qquad v = R(args)}{< m,\ \pi,\ E[R.\texttt{exec}(args)] > \ \longrightarrow\ < m,\ \pi,\ E[v] >} \quad \text{[E-Exec]}$$

$$\frac{r \notin m \qquad m' = m :::(r)}{< m,\ \pi,\ E[R.\texttt{defineMethod}(r)] > \ \longrightarrow\ < m',\ \pi,\ E[R'] >} \quad \text{[E-DefineMethod]}$$

$$\frac{args \in i(\pi_n) \qquad v = \pi_n(args)}{< args;\ \textbf{while}\ \pi_{n+1}\ \notin Nil \mid \pi > \ \longrightarrow\ < v;\ \textbf{skip}\ v >} \quad \text{[Invokes]}$$

The operator $::$ means append of subfunction the element to an array like data structure π and $:::$ is dispatch of object method with given refinable functions.

Figure 18: Semantics of λ_r.

In order to make a composition, the subject of the composition is advice function r where the object is R. It checks the argument conforms to be a subtype of q and return type to be a subtype of s. After the validation process is performed, it adds function array π and method list m to r, the type of addition it depends on type of advice/

[E-Exec] and [E-Invoke]. These two rule related to the execution of refinable function, since refinable function is consist of arbitrary numbers of subfunction, sequential execution and argument passing is essential for processing refinable function. the first rule checks argument is conforms to the inputtype of refnable function R, and invocation function R with parameter, which involves the use of rule [E-Invoke]. This rule iterate all members of π and invoke $n + 1_{th}$ element of π after typechecking until it is empty. It finally returns value v ss a result of function application of argument $args$.

[E-DefineMethod]. This rule is used for adding domain-specific refinement method for function. The addition procedure is simple, first it validates the name of method function r to m, and add method to the m.

4 Applications and Analysis

In previous sections, we present background motivation and the conceptual introduction of refinable function with its internal operating mechanism in language-independent representation, operational semantics. In this section, we map concept of refinable function as a solution to research problem of modularity in programming languages and software engineering, this is important for providing generality and applicability for the object-oriented refinement of function as a general-purpose language-based modularity mechanism. Specifically, we show:

- **Correspondence to Aspect-oriented Languages** The inheritance and extension of refinable function enable the reuse of commonality and localization of variability of function and classes, which is direct correspondence to localizing scattered and tangled cross-cutting concern in Aspect-oriented Languages. The [E-Extend] and [E-Exports] rules handles reuse and localization operation respectively.

- **Correspondence to Feature-oriented Languages** Feature-oriented Programming is a way of implementing a feature using a set of software components[4]. It is performed by many different levels of a software system such as classes or method[26], by the composition of function and classes to make a sophisticated artifact, the extensibility and safe mutability of refinable function allow dependency-free feature-oriented languages for any platform including web.

- **Solution for Feature Interaction Problem** The extension and mutation of refinable function can be also usable for resolving feature interaction problem by adding coordination code explicitly using mutation method or implicitly by function export.

- **Programmatical Aspect and Feature Composition** The interoperability provided by unified refinement method allows to create and structure refinable function programmatically just like object-oriented programming, in this sense applying arbitrary design pattern for creating and structuring refinable function is possible to implement a function or feature composition.

Table 4: Comparison of Method-level Modularity Techniques

	Composition Types	
Extension Type	Asymmetric	Symmetric
+	AOP	-
+ and -	Refinable Functions	-

Table 5: Comparison of Class-level Modularity Techniques

	Composition Types	
Extension Type	Asymmetric	Symmetric
+	AOP	Hyper/J
+ and -	DOP, RefinableFunctions	-

4.1 Separation of Concerns

Separation of concern[27] is a core goal of modularity in software engineering. The goal of separation of concern is on includes properties like low impact for change, comprehension and managed complexity and reuse in order to make software easy.

OOP helps to model the problem using classes and promotes reuse by extension of class and design pattern, thus the modularization class is the essence of modularity in OOP[28]. Several cross-cutting concern mechanism for OOP is made which is not well modularized through traditional OOP mechanism, including Aspect-oriented programming(AOP)[1]. The previous research on separation of concern shows many criteria of modularity. For composition strategy, is classified as asymmetric and symmetric approach[29]. AOP and DOP are asymmetric strategy since is have augment and refine base and core module by applying aspect and delta module, whereas he symmetric approach enables modularization from composing set of concerns, techniques include Hyper/J[28] which is originated from the Multi-dimensional separation of concern[30] and subject-oriented programming[31]. For the structuring strategy, AOP is classified as an additive by focusing on the technique is augmenting behavior of class where DOP is both additive and subtractive since the changes of module extend and shrink of the based module. Table 4 shows a comparison of modularity technique support for method-level modularity, om contrast to AOP, refinable support both additive and subtractive modularity via function extension as well as function mutation. Table 5 shows comparison on modularity technique of class-level modularity.

4.2 Cross-cutting Concern Modularization

A concern is a semantically coherent issue of a problem domain that is of interest to a stakeholder[32], and cross-cutting is a structural relationship between concern implementations that is alternative to hierarchical and block structure in horizontal way[32]. As we described in section 1, such cross-cutting is localized with various language and tooling techniques.

The function is the basic building block of functional and object-oriented programming as a form of a method. From [32]Cross-cutting for the function is that the evaluation of a function involves the evaluation of terms that belong to multiple concerns and that a concern is implemented by the evaluation if terms in different functions. Shortly, individual statement of function which corresponding to handling concern is the body is scattered across the different function.

There are several aspect-oriented programming for functional languages shares same motivation for augmenting functional behavior without modifying its code such as Aspectual Caml[33] however it requires compiler modification, which against our motivation for modularity implementation. Function inheritance[19] is introduced, despite its benefit as a general purpose solution, it leaks refinability of the composed artifact. We discuss four property of modularity on extension and composition strategy shown in Table. 4 and 5.

```
1 //reactive additive modularization
2 var read = new Refinable().add(function checkEmpty (data){ return !data; });
3 var readNumber = read.new().add(function numTypeChecker (data) { return typeof data === number; });
4 var readUserId = readNumber.new().add(function idChecker (data) { return id > 0 }).add(function dbProc (data)
      { DB.Users.read() });
5 var result = readNumberUserId(4);
```

Listing 1: Additive Example of Reactive Modularization

```
1  var read = new Refinable().add(function checkEmpty (data){ return !data; });
2  var readNumber = read.new().add(function numTypeChecker (data) { return typeof data === number; });
3  var readNumber.asBefore();
4
5  var getId = new Refinable().add(function dbProc (data) { DB.Users.read() });
6  var readUserId = readNumber.compose(getId);
7
8  var readUserName = readUserId.new();
9  readUserName.numTypeChecker.delete().checkEmpty.after(stringTypeChecker);
10 var readUserType = readUserId.new();
11 readUserType.numTypeChecker.delete().checkEmpty.after(booleanTypeChecker);
```

Listing 2: Substractive Example of Proactive Modularization

4.2.1 Types of Module Extensions

Additive Approch. Additive modularization is augmenting behavior of function by extension and mutation of function. Programmer can implementing this modularity by explicitly appling extension and mutation to the refinable functions. Fig. 1 shows additive modularization. In the example, cross-cutting concerns `checkEmpty`, `numTypeChecker`, `idChecker` and *dbProc* is added to consult feature `readNumberUserId`.

