Refining Higher-Order Functions and Aspects

Hiun Kim
Sejong University
Seoul, South Korea
hiun@hiun.org

Abstract

Higher-Order Languages (HoL) plays a key role in implementing modern software systems by treating *program as a data*. Dynamic language features and typing discipline benefits expressiveness and productivity of HoL, but these leads leak to embrace some of the fundamental software engineering and modularity concepts- aspect-orientation and step-wise refinement, which relies on quantification from static language features and typing. In this paper, we present *Function Refinement* (FR), a dynamically quantified AOP model for supporting fine-grained refinement over higher-order functions and aspects. FR model sequential compositionality of functions as a class, comprised with function list and set of methods to extends, inherit and mutates its functional elements to refine a function, just like how we refining classes. As a result, FR supports to perform arbitrary additive and shrinking types of higher-order aspect refinement over dynamic join point at runtime. The major design component of FR is comprised of recursive definition of a functional class to support the basis of refinement, and indirect, encapsulated mutation as a mechanism of refinement, and lastly, functional hierarchies to support inheritance for reuse and localization of cross-cutting concerns. In order to clearly draw correspondence and benefit of FR, compare to traditional AOP techniques, we provide operational semantics and running implementation of FR in JavaScript programming language.

CCS Concepts • **Software and its engineering → Language features**; **Software development techniques**; *Object oriented languages*;

Keywords function refinement, aspect-oriented programming, step-wise refinement, higher-order programming, object-oriented programming

1 Introduction

Higher-Order Languages (HoL) like JavaScript [1] or Python [3] play a key role in implementing modern software systems by treating *program as a data* [26]. HoLs has become a practical programming languages to engineering modern web, mobile or cyber-physical software systems [16, 38, 52, 54], from its practical benefit of productivity [55], and decent performance [4, 60]. Dynamic language features and typing discipline benefits expressiveness and productivity of HoL, but these leads leak to embrace some of the fundamental software engineering and modularity concepts- aspect-orientation [46] and step-wise refinement [28], which relies on quantification from languages with mostly static features and typing like Java [24].

1.1 Background

Aspect-Oriented Programming (AOP) [46] is profound modularity technique [25], applied from enterprise applications [32, 47] to adaptive softwares [35, 41, 69], for reuse and localization of cross-cutting concerns that are scattered and tangled across the program. For AOP in HoL, *Tucker et al.* [64] proposes a Higher-Order Aspect (HoA), which provides functional pointcut and advice by means of function composition. *Apel et al.* [23] proposes Aspect Refinement (AR) to enable extensible reuse of aspect in Object-Oriented Programming (OOP) languages by inheriting and refining of pointcut and advice just like how we refine classes. However each proposition has own limitation, as HoA is not refinable while AR does not supports fine-grained functional aspects.

For HoA, *Toledo et al.* [63] proposed AspectScript, an aspect realized in higher-order functions of JavaScript. AspectScript allows aspects can be defined as normal functions, for higher-order reuse and composition.

Batroy et al. [27] introduced the notion of Step-Wise Refinement (SWR) based algebraic model, where a development of the complex program is made of an incremental composition of simple programs. AR [23], which extends SWR, enable to incrementally develop pointcut and advice from series of refinement. When both HoA and AR combine, it is promising to deliberate AOP values into HoL. However, both HoA and AR are studied independently, does not adequately addressed when it combines as a whole, to support *higher-order aspect refinement*.

When discussing HoA and AR in a single language, due to inherent properties of HoL, several challenges are made. For productivity, HoL constructs dynamic behavior in runtime via composition of functions, which inherently limit the decomposability and refinability of definitions of individual aspects. For performance, as some of HoL are so-called *script* languages, have no intermediate representation, interpreted or directly compiled, which make difficult to manipulate program in build time using compiler extensions, like AOP and AR in static first-order languages [23, 45, 61]. Lastly,

Conference'17, July 2017, Washington, DC, USA
2018.

Table 1. Contributions of function refinement.

	Higher-Order Aspects [63, 64]	**Function Refinement**	Aspect Refinement [23]
Aspect Construct and Granularity	Functions (Fine-Grained)	**Function and Class (Multi-Grained)**	Class (Coarse-Grained)
Refinement Types	Additive	**Additive, Full Shrinking**	Additive, Partially Shrinking
Implementation	Compiler Extension	**Vanilla Runtime**	Program Transformation
Refinement Mechanism	Dynamic Refinement over Static Join Point	**Dynamic Refinement over Dynamic Join point**	Static Refinement over Static Join Point
Dynamic AOP	O	**O**	X

many script languages are developed with standards in order to maintain interoperability to a runtime environment [37], like JavaScript, that enforces program is written strictly with no custom extension to ensure compatibility of many web browsers. This makes harder to extend syntax and semantics of languages while conforming standards.

1.2 Refinable, Higher-Order Functions and Aspects

In summary, the composition of HoA is powerful, but it leaks to support refinement after compositions are tied. In the same line, SWR is effective but it is less promising to support refinement in the level of fine-grained function. On top of that, for regarding implementation, techniques proposed for HoA [63] and AR [23] require either program transformation or compiler extension, which less feasible in a case of using HoL for programs in variant runtime environments, such as web browser or commodity Internet of Things (IoT) devices.

In this paper, we present *Function Refinement* (FR), an AOP model that supports SWR over a composition of higher-order function and aspects to adequately integrate both AOP and SWR within plain language features of FP and OOP. Sec. 2 introduce background and problem definition of current AOP and SWR. Sec. 3 elaborate fn actual use of AOP with FR regarding construction, composition, refinement, and deployment of higher-order functional pointcut and advice. Sec. 4 shows implemtation and operational semantics of FR. Specifically, we deliver how FR model sequential compositionality of functions as classes, comprised with function list and set of methods[1] to extends, inherit and mutates its functional elements to refine a function, just like how we refining classes. Sec. 5 compare enablements of FR, dynamic additive and shrinking types of higher-order aspect refinement regarding existing AOP techniques. Sec. 6 concludes paper with discussions and future work.

Contributions We summarize our contributions:

- **Analysis.** We provide an analysis of current static first-order and dynamic higher-order languages to find where refinement of higher-order functions and aspects can support scalable aspect modularity for dynamic software written in HoL.
- **Model.** We provide a refinement model and operational semantics for higher-order functions and aspects by applying step-wise refinement over function composition, based on proposed mechanisms on extending inheriting, exporting and mutating functional class.
- **Implementation.** We provide a design and publically available implementation [13] for higher-order aspects that support dynamic additive and shrinking refinement with vanilla language features.

2 Background and Problem Definition

In this section, we introduce static first-order aspects in AspectJ [45] and dynamic HoA in AspectScript [63]. In order to support higher-order aspect refinement, we discuss and exploit the contributions and limitations of previous research on HoA and AR, and define problems which FR tries to address as depicted in Table 1.

2.1 Higher-Order Aspect Composition

2.1.1 First-Order to Higher-Order Aspects

One of profound implementation of AOP in static first-order languages is AspectJ [45]. Starting from its prototype implementation [46], AOP has largely been substantiated with FP [22, 34, 36, 50, 66, 67]. Specifically, the concept of HoA by *Tucker et al.* [64] and AspectScript [63] by Toledo et al. showing how pointcut and advice are implemented in HoL, using higher-order functions by leveraging function composition to weave and deploy them. However as an inherent limitation of function composition, SWR of higher-order pointcut and advice are difficult to implement since compositional order and contents of functions are fixed, and are does not modifiable.

