Refinable Function

An Object-oriented Approach to Procedure Modularity

Hiun Kim
B.S. Student
Department of Computer Science and Engineering
Sejong University
Seoul, South Korea
hiun@divtag.sejong.edu

Abstract

Modularity is the fundamental aspect of modern software engineering, however many advanced modularity techniques requires prospective technologies as part of development and operation process. In this paper, we present *Refinable Function*, an object-oriented approach to advanced language-based, symmetric modularity technique for the procedure. We conceptually compare Refinable Function to existing technique to substantiate benefits of modularity can be implemented in on well-established object-oriented language without compiler support. We introduce concepts of inheritance, encapsulation, and polymorphism of function for bringing object-orientation to procedure modularity and describe the design and implementation of Refinable Function in JavaScript to validate our approach to practical web application development. We introduce the practical aspect of Refinable Function implementation by discussing concerns of applying modularity on asynchronous processing. We tested and implemented Refinable Function to substantiate its relevance to web application development and product line implementation.

CCS Concepts • **Software and its engineering → Abstraction, modeling and modularity**; *Software development techniques*; Language features; Design patterns;

Keywords Modularity, Design, Object-orientation, Web Application

ACM Reference format:
Hiun Kim. 2017. Refinable Function. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 17 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modularity is the fundamental aspect of modern software engineering to improve comprehension and maintain its evolutionary development. Modularity is an important issue for software product line, which is a technique for deriving problem-specific software artifact by composing more smaller rolled functionality. Modularity technique used to implement variability has two types, via language or via tooling. Despite such benefit of modularity, its practical relevance in general application area is not sufficient. In this paper, we present Refinable Function as a pragmatic fine-grained modularity technique based on object-oriented programming. Refinable Function uses well-established and implemented the principle of object-oriented programming, encapsulation, inheritance, and polymorphism to the procedure to resolve the problem of practical adaptation of advanced language-based modularity in software engineering community especially. We study Refinable Function in the context of web application, by its event-driven operating traits [1]and dynamics of behavior variability[2]

1.1 Traditional Approach

Modularity for Software Product Line[20] is often implemented as two distinguished parts which are language and tool-based approach[2]. Start from a function, a language-based approach has advanced to fulfill more dedicated modularity solution that is generally applicable to programming, but such approach is often struggling from practical adaptation and relevance. For example, Aspect-oriented Programming [15], a theory and technique for modularising scattered, commonalities in software is extensively adopted in Java community, is not well spread for other language because it is tough to implement by engineer who has to know compiler extension, even compiler is implemented, such experimental technology has clear difficulty for practical adaptation. For language does support flexible language manipulation like user-defined operator is often not considered for application programming in the industry such as Haskell for leakage of library ecosystem or swift for the same reason.

Of course, the equivalent functionality can be implemented via library but it can not overcome the size of expressiveness that language can offer. The ideal solution would be a metaprogramming technique that is widely used in most

[1]Event-related, Network-related application involves lot of variabilities, in terms of feature and security modularisation for such business logic is required.
[2]Variability including API generation or Service composition in service-oriented architecture.

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

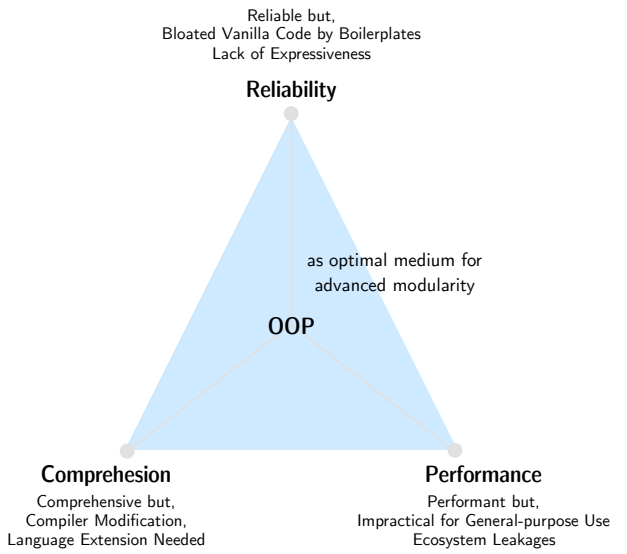

Figure 1. Constraints of Advanced Modularity Techniques

modern language which does not require compiler or other system software support but as primitive syntax. One technique that met this requirement is object-oriented programming which is a well-established technique that is extensively used for the practical environment over few decades. In the following section, we describe how we capture and met such constraint for practical relevance in object-oriented programming by introducing some of the essences of object-orientation and its correspondence to application for procedure modularity.

As figure 1 shows, existing modularity technique has strength in one of three criteria. Modularity technique implemented via vanilla code is reliable, thus it is not comprehensive because constraint of base language specification is inherent to them. Compiler or language extension can provide dedicated syntax and evaluation strategy for modularity, a well-known implementation, AspectJ[14] from Aspect-oriented Programming[15] and DeltaJ from Delta-oriented Programming[22] uses Java-based language extension to support modularity in dedicated syntax improves comprehension and composition. However, it is not always best to answer for languages or environment that does not support such modification, client-side scripting language JavaScript[3] or ActionScript[4] is good example because setting its operational environment is depending on user. Performance can be achieved safely when language itself is supporting user-defined custom operator. Languages like Haskell[5] supports custom infix operator[10] and Apple's Swift[6] supports custom unary and infix operator[11] however those language is relatively new, so the system written in such languages are not widely spread. Even old languages like Haskell has an insufficient ecosystem for practical application development. We capture three strength of modern advanced language-based modularity and its pros and cons. In this paper, we propose using of object-orientation for language-based, procedure modularity mechanism for optimal meduim for maintaining such strength and promotes practical adaptation.

1.2 Object-oriented Approach

The common sense of programming for Imperative programming is algorithm plus data, logic programming is rule plus fact. The motivation for object-oriented programming is *programming as modeling*[19]. The goal of modeling initially implemented for simulation, which motivation creates SIMULA the first object-oriented language[3]. The philosophy of object-orientation is implemented with the concept of class which models the thing in the real world and object with is realisation of class that takes message passing to invoke polymorphic behavior via dynamic dispatch, and update behavior by mechanisms like multiple dispatches.
The question of this paper is,*is it pragmatic to applying object-orientation to procedure modularity*? How can the current object-orientation concept be reconceptualised to function? and what criteria needed to models a function. Since the function is essentially a directional string of instruction which a lot different for the object, which is the mixture of function and data. This paper present Refinable Function to answer such questions.