Substractive Approch. DOP introduces substractive modularization by applying changes of deletion. In subtractive case programmer can implement functions by removing details from the boilerplate. This type of modularization is useful for making library that can refined by user Fig. 2 shows subtractive modularization. In the example, line 1-3 construct refinable function with additive as an additive manner where line 8-9 and 10-11 shows subtractive modularization using complete refinable function `readUserId`, both subtraction removes unsuitable subfunction `checkEmpty` and adds new subfunction to implement desired feature `stringTypeChecker` and `booleanTypeChecker`.

4.2.2 Types of Module Refinements

Reactive Approach. Reactive modularization is an imperative way of applying modular changes of refinable function. The refinement mechanism is defined by the user so that, the user can explicitly configure procedure of modularization. This approach is useful for build trivial modularization and small-sized application. 1 shows reactive approach of modularity by applying extension of function explicitly via `add` method call.

Proactive Approach. A proactive approach for modularization is essentially different from the reactive approach by predefine the composition mechanisms of refinable function via advice function type. This mechanism is implementable using *.asBefore*, *.asAfter*, *.asAround* for setting aspect-oriented point of exportation and *.asEntry* for setting arbitrary point of exportation, by predefining composition mechanism the user of refinable function can be different to the creator, thus proactive

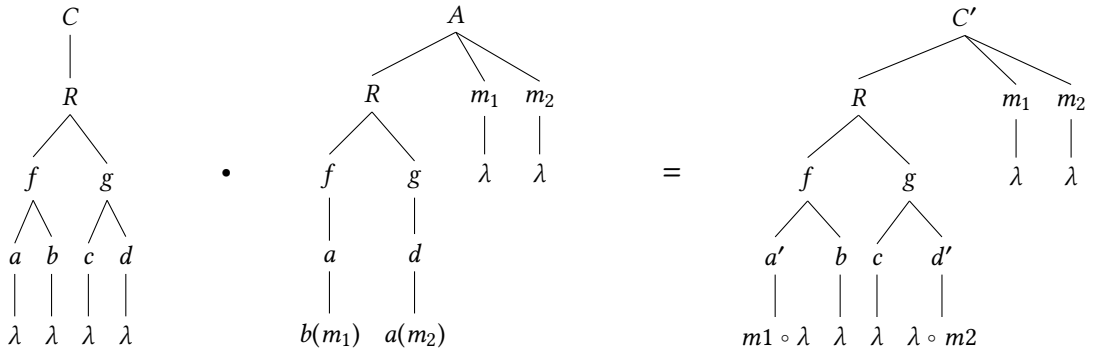


Figure 19: Superimposition of Refinable Functions Class

approach reduces the complexity of declaring explicit composition type, it is usable for large-scale application. The listing 2 shows proactive modularization of web API server. In line 3, the `readNumber` is a cross-cutting concern, thus it is defined as before advice type. In the line 5 and 6, the user of `readNumber` will compose refinable function with `compose` method without explicitly defining type of composition mechanism.

4.3 Feature-oriented Programming

$$o' := \text{new } Aspect(\{R : \{f : \{a : m_1.\text{asBefore}(), d : m_2.\text{asAfter}()\}\}\}) \tag{20.1}$$
$$o'.\text{compose}(o); \tag{20.2}$$

Figure 20: Example of Superimposition

Feature-oriented programming(FOP) is programming technique in the point of feature[34]. FOP towards automatically implementing feature based on a composition by from many partial and low-level implementation of functionalities. Function refinement introduced by refinable functions is essential for modifying the behavior of function and class without modification to the original source code in the coarse and fine-grained level of software, thus implementing feature-oriented programming in refinable function is possible. The combination of software is derived by composition is held at various level, coarse-grained in class level or fine-grained in method level[26] which is both supported by refinable function via function itself and aspect composition. The composition procedure, superimposition is a widely used technique for composing software artifact[35] which is a process of composing software artifacts of different components by merging their corresponding substructure. Superimposition in refinable function is implemented by composing aspect to class, in the aspect, a new method, and advice function to call that method is contained. This part will cover implementing superimposition using exportation of refinable function and aspect composition. We also show that example of resolve feature interaction problem using exemplary shortest path algorithm from [34] by resolving feature interaction using the reverse feature, through injecting appropriate coordination code into it.

4.3.1 Feature Composition

Feature composition is the process of merging multiple features into one, the goal of feature composition is basically on deriving new feature from existing feature but also adding coordination code for resolving feature interaction problem. Fea-

```
var loggerAdvice = new Refinable();
loggerAdvice.add(function logging (sym, arg1, arg2) {
    console.log(arg1 + sym +  arg2);
    return {args: [arg1, arg2]};
});

var loggerAspect = new Refinable.utils.aspect({
  add: loggerAdvice.logging.new().bind('+').asBefore(),
  sub: loggerAdvice.logging.new().bind('-').asBefore(),
  mul: loggerAdvice.logging.new().bind('*').asBefore(),
  div: loggerAdvice.logging.new().bind('/').asBefore()
});

var divChecker = new Refinable.utils.aspect({
  div: new Refinable().add(function divChecker (arg1, arg2
      ) {
    if (arg2 === 0) { throw 'divide by zero error'; }
    return {args: [arg1, arg2]};
  }).asBefore(),
  sqrt: new Refinable().add(function sqrtCheck (arg) {
    if (arg < 0) { throw 'sqrt bound error'; }
    return arg;
  }).asBefore()
});3
```

Listing 3: Subaspecting

```
class Calculator {
  add(arg1, arg2) { return arg1 + arg2;
      }
  sub(arg1, arg2) { return arg1 - arg2;
      }
  mul(arg1, arg2) { return arg1 * arg2;
      }
}

class CalcE extends Calculator {
  div(arg1, arg2) { return arg1 / arg2;
      }
}

class CalcEE extends CalcE {
  sqrt(arg1) { return Math.random(arg1);
       }
}

loggerAspect.extends(divChecker);

var Calc = loggerAspect.compose(CalcEE);
var CalcA = new Calc();

CalcA.sum(1,2) //  1+2
CalcA.div(1,0)
// 1/0 'divide by zero error'
CalcA.sqrt(4,2) // 4/2
```

Listing 4: Compose aspect to Classes

ture composition is occurred in the granularity on function(fine-grained) and classes(coarse-grained) depended on their needs. Many feature composition method uses software composition method called Superimposition[35] which is language independent approach to software composition based on structured representation called Feature Structure Tree(FST)[35, 36] which is abstract hierarchical structure of a software component that hides the language-specific detail of a software component. FST can represent many codes and non-code artifacts, such as class structure or documentation, or functional programs[32]. In this section, we discuss feature composition of function and method using superimposition technique. The above-mentioned superimposition technique is designed for hierarchical structure, thus feature composition is both apply a class and its contained refinable functions.

Fig. 19 shows superimposition of refinable function class, the tree C shows a class that contain refinable functions R, which is consist of sub refinable function f anf g. Both f and g contains another subrefinable function a, b and c, d respectivelty. The tree A shows aspect object that is constructed by `Refinable.utils.aspect`. Aspect A is consist of same refinable function R which is to be superimposed with $C.R$. To be supercomposed, the $A.R$ contains proactive refinable function that are ready to be composed in certain way. In this, the method m_1 and m_2 is composed as a before and after advice respectively for $C.R.f.a$ and $C.R.g.d$ respectively. The result of superimposition, the resulting class C' contains refined method R with its mutated internal subfunctions in after and before advice for sub refinable function $C.R.f.a$ and $C.R.g.d$.