[1] A full list of refinement methods are available in Appendix B

2.1.2 Higher-Order Aspects in AspectScript

HoA [64] like AspectScript [63] differs from static first-order AOP languages like AspectJ by using a standard function to defines advice and pointcut. By using bringing aspects to first-class language primitives, AspectScript provides rich dynamic AOP features including, dynamic aspect deployment, aspect propagation, and custom join point with user-defined event [58].

```
var pc = PCs.exec(sockReceiveHandler).and(PCs.cflow(PCs.exec
     (socket)));
var adv = (jp) => {
  if (!checkDataFmt(jp.args[0])) { return 'error' }
    else { return jp.args[0]; }  }
AS.after(pc, adv);
```

The above code depicts simple argument checking in AspectScript when function `sockReceiveHandler` is invoked from the `socket`. As described above, set of pointcut `exec` and `cflow` is stored in `PCs`, and used to define pointcut conditional for advice `adv`, which composed via `AS.after` with `pc` for `sockReceiveHandler`.

In AspectScript, for regarding additive refinement of aspects, extending existing pointcut and advice are simple with functional composition:

```
var pc = PCs.exec(sockReceiveHandler).and(PCs.cflow(PCs.exec
     (socket)));
var pc_ext = pc.and(negationChecker);
var adv_ext = socketValidatorFn(adv);
AS.after(pc_ext, adv_ext);
```

However, such composed functions are difficult to update and modify limits to support `shrinking` refinement, by supporting not just extending, but mutating, deleting some of the composed functions. Consider of `cflow` pointcut refinement. As above example shows`cflow` pointcut `pc` is composed with execution of `sockReceiveHandler`. The reusing of `pc` is difficult without rewriting the whole pointcut definition. For advice refinement, consider a case where we want to change the behavior of `adv_ext` which is composed of multiple advice functions. In the same line of pointcut refinement, by modifying a portion of advice will introduce the rewriting of the whole definition.

2.1.3 Leaking Support for Functional Aspect Refinement

As the ability to SWR is an essential method to managing software variation [21], however current HoA techniques are difficult to implement refinement from a complex core since such refinement involves modification and removal. For advanced reuse, HoA demand not just support the extension of aspects, but the support of update and deletion to refining and reusing aspects from a compound basis similar to product line derivation [21, 61].

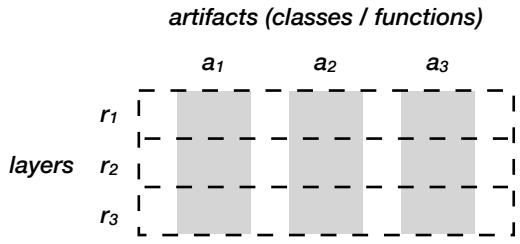

Figure 1. Step-wise refinements of classes and functions

2.2 Step-Wise Aspect Refinement

2.2.1 Aspect Composition to Refinement.

SWR is an engineering method to develop software by incrementally adding small features [68], *Batroy et al.* [27] proposes a notion of program representation as a hierarchical algebraic structure, to scalably refining them with sophisticated language extension or tooling. Since SWR promotes software engineering at scale, applying SWR for AOP are natural approaches [23, 49, 51]. AR frames advice and pointcut as typical class members and implements their refinement using mixin-based inheritance [30].

AR benefits pointcut and advice to be tolerant of future changes, however current AR have not much discussed in the context of HoA and HoL. Moreover, current AR only support *additive refinement*, that additively *extends* pointcut and advice definition inherited from parent. As we discussed in HoA, it is reasonable to support *shrinking refinement* [61], over AR that *mutates* composed pointcut and advice definition to fully adequate for bringing software engineering methods to HoL [21, 44, 57].

2.2.2 Leaking Support for Shrinking Refinement

SWR is mostly studied in the context of OOP, where refinement corresponds to implementing features by means of adding or modifying methods and data from the object. Fig. 1 illustrate SWR, a column a_1, a_2 represents individual classes where refinement layers r_1, r_2, and r_3 crosscuts that column horizontally. AR applied the notion of SWR for aspects, where pointcut and advice have a name, to be called and applied refinement later. As in code shown below, AR designates `ValidatePC` as a pointcut and `CheckerAdv` as a advice. AR reuses parent pointcut definition for inherited pointcut when it extended. For example, as below code shows, AR adds `cflow(File)` as another valid initiation points. In the same sense, advice refinement are similar to use of `super`, by *additively* extending advice with custom pre and postprocessing (`lock` and `unlock`), as a around advice for `CheckerAdv`.

2.3 Runtime Dependency

In static first-order AOP languages like AspectJ, implements the syntax and operational mechanism of AOP by extending compiler functionality. In dynamic higher-order AOP

```
aspect DataFmtChecker {
  pointcut ValidatePC (String data): target(data) && call(
       String SockHandler.receive()) && cflow(Socket)
  String after CheckerAdv (SocketHandler) : receive(data) {
    if (checkDataFmt(data)) { return 'error'; } else {
         return data; }
  }  }
```

```
refines aspect DataFmtChecker {
  pointcut validatePC(): super.syncPC() || cflow(File)
  Object SocketHandler() {
    lock(); Object res = super.DataFmtChecker(); unlock();
         return res;
  }  }
```

languages like AspectScript, and AOJS [67] implements aspects with program transformation which normally involves wrapping function with aspect-related codes. However, both approaches limit to support dynamic refinement over dynamic join point, due to the weaving phase are bounded in static build time.

2.3.1 Leaking Support for Vanilla Implementation

Current AOP and AR implementations depended on build time preprocessing which difficult to support dynamic refinement over runtime constructed join point. Runtime-based Higher-Order AOP like FR and other described in Sec. 5 has advantages to handle runtime dynamics. However, most of runtime AOP are based on a primitive mechanism of function composition, which inherently not refinable.

2.4 Towards Higher-Order Aspect Refinement

In this section, we have discussed the contributions and subsequent shortcomings of current HoA and AR languages, and their implementations. That is, (1) leaking support for dynamic, and shrinking refinement, (2) leaking support for functional aspect refinement, (3) leaking support for vanilla runtime implementation. In order to fully tackle proposed problems, we propose FR, as an AOP model and implementation that support for vanilla, dynamic, shrinking refinement for higher-order aspects without build-time dependencies.

3 Function Refinement in a Nutshell

In the previous section, we have discussed existing language approaches to AOP and SWR. We elaborated static first-order languages support AR, however, does support HoA, and dynamic higher-order languages support HoA, but difficult to support AR. In addition, both approaches limit to support dynamic refinement over dynamic join point due to build time dependencies. To address these problems, We propose *Function Refinement* (FR) as a novel AOP model and technique to support HoA with dynamic refinement.