1.3 Contributions of This Paper

In this paper, we contribute following,

- Introduction of Refinable Function as a new application of object-orientation for advanced language-based modularity technique, and validates our approach with code example
- Introduction to concept of function inheritance, function encapsulation and function polymorphism and its ability to function composition
- Introduction of design and implementation of Self, a JavaScript-based Refinable implementation with its architecture and internal operating mechanism
- Address of practical issues on code comprehension in Self in terms of modularity issue for asynchronous processing in the web application. We present automatic promisification and infix function composition as a prospective solution

[3]JavaScript - Mozilla Developer Network.
https://developer.mozilla.org/en-US/docs/Web/JavaScript
[4]Adobe ActionScript http://www.adobe.com/devnet/actionscript.html
[5]Haskell. An advanced, purely functional programming language.
https://www.haskell.org/
[6]Swift Programming Language. https://swift.org/

- Analysis of penalties of using the object-oriented system as general function primitives based on Self, by interpreting time and space complexity in terms of compiler and runtime-specific perspective.

We describe modularity technique that has novelty in practical relevance as well as exposes potential of application of object-oriented programming to non-object, a function.

2 Refinable Function

In this section, we introduce the concept of Refinable Function. We elaborating how Refinable Function applies the concept of object-orientation to implements its feature for modularity. The concepts part introduces how we apply fundamental features of OOP mention in section 1.2 into the function. We elaborate the three pillar of OOP, inheritance, encapsulation and polymorphism and how their traits are suitable for function composition to implements language-based modularity.

In the feature part, by using concept we propose concrete feature to implements refinability to function. We propose method-based composition such as .add .update and .delete, and traits[24] based compositon for the sake of large scale, stuctured refinement. We introduce more low-level solution for function manipulation including .map method for wrapping function natively and .defineMethod user-defined method for custom refinement.

2.1 Concepts

Inheritance as a Function Realisation. Inheritance is the concept of human to the reasoning for program construction by catching concrete concept to derive abstract concept[4]. Just like object, we capture the principle of inheritance can be adaptable to a function that can gradually localise variability from the requirement by making subtyped function, just like inheritance of class can be a subtyped. The key difference is that, since the function is stateless and object is stateful entity class and instance relationship does not exist as for inheritance of function.

Encapsulation as a Function Refinement. Encapsulation in object-oriented programming allows indirect and thereby enforced access to data in the object. Encapsulation is a form of information hiding which benefits security, abstraction, and interoperabilty[1]. Applying encapsulation to function is essentially simple, our implementation of Refinable Function contains an array of function and invoked sequentially to get a result, as consequences making a modification to encapsulated, function-contained array's element is a refinement of function. Refinable Function uses various method to allow different kinds of refinement. simplest example works like .before advice in aspect-oriented programming. The security means for securing program's correctness by making composition safe. Essentially such safety is achieved by encapsulation for validating new input function which is achieved by encapsulation. Abstraction is achieved by hiding refinement code in a method which also promotes security as well. Lastly, interoperability is implemented by using common unified standard or user-defined method for various function composition.

Polymorphism as a Safe Composition. Polymorphism add flexibility of applying operation with a different type of object. With reflection supported environment, composed function works polymorphically to handling variability of newly composed function with the form of ad-hoc polymorphism, function overloading applied to function. Infix polymorphism can be applicable to compose function in addition operator. Subtype polymorphism is embedded into function inheritance.

2.2 Features

In this section derive concrete feature of Refinable Function by using concept introduced in the previous section. Both conceptual and implementational part of Refinable Function is on top of object-oriented concept, the initial process must be construction, later the constructed function will hard copy by inheritance. As a step of refinement, inherited Refinable Function object localise variability by message passing through its methods. Ad-hoc polymorphism can be applied as part of refinement to handle variability come from parameters and composition based on infix or unary operator is used to improving expressiveness.[7].

2.3 Inheritance of Function

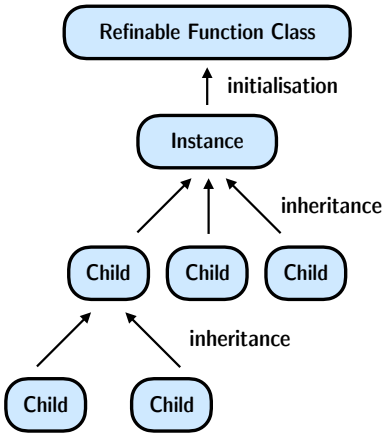

Figure 2. Diagram for Inheritance of Function

Construction of Refinable Function is done by is initiating constructor, and after variability is applied to Refinable Function instance, it can be inherited to create hardcopy instance

[7]Languages like Go support reflection to inspect internal of function, however, language like JavaScript function is black-boxed https://golang.org/pkg/reflect/

of it. Unlike inheritance is created for class and object relationship, Refinable Function has no concept of class, the object is directly inheritable since the function is stateless. Inheritance works as new boilerplate for function realisation by the method. The figure 2 shows derivation process of Refinable Function in hierarchical manner. Original constructor instantiates Refinable Function and it is re-inherited to implement more sophisticated behavior. This approach is implemented in multi-level inheritance.

2.4 Encapsulation of Function

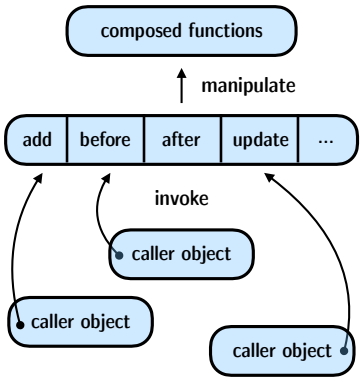

Figure 3. Diagram for Encapsulation of Function

Encapsulation provides standard and interoperable ways to change the state of the object. The Same principle can be applied to Refinable Function since it consists of a set of composed function, refinement means for reordering and modifying an element in an array of functions. Encapsulation allows such adjustment and modification in a safe manner and standardised way. The interoperability also allows dynamic and automated generation of Refinable Function by using standardized refinement API from caller object. The figure 3 shows encapsulation diagram fo function. the composed function is function array and caller object indirectly manipulate an array by calling standard or custom method. The detail of method available in table 1.