4.3.2 Aspect-oriented Programming

The notion of class-level feature composition shares same motivation as function composition in many functional programming, to build a complex logic based on functions of simpler role, however as mentioned in 1, the result of these techniques are concrete, tangled function that can not be altered in further and which is core motivation of refinable function to build optimal language-based modularity technique using OOP.

In this part, we introduce to implementing aspect-oriented programming using refinable functions, by the seamless integration of refinable function allows to use refinable functions as a method of classes, and exportation mechanism enables to make the aspect to be extended via suspecting.

Fig. 3 introduce aspect and subaspect definition. In has no different in declaration because the hierarchy is depended aspect extension method call. Line 7 to 12 in Fig. 3 shows aspect declaration of `loggerAspect` for Calculator class which is comprised of logger method of `add`, `sub`, `mul` and `div`. On the value field, refinable function called `loggerAdvice` is placed with new instances, the argument is bounded in arithmatic symbol to helps logging activity in `loggerAspect`. To supprt logging before the arithematic computation, every 4 logger is exported as a before advice via `.asBefore` method call. As the class in line 1 to 5 of 4 shows, the bounded argument using `bind` is consulted to be non-invasive logging. Another aspect `divChecker` is maded by same procedure as `loggerAspect`. The class `Calculator` is extended twoice to add additional method, in the line 15 of Fig. 4 `loggerAspect` is extended by adding `divChecker` advice. Thus, it internally perform logging and input validation sequentially. The line 17 shows, compsotiom of aspect to the class `CalcEE` via `compose` method.

4.3.3 Resolving Feature Interaction Problem

A feature interaction is a problem of the inadvertent behavior of feature when two more feature is combined, but it cannot be easily deduced from the behaviors associated with the individual features[4]. Most notable example includes problems like call forwarding and call waiting requires coordination of mutually exclusive coordination and, Fire and flood control need coordination on explicit priority. The combine does not always imply physically composing two feature but it also entails sharing mutually exclusive resources like state or network listening like a signal as resources in call forwarding and waiting problem or even beyond the level of individual software like flood and fire control system. As a result, the solution of resolving feature interaction is converged on adding coordination code in both function and method level, its causes are varied. In this part, we show the implementation of feature interaction problem from [34] on maintaining reference chain when composing a singly linked list and doubly linked list.

Fig. 5 shows an example of resolving feature interaction problem using refinable function extension on feature composition of singly and reverse a linked list. Line 1 and 2 represent abstract data type of single and reverse linked list where initiating by creating a reference from next and previous reference. However, the problem occurs when two single and a reverse linked list is combined. When the method `push` on `SinglyLinkedList` is invoked, the reference to previous chain(`this.next`) from current chain(`this.first`) is created; however it does not maintain reference to the backward, thus, additional invocation of `shup` from `RevLinkedList` occur error because the reference is not coordinated. The refinable function creates before advice on `CoordAspect` for maintain backward reference by `this.first.prev` is `n` which is used by line 2. , thus, simultaneous invocation of `push` and `shup` will maintain chain of reference. Refinable functions enable to inject such kinds of coordination code based on function extension and mutation, and support composition by aspect composition.

```
class SinglyLinkedList { push(n) { n.next = this.first;  this.first = n; } }
class RevLinkedList { shup(n) { n.prev = this.last; this.last = n; } }
var CoordAspect = {
  SinglyLinkedList: new Self().add(function pushCoord (n) {
    if  (this.first === null) this.last = n; else this.first.prev = n; return n;
  }).asBefore(),
  RevLinkedList: new Self().add(function RevLinkedList (n) {
       if  (this.last === null) this.last = n; else this.last.next = n; return n;
  }).asBefore()
}

var DoublyLinkedList = Object.assign(SinglyLinkedList.prototype, RevLinkedList.prototype);
CoordAspect.compose(DoublyLinkedList);
```

Listing 5: Resolving Feature interaction problem using fuction extension

4.4 Using Design Patterns for Programmatical Refinement

As the size of the project is larger, the domain knowledge of functionality is distributed, by providing function refinement programmatically, the system can be developed in scale like many libraries of language does, creating functions and pattern for constructing and structuring refinable function can be possible. This work is benefitting from refinability and reflectivity of refinable function since it allows extension and mutation of function. Secondly, the method based refinement has the strong advantage of interoperability[37], providing a promise for refinement by standardized and programmatically invocable method.Since Refinable Function is the object-oriented notion of function, the process of refinement can be separated from the refinable function implementation.

Design pattern towards a reusable solution for the commonly occurred problem in software design[38], if there is the type of extension and mutation to a common part of the arbitrary refinable function, leveraging design pattern for making the solution seems obvious. Conversely, implementing design pattern using refinable function is possible, there is previous research using AspectJ[5] are exist[39] but is not in the scope of this paper, we will discuss it later.

In later parts, we show factory and bridge design pattern for constructing and structuring extension and mutation of refinable function programmatically.

4.4.1 Factory Method Pattern as a Refinable Function Generator

The essence of factory method pattern is to simplify the creation of an object in the aspect of the problem domain. It's like the implicit construction of object depends on the parameter as a variability and other members of the class as commonalities. Using factory pattern to generating refinable function is especially useful for constructing identical structure of refinable function. For example, Fig. 13 shows factory function of crawler which contains commonalities of HTML character escaping and save to database. The function `crawlerFactory` mutates new instance of refinable function `RF` without concerning concrete detail of `RF` except it is general form, a duck type of crawler. It append *HTMLEscape* and *SaveToDB* into tail of function.

```
var RF = new Refinable().add(afterConcern);
var coreConcern = new Refinable().add(HTMLEscape)
    .add(SaveToDB)

function crawlerFactory (Refinable, afterConcern,
     coreConcern) {
  var newRF = Refinable.new();

  if (coreConcern && afterConcern) {
    if (newRF.URLChecker) {
     newRF.URLChecker.after(afterConcern);
     return newRF.add(HTMLEscape).add(SaveToDB);
     } else { throw new TypeError(`URLChecker')}
  } else { throw new TypeError(`Param'); }
}
```

Listing 6: Factory Pattern for Refinement

```
function RequestChanger (rf) {
  //exposes default methods
  this = rf;
  this.prototype = rf.prototype;

  this['HTTPRequest'].delete = (newReq,
      newChecker) => {
    this.HTTPRequest.update(newReq);
    this.URLChecker.update(newChecker);
  }
}

var FileCrawler = new RequestChanger(PlainCrawler
    );
FileCrawler.HTTPReqeust.delete(fileReqm,
     FileURIChecker);
```

Listing 7: Bridge Pattern for User-defined Refinement

4.4.2 Bridge Pattern as a Refinement Structurer

The goal of bridge pattern is to improve the independence of interface and implementation by abstracting object-specific details and focusing on the use of an object on the user. It's like the implicit invocation of method call whereas factory method pattern is the implicit construction of the object.

The independence allows to create high-level refinement method to perform sophisticated refinement operation without making custom operation using `defineMethod`. Using bridge pattern for making imperative and repetitive refinement is into simple and declarative.

The notable usage of bridge pattern is performed refinement of dependent subfunction, for example, when we delete some core or crosscutting concern, that concern might depended or dependent on another concern, in this case by making builder pattern for specific subfunction, the programmer can refine function easily and safely. Conversely, we can augment extension of specific subfunction that requires some feature or operation specific concern. For example, an SQL query must need to be escaped before evaluated. In that sense, the escaping module is a dependency of query evaluator. Fig. 7 shows abstract code example of Dependent subfunction refinement via bridge pattern.