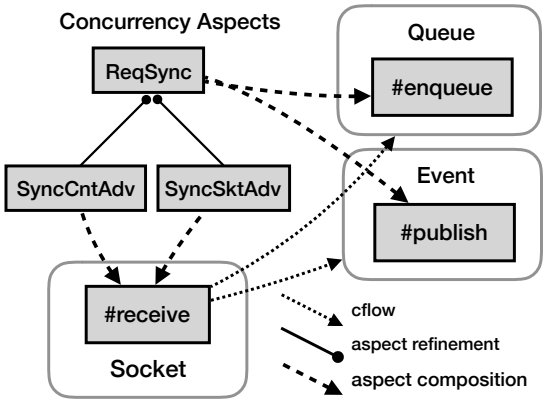

Figure 2. Diagram of aspect refinement.

```
1  // class declaration
2  var Queue = {
3    conn: // ..
4    enqueue : (e) => { this.conn.push(e); }  };
5  var Event = {
6    subscriber = // [..]
7    send: // ..
8    publish: (data) => { this.send(data) }  };
9  var Socket = {
10   receive: () => {
11     var queue = new Queue();
12     var event = new Event();  }  };
13
14 // aspect definition
15 var ReqSync = new RF().add(lock).add(unlock);
16 ReqSyncAfter = ReqSync.new().lock.asAfter();
17
18  // aspect composition
19 ReqSyncAfter.compose(Queue.enqueue);
20 ReqSyncAfter.compose(Event.publish);
21
22 // aspect refinement
23 var receive = Socket.receive;
24 SyncCntAdv = ReqSync.new()
25   .lock.after(() => { this.count++; }, 'count');
26 receive = SyncCntAdv.lock.after(receive);
27
28 SyncSktAdv = ReqSync.new()
29   .lock.before(cflow('Socket'));
30 receive = SyncSktAdv.lock.after(receive);
31
32 // aspect deployment
33 RF.utils.deploy('receive*', Socket, ReqSyncAfter);
```

Figure 3. Example of aspect extension, inheritance, and deployment in FR.

To elaborate the concepts of FR, we use the concurrency synchronization problem of AR in [23] to addressing how additive refinement of HoA are made. In the latter part, we

extend this example to elaborate how shrinking refinement of HoA are made for dynamically constructed join point.

3.1 Additive Refinement of Functions

For additive refinement of functions, We show an example of inheriting and extending advice and pointcut using concurrency control of Queue and Event as illustrated Fig. 2. The corresponding code of refinement is depicted in Fig. 3 including construction, refinement, and deployment of aspect:

- **Refinement Target.** A class Queue and Event are introduced, which contains enqueue method to put items, and publish method to sending message to clients (Line 2-13). Both method call is synchrnozed via ReqSync aspect that exported with *after* advice type: ReqSyncAfter (Line 16-21).
- **Refinement Subject.** A class Socket is introduced, that uses both Queue and Event class (Line 9-13), to initiate enqueue or dequeue method call in receive method. For advice refinement, SyncCntAdv and SyncSktAdv is inherited from ReqSync advice and composed to Socket.receive. In SyncCntAdv, additional for synchronization counting aspect is composed (line 25-27). At SyncSktAdv, cflow pointcut is composed by synchronizing call of enqueue or dequeue method call from Socket.receive (Line 29-31). Lastly, SyncCntAdv is deployed into all methods of Socket (Line 34).

We have briefly shown how shrinking and additive refinement is made over higher-order functions and aspects. In the following part, we elaborate on each concept of function extension, inheritance, exportation, and mutation to show how dynamic aspect refinement can be expressed and implemented by using OOP and FP features.

3.1.1 Higher-Order Function Extension

Function extension corresponds to pointcut and advice refinement in [23] by extending or definition of aspect. However, the difference is function extension supporting higher-order functions and aspects and offer incremental refinements for aspects, instead of overriding and defining aspect from scratch.

Construction of Functional Aspects. As Fig. 3 shows, ReqSyncAdv is introduced, that support synchronization for enqueue, dequeue of Queue and Socket. ReqSyncAdv is simply constructed by sequentially appending lock and unlock advice via add methods:

```
var ReqSyncAdv = new RF().add(lock).add(unlock);
```

Composition of Functional Advice. Fine-grained functional advice composition is made by invoking function extension methods from function and pass join point as an argument. Like below, the invocation of after refinement method with receive join point construct SyncReceive, which is function that Socket.receive is composed between lock and unlock. Similarily, by predefining advice with asAfter(), the extension is done by compose method call.

```
ReqSyncAdv.lock.after(Socket.receive);

var ReqSyncAdvAfter = ReqSyncAdv.asAfter();
ReqSyncAdvAfter.compose(Socket.receive)
```

Coarse-grained, class-level advice composition is made by defining advice type, with asAfter call in function exportation methods. The exported function ReqSyncAfter us delopyed to class with deploy method in RF utility. The deploy methods allows to compose ReqSyncAfter into every method in Socket when a method name is matched in the pattern. In this case, all method prefixed in receive will be affected.

```
ReqSyncAfter = ReqSync.new().lock.asAfter();
RF.utils.deploy('receive*'. Socket, ReqSyncAfter)
```

Composition and Refinement of Functional Pointcut. Similar to advice, the functional pointcut is composable as a predicate function to determine to proceed or not to invoke advice. Like advice, pointcut in FR is a standard function, except pointcut must throw an error when the predicate is evaluated as false. Below code displays abstract version of cflow pointcut for FR. Similar to AspectScript, pointcut returns predicate function for evaluation after binding pattern.

```
function cflow (pattern) {
  return () => {
    if (getLeastCall(new Error().stack()) == pattern) {
      return new Error('cflow error');
    } } }
```

As an example of pointcut refinement, consider a SocketPC aspect that contains cflow pointcut for Socket. SocketPC is inherited into child pointcut and construct SocketAuthPC that has additional authenticaton checking, by adding authCheck poincut. Later, SocketAuthPC pointcut can be refined to update cflow target for Queue, and reusing authCheck.

```
var SocketPC = new RF().add(cflow('Socket'));
var SocketAuthPC = SocketPC.new().add(authCheck);
var AuthPC = SocketPC.new().cflow.update(cflow('Queue'));
```

For coarse-grained pointcut composition, like advice, pointcut itself can composable with pattern matching. Below code shows composition of pointcut SocketAuthPC into all methods in Socket.

```
RF.utils.deploy('Socket.*', Socket, SocketAuthPC);
```

3.1.2 Higher-Order Function Inheritance

We discussed function extension and exportation as a mechanism of implementing additive refinement over advice and pointcut. However, as extension and exportation directly modifies function, that degrades the reusability of the original definition to creates multiple instances of functions that differently refined further. A natural mechanism to handle such variability is *inheritance* [29] that aims to resuing commonalities, and localizing variabilities, similar motivations of software product line engineering [21, 57]. Inheritance of function basically constructs an independent instance of function to allows reuse of current configuration, while able to take refinement differently from parent function. [2] As Fig. 2 shows, function `SyncCntAdv` and `SyncSktAdv` is inherited from parent function`ReqSync`, which is realized in invocation of `new` method code in 3.

3.1.3 Higher-Order Function Exportation

In FR, as advice is normal functions, an advice type needs to be defined for both fine-grained and coarse-grained composition. With function exportation, the `before`, `after` and `around` join point mechanism is wrapped with function and directly composable to functions or classes with unified method call `compose`. For coarse-grained refinement, `compose` invoked remotely by `deploy` method. An example of function exportation is realized with `asAfter` in Fig. 2.

3.2 Shrinking Refinement of Functions

3.2.1 Higher-Order Function Mutation

In the previous part, we have discussed extension, inheritance, and exportation of function and shows correspondence to AOP and AR. However, in a line with the previous tehniques [63, 64], currently proposed refinement methods are good at supporting additive refinement, but limited to support shrinking refinement by means of update or deletion, which is the core properties of generative programming such as software product line [21, 61]. In this part, in a line with pointcut refinement previously, we elaborate function mutation to show shrinking refinement is made on functional advice using refinement methods of FR.