Table 1. Standard Method in Refinable Function

Method Name	Description
Behavior#add	Adds new sub-behavior.
Behavior#sub#before	Add new behavior before specified behavior
Behavior#sub#after	Add new behavior after specified behavior
Behavior#sub#update	Update specified behavior with given behavior
Behavior#sub#delete	Delete specified behavior, this function takes no argument
Behavior#sub#map	Wrap specified behavior with given function-returning function
Behavior#exec	Delete specified behavior, this function takes no argument
Behavior#catch	Catch errors while invoking promise
Behavior#new	Create inherited behavior. The inherited behavior is deep clone of parant behavior and does not reference or link any property
Behavior#assign Behavior#sub#assign	Apply traits to behavior Traits means set of object-independent, composable behavior Traits override existing sub-behavior with given sub-behavior by name
Behavior#interface	Apply interface to behavior to substantiate structure of behavior by naming
Behavior #defineMethod	Define new method for perform refinement of sub-behavior contained array directly

[a] The detail parameter and code example can be founded in appendix A

2.5 Polymorphic Refinement of Function

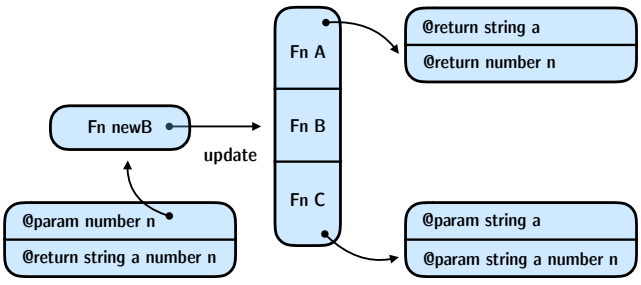

Figure 4. Multi-level Inheritance on Behavior of Database Module

```
1  var Read = new Behavior().add(data => { return !data; }, 'checkEmpty');
2  var ReadNumber = Read.new().add(data => { return typeof data === number; }, 'numTypeChecker');
3  var ReadNumberUserId = ReadNumber.new().add(data => { return id > 0 }, 'idChecker');
4  var Result = ReadNumberUserId.add(data => { DB.Users.read() }, 'dbProc');
```

Listing 1. Inheritance of Function

```
1  var ReadNumberPostId = ReadNumberUserId.new().idChecker
2      .update(data => {data[0] === 'p' && parseInt(data.substr(1))})
3      .after(data => { var result = Cache.read(data);
4                  if (result) {return Promise.resolve(reuslt);} else {return data} } )
```

Listing 2. Refinement of Function

One of the main concern of using function composition is interoperability of parameter structure. Unlike currying, a primitive language-based solution, Refinable Function can be partially updateable and dynamically composed by the system so parametric polymorphism and ad-hoc polymorphism can be applicable for filling such gap. Refinable Function introduce .map a method for manual wrapping of function. Since our implementation is written in vanilla JavaScript, type checking is not supported in language basis, it is possible to implement such feature in TypeScript[8] or Flow[9] but snow does not provide systemic support.

3 Blog as a Refinable Web Application

In this section, we introduce concrete example on Refinable Function application by implementing the some of the base functionality of blog with Refinable Function. Blog is a type of content management system that satisfies crucial aspect of three-tier architecture[6] which is dominant architecture for a web application. Since CRUD operation and routing is the main concern of blog and three-tier web application, a technique that gives utility for a blog is general enough to apply more many web application. We introduce some core functionality of web application implemented with Refinable Function elaborate with the concept with concrete code example.

3.1 Requirements with Domain Analysis

We designed blog to supports CRUD operation for 4 entity in the schema as in Appendix C. Users, Posts, Comments, Messages and Images. We elaborate the construction of behavior to implements it requirements by using the feature we previously described.

3.2 Function Inheritance for Commonalities

Inheritance is used to for localising commonalities to parental function. As implementers perspective typical type of operation to think about a blog is *reading*. We can classify read as operation type. By using Refinable Function programmer can gradually inject commonality and variability at corresponding inheritance hierarchy of function. The first line of code in listing 5 shows .add for .checkEmpty to implements operation type commonalities. The second line of code shows ReadNumber inheritance from Read to implements type-specific commonalities, in this a commonality is a number. Depending on the domain, however, the last level inheritance is held for object level which represented in the schema, in this case, domain specific type check for userId is performed in the third line of code. We perform inheritance from abstract functionality to more specific functionality for localising each behavior's common roles. Now to reuse such function, next section describes refinement for implementing variabilities by using inherited, hardcopied Refinable Function instance.

3.3 Function Refinement for Variabilities

Refinement is a process of making the realisation of function by manipulating its behavior and ordering. In the inheritance section, we present three type of refinement technique in Refinable Function. In the previous example, the last refinement is about the realisation of object-specific operation. A validator detects the number is greater than zero. Let's consider we want to check more complicated rule such as the character starting from P instead of a just number. To implement this feature we made refinement uses .after method. Listing 2 shows refinement for transform ReadNumberUserId to ReadNumberPostId by updating idChecker to sophisticated function.

Refinement in Refinable Function is essentially adjusting the order of functions. The method .defineMethod allows user to define a custom method for actuating refinement. In this case, to handle the error, let's assume simple filter and

[8]Typed JavaScript for Scalable Development, Superset of JavaScrpt with Transpiling. https://www.typescriptlang.org/
[9]Static typing for JavaScript https://github.com/facebook/flow

```
ReadNumberUserIdExec.interleave((arg) => {
        if (!arg) { return Promise.reject(new Error('user input type error')) } else { return arg }
})

Read.defineMethod('interleave', (Fn) => {
  this.BehaviorStore.Behaviors.reduce((accu, behavior, index) => {
    if (index % 2 === 0) {
      this.BehaviorStore.Behaviors.splice(index, 0, Fn);
    }
  });
});
```

Listing 3. User-defined Refinement of Function

```
LoadArticle.interface(['payloadCheck', 'authCheck']);

//traits example
//using traits
var publicApiTraits = {authCheck: null};
var publicLoadArticle = LoadArticle.new().assign(publicApiTraits);
```

Listing 4. Interface and Traits for Function

```
singleFn.map((singleFn) => {
  return inputs.forEach(singleInput => {
    singleFn(singleInput);
  });
});
```

Listing 5. Naive Example of Multi-argument Application

pipe apps that return false by the checking of data's truth, In order to insert error occurring function. The listing 3 shows interleave message with is constrained version of a function that obeys negative to error, but powerful metaprogramming also occurs leaks of safety.

