Visibility of Context-oriented Behavior and State in L

Robert Hirschfeld Hidehiko Masuhara Atsushi Igarashi
Tim Felgentreff

One of the properties of context-oriented programming languages is the composition of partial module definitions. While in most such language extensions the state and behavior introduced by partial definitions are treated equally at the module level, we propose a refinement of that approach to allow for both public and restricted visibility of methods and local and shared visibility of fields in our experimental language *L*. Furthermore, we propose a new lookup mechanism to reduce the risk of name captures.

1 Introduction

Context-oriented Programming (COP) is an approach to software modularity [8]. COP languages and systems provide constructs and mechanisms to combine and abstract behavioral variations, which can be activated and deactivated according to computational context at run-time.

Most COP language extensions are add-ons to other modularity mechanisms provided by the host language—usually classes in a contemporary object-orienged environment.

L is our exploration of the design of a COP language that tries to avoid asymmetry between module constructs for capturing partial or full implementations of system properties [10]. Instead of having COP constructs adding behavioral variations to a base system implemented using other composition mechanisms, *L* systems are based only on partial objects and layers.

So far, state and behavior introduced by different partial definitions are treated almost as if they originate from one and the same defining module, with the exception that they can be dynamically activated and deactivated depending on the execution context on an individual basis. Visibility control respects the constraints imposed by the host language, but usually does not go beyond that[†1].

In a more dynamic execution environment where composition units can be added to and withdrawn from the system at run-time, it seems desirable to assist developers with additional visibility constructs that allow them to describe which units of behavior and state to be made available to or hide from other parts of a composition.

With L_{four} (our fourth version of *L*) we propose a new visibility mechanism and lookup mechanism that allow for that.

In the following we give a short overview on how *L* developed and present our proposal in more detail by discussing an example and describing a revised

文脈指向言語 L における挙動と状態のアクセス制御
Robert Hirschfeld and Tim Felgentreff, Hasso Plattner Institute, University of Potsdam.
増原英彦, 東京工業大学大学院情報理工学研究科, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology.
五十嵐淳, 京都大学大学院情報学研究科, Department of Communications and Computer Engineering, Kyoto University.
コンピュータソフトウェア, Vol.32, No.3 (2015), pp.149–158.
[研究論文] 2014 年 10 月 1 日受付.
大会同時投稿論文

†1 JCop [1] is an exception in that it introduced the keyword `thislayer` to explicitly refer to a partial method definition provided by the same layer the currently executed code belongs to. While this allows to achieve a limited form of visibility control, it burdens the caller to be explicit about that for every single activation.

```
object Person {
  var name;
  var address;
  // constructors...
}
layer LPerson {
  object Person {
    toString() {
      ↑ "Name : " + name;
    }
  }
}
layer LResidence {
  object Person {
    toString() {
      ↑ next()
        + " Address : " + address;
    }
  }
}
```

Listing 1 L_{one} **Example.**

```
layer LPerson {
  object Person {
    var name;
    // setters...
    toString() {
      ↑ "Name : " + name;
    }
  }
}
layer LResidence {
  object Person {
    var address;
    // setters...
    toString() {
      ↑ next()
        + " Address : " + address;
    }
  }
}
```

Listing 2 L_{two} **Example.**

lookup mechanism in its support.

2 L So Far

L went through a series of design steps, each focusing on a particular aspect of the language. We now briefly describe the key properties of each version—simply named after their respective version numbers ranging from L_{one} to L_{four}—using a running example.

L_{one} [10] was our first attempt to work on a symmetric approach to modularity in a layer-based language. Here we moved all partial method definitions into layers, leaving only the declaration of state to object definitions forming the base layer. In L_{one} methods are public and state is private but shared among all partial method definitions of an object. While this already removed asymmetry in the definition of behavior, it still left us with a base layer with respect to state.

In Listing 1, object `Person` defines *all* of its fields, here `name` and `address`—*centrally* and *outside of any layer*. Behavior—here the generation of a `String` representation via `toString`—is partially defined in layers `LPerson` and `LResidence`.

In L_{two} [10], we removed the concept of a base layer entirely by moving state declarations into partial object definitions. We decided that all such declarations need to be repeated in all partial object definitions that access the respective state—to both remove centers and dependencies on them as formerly introduced by base layers and help programmers understand a particular code fragment by providing necessary information as close as possible. All declarations of fields using the same name refer to the same (shared) state, both allowing and requiring shared field definitions distributed across layers. Due to the distributed nature of state declaration we gave up on the notion of constructors but assume state to be initialized via regular accessors where appropriate.

In Listing 2, all field declarations are now located in partial object definitions that require access to the respective fields—here the part of object `Person` defined in layer `LPerson` and also another part of the same object defined in layer `LResidence`.

With L_{three} [11], we introduced refinement relationships for both layers and objects to allow for code sharing. We tried to avoid many of the problems associated with multiple inheritance and mixins [5] by explicit conflict resolution at development-time via static Traits-like flattening [13]. Flattening ensures that all partial definitions imported from other layers or objects are treated as if they were implemented directly in the refining layers or objects. Conflicts are resolved by selectively aliasing or hiding definitions.