Refinement of Functional Advice. For shrinking refinement of functional advice, we extend the previous example to make shrinking refinement for a method of `Socket`, respect to state of `Queue` and `Event`. The example motivated from *Typestate-Oriented Programming* [19, 62] that is a programming technique that dynamically enables or disable availability of method respect to the state of that object.

```
1  Queue.on('on', function statHandler (target) {
2  return (args) => {
3      if (msg === 'full') {
4        target['enqueue'].queueSender.delete();
5         return function (...args) {
6            let result = target['enqueue'].apply(this,
                  args);
7            return result;
8          } else {
9            return target['enqueue'].prepend(
                  queueCrosscutts);
10         }
11     if (msg === 'no-subscriber') {
12        target['publish'].eventEmitter.delete();
13        return => (...args) {
14           let result = target['publish'].apply(this,
                  args);
15           return result;
16       } else {
17         return target['publish'].prepend(eventEmitter);
18 }  }  }  }) ;
```

Figure 4. Diagram of dynamic aspect mutation in FR.

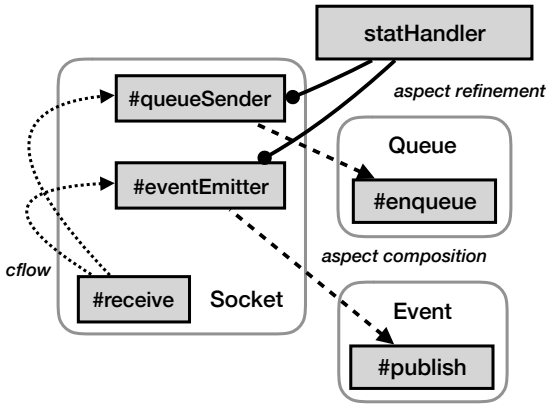

Figure 5. Example of dynamic aspect mutation in FR.

Recall our example that, when the `enqueue` or `publish` methods failed by `Queue` is full, or no subscriber exists in `Event`. For making early failure, we can dynamically add and remove method `enqueue` or `publish` from caller depending on the state of `Queue` or `Event`.

Fig. 6 depicts design of method refinement for `Queue` and `Event`. As event handler `statHandler` is constructed for *Socket* that listen state of `Queue` and `Event`. Respect to the result of state from `statHandler`, `enqueue` and `publish`, which is cross-cutting concern of `receive`, is added or removed dynamically.

[2] Inheritance provided in FR is different from function inheritance by *Brown et al.* [31], which is a technique to adding extensibility for function composition by supporting modification called *mixin functions*. Mixin functions correspond to aspects and reusable, since functions binding of arguments until determined before final invocation. However, composed functions are not refinable, due to the inherent limitation of native composition mechanism. Function inheritance is more generally usable, plain solution for HoA, while FR is much more flexible, but more ad-hoc by only supporting consulted types and structure for HoA.

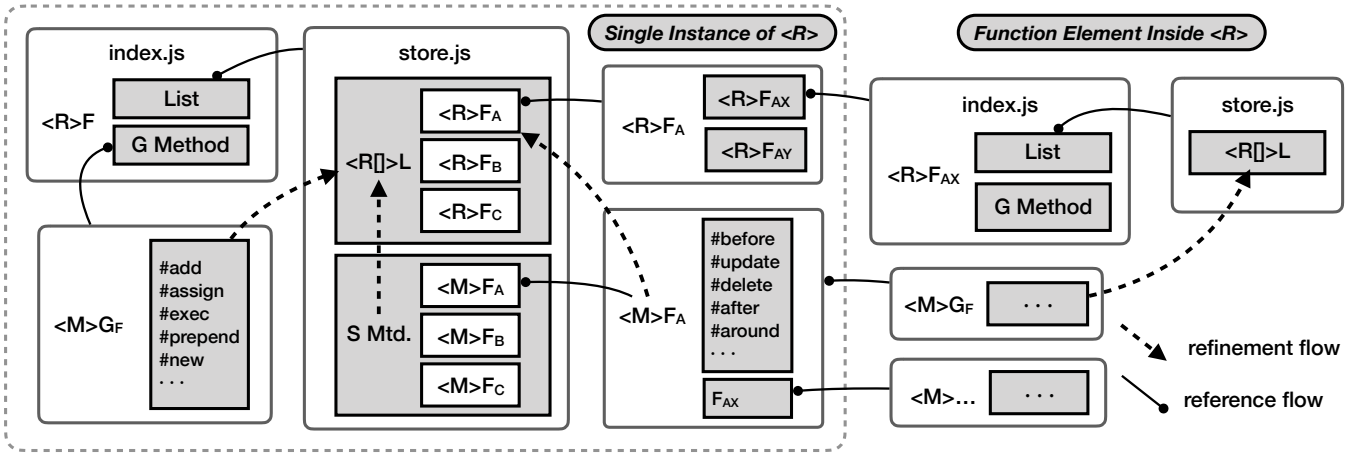

Figure 6. Design and implementation of function refinement.

$$
\begin{array}{rcl@{\qquad}rcl}
R \in Func & = & Addr \hookrightarrow (L \times M) & L_p \in Prec & = & Addr \hookrightarrow (Func \times Addr) \\
L \in List & = & (Func \times Addr) & L_s \in Succ & = & Addr \hookrightarrow (Func \times Addr) \\
M \in Methods & = & Addr \hookrightarrow Object & R_t \in Targets & = & Addr \hookrightarrow (Func \times Addr) \\
R_g \in Guest & = & Addr \hookrightarrow (Func \times Addr) & T_i \in Input & = & Types \\
L_n \in ListItem & = & Addr \hookrightarrow (Func \times Addr) & T_o \in Output & = & Types
\end{array}
$$

Figure 7. Runtime of function refinement.

$$\frac{R_g \notin L \qquad T_i(R_g) \in T_o(L_p) \qquad T_o(R_g) \in T_i(L_s) \qquad L' = L :::: (R_g) \qquad M' = M :: (R_g)}{< M,\ L,\ E[R.\mathtt{add}(R_g)] > \longrightarrow < M',\ L',\ E[R'] >}\ \text{E-Extends}$$

$$\frac{R_t \in L \qquad T_i(R_g) \in T_o(L_p) \qquad T_i(L_s) \in T_o(R_g) \qquad L' = L :::: (R_g, R_t) \qquad M' = M :: (R_g)}{< M,\ L,\ E[R.R_t.\mathtt{update}(R_g)] > \longrightarrow < M,\ L',\ E[R'] >}\ \text{E-Mutates}$$

$$\frac{R_t \in L \qquad L' = L :::: (R_g)}{< M,\ L,\ E[R.R_t.\mathtt{asAround}()] > \longrightarrow < M,\ L',\ E[\lambda.argR'] >}\ \text{E-Exports}$$

$$\frac{M^* = M \qquad L^* = L}{< M,\ L,\ E[R.\mathtt{new}()] > \longrightarrow < M^*,\ L^*,\ E[R^*] >}\ \text{E-Inherit}$$

$$\frac{R_g \notin M \qquad M' = M :::: (R_g)}{< M,\ L,\ E[R.\mathtt{defineMethod}(R_g)] > \longrightarrow < M',\ L,\ E[R'] >}\ \text{E-Dispatch}$$

$$\frac{args \in T_i(R) \qquad v = R(args)}{< M,\ L,\ E[R.\mathtt{exec}(args)] > \longrightarrow < M,\ L,\ E[v] >}\ \text{E-Exec}$$

$$\frac{args \in T_i(L_{n-1}) \qquad v = L_n(args)}{< args;\ \textbf{while}\ L_{n+1} \notin Nil \mid L > \longrightarrow < v;\ \textbf{skip}\ v >}\ \text{Exec-Iteration}$$

Figure 8. Operational semantics of function refinement. (::: denotes addition of right function into L. :: denotes dispaching of refinement method of right function into M).