Back to the crawler example, lets consider we build same crawling mechanism for local storage instead of internet, then it is much faster to use file io instead of network io, thus we replace `HTTPRequest` to fileRead, however as dependencies, by replacing `HTTPRequest` will result in malfunctioning of `URLChecker` which is totally dependent its usage by `HTTPRequest`. In this case, we create a bridge object that wrapped single method that implicitly calls two methods of delete and replacing `HTTPRequest` to another request, like file IO and also replace URL checker with another check, like file URL scheme.

5 Design and Implementation

Since Refinable Functions is research on pragmatic language-based modularity, elaboration of its internal design and implementation mechanism is essential. This section introduces design and implementation of JavaScript-based Refinable Function

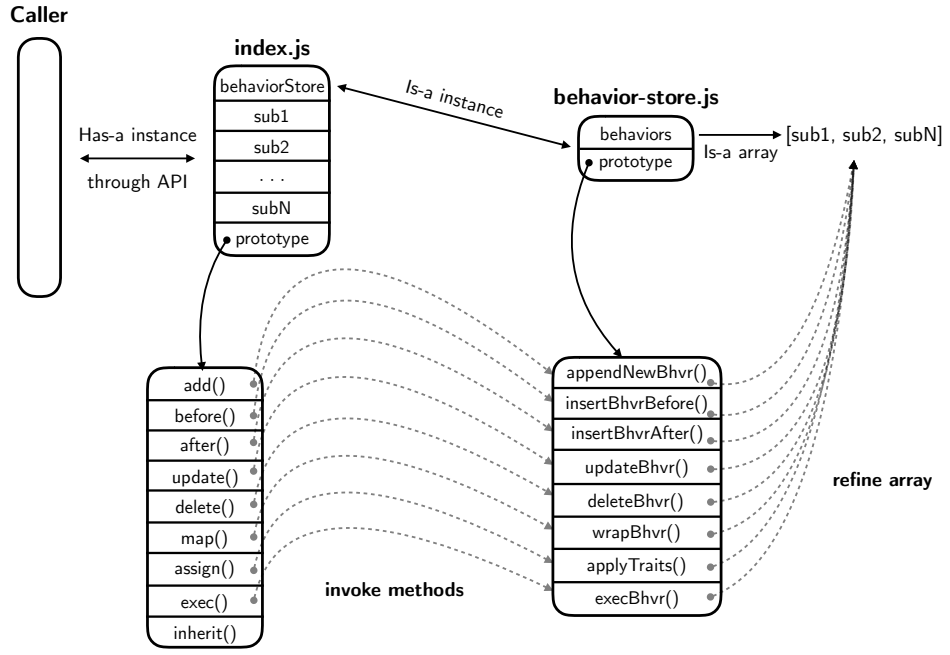


Figure 21: Architecture of RefinableJS, JavaScript-based Refinable Function Implementation

constructor Self[40]. We introduce high-level architecture and its corresponding relationship to features introduced in the previous section. By the strong interests of asynchronous processing in web application community, we also elaborate how RefinableJS process heterogeneous function formation that used in Self and determination logic behind the sequential processing of the synchronous and asynchronous operation.

5.1 The JavaScript Language

Since reliability and performance is major issue of web application, it is very conservative for new technology as part of development and operation process, moreover porting modularity technique mainly for Java to other languages like JavaScript has significant grounding risk sometimes impossible due to environmental constraint[3]. The industry needs universally accepted and thus grounded technologies to implements advanced language-based modularity for software product line. We discover reinterpretation of object-orientation is an optimal medium to enable such possibilities.

5.2 Object-oriented Function Architecture

The object-oriented function is essentially implemented by constructing an object that contains an array of function with a method for performing refinement.

Like object-oriented language are essentially built on top of procedural language, we implemented Self based on JavaScript, which prototype-based[41] OOP. We designed Self into an object-oriented design to implements separation of concern. We built RefinableJS based on two distinguished objects by its role which is constructor and store.

[3]Client-side scripting language like JavaScript or ActionScript the control of runtime environment is deterministic for developer.

```
  var child = new Behavior(this.behaviorStore.getAllBehavior());
  var props = Object.keys(Behavior.prototype).concat(Object.keys(this));
  props.forEach((trait) => {
    if (Behavior.prototype[trait]) { child[trait] = Behavior.prototype[trait]; }
    if (this[trait] && trait !== 'behaviorStore' && trait !== 'catchEvent') { child[trait] = this[trait]; }
  });
  return child;
```

Listing 8: Implementation of Refinable Functions Inheritance

```
this.behaviors.push(newBehaviorObj);

this.behaviors.forEach((behavior, index) => {
  if (behavior.name === name) { this.behaviors[index] = { name: behavior.name, behavior: behavior }; }
});

var newBehavior = _namedFuncChecker.call(this, behavior, name);
  this[newBehavior.name] = { name: newBehavior.name };
  var props = Object.keys(Behavior.prototype);
  props.forEach((trait) => {
    if (Behavior.prototype[trait]) { this[newBehavior.name][trait] = Behavior.prototype[trait]; }
  });
  this[newBehavior.name]['behaviorStore'] = this['behaviorStore'];
  if (behavior instanceof Behavior) {
    var arr = behavior.behaviorStore.behaviors.slice();
    var clonedBehaviorStore = new BehaviorStore(arr);
    for (let subBehavior of Object.keys(behavior)) {
        this[newBehavior.name][subBehavior] = {};
        this[newBehavior.name][subBehavior] = Object.assign({}, behavior[subBehavior]);
      }
  }
```

Listing 9: Implemention of Refinement

The Constructor. The Constructor, positioned in `index.js` in the figure 21, is a typical object-oriented class that can exported callable from user. Constructor provides user-visible API for construct, inherit and refine the behavior of Refinable Function. The constructor returns a behavior instance is a runnable object. The initialized Refinable Function object stores hard copied behavior by its own initialized behavior store instance. The user-visible method of an object is internally connected to the method of behavior store see figure 21. Method on constructor mainly roles to verifying the correctness of user input and passes input to store object where actual refinement occurs to the array of function.

This made key different from most of the function composition technique, those have no transparency and thus no modifiability of already composed function. Refinable Function supports modifiability in the manner of OOP.

The Behavior Store. The Behavior store represented as `behavior-store.js` in a figure 21. Is essentially stores feature associated with an entire life cycle of Refinable Function. The API of behavior store is private only callable by Refinable Function instance, by that, behavior store delegates input verification to Refinable Function instance. The vast of operation in behavior store is simple, adjust or update ordering of array that contains composed function.

```
1  Behavior.prototype.asBefore = function () { return _returnObjGenerator.call(this, 'before'); };
2  Behavior.prototype.asAfter = function () { return _returnObjGenerator.call(this, 'after'); };
3  function _returnObjGenerator (adviceType) {
4    return { type: 'advice', behavior: adviceType,
5      compose: _AdviceComposeGenerator.call(this, adviceType), get: () => { return this.new(); } }
6      ...
7  function _AdviceComposeGenerator (hostAdviceType) {
8    return (targetFunc) => {
9      if (targetFunc.adviceType) { let targetAdviceType = targetFunc.adviceType; targetFunc = behavior;
10     }
11     return (...args) => {
12       let newFunc = this.new();
13       if (!targetAdviceType) {
14         switch (hostAdviceType) {
15           case 'before': newFunc.add(targetFunc); break;
16           case 'after': newFunc.prepend(targetFunc); break;
17           case 'around': newFunc.bind(targetFunc); break;
18           case 'entry': this.behaviorStore.insertBehaviorAfter(this.name, targetFunc); break;
19         }
20       } else {
21         if (hostAdviceType === targetAdviceType) {
22           if (hostAdviceType === 'before') { newFunc.prepend(targetFunc); }
23           if (hostBehavior === 'after') { newFunc.add(targetFunc); }
24         } else { throw 'TypeError aspect type must be the same' }
25       }
26       return newFunc.exec(args)
27       ...
```

Listing 10: Implemention of Exportion

5.3 Implementing Refinable Function Inheritance

Inheritance mechanism of function promotes reuse of function by making a deep copy of refinable function instances. The inheritance mechanism is simple that creates a new refinable function with a copy of `behaviorStore` and methods on parent refinable function. The deep copy is important for ensuring independent extension and mutation of function.

