Context Holders: Realizing Multiple Layer Activation Mechanisms in a Single Context-Oriented Language

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@is.titech.ac.jp

Testuo Kamina
University of Tokyo
kamina@acm.org

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@is.titech.ac.jp

Abstract

We propose LamFJ, a calculus for expressing various layer activation mechanisms in context-oriented programming languages. LamFJ extends FeatherweightJava with *context holders*, which are the abstraction of dynamic layer activation. By encoding programs with different layer activation mechanisms into a program manipulating context holders, LamFJ serves as a foundation to reason about interactions between different mechanisms. This paper presents a sketch of the context holders and encodings of existing layer activation mechanisms.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Context-oriented programming, layer activation mechanisms

1. Introduction

Context-oriented programming (COP) [4] is an approach to modularize behavioral variations of a program from the viewpoint of the *context* that changes during execution of a program. Changing behavior with respect to changes of the context is achieved by (1) changing the compositions of behavioral variations when the context changes and (2) finding a composition of behavioral variations and selecting a method body with respect to not only the dynamic type of the receiver object but also the composition when a method is called.

COP languages provide *partial methods* and *layers* to modularize behavioral variations and *layer activation mechanisms* to change the behavior of the program. A *partial method* in a *layer* describes a behavioral variation of a method. A *layer activation mechanism* specify how and when compositions of behavioral variations change.

Different layer activation mechanisms use different rules for finding a composition of behavioral variations, which can be explained as an analogy to scoping of variable bindings. ContextJ [9], JCop [1], and PyContext [11] employ *dynamic scoping*; EventCJ [7] and ContextErlang [10] employ *per-object activation*; and Subjective-C[3] employs *imperative activation*. In ContextJS [8], one can implement his or her own layer activation mechanism through meta programming in JavaScript.

FOAL '14, April 22, 2014, Lugano, Switzerland.

http://dx.doi.org/10.1145/2588548.2588552

```
1 class C{
2   void m(){}
3   layer L{ void m(){new C().m();} }
4 }
```

Figure 1. Example of a class with a layer and a partial method

```
1 C c=new C();
2 activate(L);
3 c.m();
4 deactivate(L);
5 c.m();
```

Figure 2. Example using imperative activation

In this study, we propose LamFJ, a calculus for expressing various layer activation mechanisms. It extends FeatherweightJava [6] with layers, partial methods, *context holders*, *contextual method invocation*, and *let-binding*. Using context holders enables us to not only realize each layer activation mechanism separately but also use several mechanisms in one programming language. In this sense, LamFJ serves as an intermediate language for COP languages and foundation to reason about interactions between different mechanisms.

In this paper, we sketch LamFJ and the context holders, and show how we can realize each layer activation mechanism and use several mechanisms by using them.

2. Activation Mechanisms

This section gives a brief overview of the three layer activation mechanisms in COP languages, namely imperative activation, per-object activation, and syntactic activation.

Figure 1 shows class `C` that is commonly used to explain each mechanism. Class `C` has base method `m` and layer `L`, which defines a partial method to `m` in class `C`. When method `m` is invoked on an object of `C`, the program goes to line 3 if layer `L` is active and to line 2 otherwise. The active layers are usually managed as a sequence $\mathtt{L}_1, \mathtt{L}_2, \cdots, \mathtt{L}_n$. Partial methods in $\mathtt{L}_i$ precede the ones in $\mathtt{L}_j$ if $i < j$, and $\mathtt{L}_i$ is activated more recently than $\mathtt{L}_j$ in most COP languages.

