Layer and Object Refinement for Context-oriented Programming in L

Robert Hirschfeld\textsuperscript{1,a)} Hidehiko Masuhara\textsuperscript{2,b)} Atsushi Igarashi\textsuperscript{3,c)}

Abstract: Context-oriented programming (COP) languages provide layers as an abstraction mechanism for modularizing context-dependent behavioral variations. While existing COP languages offer layers in addition to other constructs like classes asymmetrically, we propose an experimental language called L that removes such asymmetry. The design of L started from ContextFJ, our minimalistic COP language, with extensions for state and refinement. This proposal presents one such refinement mechanism as a first step towards a small yet practical COP kernel.

1. Introduction

This proposal builds on our previous efforts to design a COP language called L that tries to avoid asymmetry between modularity constructs such as classes and layers COP-based object-oriented systems [2].

When allowing layers to refine one another similarly and in addition to regular inheritance in common object-oriented programming systems, the resulting languages offer at least three different ways to compose object behavior: layer composition via the family of \texttt{with} constructs, inheritance between objects or classes, and inheritance between layers.

Examined in isolation, each of these features seems straightforward to both explain and apply. In combination, however, they show interactions that make their semantics quite complicated and lead to difficult problems that resemble those of multiple inheritance.

In several attempts to resolve those issues, we were left with systems that had a few reasonable properties with respect to object composition but that also displayed puzzling behavior in many rather common situations. And instead of following up on such complicated semantics, we decided to simplify our design by reducing the number of composition mechanisms.

Therefore the current version of L (also referred to as L\textsubscript{three} since this is our third version of L) provides (i) only a single means to activate another partial method definition provided by another layer composed via \texttt{with}, (ii) object and layer refinement, both confined to the refining layer, and (iii) flattening of both refinements including explicit conflict resolution if necessary [3].

In the following we will present a short but illustrative example to introduce object and layer refinement. After that, we will use a more abstract example to lay out some of the cases of interacting refinements.

From version to version we also experiment with L’s syntax. In this paper we make our language resemble some of the more dynamically typed ones – mainly for reasons of compactness. Since we assume the few language features to be self-explanatory, we only briefly describe them if needed.

2. Refinements

Refinements in L can be established both between layers and between objects. They allow for reusing existing partial definitions by importing them into the layer or object under consideration. Refinements in L are influenced by Traits [3] in that they \texttt{flatten} all imported definition and that they require explicit conflict resolution.

The flattening property asserts that all partial definitions imported from other layers or objects are treated as if they were implemented directly in the refining layers or objects.

Since there can be more than one source for importing such definitions, there is the possibility of importing similar definitions and with that a chance for conflicts that need to be resolved. L requires such conflicts to be resolved explicitly. For that, it provides means to alias or hide involved definitions.

In Listing 1 we adapted some code from our work on ContextFJ [1] to show the application of our newly introduced language constructs \texttt{refine}, \texttt{alias}, and \texttt{hide}.

Object \texttt{Person} is defined in layer \texttt{LPerson} and has one variable \texttt{name}, a corresponding setter method, and an implementation of method \texttt{toString()}, which simply returns the name of the person. Note that any object that does not refine another object explicitly refines object \texttt{Object} by default. With that the runtime can provide default behavior via a bootstrapping layer on start-up.

\textsuperscript{1} Hasso-Plattner-Institute, University of Potsdam, Germany
\textsuperscript{2} Tokyo Institute of Technology, Japan
\textsuperscript{3} Kyoto University, Japan
\textsuperscript{a)} hirschfeld@hpi.uni-potsdam.de
\textsuperscript{b)} masuhara@acm.org
\textsuperscript{c)} igarashi@kuis.kyoto-u.ac.jp
We decided to refine layer LPerson into layer LResidence using refine to extend its entities (here only object Person) with behavior related to residential information (Fig. 1). In the new layer, Person receives a new variable address with setter method and its own implementation of toString(). This version of toString() would like to make use of the imported code from Person in LPerson, which is however shadowed by the new implementation and with that unfortunately not available.

To resolve that situation, we provide an alias for toString() from LPerson named LPerson_toString(). Calling our aliased method allows the new toString() to activate LPerson’s definition from within LResidence.

Note that because of flattening there are no super calls.

In Listing 2 we provide two more layers named LStudent and LEmployment that we will use for dynamic sideways composition via the with (...) construct.

Layers LStudent and LEmployment provide two more properties (university and employer respectively) to our Person object. Both layers are intended to be used via with (...) and to activate other implementations of toString() from their own implementation using next() (similar to CLOS’ call-next-method or ContextFJ’s proceed()).

Calling next() from within an aliased method will proceed to the next method with the original name, that is the name of the method prior to its aliasing.

In Listing 3 we now create an instance of Person and send messages to it in the context of different layer compositions (Fig. 1).

Creating a new Person without any layers applied to computations in which it is involved will leave such instance with only the basic behaviors provided to all objects by the runtime.

After applying layer LResidence, both name and address are available to our instance atsushi. Invoking toString() calls the version directly implemented in LResidence which in turn calls the aliased version imported from LPerson by using its newly assigned name.

Adding LEmployment to the previous configuration will extend the set of properties of atsushi with employer.

Another means to resolve conflicts, namely the ones which arise if there are more than one sources for a definition, is to hide all unwanted implementations. The hide construct is used in the next section.

3. More Refinements

We now show on a rather synthetic example how refine between layers and classes on with (...) interact when all used in the same code base. For that we successively enhance our example with new partial definitions and conflict
resolutions. While we do not address all possible interactions, we hope to cover some of the interesting ones.