Another type of function refinement technique we present is traits. Refinement for apply traits accompanying with interface[10]. Trait is a composable unit of behavior to object[24]. In object-oriented programming, traits are used to compose same behavior to object with different classes. We apply traits to similar fashion for enforcing favor of function by name. For example, traits for implementing public web API may containing DDoS protection and may do not contain authentication mechanism. Traits implicitly replace such domain-specific refinement. Function or null can be used as value traits object, the name of value used for replacement target function. The traits object represent absence or existence of behavior.Traits iterate array to assign matched function name to properties value or remove it. The code in listing 4 publicApiTraits assign null authCheck defined by interface. Traits deals with a bulk assignment, interface plays exact opposite role to them. Interfaces enforce order and name of a function that needs to must implements.

3.4 Function Polymorphism for Safety

Explicit ad-hoc polymorphism is partially implemented because our target language JavaScript does not support Reflection feature to function. With .map method and use of functional programming technique currying, explicit handling argument's variability are possible.

4 Design and Implementation

This section introduces design and implementation of JavaScript-based Refinable Function constructor Self[16]. 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 Self process heterogeneous function formation that used in Self and determination logic

[10]Interface is not implemented in the library yet

```
var checkObsticle = new Behavior().add(checkStaticObsticle).add(checkDynamicObsticle)
var moveRight = new Behavior().add(checkBattery).add(forwardInDegree.bind('90'))
moveForward('right')

var moveForword = new Behavior().add(checkObsticle).add(moveRight);

//add new function with deep method containment
moveForward.checkObsticle.checkStaticObsticle.after(CheckEnvironmentalObsticle);
moveForward.moveRight.checkBattery.after(checkFuel);
```

Listing 6. Static Deep Method Traverse on Robot Programming

```
var decoratorBuilder = (preFn, postFn) => {
  return (coreFn) => {
    return new Behavior().add(preFn).add(coreFn).add(postFn);
  }
};

var HTTPDecorator = decoratorBuilder(httpScaffolder, errorHandling);
var WebsocketDecorator = decoratorBuilder(wsScaffolder, errorHandling);
HTTPDecorator.postFn.update(newPostFn);
```

Listing 7. Dynamic Deep Method Traverse for Design Pattern Building

behind the sequential processing of the synchronous and asynchronous operation.

4.1 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[11]. 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 for enable such possibilities.

4.2 High-level Architecture

Like object-oriented language are essentially built on top of procedural language, we implemented Self based on JavaScript, which prototype-based[26] OOP. We designed Self into an object-oriented design to implements separation of concern. We built Self based on two distinguished objects by its role which is constructor and store.

The Constructor. The Constructor, positioned in index.js in the figure 5, 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 initialised Refinable Function object stores hard copied behavior by its own initialised behavior store instance. The user-visible method of an object is internally connected to the method of behavior store see figure 5. 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. The Refinable Function exposes the name of its composed function with a method, if composed function is Refinable Function instead of the native function, it also contains method. This containment relationship is recursive so the method hierarchy of Refinable Function is enlarged see the example on figure 6. By using Refinement in Method, Refinable Function the functionality of Refinable Function is extended or shrank. Also, this reflective feature transforms the role of function transparent, which means composed function can be decomposed partially.

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 5. Is essentially stores feature

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

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. Behavior store also covers execution of functions, which processes both sync and async function sequentially then returns promise. The detail to the operating mechanism is described in chapter 4.3.

4.3 Asynchrony Support

Asynchrony is the core concept of a computer, the asynchronous mechanism spread across computer architecture at a variety of level. For example instruction pipelining and design of memory hierarchy is a way of diminishing bottleneck by the asynchronous approach. Since the philosophy of asynchronous processing is *overlapping the time*, its adaptation for asynchronous processing on web application domain gets significant interest in last decade mainly for overlapping time to waiting for network IO on the server. Many popular languages and framework support asynchronous processing, notably Node.js[12] and uvloop[13] on Python, both techniques are based on Libuv[14], an operating system abstraction for asynchronous processing. Note that synchronous processing means every operation that is performed in-memory so synchronous operation is typically hundred or thousand times faster then asynchronous operation.

4.3.1 Asynchronous Primitives

Asynchronous programming in JavaScript run it on event flow, which corresponding to IO operation is complete, this event flow is typically handled by lambda expression also known as the anonymous function in JavaScript community. While nested anonymous function, in this case, callback function occur negative issue code's readability and comprehension, it called callback problem[15]. JavaScript uses Promise[18] asynchronous primitives for resolving nested callback hell. Promise is a type of coroutine that eventually return resolve or reject, and proceed computation with then method chaining.

4.3.2 Asynchronous Function Handling

The web application is consist of series of IO operation and non-IO operation such as data checking scaffolding. In the design of Refinable Function, we detect whether a function belongs to sync or async by determining function is belong to native, promise object or refinable function object. As the code in Appendix D shows we use partial application to determine asynchronous and synchronous function. We partially apply function and capture it result to proceed to next step. The code shows that, if result is non-function, that is final value, If result returns object with .then method, Self retrieve the value by receive event from .then method. Lastly, If the type of function is an object, that means type function is Refinable Function, then Self recursively execute Refinable Function.

[12]Node.js JavaScript Runtime. http://nodejs.org

[13]Ultra fast implementation of asyncio event loop on top of libuv http://https://github.com/MagicStack/uvloop

[14]Libuv. Cross-platform asynchronous I/O. https://github.com/libuv/libuv

[15]Callback Hell in JavaScript. http://callbackhell.com/

5 Discussion

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.

5.1 Microbenchmark

To discover penalties of 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 analyse 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. In theoretical computer science, time complexity is used as general criteria for the efficiency of algorithm and system analysis. The detail information for the benchmark is available at Appendix B.