2.1 Imperative activation

In the imperative activation mechanism, there is exactly one sequence of active layers and its change affects every method dispatch process in the program. Figure 2 shows a simple example that uses imperative activation. Layer `L` is activated and deactivated in lines 2 and 4, respectively. The method invocation in line 3 and the subsequent method invocations execute partial method `m` in `L`

```
1 C c1=new C();
2 C c2=new C();
3 activate(L,c1);
4 c1.m();
5 c2.m();
```

Figure 3. Example using per-object activation

```
1 C c=new C();
2 with(L){ c.m(); }
```

Figure 4. Example using dynamic scope activation

Signatures	Behavior
`CH newCH()`	creates an empty context holder
`CH activate(Layer,CH)`	activates the layer
`CH deactivate(Layer,CH)`	deactivates the layer
`CH clone(CH)`	clones the context holder
`CH merge(CH,CH)`	merges the two context holders
`Obj dispatch(m,CH,`$\overline{\texttt{Obj}}$`)`	executes a partial or base method

Table 1. Operations on `CH`

(line 3 in Figure 2) because they appear after the layer activation of `L`. On the other hand, the method invocation in line 5 executes base method `m` (line 2 in Figure 1) because it appears after the layer deactivation of `L`.

2.2 Per-object activation

In the per-object activation mechanism, layers are activated on each object. To activate a layer, the programmer specifies the object along with the name of the layer. Figure 3 shows a simple example that uses per-object activation. Layer `L` is activated on object `c1` in line 3. The method invocation in line 4 executes the partial method `m` in layer `L` (line 3 in Figure 1) because `L` is active on object `c1`. The subsequent method invocation in partial method `L.m` (line 3 in Figure 1) executes base method `m` because `L` is not activated on the receiver objects. Similarly, the method invocation in line 5 executes `m`.

2.3 Dynamic scoping

In the dynamic scoping activation mechanism, layers are activated using a block structure, namely the `with`-block, and the layers are active during the execution of the body of the block. Figure 4 shows a simple example that uses dynamic scoping activation. Line 2 activates the layer `L` by using the `with`-block, and the method invocation in the body executes partial method `L.m`. Subsequent method invocations also execute the partial method, because they are evaluated during the execution of the body of the `with`-block.

3. Context Holders

In this section, we informally explain *context holders* and *contextual method invocation*, and how we can realize in LamFJ the layer activation mechanisms described in Section 2.

Context holders are instances of the abstract data type `CH` defined as a collection of pairs of an activation event ϵ and time `t` and six operations, namely `newCH`, `activate`, `deactivate`, `clone`, `merge`, and `dispatch`, which are summarized in Table 1. An activation event is either $\alpha_{\texttt{L}}$, denoting that layer `L` is activated, or $\delta_{\texttt{L}}$, denoting that `L` is deactivated. `newCH()` creates an empty context holder. `activate(L,h)` and `deactivate(L,h)` evaluated at `t` take a layer name `L` and context holder `h` and add a pair $(\alpha_{\texttt{L}}$`,t)` and $(\delta_{\texttt{L}}$`,t)` to `h`, respectively. `clone(h)` takes a context holder `h` and returns a deep clone of `h`. `merge(h,h')` takes two context holders `h` and `h'` and returns a new context holder that has every pair $(\epsilon$`,t)` in `h` or `h'`. `dispatch(m,h,`$\overline{\texttt{o}}$`)` takes a method name `m`, context holder `h`, and objects $\texttt{o}_1, \cdots, \texttt{o}_n$, and executes a base or partial method on $\texttt{o}_1$ with respect to the sequence of active layers

```
1  class C{
2    Obj m(CH h){...}
3    layer L{
4      Obj m(CH h){
5        new C().m⟨h⟩();
6      }
7    }}
8  Obj main(){
9    let h=newCH() in
10   let c=new C() in
11   activate(L,h);
12   c.m⟨h⟩(h);
13   deactivate(L,h);
14   c.m⟨h⟩(h)
15 }
```

Figure 5. Imperative activation using context holders

```
1  class C{
2    CH h=newCH();
3    Obj m(){...}
4    layer L{
5       Obj m(){
6         let c=new C() in
7         c.m⟨c.h⟩()
8       }
9    }}
10 Obj main(){
11   let c1=new C() in
12   let c2=new C() in
13   activate(L,c1.h);
14   c1.n⟨c1.h⟩();
15   c2.n⟨c2.h⟩()
16 }
```

Figure 6. Per-instance activation using context holders

computed from `h` as follows. Let $\epsilon_{\texttt{L}}$ and $\texttt{t}_{\texttt{L}}$ be the latest activation event on `L` and its time in `h`, respectively. Then in the resulting sequence of active layers, (1) every `L` such that $\epsilon_{\texttt{L}} = \alpha_{\texttt{L}}$ appears and (2) `L` appears in front of `L'` if $\texttt{t}_{\texttt{L}} \geq \texttt{t}_{\texttt{L}'}$.