As line 4, 9, 12, 17 in Fig. 4 shows, refinement dynamically deletes or extends enqueue and publish method via delete and prepend method call on join point respect to the state variable msg from statHandler.

4 Implementation and Analysis

In this section, we discuss how features of FR proposed in Sec. 3 are corresponds to resolves problems defined in Sec. 2 . We elaborate implementation and semantics on major design components of FR, including the *recursive definition of a functional class* to support the basis of refinement, and *indirect, encapsulated mutation* as a mechanism for refinement, and lastly, we introduce *functional hierarchies* to support function inheritance for aspect reuse and localization.

Problems Revisited Recall that we identified the following problems in Sec. 2: (1) leaking support for dynamic, and shrinking refinement, (2) leaking support for functional aspect refinement, (3) leaking support for vanilla runtime implementation.

```
function RF (initFn) {
  this.store = new Store();
  if (initFn) {
    if (initFn.data) {
      initFn.data.forEach((fn) => {
        this.store.list.push(fn);
      });
    } } }
```

Figure 9. Construction of Functional Class.

In the following, we elaborate how these problems are tackled in FR.

4.1 Dynamic and Shrinking Refinement

We proposed function extension in Sec. 3.1.1 to support dynamic refinement of functional aspects. We have showed an example using add method, to incrementally composing count advice after lock and unlock advice into ReqSyncAdv aspect. Similarily, we demonstrated extension for pointcut by incrementally composing cflow and custom pointcut authCheck for constructing SocketAuthPC aspect.

Dynamic and shrinking refinement is supported in a basis of functional class in FR, which conducted to apply refinement using method to functions stored in the list. At the implementation perspective, extension, mutation, and inheritance for refining order and body of serialized functions is substituted for pushing, splitting, and duplicating functions in the list. For instance, invocation of delete and update as shown in 4.2 will results removing and replacing corresponding functions in the list. Dynamic refinement and join point allowed, by only using runtime language features.

4.1.1 Recursive Definition of Functional Class

Design. As we discussed in Sec. 2, higher-order functions and their composition is a basis for implementing HoA.In order to bringing refinement into higher-order function composition, we model the *compositionality* of functions with recursive definition of *functional class* in FR. On the left side of Fig. 6 shows, the functional class denoted as type R, which possibly contain other functional class typed as R[3]. The instance of R, F has function list L that composed of low-level functional class F_A, F_B, and F_C. In the same manner, functional class F_A is composed of functional class F_{AX} and F_{AY}.

Semantics. FR framed classes as functions to uses refinement of classes as a refinement of functions. We designed this functional class by modeling composition of function as a composition of recursively typed classes. As a result, we can use classes in terms of a unit of reuse, composition, and execution, just like functions, while allowing refinability, just like classes.

[3]R can be typed as primitive function. However in this exampl,e we assume all R is typed as functional class.

```
RF.prototype.add = function (fn, name) {
 var newFn;
 if (typeof fn === 'function') {
   newFn = {name: fn.name, fn: fn};
 } else {
   var hardcopiedFn = fn.store.list.slice();
   newFn = {name: name, fn: hardcopiedFn};
 }
 this.store.list.push(newFn);
 this[newFn.name] = {  name: newFn.name  };
 var props = Object.keys(RF.prototype);
 props.forEach((trait) => {
   if (RF.prototype[trait]) {
     this[newFn.name][trait] = RF.prototype[trait];
   }  });
 this[newFn.name]['store'] = this['store'];
 if (fn instanceof RF) {
 var arr = fn.store.list.slice();
 var clonedStore = new store(arr);
 Object.keys(fn).forEach((subfn) => {
   this[newFn.name][subfn] = {};
   props.forEach((method) => {
    if (RF.prototype[method]) {
       this[newFn.name][subfn][method] =
          RF.prototype[method].bind({store: clonedStore
               });
     }  });  });  }
  return this;
};
```

Figure 10. Implementation of Function Extension.

Fig. 7 shows runtime of FR, a functional class R is comprised of function list L with set of refinement methods M. A guest function R_g, is refinement subject to R, and L_p and L_s is previous and succeeding function of R_g, after R_g is serialized in L. For function extension, E-Extends in Fig. 8 shows operational semantics of add, that append R_g into L using add method call in M for R. In order to maintain compositional safety, much of refinement takes type checking based function subtyping rule [56], which briefly saying, input type of R_g is supertype (contravariant) to output type of previous function, L_p, and output type of R_g is subtype (covariant) to input type of succeding functions L_s [4].

Implementation. Fig. 9 depicts construction of functional class in implementation refinable.js [13] [5]. The construction of functional class construct function list Store and

[4]For functions composed in first or last position of list, type checking is skipped for either L_p or L_s which does not exist in the edge of list .

[5]The implementation is written in prototype-based programming languages [65] JavaScript [1]. In JavaScript, a class is implicitly constructed by defining a constructor function, hence this construction mechanism is not related to the functional class design of FR.

bound as member, which is extended and mutated upon refinement method invokes. Construction has one optional argument, that take function list from another functional class for function inheritance which will be discussed later.

Fig. 10 depicts add method for function extension. When methods invoke, given R_g is pushed to the L in member variable `this.store.list` (Line 9) before setting the name (Line 10), which used as anchor of refinement later. If the function is a typed as R, instead of a primitive function, duplicate the L of R_g (Line 2-8), this allows independent refinement of function by deleting the reference chain to previous L of R_g. To set the M for R_g into host R, all current M of host R is attached under the method anchor (such as `newFunc` in `R.newFunc.add`) named R_g in host R (Line 12-15) and bound `this.store` as a context of new M to direct refinement target into L of R_g in host R (Line 16). If a R_g is typed as of R (Line 17), for each element in L in R_g, corresponding methods are attached into M of R_g, and set copy of L in execution context (Line 18-26), like previous method dispatch, to direct refinement target to L of R_g instead of host R.

As code depicts, the composition of functions and aspects are made dynamically in runtime, and like function composition, the R can be composed of multiple, nested set of other R. With this structure, similar to additive refinement from function extension shrinking refinement is supported, by simply remove or replace existing elements in L via M which will be discussed in next part.

4.2 Aspect Refinement in Functional Granularity

In Sec. 3.1.1 and 3.2.1, we have showed FR can support both additive and shrinking type of refinement for refining functional aspect.

As we discussed In the previous section, function mutation essentially to modifying functional elements of the list in functional class. For example, In Sec. 3.2.1, we have showed the example of shrinking refinement, to dynamically attach or detach `queueSender` and `eventEmitter` advice with `delete` and `prepend` methods respect to the state message from the `Queue` and the `Event`. We also demonstrated shrinking refinement of `cflow` pointcut in aspect `PC` by invoking `update` method for replacement.

In this part, we showed how various shrinking aspect refinement in functional granularity is implemented based on the functional class design presented in the previous part.