5.2 Time Complexity

Processing object is a heavy task for a computer, even compiler optimisation is performed, figure 6 shows its average is higher than native function approximately 3 times. For the IO operation, followed by figure 7 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

5.3 Space Complexity

Space complexity is important to measure the reliability of Refinable Function in terms of memory usage. Refinable Function use memory to store dynamically dispatched method and references of functions that duplicated when making inheritance of function instance. We measure heap size of function in both arithmetics synchronous and asynchronous file io operation. The heap size of the function is growing and not easily freed even explicitly deleting an

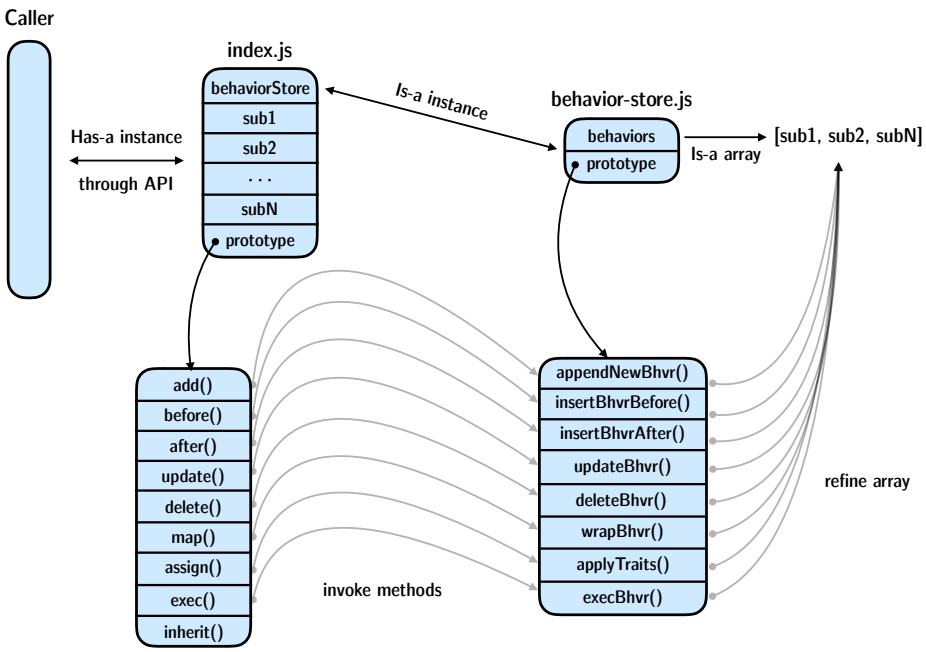

Figure 5. Architecture of Self, JavaScript-based Refinable Function Implementation

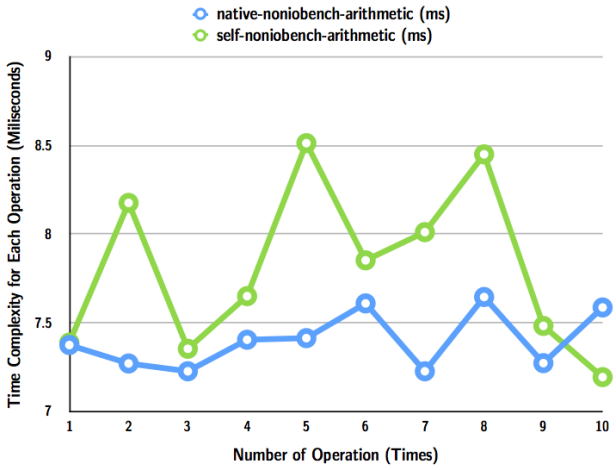

Figure 6. Time Complexity for Non-IO Operation

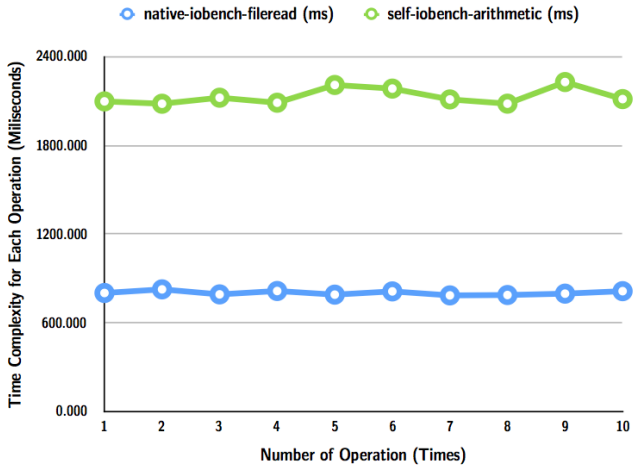

Figure 7. Time Complexity for IO Operation

object, this is a crucial issue for low memory devices. We present the solution in next chapter.

5.4 Prospective Issues

Tick Leap for Explicit Garbage Collection. By extensive usage of object copying and dynamic dispatch, As benchmark shows Refinable Function has inefficient memory consumption than normal function. The result generally shows optimisation is not applied to Refinable Function. However, in IO operation, the gap is being dismissed and the continuous growth of memory consumption is seemingly resolved. The performance of Refinable Function is not supreme but it has competitive for IO operation, such as event-based network application or any application that has dynamic behavior generation. We capture this issue by increasing benchmark phase for 100 times to 500 times, at the same time we reduced iteration of forloop by 100 to 10 times. This modification gives us insight about compiler of modern language has difficult to optimise object related computation in a seamless processing without tick leap, but event-based software like web application and robotics, have many tick between operation and could overcome this weakness. The reason because tick resolves memory usage is that, a tick may break branch prediction and consequently break cache