Fig. 8 shows implementation of refinable function inheritance. The line 1 construct new refinable function with argument which retrieving deep copied array of `behaviorStore` from current, a parent refinable function using `getAllBehavior` method call. In order to attach all method definition of function extension and mutation, the second line collects all property of current refinable function instances and prototypal method definition. From the line 3 to 6 perform a copy of method and data structure from instance and method definition. Line 7 returns constructed new refinable function.

5.4 Implementing Refinable Function Refinement

Applying refinement of function is essentially implemented by extension and mutation of `behaviorStore` after validation of name and type compactibility.

Fig. 9 shows implementation of function refinement. The line 1 shows implementation of extends refinement called append, which is exposed as `add` method in Table 9. The mechanism is straightforward that simpy append new behavior `newBehaviorObj` into array `this.behaviors`. Line 3-5 shows implementation of `update` method where the `index` repre-

```
1  async function execBehavior (...input) {
2    for (behavior of this.behaviors) {
3        var hostBehavior = behavior.behavior;
4        if (hostBehavior !== null) {
5          if (typeof hostBehavior === 'function') {
6              input = await hostBehavior.apply(null, input);
7              if (Array.isArray(input.args) && Object.keys(input).length === 1) {
8                input = input.args;
9              } else { input = [input]; }
10         } else if (Array.isArray(hostBehavior)) {
11           var context = this;
12           context['behaviors'] = hostBehavior;
13           try { input = await execBehavior.apply(context, input);
14           } catch (e) { throw e; }
15         }
16       }
17       if (!Array.isArray(input)) { input = [input]; }
18   }
19   return input;
20 }
```

Listing 11: Implemention of Custom Method Definition

sent order of element of target behavior. Codes after line 7 shows common procedure when refinement occur, in order to support refinement of newly added function. The `_namedFuncChecker` checks naming conficts and the iteration of `prop` adds refinement method under the property's name `newBehavior.name`. If new function is type of refinable function, it hardcopies subfunctions and exposes all method related to that refinable function.

5.5 Implementing Refinable Function Exportion

Function exportation exposes function with the predefined method of composition type, which is compatible with another refinable function and aspect. The export mechanism wraps refinable functions into normal JavaScript function that contains composition mechanism defined by export methods.

Fig. 10 shows implementation of exportion mechanism of refinable function. As we described earlier, the exportion allow to use refinable function as a advice function as described in Table 9. The line 1 and 2 in Fig. 10 shows method definition of refinble function exportion as `before` and `after`. It internally invokes `_returnObjGenerator` function with given advice type, the function returns an object with composition mechanism that are compactible for `compose` method via aspect. The `_AdviceComposeGenerator` returns a function that contains composition handling code that performs creation of new refinable function and add or bind given refinable function into specific *aspect* to target refinable function via aspect or function level composition, in line 23-26. The line 29-31 shows composition of aspect via advice type composition rule described in Fig X. After construction finished, it will be invoked via `exec` method.

5.6 Implementing Refinable Function Invocation

Function Invocation of refinable functions performed by sequential execution of array which is similar to the execution of asynchronous construct promise[42] in JavaScript. For example, a `behaviorStore` contains array of functions and refinable

```
1  if (!(targetObj instanceof Aspect)) { throw new Error('aspect type error'); }
2  if (targetObj instanceof Aspect) { targetObj = targetObj.get(); }
3
4  for (key of Object.keys(targetObj)) {
5    if (targetObj[key].type === 'advice' && this.aspectObj[key]) {
6      if (this.aspectObj[key].type === 'advice') {
7        this.aspectObj[key] = targetObj[key].compose(this.aspectObj[key]);
8      } else if (typeof this.aspectObj[key] === 'function' || this.aspectObj[key].exec) {
9        this.aspectObj[key] = targetObj[key].compose(this.aspectObj[key]);
10     } else { this.aspectObj[key] = targetObj[key]; }
11   } else if (!this.aspectObj[key] && targetObj[key]) {
12     if (this.aspectObj.prototype && targetObj[key].type !== 'advice') {
13       this.aspectObj[key] = targetObj[key];
14     } else { this.aspectObj[key] = targetObj[key]; }
15   }
16 }
```

Listing 12: Implementation of Inheritance and Extension of Aspect

```
1  Behavior.prototype.defineMethod = function (methodName, method) {
2    Behavior.prototype[methodName] = method.bind(this);
3    return this;
4  };
```

Listing 13: Implemention of Custom Method Definition

function. The element f_1 is invoked by initial argument, where f_2 is invoked by result of f_1. The function `execBehavior` is `async` function that operates on asynchronous runtime in JavaScript. When it invokes, the function integrates all elements in behavior store. If the element is a normal function, it invokes and saves the argument intermediately as line 5 to 9 shows. When the function is refinable functions, it set subfunction array `this.behaviors` as an execution context of recursive execution of `execBehavior` as in line 10 to 15 showed This recursive execution is executed unless hierarchy of refinable functions is ended with native functions.

5.7 Implementing Custom Refinement Methods

method definition enables custom use can easily adds custom method by call `defineMethod`. The implementation mechanism of `defineMethod` is done by simply adding method. The method binds the context to make passed function is pointing to right `behaviorStore` instance. The Fig. 13 shows implementation of custim method definition in RefinableJS. As line 2 shows it simply adds function `method` to dispatch method definition of refinable function. The `.bind` is JavaScript method that bound context of method to extends and mutates subfunction array that attached to given refinable function.

5.8 Aspect-oriented Programming

5.8.1 Aspect Inheritance and Extension

In this part, we describe an extension of aspect using inheritance. In AOP utility we provided, the aspect can be inheritable using the built-in extension method. The Fig. 12 shows implementation of inheritance and extension of aspect. Line 1

```
1   var props = {};
2   (function retrievePrototype (classObj) {
3     if (typeof classObj === 'function' && classObj.prototype) {
4       for (prop of Object.getOwnPropertyNames(classObj.prototype)) {
5         if (!props[prop]) { props[prop] = classObj.prototype[prop]; }
6         retrievePrototype(Object.getPrototypeOf(classObj));
7       }
8     } else { return; }
9   })(Object.getPrototypeOf(classObj));
10
11  this.extends.call({ aspectObj: props }, this.aspectObj);
12  var newClassObj = classObj;
13
14  Object.assign(newClassObj.prototype, {})
15  Object.assign(newClassObj.prototype, props);
16  return newClassObj;
```

Listing 14: Aspect to Class Composition Codes

validates whether the input is proper aspect object. After line 3, the code shows three composition types of members in the aspect depending on its existence and types. 1) advice to advice composition 2)advice to function composition and 3)null type composition. Listing 12 shows exportation mechanism for advice that used in many aspect-oriented languages. It uses three methods, asBefore, asAfter and asAround.