4.2.1 Indirect, Encapsulated Mutation

Design. In FR, functions are lazily composed when and stored inside the list data structure. This enables FR to take arbitrary method-based mutation at runtime. On the left side of Fig. 6, $< M > G_F$ shows refinement methods for F, and $< M > F_A$ shows refinement methods for F_A which is composed in F. By concerning method as mutation, and functions list as a state, method-based refinement offers encapsulation and information hiding to improve interoperability of functional mutation [18]. As a result, mutation can be taken not just imperative and but programmatical manner. This possibly enables to use design patterns [39] like Factory or Proxy for constructing and refining functions. Furthermore, we can define custom refinement method by dispatching method list.

```
1  RF.prototype.before = function (fn, name) {
2    this.store._findAndUpdate(name, (index) => {
3        var elem = {name: fn.name, fn: fn};
4        this.store.list.splice(index, 0, elem);
5      });
6    return this;  };
7
8  RF.prototype.map = function (wrapper, name) {
9    this.store._findAndUpdate(name, (index) => {
10     var oriFn = this.store.list[index];
11     var newFn = wrapper(oriFn.fn);
12     newFn.name = name;
13     this.store.list[index] = {
14       name : name,
15       fn: newFn
16     };  });
17   return this;  };
18
19 RF.prototype.update = function (newFn) {
20   this._findAndUpdate(newFn.name, (index) => {
21     var elem = { name: newFn.name,
22                  fn: newFn };
23     this.store.list[index] = elem;
24   });
25   return this;  };
26
27 function Store (initLst) {
28   this.behaviors = Array.isArray(initLst) ? initLst :
         [];
29 };
30
31 Store.prototype._findAndUpdate = (name, callback) => {
32 this.list.forEach((fn, index) => {
33   if (fn.name === name) { callback(index); }
34 });  };
```

Figure 11. Implementation of Function Refinement.

Semantics. E-Mutates in Fig. 8 shows semantics of function mutation for `update` method which replaces R_t into R_g in L. To support refinement method for R_g, method list M is updated. Functional exportation allows defining user-defined mutation mechanism, using predefined advice typing, by embedding advice composition mechanism as a thunk of delayed evaluated function. As E-Exports shows semantics of `asAround`, which is done by checking R_t against L and push R_g into the position of R_t in L, after wrapping R_t with R_g denoted as λ. As left side of Fig. 6 shows, functional class

F has its own refinement methods in typed in M, which is divided into general and specific refinement method. General methods denoted $< M > G_F$ to perform refinement in absolute position, while specific methods denoted `S Mtd.` for refining specified function, by prefixing name of aspect in front of the method call like $R.R_t.update$. As both R and *store* are *has-a* relationship and tangled in as a single class, this refinement relationship is preserved, after R_F is composed to another functional class. Lastly, In E-Dispatch, depict semantics of custom refinement method that dispatching M after checking naming conflict.

Implementation. Fig. 11 shows implementation of function mutation methods in FR [6]. For example, with R_g and their name (Line 1) `before` method calls `findAndUpdate`. `findAndUpdate` is internal method in `this.store` is that returns index of target function R_t (Line 28-31). Using that index value, application of before advice done trivially by inserting R_g before R_t (Line 4), as the L is essentially array data structure, that can be constructed with other L in a case of inheritance (Line 27-29). The implementation of `map` and `update` takes similar to `before`, using the index, wrapping (Line 10-16) or replacing(Line 21-23) R_g to L. With functional class design, refining functional aspect is implemented by mutating items in an array.

4.3 Vanilla Runtime Implementation

As Sec. 4 shows, unlike traditional HoA and AR mechanism, FR does not require a compiler extension [23, 45, 64] or program transformation [63] in order to declare new syntax or special operational mechanism but applying built-in languages features of FP and OOP in a novel way. Like FR, uses of plain language promotes the applicability of external tooling as we discussed in Sec. 2. In this section, we introduce the coarse-grained design of FR, functional hierarchies, which enable inheritance and execution of the functional class to implement AR within vanilla runtime mechanism.

4.4 Functional Hierarchies

Design. Since FR adapt class mechanism, it naturally introduces inheritance hierarchy for functions by instantiating child functional class from parental functional class. Such hierarchical offer modularity benefit by inheritance. For instance, in Fig. 6, reusing commonaliites of functional class F by inheriting body function $< M > G_f$ and *List*. After inheritance, the function can be localized variabilities independently via function mutation.

Semantics. As E-Inherit in Fig. 8 shows, function inheritance instantiate M' and L', from M and L which is the independent copy of L and M. Execution of functional class also done by traversing breadth-first search from the functional

[6]We omitted validation and name set up and other non-functional part for space reason

```
1  RF.prototype.exec = async function exec (...input) {
2    for (fn of this.list) {
3        var hostFn = fn.fn;
4        if (hostFn !== null) {
5          if (typeof hostFn === 'function') {
6            result = hostFn.apply(null, input);
7            if (typeof result.then === 'function') {
8              input = await hostFn.apply(null, input);
9            } else {  input = [result]; }
10         } else if (hostFn instanceof RF) {
11           var context = this;
12           context['list'] = hostFn;
13           try {
14             input = await hostFn.exec.apply(context,
                     input);
15           } catch (e) { throw e; }  }  }
16       if (!Array.isArray(input)) {
17         input = [input];
18       }  }
19   return input;  };
```

Figure 12. Aspect composition and deployment in FR.

hierarchy. As E-Exec shows, by applying a R with given initial arguments *args*, iteration is performed by sequentially invoking function elements in L. The invocation rule Exec-Iteration which shows, FR invoke function L_n (either typed R or primitive), with *args* from the result of its previous function (L_{n-1}). The iteration ends when all elements in L are traversed.

Implementation. The Fig. 12 depicts implementation of function execution in R. The method first traverse L (Line 2). For each function, `hostFn`, is normally applied with given arguments `args`, if it is primitively typed or deferred function like Future or Promise [2] (Line 5 8). If `hostFn` is typed as R (Line 10), Recursive application of `exec` are made for `hostFn` (Line 14) to traverse and invoke new functional hierarchy in `hostFn`. In the application, setting L in the context of`hostFn` is important for apply functions against L of `hostFn` (Line 11-12). Fig. 13 depicts implementation of function inheritance. Inheritance of host R is made by constructing new R with by with the copy of L (Line 2). Later, standard and custom M is dispatched to new R (Line 10-4).

5 Related Works

In this section, We describe related works to FR from the area of extension-based AOP and runtime-based AOP for JavaScript, and AOP for functional languages.

AOP Extensions for JavaScript. AOP extensions for JavaScript approaches take into account preprocessing or program transformation method in order to integrate aspect with the host language. AOJS [67] supports AOP with proxy-based runtime weaving using practical pointcuts like variable assignment.

```
RF.prototype.new = function () {
  var child = new RF(this.store.list.slice());
  var props = Object.keys(RF.prototype).concat(Object.
       keys(this));
  props.forEach((trait) => {
    if (RF.prototype[trait]) {
      child[trait] = RF.prototype[trait];  }
    if (this[trait] && trait !== 'store') {
      child[trait] = this[trait];  }
  });
  return child;
};
```

Figure 13. Implementation of Function Inheritance.

Unlike AOJS, AsoectScript [63] support pointcut and advice in higher-order functions, AspectScript allows to define and compose advice and pointcut just like normal function composition, that is different from AOJS that uses statically defined XML external specification which degrades the ability of quantification and reasoning about the program. AspectScript supports dynamic aspect deployments and aspect propagation to fully tackle dynamic, HoA within JavaScript.

As these techniques rely on program transformation, inherently, aspect construction is only done at build time which makes difficult to support dynamic aspect refinement and application to dynamic join point. Also, such a transformation phase difficult to making static reasoning, without explicit tooling support.