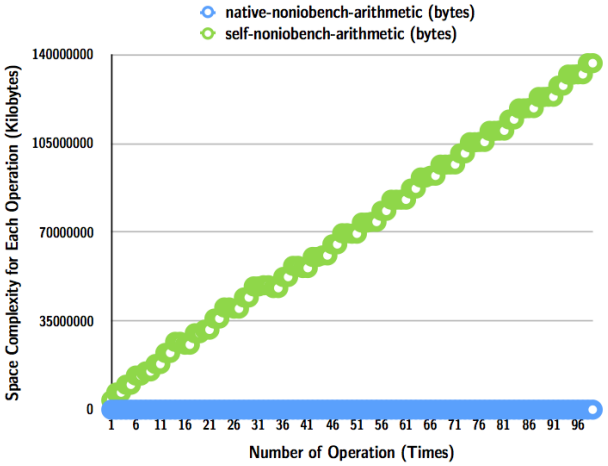

Figure 8. Space Complexity for Non-IO Operation

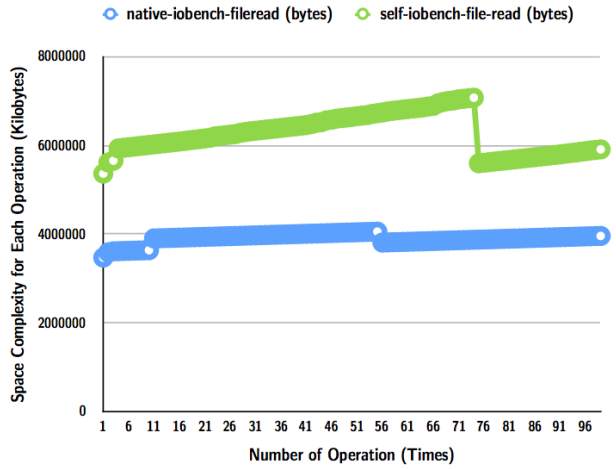

Figure 9. Space Complexity for IO Operation

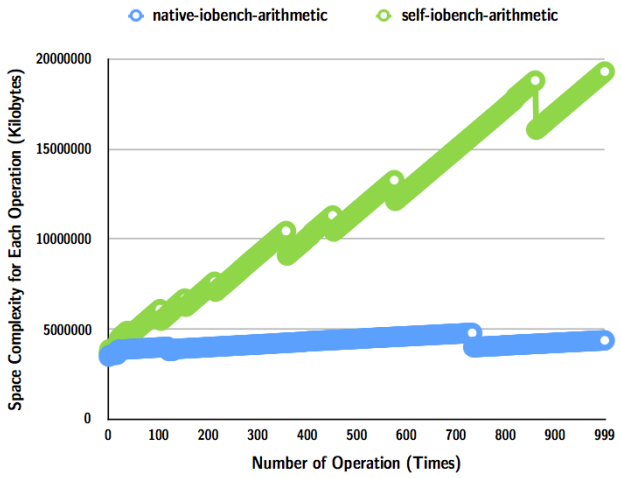

Figure 10. Space Complexity for IO Operation with More Tick

locality. Operating system may resize heap size of given application. As a solution, explicit tick leap can be used to operating heavy object-oriented program.

Just-In-Time Function Inheritance. Useless reference chain is created without reasoning for a program. For example, as appendix add the only program still get all reference from parent behavior. program with no further manipulation has its own copy of the function. and such function may consume memory space. To resolve this, an idea is dynamically to connect chain if a function needs to manipulated and also do not hard copying function array but just reference it and if actual changes have made then copy entire behavior in specified runtime.

5.5 Implementation Issue for Code Comprehension

5.5.1 Automatic Promisification

Promise is constructed that represents the eventual completion of an asynchronous operation[18]. Promise is used for synchronising program execution in language. The initial usage of promise is synchronising lazily evaluated computation, while the contemporary usage of Promise is focused on reducing the delay of IO operation without continuous passing style[25] in an asynchronous programming environment. The Continous Passing Style(CPS) uses Lambda Calculus[25] to continue the event flow. CPS benefits syntactic elegance for concurrent programming but deep CPS defects comprehension and debuggability of code. The Promise in JavaScript resolve the issue of CPS nesting by serialising event flow with .then method[16].

JavaScript uses Callback Function[17] to communicating with underlying CPP module for any IO operation. Such as file or network IO. Static code analysis to identifying CPP module, con contribute automatic promisification of function. This is important for JavaScript community still using library interface based on CPS(callback) style. And it reduces the cost of manual Promisification. Although it is technically possible to runtime code analysis, but its downside risk is performance and robustness

5.5.2 Composition using Infix Operator

Function composition using primitive operator is essential for reducing boilerplate of library initialisation and using thus improve expressiveness by supporting polymorphic behavior on the operator. JavaScript does not allow declaration of the new operator, so overring operator for specified subtype is needed. ECMAScript, the standardised language for JavaScript internally convert type of object to string (9.1 in [7]), when object is used for any arithmetic operator (11.6.1 in [7]). Since object-orientation for JavaScript is implemented as prototype fashion[26], by replacing prototype method

[16]Promises/A+ specification https://promisesaplus.com/

which invoked by evaluating antiemetic operator can be applicable for function composition.

6 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 mininal 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[13] from fine-grained solution to coarse-grained solution. In this section we discuss position of Refinable Function compare to related techniques.

Currying and Partial Application. 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. Partial application takes function with multiple parameters and returns function with fewer parameters, currying returns function with one parameter. There is programming style called pointfree for composing function using partial application[17]. The key difference between refinable function is upgradeability of function, once it is composed the generated function could not be restructured in order to localise variability further.

Aspect-oriented Programming. Aspect-oriented Programming(AOP) towards separation of concern by localising cross-cutting concern that is scattered and tangled across program [15]. At the implementation perspective, a popular implementation like AspectJ[14] supports composition of cross-cutting 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, researches from AOP community is essential to advance composition mechanism of Refinable Function such as heterogeneous cross-cutting concern[5]

Decorator Design Pattern. Design Pattern refers to a general reusable solution to a commonly occurring problem in software design[8]. Decorator pattern is structural solution object-oriented software to localise 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 whiteboxed function dynamic modification of decorator is contrast parts of traditional function-based decorator

[17]Haskell Pointfree Style https://wiki.haskell.org/Pointfree

Data, context and interaction. Data, context and interaction architecture(DCI) is object-oriented design approach to prevent object-oriented collaboration that cause unpredictable system behavior in runtime[21]. 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.

GraphQL. GraphQL is declarative query language for information retrieval from datastore[12]. 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 of 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 can not 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.

Feature-oriented Programming. Feature-oriented Programming(FOP) is composition-based approach for building software product line on the notion of feature[2]. 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)[22] 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[23]. Refinable Function also uses similar Interfaces or possibly DCI architecture[21] 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. Context-oriented Programming [9] 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 context-oriented program to do polymorphic behavior[2]. 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 Conclusion

In this paper, we introduce Refinable Function, optimal medium for advanced language-driven modularity technique based on object-oriented programming. Refinable Function towards a pragmatic solution for languages and environment that has struggled for adapting advanced modularity technique by various technological issues. In addition to practical relevance, we introduced how concepts of object-orientation can be applied to function by introducing inheritance, encapsulation, and polymorphism for function. We and provides concrete feature derived from the concept that can be applied to the web application. We show a prototypal implementation of Refinable Function in JavaScript and performed benchmark and case studies of Refinable Function with a various aspect. We capture new application of object-oriented programming for modularity and explored both theoretical and practical issues of adapting modern modularity to the web application.