5.8.2 Aspect Composition for Class

Class refinement is done by traversing prototype chain recursively by retrieving full prototype properties for the method, because of composing Aspect directly to prototype declaration is dangerous for making side effect, RefinableJS inherit class additional time and compose Aspect by overriding method definition. Self uses `Object.getPropertyOf` for traverse prototype chain of parent class, and uses `Object.getOwnPropertyNames` for iterate all method of classes. The fig. 14 shows internal mechanism of prototype traverse.

The fig. 14 shows internal mechanism of Aspect composition. The procedure of composing aspect to classes is same as the procedure of inheriting and composing aspect to aspect. In the composition of aspect, the advice function must have a corresponding method, otherwise, the composition is not made. On the other hand, aspect object can introduce the new member of target class if the only exists in the classes. In order to process this mechanism, refinable function reuses mechanism of `extends` method that is used in aspect to aspect composition. To utilize `extends` method, line 2 to 7, refinable functions flatten class member in order to clarify the existence of individuals. As JavaScript is based on prototype-based OOP, the flattening works are needed and the is done by traversing prototype chain via recursive function `retrievePrototype`. In the recursion, it fills `prop` object for every property in the given class `classObj`, it prioritizes member of classes that are close from class, thus as line 5 shows, it skips member of the class when it is already filled. In line 11, the filled `props` object plays a role of host aspect and the original aspect is working as target aspect. After aspect composition is made, it returns a `newClassObj` with attaching augmented property as in line 14 and 15.

```
1    for (tmember in target) {
2      if (typeof host[tmember] === 'object' && typeof target[tmember] === 'object') {
3        proc(host[tmember], target[tmember])
4      } else {
5        if (!host[tmember]) { throw new TypeError('not subtype'); }
6        if (target[tmember] !== host[tmember]) { throw new Error('member type does not match') }
7      }
8  }
```

Listing 15: Covariance/Contravariance Typechecker for Compound Type

5.9 Implementing Typechecking of Refinement

The typechecking mechanism using subtyping rule discussed in 12 are effectively checks compactibility of sequential invocation on refinable function. The implementation is simple via check containment to return set to the parameter set to another. The Fig. 15 shows implementation of compound type(object) `target` is wheater covariance or contravariance to another compound type `host`. The checking mechanism is simple that to verify `target` is fully contains, subtype to `host`. In the iteration of members in `target`, the line 2 shows recursion when the compound type object contains another object when both members of `target` and `host` is type of `object`. The type `target` is subtype of type `host` when type `target` is only comprised of members of type `host` with same data type which is checked by line 5 and 6.

6 Discussions

Modularity plays important role in modern software engineering, there are a tremendous amount of works in modularity however its practical relevances are not met to the size of research maded[4]. One of the reasons for this is, the implementation of modularity is largely biased to the compiler-based languages like Java and requires compiler extension. Due to the extensive use of script languages, 5 of 10 of new applications for web and tools are written in script languages, this because the cost of development is getting more important than performance including the field of web applications, tool building or machine learning.

However, the advanced modularity mechanism developed previously is cannot be generalized to script languages because new syntax introduced in these techniques is largely depended on compiler extension. It is inevitable that, to modify compiler for support new syntax. This circumstance made implementation constraint of modularity to support variance in runtime, which means that as script language is operated in identical compile and execution environment, the extension of the compiler, should be not allowed for making application to be runnable in general environment.

In order to support modularity for most of the major script programming languages using language-primitives syntax in runtime variance environment, we framed one of core modularity element of computer programming function or method as classes and allows refinement of classes based on built-in and custom method. We introduce language feature refinable function, that contains an array of subfunction and set of refinement method to perform manipulation and the concept of function inheritance and refinement as a commonality reuse and variability localization method which is a common attribute of language-based modularity. We formalized both, a language constructs refinable function and mechanism of function inheritance, refinement into both perspectives of software engineering using diagram and use case study and programming languages using syntax and semantics as the view of refinable function as a language feature.

To shows applicability and impact of refinable function, we present applications of a refinable function to implement equivalent functionality of AOP and FOP. We showed crosscutting modularization for aspect-oriented languages and feature composition and solution for the feature-interaction problem on feature-oriented languages with composition method of use refinable function as a method. In addition, we present the novelty of refinable function by showing the programmatical refinement of function in creational, structural design pattern such as factory and builder, these examples show repetitive refinement can be programmatically possible when refinable functions is compatible types. As a result of experiments, we can prove that object-oriented function refinement has a direct correspondence to language-based modularity technique as aspect-oriented and feature-oriented languages.

Refinable function shows that the principle of reuse and localization of object can be effective as for the object but also for the function which configures the object as a form of a method.

As the implementation of refinable function construct requires common object-oriented functionality, it supports runtime variances and enable to use such modularity technique in script languages;

The concept of function refinement allows to not only augment function's behavior but transparently update and delete in partial, this reflective characteristic of function refinement can be applicable for disciplines like metaprogramming, softwarization and self-adaptive system however further research is needed.

Currently the concept of refinable function is validated through its functionality, however, more empirical validation is required to see the actual effectiveness of refinable function. We will discuss performance issue in the next part. Implementing refinable functions non-prototype-based OOP language(class-based) like python may discuss the generalization of refinable function. Also as for current, the refinable function aims to technique for achieving compiler independent modularity, however as in conceptual/theoretical aspect comparing refinable function to other modularity technique in purely conceptual, theoretical aspect could be helpful The implementation language, JavaScript is really flexible, that it can manipulate a member of classes as a normal property, however, there is language that has less flexible for such issues. More implementation aspect of research is needed to make syntactic support of these languages.

In the discussion section, we elaborate performance and comprehension issue of Refinable Function. We performed microbenchmark to measure performance penalties of using heavy object system as well as determine processor to capture the effectiveness of code optimisation that modern JavaScript engine provides.

6.1 Performance benchmarks

To discover penalties for using Refinable Function, we perform microbenchmark to Refinable Function. Since our concern is processing performance of both synchronous and asynchronous function, we measured simple arithmetic operation and asynchronous IO file operation. In order to measure performance and interpretation of the result, we analyze both time complexity and space complexity in both non-IO and Io operation. Time Complexity means the volume of computation needs to perform in order to achieve the desired result by a program. The detail information for the benchmark is available in Appendix A.

6.2 Time Complexity

Processing object is a heavy task for a computer, even compiler optimization is performed, figure 22 shows its average is higher than native function approximately 3 times. For the IO operation, followed by figure 23 Refinable Function is still slower but the execution time is much more reliable then non-IO function. We capture the originating cause is that time needed for IO operation is prevailed time variability by processing object. We can derive Refinable Function is much more reliable for IO operation.