A contextual method invocation is a method invocation that takes one context holder `h` along with the receiver and argument objects, i.e., $\texttt{e}_0\texttt{.m}\langle\texttt{e}_h\rangle\texttt{(e}_1\texttt{,}\cdots\texttt{,e}_n\texttt{)}$, where $\texttt{e}_h$ is an expression evaluated to a context holder and $\texttt{e}_0\texttt{,}\cdots\texttt{,e}_n$ are the arguments. The semantics of contextual method invocation $\texttt{e}_1\texttt{.m}\langle\texttt{e}_h\rangle\texttt{(e}_2\texttt{,}\cdots\texttt{,e}_n\texttt{)}$ is given by `dispatch(m,h,`$\overline{\texttt{o}}$`)` where $\texttt{o}_i$ is the value obtained by evaluating $\texttt{e}_i$ for each $i \in \{1, \cdots, n\}$ and `h` is the value obtained by evaluating $\texttt{e}_h$.

The following subsections show how we can realize each layer activation mechanism by using context holders in LamFJ.

3.1 Imperative activation

The imperative activation mechanism can be realized by creating exactly one context holder at the beginning of the execution of the program and using it for every method invocation and layer activation. In Figure 5, we rewrite class `C` (Figure 1) and Figure 2 to use context holders in this manner. In the declaration of class `C` (lines 1–7), the method invocation (line 5) in partial method `L.m` is changed so that it specifies the context holder `h`. Line 9 creates the context holder that are used by every layer activation and method invocation in the program. Lines 10–14 correspond to the code in Figure 2. To activate and deactivate layer `L`, the context holder `h` is explicitly specified (lines 11 and 13). The method invocations in lines 12 and 14 are also extended to specify the context holder `h` for method dispatching.

3.2 Per-object activation

The per-object activation mechanism can be realized by having a context holder for each object. In other words, we can get a program that uses the context holders by translating each object to a pair comprising the object and a context holder.

In Figure 6, we rewrite class `C` and Figure 3 to use context holders. Instead of using the pairs, each object of `C` has its context holder as a member (field `h` in line 2). Each method invocation to an object of `C` then uses the context holder by accessing its field `h` (lines 7, 14 and 15). Activating layer `L` on object `c1` (line 13) is also changed so that it specifies the context holder in `c1` instead of the object itself.

```
1  class C{
2    Obj m(CH h){...}
3    layer L{
4      Obj m(CH h){
5        new C().m⟨h⟩(h);
6      }
7    }}
```

```
8  Obj main(){
9    let h=newCH() in
10   let c=new C() in
11   let h'=clone(h) in
12   activate(L,h);
13   c.m⟨h⟩(h)
14 }
```

Figure 7. Dynamic scoping using context holders

```
1 C c=new C();
2 activate(L);
```

```
3 without(L){c.m();}
4 c.m();
```

Figure 8. Example using imperative activation and dyn. scoping

```
1  class C{
2    Obj m(CH gh,CH h){...}
3    layer L{
4      Obj m(CH gh,CH h){
5        let mh=
6            merge(gh,h) in
7        new C().m⟨mh⟩(gh,h);
8    }}}
9  Obj main(){
10   let gh=new CH() in
```

```
11   let h=new CH() in
12   let c=new C() in
13   activate(L,gh);
14   let h'=clone(h) in
15   deactivate(L,h);
16   let mh=merge() in
17   c.m⟨mh⟩(gh,h);
18   let mh=merge(gh,h') in
19   c.m⟨mh⟩(gh,h')
20 }
```

Figure 9. Imperative activation and dyn. scoping using context holders

3.3 Dynamic scoping

The dynamic scoping activation mechanism can be realized by creating a context holder for each dynamic scoping. `clone` plays an important role to yield the same behavior as the `with`-block using the context holders. Let `h` be a variable that is bound to a context holder for the current dynamic scope. Then, `with(L){o.m();}` is translated into the following code:

```
1 let h'=clone(h) in
2 activate(L,h);
```

```
3 o.m⟨h⟩();
4 let h=h' in ...
```

Line 1 backs up the context holder `h` for latter use. Line 2 activates layer `L` on `h` and line 3 invokes method `m` by using `h` so that layer `L` affects the method dispatching. At the point where the `with`-block closes, layer activation and deactivation on `h` are reverted by simply binding `h` to the context holder `h'` (line 4).