AOP for Functional Languages. AOP for HoL is discussed in the context of functional language, including Scheme [17], Standard ML [53] and Caml [48].

AspectScheme [36] is a dynamically typed language. Like FR, pointcut and advice are normal functions, selected from an identity basis. FR covers all these features with dynamic refinement and join point.

AspectML [34] not using function identify-based pointcut, instead uses function name and agruments types to match functions in lexical scope. As discussed by AspectScript, relying on functions name in first-class functions are error-prone. Like AspectScript, FR uses identity-based functional pointcut in fine-grained composition, but in coarse-grained composition, FR uses name-based on pattern matching, which needs to improved using identity-based later.

Aspectual Caml [50] is Caml extension for AOP with type system integration, which currently FR does not study well. Designing aspect refinement based on types in HoL is important as many HoL supports static typing. Aspectual Caml supports pointcut matching over method name, which is similar to coarse-grained deployment in FR. Lastly, first-class pointcut and can advice are anonymous functions, which is not possible in FR, as refinement require to have named advice and pointcut as an anchor.

Three language-based approach has its own design specialties, such as type systems, however difficult to consult for AR in HoL.

Runtime AOP for JavaScript. For a runtime solution, much of approaches took the the wrapper pattern [39] based approach for AOP to functions and methods in JavaScript [5–8, 10–12], by taking functions as arguments and adds pre- or post-processing. While wrapping is the primitive mechanism to augment behaviors of functions, by simply intercepting argument and scope information, allowing dynamic functional construction in a higher-order manner. Runtime AOP for JavaScript is simple and lightweight by using built-in language features, and thus it can support dynamic AOP in partial. However, it is not quantifiable and not refinable, since composed functions are tangled.

6 Discussion and Future Work

Higher-order functions are popular traits of modern programming languages, much used in the construction of modern software systems [20, 40, 55]. Due to the rising dynamic web, mobile, and cyber-physical software systems built on top of HoL, we revisit aspect-orientation and step-wise refinement in terms of HoL, to apply AOP and SWR as a method to model and implement dynamic variabilities and requirements constructed from higher-order programs.

We present function refinement, a dynamically quantified AOP model for supporting fine-grained refinement over higher-order functions and aspects. In FR, we model functions as a class with sequential compositionality that comprised with function list and set of methods to extends, inherit and mutates its functional elements to refine a function, just like how we refining classes. We showed FR supports arbitrary additive and shrinking types of higher-order aspect refinement over dynamic join point at runtime. We explored the implementation of FR with its underlying mechanism in both levels of operational semantics and implementation codes. We draw correspondence and contributions of FR, related to traditional AOP techniques by comparing against three its design components that is recursive definition of a functional class to support the basis of refinement, and indirect, encapsulated mutation as a mechanism of refinement, and lastly, functional hierarchies to support inheritance for reuse and localization of cross-cutting concerns.

To improve conceptual and practical relevances, we are currently working on FR with stateful and symmetric AOP support, developing practical implementation with rich join point, and supporting runtime type checker for compositional safety.

Advanced AOP Support. Currently, aspects provided by FR are stateless, to support implicit cross-cutting requirements for dynamic adaptive behaviors of modern software, we are considering bringing stateful aspects [33] into functional aspect. As FR allows programmable refinements, which

are related to the dynamic software product line [42], we can further extend current asymmetric aspects concept, into symmetric aspects [43] for using FR to generating or synthesizing features in runtime from specification.

Practical Implementation. We are currently working on FR to provides more rich pointcut model and custom join point [59]. In order to support AR into multiple and scattered deployed aspects, remotely deployed aspects for refinement in global namespace are useful. Lastly, we are working on to provide more concrete implementation by tackling proposed features of FR, including support of `deploy`, `compose` methods and context synchronization between functions, which are implemented using primitive FR features but are not included in our current stable release of implementation.

Compositional Verification. Currently, FR is designed for dynamically typed JavaScript language, however many higher-order languages have typed, even JavaScript can achieve such enforcement using runtime type checking with third-party libraries [9, 14, 15]. As we discussed in Sec. 4, runtime type checker for JavaScript function with function subtyping rule [56] potentially useful for type-safe refinement and composition.

We showed that FR enables and offer the incremental refinement of higher-order functions and aspects, using built-in language features, to make functions have high reusability and tolerance to dynamic variabilities. As functions are the heart of higher-order languages, we believe enabling disciplined engineering for a function will flourish disciplined engineering for higher-order languages onwards.

References

[1] 2017. JavaScripe Programming Language. A lightweight interpreted or JIT-compiled programming language with first-class functions. (2017). Retrieved 2017-11-18 from https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] 2017. JavaScripe Promise. (2017). Retrieved 2017-11-18 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[3] 2017. Python Programming Language. A programming language that lets you work more quickly and integrate your systems more effectively. (2017). Retrieved 2017-11-18 from https://www.python.org/

[4] 2017. V8 JavaScript Engine. (2017). Retrieved 2017-11-18 from http://code.google.com/apis/v8/design.html

[5] 2018. Ajaxpect. A JavaScript framework for aspect-oriented programming. (2018). Retrieved 2018-06-24 from http://code.google.com/p/ajaxpect/

[6] 2018. AspectJS. A JavaScript framework for aspect-oriented programming. (2018). Retrieved 2018-06-24 from http://zer0.free.fr/aspectjs/

[7] 2018. aspect.js. JavaScript library for aspect-oriented programming using modern syntax. (2018). Retrieved 2018-06-24 from https://github.com/mgechev/aspect.js

[8] 2018. AspectJS. Proxies in JavaScript. (2018). Retrieved 2018-06-24 from http://www.aspectjs.com/

[9] 2018. assert. A runtime type assertion library. (2018). Retrieved 2018-07-01 from https://github.com/angular/assert/

[10] 2018. CernyJS. A javascript framework for method-call interception. (2018). Retrieved 2018-06-24 from http://www.cerny-online.com/cerny.js/

[11] 2018. Humax. A JavaScript framework for aspect-oriented programming. (2018). Retrieved 2018-06-24 from http://humax.sourceforge.net/

[12] 2018. Prototype. A JavaScript library that aims to ease development of dynamic Web applications. (2018). Retrieved 2018-06-24 from http://prototypejs.org/

[13] 2018. refinable.js. Refinable Functions Constructor for Aspect and Feature Modularity. (2018). Retrieved 2018-06-24 from https://hiun.org/refinable/

[14] 2018. RuntimeTypeChecks. Runtime type checks for JavaScript and TypeScript. (2018). Retrieved 2018-07-01 from https://github.com/vsavkin/RuntimeTypeChecks

[15] 2018. typify. Runtime type checking for JavaScript. (2018). Retrieved 2018-07-01 from https://github.com/phadej/typify

[16] Todd Abel. 2016. *ReactJS: Become a Professional in Web App Development.* CreateSpace Independent Publishing Platform, USA.

[17] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. 1998. Revised5 Report on the Algorithmic Language Scheme. *SIGPLAN Not.* 33, 9 (Sept. 1998), 26–76. https://doi.org/10.1145/290229.290234

[18] 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 (Onward! 2013).* ACM, New York, NY, USA, 101–116. https://doi.org/10.1145/2509578.2514738

[19] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* ACM, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

[20] Marc Andreessen. 2011. Why Software Is Eating the World. (2011), C2 pages.