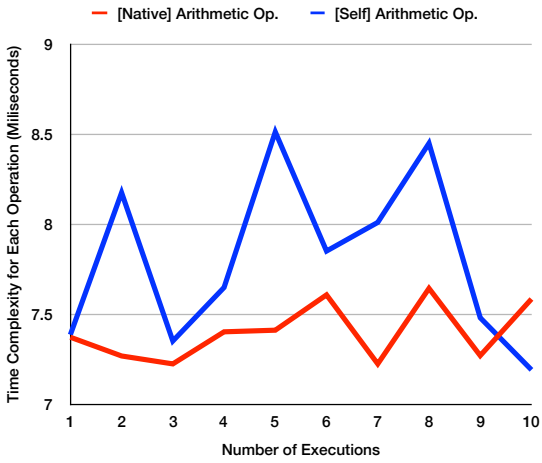


Figure 22: Time Complexity for Non-IO Operation

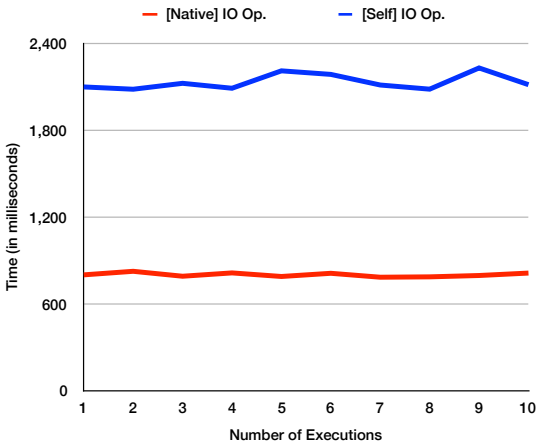


Figure 23: Time Complexity for IO Operation

7 Related Works

Refinable function essentially classifiable to a technique for function composition. Function composition is a rooted technique for many object-oriented programming and functional programming because the function is generally treated as a minimal unit of software design. For the application of function composition to implement a feature, there are sufficient previous works exist with different granularity aims to[26] from fine-grained solution to coarse-grained solution. In this section, we discuss position of Refinable Function compare to related techniques.

7.1 Language-based Approach

Partial application is multi-staged function reduction generate more sophisticated functionality to promote reusability of common parts of the function. Currying is a special form of partial application. The partial application takes a function with multiple parameters and returns function with fewer parameters, currying returns function with one parameter. The key difference between refinable function is upgradeability of function, once it is composed the generated function could not be restructured in order to localize variability further.

Aspect-oriented Programming(AOP) towards separation of concern by localizing cross-cutting concern that is scattered and tangled across program [1]. At the implementation perspective, a popular implementation like AspectJ[5] supports com-

position of crosscutting concerns in language driven by the weaver. While it AOP benefits performance by static program compilation but AOP cannot be applied to the general environment such as web client application whose runtime is heterogeneous and not owned by the developer. Despite core motivation of Refinable Function is universal language-based modularity, however, research from AOP community is essential to advance composition mechanism of Refinable Function such as heterogeneous cross-cutting concern[43]

Feature-oriented Programming(FOP) is composition-based approach for building software product line on the notion of feature[4]. Modularity positioned the essential issue of FOP, because the degree of modularity is important to compose feature. Refinable Function is usable for implementing the method on FOP in smaller granularity. For example, Delta-oriented Programming(DOP)[7] is FOP technique that asymmetrically applies delta module into the base class module to refine its members. The key concept of DOP is shrinking refinement, applying delta module can serve deletion of the member in the object. DOP's shrinking refinement is equivalently for `.delete` or `.update` method in Refinable Function. However, DOP and Refinable Function also shares the same limitation to program understandability problem by program fragment is spread across the project. DOP alleviate this problem by applying type checking[44]. Refinable Function also uses similar Interfaces or possibly DCI architecture[45] to fragmentation of composition caused by object-oriented collaboration. The key difference between DOP and Refinable Function is granularity, DOP handles class-level, however, Refinable handles function-level.

Context-oriented Programming [46] enable program to perform context-dependent behavior to program. Variability of behavior is used for different requirement come from context. Context like operating system status or sensory data like location enables the context-oriented program to do polymorphic behavior[4]. Refinable Function is usable for implementing such variability to member method function, thus the notion of context-orientation has higher granularity then Refinable Function.

7.2 Architecture-based Approach

GraphQL[47] is declarative query language for information retrieval from datastore. GraphQL is been popular for diminishing cost of business logic, Business logic is traditionally considered fundamental component in the tier in three-tier architecture and replacing business logic is a significant impact on web application development. Although GraphQL has some practical relevance for a product that needs to develop and evolve rapidly or has a scale and dynamic demands on API. However, GraphQL for practical usage still far means from the performance. The visibility of logic is not presented to the program, thus the optimisation of database query cannot be implemented. Also, automation of business logic made difficult to add another tier, such as cache layer into business logic. As result, many modularity techniques, including Refinable Function is still valid.

Design Pattern refers to a general reusable solution to a commonly occurring problem in software design[38]. Decorator pattern is structural solution object-oriented software to localize commonalities in the single entry point. Decorator pattern is an architectural pattern so, it has larger granularity then Refinable Function. Refinable Function can implements decorator pattern, notably using `.before` and `.after` method. Refinable Function is white boxed function dynamic modification of decorator is contrast parts of the traditional function-based decorator.

Data, context and interaction architecture(DCI) is object-oriented design approach to prevent object-oriented collaboration that cause unpredictable system behavior in runtime[45]. The goal of DCI is to explicitly control collaboration between object. Communication in DCI is indirectly passed to context object which maps object and its role explicitly. DCI can be used

for an architectural pattern for Refinable Function to an explicit structuring of collaboration between function collaboration to ensure the validity of composition as well as readability of source code. For example, in the case of API application, DCI architecture prevalidate composition of roles such as authentication or database operation.

7.3 Tool-based Approach

Tool-driven approach to modularity allows a clear separation between crosscutting and feature implementation but it requires the programmer to handle source code related to modularity in the different context. Beside of language-based modularity, there are several approaches to implement modularity within the tools like IDE[48, 49, 50, 51, 52].

8 Conclusion

In this paper, we present a language feature, refinable functions an object-oriented implementation of advanced language-based modularity technique. At the essence, modularity is means of reuse and localization of code, refinable functions implement such property of modularity by harness object not only for the combination of data and method to model the computation but also the combination of subfunctions as a data and method as a refinement. OOP is one of the highest commodity technology spread in the programming languages and it is extensively adapted in both academia and industry over a 2 decades, by leveraging well-established technology, most of the modern programming languages can implement refinable function including script languages with identical compile and runtime environment which is hard to apply the traditional modularity implementation that requires to extends the compiler functionality. We validated the impact of refinable function by showing its correspondence to aspect-oriented and feature-oriented languages by implementing a motivating example of each techniques cross-cutting modularization, feature composition and solution for feature interaction problem. We show the novelty of refinable function as a programming language aspect by providing its formal model as a language feature and programmatical refinement using design pattern, we also give a notion of seeing the independent evolution of method and classes in software engineering perspective as the modularity is focusing on the method on the classes. We shared the practical problem of a refinable function including performance and generalization mechanism for other programming languages which requires further investigation. The result of this paper shows that refinable function can resolve modularity implementation issues for script languages as well as usable as a novel language feature essences on function refinement which can be done with and without object-oriented way shown in this example, thus more programming language and software engineering research investigation is needed. From the result, we show bringing refinement to function in object-oriented way can enable advanced and practical language-based modularity mechanism for script languages.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming. ECOOP'97 — Object-Oriented Programming: 11th European Conference*. Springer Berlin Heidelberg, 1997, pages 220–242. DOI: 10.1007/BFb0053381.

[2] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. *ACM Sigplan Notices*, volume 43 of number 10, pages 543–562. ACM, 2008.