References

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

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-oriented software product lines: concepts and implementation*. Springer Science & Business Media.

[3] Andrew P Black. 2013. Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation* 231 (2013), 3–20.

[4] Andrew P Black, Kim B Bruce, and James Noble. 2016. The Essence of Inheritance. *arXiv preprint arXiv:1601.02059* (2016).

[5] Adrian Colyer, Awais Rashid, and Gordon Blair. 2004. *On the separation of concerns in program families*. Technical Report. Technical report, Computing Department, Lancaster University.

[6] Wayne Eckerson. 1995. Three tier client/server architecture: Achieving scalability, performance and efficiency in client server applications. *Open Information Systems* 10, 1 (1995).

[7] ECMA ECMAScript, European Computer Manufacturers Association, et al. 2011. Ecmascript language specification. (2011).

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference*. 406–431. https://doi.org/10.1007/3-540-47910-4_21

[9] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *Journal of Object technology* 7, 3 (2008).

[10] Paul Hudak and Joseph H Fasel. retrieved on July 2017. A Gentle Introduction to Haskell, Version 98, Values, Types, and Other Goodies. (retrieved on July 2017). https://www.haskell.org/tutorial/goodies.html

[11] Apple Inc. retrieved on July 2017. The Swift Programming Language (Swift 4), Advanced Operators. (retrieved on July 2017). https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AdvancedOperators.html

[12] Facebook Inc. retrieved on July 2017. (retrieved on July 2017). http://facebook.github.io/graphql

[13] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 311–320.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *European Conference on Object-Oriented Programming*. Springer, 327–354.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented programming*. Springer Berlin Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

[16] Hiun Kim. retrieved on July 2017. Self - Refinable Function Constructor for Better Modularity Webpage. (retrieved on July 2017). https://hiun.org/self

[17] Mozilla Developer Network. retrieved on July 2017. Declaring and Using Callbacks. (retrieved on July 2017). https://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes/Using_js-ctypes/Declaring_and_Using_Callbacks#Using_Callbacks

[18] Mozilla Developer Network. retrieved on July 2017. JavaScript Promise. (retrieved on July 2017). https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

[19] Oscar Marius Nierstrasz. 2010. Ten things I hate about object-oriented programming. *Journal of Object Technology* 9, 5 (2010).

[20] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[21] Trygve Reenskaug and James O Coplien. 2009. 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).

[22] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines*. Springer, 77–91.

[23] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. 2011. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*. ACM, 43–56.

[24] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274.

[25] Gerald Jay Sussman and Guy L Steele. 1998. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation* 11, 4 (1998), 405–439.

[26] David Ungar and Randall B Smith. 1987. *Self: The power of simplicity*. Vol. 22. ACM.

A Self API in Depth

A.1 Behavior.add(...args)

Adds new sub-behavior. The argument can be either

- @param Object [initBehaviors]
- @param Object[] [initBehaviors.data] - Array of sub-behaviors
- @param String initBehaviors.data[].name - Name of sub-behavior.

- @param Function|Object initBehaviors.data[].behavior - Function or behavior.

Returns a new behavior instance that will be the subject of further manipulation.

- @return Object behavior An new behavior object contain empty or given sub-behvior

```
var LoadArticle = new Behavior().add(
    inputScaffolding).add(authCheck).add(
    cacheCheck).add(loadQuery);
```

A.2 Behavior.prototype.before(...args)

Add new behavior before specified behavior

- @param Function|Object - New function or behavior object
- @param String - name of behavior (if function name does not exist)
- @return Behavior - for chaining

```
WriteDBQuery.validate.before(checkEmpty);
```

A.3 Behavior.prototype.after(...args)

Add new behavior after specified behavior

- @param Function|Object - New function or behavior object
- @param String - name of behavior (if function name does not exist)
- @return Behavior - for chaining

```
WriteDBQuery.validate.after(checkEmpty);
```

A.4 Behavior.prototype.update(...args)

Add new behavior after specified behavior

- @param Function|Object - New function or behavior object.
- @return Behavior - for chaining

```
WriteDBQuery.monitoring.update(cacheMonit);
```

A.5 Behavior.prototype.map(...args)

Wrap specified behavior with given function-returning function

- @param Function - A function for manipulating behavior, this function wraps function which contains original behavior
- @return Behavior - for chaining

```
WriteDBQuery.validate.map(() => {
    return (validate) => {
      validateWrapper(validate);
}
});
```

A.6 Behavior.prototype.delete(...args)

Delete specified behavior, this function takes no argument

- @return Behavior - for chaining

Below example deletes auth sub-behavior from ReadDBQuery

```
ReadDBQuery.auth.delete();
```

A.7 Behavior.prototype.exec(...args)

Execute behavior by serially invoke sub-behavior in array

- @param ...* - argumenets for execution
- @return Behavior - for chaining

```
var ReadDBQuery.exec({userID: 14});
```

A.8 Behavior.prototype.catch(...args)

Catch errors while invoking promise

- @param Function - Function to perform error handling
- @return Behavior - for chaining

```
var ReadDBQuery.exec({userID: 14}).catch(() => {
    return http.res(500) });
```

A.9 Behavior.prototype.new(...args)

Create inherited behavior. The inherited behavior is deep clone of parant behavior and does not reference or link any property

- return Behavior - deeply cloned instance of behavior

```
var ReadDBQuery = DBQuery.new();
var WriteDBQuery = DBQuery.new();
```

A.10 Behavior.prototype.interface(...args)

Apply traits to behavior Traits means set of object-independent, composable behavior Traits override existing sub-behavior with given sub-behavior by name

- @param Object[] - name containing array
- @return Behavior - for chaining

```
LoadArticle.interface(['payloadCheck', 'authCheck'
    ]);
```

A.11 Behavior.prototype.assign(...args)

Apply traits to behavior Traits means set of object-independent, composable behavior Traits override existing sub-behavior with given sub-behavior by name

- @param Object - traits object
- @return Behavior - for chaining

The below examples shows publicApiTraits embodies traits of behavior that, remove all auth module, which of course effects all sub-behavior originated from WriteDBQuery.

```
var publicApiTraits = {
    auth: null
};

WriteDBQuery.assign(publicApiTraits);
```

A.12 Behavior.prototype.interface(...args)