In Listing 4 we create two instances of O1 and O2 that we will use in our running example.

```javascript
var o1 = O1.new();
var o2 = O2.new();
```

Layer L1 in Listing 5 contains definitions for objects O1 and O2 with methods m1() to m4() implemented self-sufficiently. Activating any of the methods on any of the objects will yield a string describing in which layer and which object the particular method is located. Because there are no refinements in Listing 5, o1.m1() with L1 activated returns 'L1-O1-m1' (<1>).

Layer L2 in Listing 6 is a refinement of layer L1 from Listing 5. Since there are no new implementations for m4() and m5(), the ones received from L1 are available. Because of that, evaluating with (L2) { o1.m4(); } calls the m4() defined in O1 of L1 and so returns 'L1-O1-m4' (<2>). However, the new implementations for m1(), m2(), and m3() shadow the ones imported from L1. Evaluating with (L2) { o2.m1(); } executes m1() defined in O2 of L2 and returns 'L2-O2-m1' (<3>).

Layer and object refinements in Listing 7 are similar to those in Listing 6. In addition to that, L3 hides m4() originating from O1 in L1. If m4() is now called on o1 with L2, there will be an error since there is no m4() available – neither imported via refinements nor implemented directly (<4>). However, to make the implementation of m4() from O1 available via another name (here L1.m4()), we provide an alias. With that, evaluating with (L3) { o2.m4(); } leads to the execution of L1.m4(), the relabelled behavior originally defined on O2 in L1 (<5>).

Evaluating with (L4) { o2.m6(); } in Listing 8 yields 'L1-O1-m4' (<6>) because O1’s implementation of m4() in L4 is that imported from L1, which in turn is imported from O2 in L4 via its refinement of O1.

In Listing 9 we compose a sequence of layers into the system with L1 being the first layer activated and L4 the last. This illustrates with (L1, L2, L3, L4) { o1.m3(); } (<7>) will call L4’s m3(), which immediately proceeds via next() to L3’s m3(), which in turn proceeds via next() to L2’s m3(), which also proceeds via next() to the implementation of m3() in L1 where there is eventually an implementation that returns a result ('L1-O1-m3').

4. Related Work

L’s refinements are based ideas from Traits [3] such
layer L2 refines L1 {
    object O1 {
        m1 () {
            ↑ 'L2-O1-m1'
        }
        m2 () {
            ↑ m1();
        }
        m3 () {
            ↑ next();
        }
        // *** NO m4 () { ... }; ***
    }
    object O2 refines O1 {
        m1 () {
            ↑ 'L2-O2-m1'
        }
        m2 () {
            ↑ m1();
        }
        m3 () {
            ↑ next();
        }
        // *** NO m4 () { ... }; ***
        // *** NO m5 () { ... }; ***
    }
} with (L2) {
    o1.m4 (); // == > 'L1-O1-m4' // <2> from other
    o2.m1 (); // == > 'L2-O2-m1' // <3> replacement
} Listing 6

layer L3 refines L1 {
    alias {
        O2: m4 () -> L1_m4 ();
    }
    hide {
        O1: m4 ();
    }
    object O1 {
        m1 () {
            ↑ 'L3-O1-m1'
        }
        m2 () {
            ↑ m1();
        }
        m3 () {
            ↑ next();
        }
        // *** NO m4 () { ... }; ***
    }
    object O2 refines O1 {
        m1 () {
            ↑ 'L3-O2-m1'
        }
        m2 () {
            ↑ m1();
        }
        m3 () {
            ↑ next();
        }
        m4 () {
            ↑ L1_m4 ();
        }
        // *** NO m5 () { ... }; ***
    }
} with (L3) {
    o1.m4 (); // == > 'L3-O1-m4' // <4> imported alias
    o2.m6 (); // == > 'L1-O2-m1' // <5> via alias
} Listing 7

layer L4 refines L1 {
    object O1 {
        m1 () { ↑ 'L4-O1-m1' ;
        m2 () { ↑ m1();
        m3 () { ↑ next();
        // *** NO m4 () { ... }; ***
    }
    object O2 refines O1 {
        alias {
            L1: m4 () -> O1_m4 ();
        }
        hide {
            L1: m4 ();
        }
        m1 () { ↑ 'L4-O2-m1'
        m2 () { ↑ m1();
        m3 () { ↑ next();
        // *** NO m4 () { ... }; ***
        // *** NO m5 () { ... }; ***
        m6 () { ↑ O1_m4 ();
        }
    }
} with (L4) {
    o2.m6 (); // == > 'L1-O1-m4' // <6> imported alias
} Listing 8

with (L1, L2, L3, L4) {
    o1.m3 (); // == > 'L1-O1-m3' // <7> next()
} Listing 9

as flattening and explicit conflict resolution. Internal to the implementation of a particular layer, these properties allowed us to keep method lookup in our COP language simple and manageable.

5. Outlook
In COP layers are a set of partial behavioral definitions that are often composed at run-time. Each such partial behavioral definition can be implemented by several different means.

For $L_{three}$ we decided to allow for refinement relationships between both layers and objects to help to avoid code duplication.

With the three composition mechanisms available with $L_{three}$—layer refinement, object refinement, and layer composition—our design goals were to simplify method lookup and to avoid complicated interaction between these mechanisms. Flattening and manual conflict resolution via aliasing and hiding allowed us to gain confidence in achieving that goal.

However, $L_{three}$ is only one intermediate step and more of a concept design prototype than a practical artifact or theoretically sound model. We therefore will continue our work by both designing more interesting systems using $L$ and clarifying the theoretical foundations.

Acknowledgements
This paper is based upon work supported in part by the Japan Society for the Promotion of Science (JSPS) Invitation Fellowship Program for Research in Japan.

References