[3] E. K. Piveta, A. Moreira, M. S. Pimenta, J. Araújo, P. Guerreiro, and R. T. Price. An empirical study of aspect-oriented metrics. *Science of Computer Programming*, 78(1):117–144, 2012.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented software product lines: concepts and implementation*. Springer Science & Business Media, 2013.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[7] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. *International Conference on Software Product Lines*, pages 77–91. Springer, 2010.

[8] Expressjs. fast, unopinionated, minimalist web framework for node.js. URL: `https://expressjs.com/` (visited on 11/18/2017).

[9] Django. python web framework that encourages rapid development and clean, pragmatic design. URL: `https://www.djangoproject.com/` (visited on 11/18/2017).

[10] Rudy on rails. web-application framework for database-backed model-view-controller application. URL: `http://rubyonrails.org/` (visited on 11/18/2017).

[11] Javascripe programming language. a lightweight interpreted or jit-compiled programming language with first-class functions. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript` (visited on 11/18/2017).

[12] Python programming language. a programming language that lets you work more quickly and integrate your systems more effectively. URL: `https://www.python.org/` (visited on 11/18/2017).

[13] Ruby programming language. a dynamic, open source programming language with a focus on simplicity and productivity. URL: `https://mil-tokyo.github.io/webdnn/`.

[14] Tensorflow. computation using data flow graphs for scalable machine learning. URL: `http://tensorflow.org` (visited on 11/18/2017).

[15] Pytorch. tensors and dynamic neural networks in python with strong gpu acceleration. URL: `http://pytorch.org` (visited on 11/18/2017).

[16] Convnetjs. train convolutional neural networks (or ordinary ones) in your browser. URL: `http://convnetjs.com` (visited on 11/18/2017).

[17] Webdnn. fastest dnn execution framework on web browser. URL: `https://mil-tokyo.github.io/webdnn/` (visited on 11/18/2017).

[18] W. Cook and J. Palsberg. *A denotational semantics of inheritance and its correctness*, volume 24 of number 10. ACM, 1989.

[19] D. Brown and W. R. Cook. *Monadic memoization mixins*. Computer Science Department, University of Texas at Austin, 2007.

[20] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[21] A. P. Black, K. B. Bruce, and J. Noble. The essence of inheritance. *arXiv preprint arXiv:1601.02059*, 2016.

[22] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: composable units of behaviour. *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[23] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. *European conference on Object-oriented programming*, pages 126–150. Springer, 2010.

[24] M. Madsen, O. Lhoták, and F. Tip. A model for reasoning about javascript promises. *Proc. ACM Program. Lang.*, 1(OOPSLA):86:1–86:24, Oct. 2017. ISSN: 2475-1421. DOI: 10.1145/3133910. URL: http://doi.acm.org/10.1145/3133910.

[25] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[26] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on Software Engineering*, pages 311–320. IEEE, 2008.

[27] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[28] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM, 2000.

[29] W. Harrison, H. Ossher, P. Tarr, and W. Harrison. Asymmetrically vs. symmetrically organized paradigms for software composition. *IBM Rsch. Rpt. RC22685 (W0212-147)*, 2002.

[30] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[31] W. Harrison and H. Ossher. *Subject-oriented programming: a critique of pure objects*, volume 28 of number 10. ACM, 1993.

[32] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (de) composition in functional programming. *Software Composition*, pages 9–26. Springer, 2009.

[33] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. *ACM SIGPLAN Notices*, volume 40 of number 9, pages 320–330. ACM, 2005.

[34] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[35] S. Apel and C. Lengauer. Superimposition: a language-independent approach to software composition. *International Conference on Software Composition*, pages 20–35. Springer, 2008.

[36] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: language-independent, automated software composition. *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[37] J. Aldrich. The power of interoperability: why objects are inevitable. *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 101–116. ACM, 2013.

[38] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: abstraction and reuse of object-oriented design. *ECOOP'93 - Object-Oriented Programming, 7th European Conference*, pages 406–431, 1993. DOI: 10.1007/3-540-47910-4_21.

[39] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*. Pages 161–173, 2002. DOI: 10.1145/582419.582436. URL: http://doi.acm.org/10.1145/582419.582436.

[40] H. Kim. Self - Refinable Function Constructor for Better Modularity Webpage, retrieved on July 2017. URL: `https://hiun.org/self`.

[41] D. Ungar and R. B. Smith. *Self: The power of simplicity*, volume 22 of number 12. ACM, 1987.

[42] Javascript promise. mozilla developer network. URL: `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global%5C_Objects/Promise` (visited on 07/01/2017).

[43] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical report, Technical report, Computing Department, Lancaster University, 2004.

[44] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 43–56. ACM, 2011.

[45] T. Reenskaug and J. O. Coplien. The DCI architecture: a new vision of object-oriented programming. *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

[46] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object technology*, 7(3), 2008.

[47] Graphql : a query language for your api - working draft. URL: `http://facebook.github.io/graphql` (visited on 07/01/2017).

[48] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto. Do we really need to extend syntax for advanced modularity? *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 95–106. ACM, 2012.

[49] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[50] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180. IEEE, 2004.

[51] T. Hon and G. Kiczales. Fluid aop join point models. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 712–713. ACM, 2006.

[52] C. Kästner, S. Trujillo, and S. Apel. Visualizing software product line variabilities in source code. *SPLC (2)*, pages 303–312, 2008.

A Supplementary Information for Microbenchmark

A.1 Benchmark Environment

- Computer Type : MacBook Pro (Retina, 13-inch, Late 2012)
- CPU : Dual-core Intel i5 2.5GHz with 3MB shared L3 cache
- Memory : 8GB of 1600MHz DDR3L onboard memory
- Storage : 128GB Solid State Drive
- Runtime Environment : Node.js v7.7.2 (Based on V8 JavaScript Engine v5.5.372.41)

A.2 Time Complexity Benchmark Code

A.2.1 Normal Benchmark

To measure space complexity, in normal benchmark for non-io operation, we performed addition operation for 100 iteration with 100 phases.

```
var addOne = (n) => {return n + 1};

function proc (n) {
  if (n !== 100) {
    console.log(process.memoryUsage().heapUsed);
    var a = 0;
    for (var i = 0; i < 100; i++) {
      a = addOne(a);
    }
    console.log(process.memoryUsage().heapUsed);
    return proc(n + 1)
  } else {
    return;
  }
}
```

Listing 16: Native Function for Non-IO operation

```
var addOne = new Behavior().add((n) => {return n + 1}, 'a');

var a = 0;
console.time('test');
for (var i = 0; i < 100; i++) {
  addOne.exec(a).then(n => {a = n;})
}
console.timeEnd('test');
```

Listing 17: Refinable Function for Non-IO operation

For IO Operation, we measure performance by reading 1KB size file in 10K time.

```
var fs = require('fs');

function read (n) {
  return fs.readFile('./file.txt', (e) => {
    if (!e && n !== 9999) {
      read(n+1);
    } else{
      return console.timeEnd('test');
    }
  });
}

console.time('test');
read(0);
```

Listing 18: Native Function for IO operation

```
var ReadFileRefinable = new Behavior()
ReadFileRefinable.add(readFilePromisified);

```

```
4  function readA (n) {
5
6    ReadFileRefinable.exec('./file.txt').then(k => {
7      if (n !== 9999) {
8        readA(n+1);
9      } else {
10       return console.timeEnd('time');
11     }
12   });
13
14 }
15
16 console.time('time');
17 readA(0);
```

Listing 19: Refinable Function for IO operation