Apply interface ot behavior to substantiate structure of behavior by naming

- @param Object - traits object

The below examples shows publicApiTraits embodies traits of behavior that, remove all auth module, which of course effects all sub-behavior originated from WriteDBQuery.

```
var publicApiTraits = {
    auth: null
};

WriteDBQuery.assign(publicApiTraits);
```

A.13 Behavior.prototype.defineMethod(...args)

Define new method for perform refinement of sub-behavior contained array directly

- @param String - method name
- @param Function - method function
- @return Behavior - for chaining

```
Formula.defineMethod('deleteAddition', function () {
  var self = this;
  this.behaviorStore.behaviors.forEach(function (behavior) {
    if (behavior.name.slice(0, 3) === 'add') {
      self.delete.apply({name: behavior.name, behaviorStore: self.behaviorStore});
      // or by using private API
      //self.behaviorStore.deleteBehavior(behavior.name);
    }
  });
});
```

B Supplementary Information for Microbenchmark

B.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)

B.2 Time Complexity Benchmark Code

B.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 8. 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 9. Refinable Function for Non-IO operation

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

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

```
2
3  function read (n) {
4    return fs.readFile('./file.txt', (e) => {
5      if (!e && n !== 9999) {
6        read(n+1);
7      } else{
8        return console.timeEnd('test');
9      }
10   });
11 }
12
13 console.time('test');
14 read(0);
```

Listing 10. Native Function for IO operation

```
1  var ReadFileRefinable = new Behavior()
2  ReadFileRefinable.add(readFilePromisified);
3
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 11. Refinable Function for IO operation

B.3 Space Complexity Benchmark Code

B.3.1 Normal Benchmark

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

```
1  var addOne = (n) => {return n + 1};
2
3  function proc (n) {
4    if (n !== 99) {
5      var prev = process.memoryUsage().heapUsed;
6      var a = 0;
7      for (var i = 0; i < 100; i++) {
8        a = addOne(a);
9      }
10     var curr = process.memoryUsage().heapUsed;
11     console.log(curr - prev);
12     return proc(n + 1)
13   } else {
14     return;
15   }
16 }
17
18 proc(0);
```

Listing 12. Normal Function for Non-IO operation

```
1
2  var addOne = new Behavior().add((n) => {return n +
       1}, 'a');
3
4  function proc (n) {
5    if (n !== 100) {
6      console.log(process.memoryUsage().heapUsed);
7      var a = 0;
8      for (var i = 0; i < 100; i++) {
9        addOne.exec(a).then(result => {
10         a = result;
11       })
12     }
13     console.log(process.memoryUsage().heapUsed);
14     proc(n + 1)
15   } else {
16     return;
17   }
18 }
```

Listing 13. Refinable Function for Non-IO operation

```
1  var fs = require('fs');
2
3  function read (n, cb) {
4      if (n < 100) {
5        return fs.readFile('./file.txt', function ()
             {
6          cb(n+1);
7        });
8      }
9  }
10
11 console.log(process.memoryUsage().heapUsed);
12
13 read(0, function proc (n){
14       console.log(process.memoryUsage().heapUsed);
15       read(n, proc);
16 });
```

Listing 14. Native Function for IO operation

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

function read (n, cb) {

  ReadFileRefinable.exec('./file.txt').then(k => {
    if (n < 100) {
      cb(n+1);
    }
  });

}

console.log(process.memoryUsage().heapUsed);
read(0,  function proc (n){
      console.log(process.memoryUsage().heapUsed);
      read(n, proc);
});
```

Listing 15. Refinable Function for IO operation

B.3.2 Alleviated Benchmark

The callstack and memory usage of normal benchmark is drastically high does, it is not suitable for application in practices, In the alleviated benchark. We decreased number of iteration to 10 times and increased test phase for 500 times.

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

function proc (n) {
  if (n !== 500) {
    console.log(process.memoryUsage().heapUsed);
    var a = 0;
    for (var i = 0; i < 10; i++) {
      addOne.exec(a).then(result => {
        a = result;
      })
    }
    console.log(process.memoryUsage().heapUsed);
    proc(n + 1)
  } else {
    return;
  }
}

proc(0);
```

Listing 16. Native Function for Non-IO operation

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

function proc (n) {
  if (n !== 500) {
    console.log(process.memoryUsage().heapUsed);
    var a = 0;
    for (var i = 0; i < 10; i++) {
      a = addOne(a);
    }
    console.log(process.memoryUsage().heapUsed);
    return proc(n + 1)
  } else {
    return;
  }
}

proc(0);
```

Listing 17. Refinable Function for Non-IO operation

C Schema for Example Blog Application

```
var Schema = {
  Users: {
    id: Number,
    email: String,
    name: String
  },
  Posts: {
    author: Number,
    body: String,
    bodyImages: [Number],
    comments: [Number]
  },
  Comments: {
    author: Number,
    body: String,
    bodyImages: [Number]
  },
  Messages: {
    author: Number,
    body: String,
    bodyImages: [Number]
  },
  Images: {
    id: Number,
    url: Number
  }
};
```

Listing 18. Refinable Function for Non-IO operation

D Sync/Async Function Handling

```
Behaviors.prototype.execBehavior = async function execBehavior (...input) {

  //for each behavior
  //use for..of to cover use of async await
  for (behavior of this.behaviors) {

      var hostBehavior = behavior.behavior;

      //if single behavior exist (not multiple & not assigned null by .assign)
      if (hostBehavior !== null) {

        //for behavior of promise and native function
        if (typeof hostBehavior === 'function') {

          //exec promise or native function
          result = hostBehavior.apply(null, input);

          //if function returns promise
          if (typeof result.then === 'function') {

            input = await hostBehavior.apply(null, input);

          }

          //if function is not promise
          else {
            //perform assign after computation complete
            input = [result];
          }

        }

        //if behavior has sub behavior
        else if (Array.isArray(hostBehavior)) {

          //setting up context
          var context = this;
          context['behaviors'] = hostBehavior;

          //resursive
          //explicit error handling to prevent error shallow of promise
          try {
            input = await execBehavior.apply(context, input);
          } catch (e) {
            throw e;
          }

        }

      }

      //formatting to array after finish execution
      if (!Array.isArray(input)) {
        input = [input];
      }
  }

  //return final reult
  return input;
};
```

Listing 19. Code for Recursively Determining and Executing Three type of function