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

[22] Sven Apel, Christian Kästner, Armin Grösslinger, and Christian Lengauer. 2009. Feature (De)Composition in Functional Programming. In *Proceedings of the 8th International Conference on Software Composition (SC '09).* Springer-Verlag, Berlin, Heidelberg, 9–26. https://doi.org/10.1007/978-3-642-02655-3_3

[23] Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake. 2007. Aspect refinement-unifying aop and stepwise refinement. *Journal of Object Technology* 6, 9 (2007), 13–33.

[24] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java Programming Language.* Addison Wesley Professional.

[25] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. 2008. A Theory of Aspects As Latent Topics. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08).* ACM, New York, NY, USA, 543–562. https://doi.org/10.1145/1449764.1449807

[26] Henk Barendregt. 1991. Theoretical Pearls: Self-interpretation in lambda calculus. *Journal of Functional Programming* 1, 2 (1991), 229âĂŞ233. https://doi.org/10.1017/S0956796800020062

[27] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2003. Scaling Step-wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03).* IEEE Computer Society, Washington, DC, USA, 187–197. http://dl.acm.org/citation.cfm?id=776816.776839

[28] D. Batory, J. N. Sarvela, and A. Rauschmayer. 2004. Scaling step-wise refinement. (June 2004), 355-371 pages. https://doi.org/10.1109/TSE.2004.23

[29] Andrew P Black, Kim B Bruce, and James Noble. 2016. The essence of inheritance. In *A List of Successes That Can Change the World.* Springer,

73–94.

[30] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*. ACM, New York, NY, USA, 303–311. https://doi.org/10.1145/97945.97982

[31] Daniel Brown and William R Cook. 2007. *Monadic memoization mixins*. Computer Science Department, University of Texas at Austin.

[32] Adrian Colyer and Andrew Clement. 2004. Large-scale AOSD for Middleware. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development (AOSD '04)*. ACM, New York, NY, USA, 56–65. https://doi.org/10.1145/976270.976279

[33] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. 2007. Stateful Aspects: The Case for Aspect-oriented Modeling. In *Proceedings of the 10th International Workshop on Aspect-oriented Modeling (AOM '07)*. ACM, New York, NY, USA, 7–14. https://doi.org/10.1145/1229375.1229377

[34] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. 2008. AspectML: A Polymorphic Aspect-oriented Functional Programming Language. *ACM Trans. Program. Lang. Syst.* 30, 3, Article 14 (May 2008), 60 pages. https://doi.org/10.1145/1353445.1353448

[35] Pierre-Charles David and Thomas Ledoux. 2006. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *International Conference on Software Composition*. Springer, 82–97.

[36] Christopher J. Dutchyn. 2012. AspectScheme: Aspects in Higher-order Languages. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12)*. ACM, New York, NY, USA, 65–66. https://doi.org/10.1145/2661103.2661110

[37] ECMA ECMAScript, European Computer Manufacturers Association, et al. 2017. Ecmascript 2015 language specification. (2017). Retrieved 2017-11-18 from https://www.ecma-international.org/ecma-262/6.0/

[38] Thomas Erl. 2005. *Service-oriented architecture*. Vol. 8. Prentice hall New York.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[40] Paul Graham. 2004. *Hackers & painters: big ideas from the computer age*. O'Reilly Media, Inc.

[41] Robrecht Haesevoets, Eddy Truyen, Tom Holvoet, and Wouter Joosen. 2010. Weaving the Fabric of the Control Loop through Aspects. In *Self-Organizing Architectures*. Springer, 38–65.

[42] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (April 2008), 93–95. https://doi.org/10.1109/MC.2008.123

[43] William H. Harrison, H. L. Ossher, and Peri L. Tarr. 2002. Asymmetrically vs . Symmetrically Organized Paradigms for Software Composition. In *IBM Research Report RC22685 (W0212-147)*.

[44] 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.

[45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, London, UK, UK, 327–353. http://dl.acm.org/citation.cfm?id=646158.680006

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*. Springer, 220–242.

[47] Ramnivas Laddad. 2009. *AspectJ in Action: Enterprise AOP with Spring Applications* (2nd ed.). Manning Publications Co., Greenwich, CT, USA.

[48] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2011. Language Objective Caml: Caml supports functional, imperative, and object-oriented programming styles. (2011).

[49] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. 2006. A Disciplined Approach to Aspect Composition. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*. ACM, New York, NY, USA, 68–77. https://doi.org/10.1145/1111542.1111554

[50] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. 2005. Aspectual Caml: An Aspect-oriented Functional Language. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 320–330. https://doi.org/10.1145/1086365.1086405

[51] Tom Mens, Kim Mens, and Tom Tourwé. 2004. Aspect-oriented software evolution. *ERCIM News* 58 (2004), 36–37.

[52] Michael S Mikowski and Josh C Powell. 2013. Single page web applications. *B and W* (2013).

[53] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The definition of standard ML: revised*. MIT press.

[54] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc.

[55] John K Ousterhout. 1998. Scripting: Higher level programming for the 21st century. *Computer* 31, 3 (1998), 23–30.

[56] Benjamin C Pierce. 2002. *Types and Programming Languages*. MIT press.

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

[58] Hridesh Rajan and Gary T. Leavens. 2008. Ptolemy: A Language with Quantified, Typed Events. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. Springer-Verlag, Berlin, Heidelberg, 155–179. https://doi.org/10.1007/978-3-540-70592-5_8

[59] Hridesh Rajan and Gary T Leavens. 2008. Ptolemy: A Language with Quantified, Typed Events. In *European Conference on Object-Oriented Programming*. Springer, 155–179.

[60] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[61] 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.

[62] R E Strom and S Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.* 12, 1 (Jan. 1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[63] Rodolfo Toledo, Paul Leger, and Éric Tanter. 2010. AspectScript: Expressive Aspects for the Web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1739230.1739233

[64] David B. Tucker and Shriram Krishnamurthi. 2003. Pointcuts and Advice in Higher-order Languages. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development (AOSD '03)*. ACM, New York, NY, USA, 158–167. https://doi.org/10.1145/643603.643620

[65] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM, New York, NY, USA, 227–242. https://doi.org/10.1145/38765.38828

[66] Meng Wang, Kung Chen, and Siau-Cheng Khoo. 2006. Type-directed Weaving of Aspects for Higher-order Functional Languages. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation*

and Semantics-based Program Manipulation (PEPM '06). ACM, New York, NY, USA, 78–87. https://doi.org/10.1145/1111542.1111555

[67] Hironori Washizaki, Atsuto Kubo, Tomohiko Mizumachi, Kazuki Eguchi, Yoshiaki Fukazawa, Nobukazu Yoshioka, Hideyuki Kanuka, Toshihiro Kodaka, Nobuhide Sugimoto, Yoichi Nagai, and Rieko Yamamoto. 2009. AOJS: Aspect-oriented Javascript Programming Framework for Web Development. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '09)*. ACM, New York, NY, USA, 31–36. https://doi.org/10.1145/1509276.1509285

[68] Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (April 1971), 221–227. https://doi.org/10.1145/362575.362577

[69] Xing Xie, Chong Wang, Li-Qun Chen, and Wei-Ying Ma. 2005. An Adaptive Web Page Layout Structure for Small Devices. *Multimedia Syst.* 11, 1 (Nov. 2005), 34–44. https://doi.org/10.1007/s00530-005-0188-1

A Syntax and Evaluation Contexts

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

Figure 14. Syntax and evaluation contexts of function refinement.

B List of Refinement Methods

Table 2. Standard Method of Function Refinement

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

[a] The keyword this denotes local context of an object, returned for method chaining.