In Figure 7, we rewrite class `C` and Figure 4 in the above manner. The base and partial methods in class `C` take one context holder as the argument to propagate the context holder of the current dynamic scope. Line 9 creates the topmost context holder `h` by using `newCH`. Lines 2–12 correspond to the beginning of the `with`-block in Figure 4, and thus they create a backup context holder `h'` by using `clone` and activate layer `L` on `h`. The following method invocation (line 13) uses the context holder `h` as a parameter of method dispatching and the argument. Because `L` is active on `h`, it executes partial method `L.m`. The subsequent method invocations also execute `L.m` infinitely because they use the same context holder.

3.4 Mixed activation

This section shows that context holders enable us to use several activation mechanisms in one language thanks to `merge`.

Suppose we have a hypothetical COP language that employs imperative activation and dynamic scoping. Figure 8 is an example (see Figure 1 for the definitions of class `C` and layer `L`). It first creates an object of `C` and then activates layer `L` by using imperative activation (line 2). The `without`-block in line 3 deactivates layer `L` within the block. Therefore `L` is not active when `m` is invoked in line 3. On the other hand, `L` is active when `m` is invoked in line 4.

In Figure 9, we rewrite class `C` and the code in Figure 8 to use context holders. Base and partial methods `m` and `L.m` take two context holders, namely `gh` and `h`; `gh` records activation events fired using imperative activation; and `h` records activation events fired using dynamic scoping. In partial method `L.m` (lines 4–8), we use `merge` to merge the sequences of layer activation events fired using imperative activation and dynamic scoping. Line 7 specifies the merged context holder `mh` to invoke `m`. Method `main` activates layer `L` on `gh` (line 13) and deactivates `L` on `h` (line 15). After line 15 is executed, `gh`, `h`, and `h'` are $\{(\alpha_{\mathrm{L}}, \mathtt{t})\}$, $\{(\delta_{\mathrm{L}}, \mathtt{t}')\}$, and empty, respectively, where $\mathtt{t} < \mathtt{t}'$. The method invocation in line 17 executes base method `m` because it depends on context holder `mh`, which is $\{(\alpha_{\mathrm{L}}, \mathtt{t}), (\delta_{\mathrm{L}}, \mathtt{t}')\}$. On the other hand, the method invocation in line 19 executes `L.m` because the specified context holder `mh` is $\{(\alpha_{\mathrm{L}}, \mathtt{t})\}$.

4. Conclusions and Future Work

In this paper, we proposed LamFJ, a calculus for expressing various layer activation mechanisms in context-oriented programming languages. Context holders in LamFJ abstract dynamic layer activation and enable us to not only realize each layer activation mechanism separately but also use several mechanisms in one programming language. There are several directions for future work. One direction is to support implicit layer activation [2, 11]. Another direction is to develop a verification technique for context-oriented programs based on context holders. By considering context holders as resources, we think resource usage analysis [5] is applicable.

References

[1] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC '10*, pages 50–65, 2010.

[2] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: A fresh look at programming context-aware applications. In *Onward! '12*, pages 67–84, 2012.

[3] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: bringing context to mobile platform programming. In *SLE'10*, pages 246–265, 2011.

[4] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[5] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *TOPLAS*, 27(2):264–313, 2005.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[7] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[8] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, December.

[9] Michael Haupt Malte Appeltauer, Robert Hirschfeld and Hidehiko Masuhara. ContextJ: Context-oriented programming with java. *Computer Software*, 28(1):272–292, 2011.

[10] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, pages 191–202, 2012.

[11] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *ICDL '07*, pages 143–156, 2007.