In Listing 3, layer `LResidence` statically refines layer `LPerson` by adding to the implementation of its `toString` method of object `Person`. Since there cannot be more than one version of `toString` offered by one partial object definition, we alias the partial method definition originating from `LPerson` in `LResidence` as `LPerson_toString` and with that resolve a name conflict.

With the visibility control of behavior and state

```
layer LPerson {
  object Person {
    var name;
    // setters...
    toString() {
      ↑ "Name : " + name;
    }
  }
}
layer LResidence refines LPerson {
  alias {
    Person :
      toString() -> LPerson_toString();
  }
  object Person {
    var address;
    // setters...
    toString() {
      ↑ LPerson_toString()
        + " Address : " + address;
    }
  }
}
```

Listing 3 L_{three} **Example.**

```
layer LPerson {
  object Person {
    shared var name;
    // setters...
    public toString() {
      ↑ "Name : " + name;
    }
  }
}
layer LResidence {
  object Person {
    shared var name;
    local var address;
    // setters...
    public toString() {
      ↑ next()
        + formattedAddress();
    }
    restricted formattedAddress() {
      ↑ " Address : " + address;
    }
  }
}
```

Listing 4 L_{four} **Example.**

as introduced in L_{four}, our example can be implemented as shown in Listing 4. An explanation of how `local`, `shared`, `public`, and `restricted` work is provided in the remainder of the paper.

3 Visibility

We want to control visibility for both behavior and state with respect to the objects and layers they are defined in and used. (In this text we use the terms behavior/methods and state/fields interchangebly.)

3.1 Behavior

For behavior we want to allow for partial method definitions to contribute to the public interface of an object, either by changing the way the object responds to a message received or by allowing an object to respond to entirely new messages it was not able to understand so far.

On the other hand, we want to explicitly mark methods to be only accessible from within a set of partial definitions. To not complicate matters unnecessarily, we want to ensure that the visibility of methods can be restricted to a particular layer implementation.

We introduce two method markers named *public* and *restricted—public* for partial method definitions of the former kind and *restricted* for the latter.

Methods marked *public* can be accessed not only from anywhere in the layer they are defined in but also from any other partial definition in any other layer, assuming that the current layer composition—usually established by a sequence of preceding `with` or `without` constructs—was set up accordingly.

If a method is marked *restricted*, it can be activated only if the corresponding message received originated from the same layer that defines that method. It is important to note that *restricted* methods cannot be called via `next`, or put differently, methods from one layer can only proceed to *public* methods if available. (`next` [11] is similar to CLOS' `call-next-method` [14] or ContextFJ's `proceed`) [10] in that it invokes the next available partial method defintion of the same name and signature in the current composition.)

Also, *restricted* methods, when called within their defining layer, have precedence over other public partial definitions of the same method from outer layers to prevent name captures.

For example as shown in Figure 1, method `m2` of object `O1` and layer `L3` calls `m1` and `m3` directly since both methods are declared restricted and in the same layer as the calling `m2` and so are considered first. (The code accompanying the composition in Figure 1 is presented in Listings 5 to 8 of Section 4.)

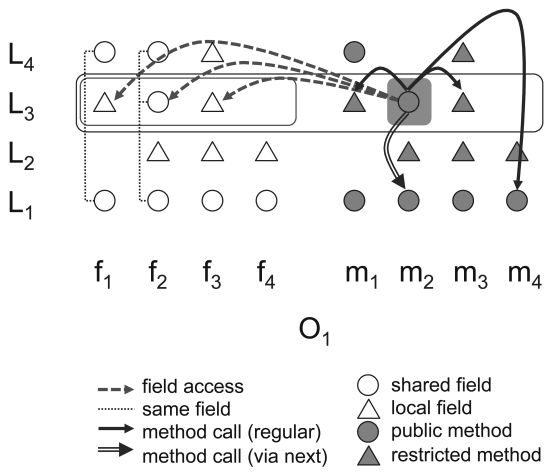

Fig. 1 Composition `with (L4 L3 L2 L1).`

3.2 State

For state we want to enable the interaction of partial definitions via side effects since sharing fields between methods has been shown convenient and is common practice in object-oriented programming from its inception.

But we also want to confine state so that it is only visible from within a particular partial definition—for now from within a partial object definition.

We provide two field markers named *shared* and *local*—*shared* for field declarations of the former kind and *local* for the latter.

Fields marked *shared* can be accessed from all partial definitions of an object if declared there as such. All these partial definitions then refer to one and the same field meaning that, if changed from within one partial definition, this modification is the same to all partial definitions that participate in sharing that field.

From outside a partial object definition, all fields marked *local* cannot be accessed directly. With that, state is not only confined to a particular object but also to a specific layer.

In Figure 1 one can see that only the fields declared in `L3` (`f1`, `f2`, and `f3`) can be accessed from methods defined there.

When layered, field declarations can shadow each other. While there is no such mechanism like `next` for fields, sharing resembles `next` since it allows implicit state propagation in-between participating partial definitions. However, there is an important difference between the two: While control propagation via `next` can only proceed to layers more inner than the current one, state propagation (via side effects) goes both inward and outward.

Note that the visibility of fields is orthogonal to their lifetime and that local fields keep their values if their defining layers are inactive.

4 Composition Example

We illustrate some of the consequences of the application of *public*/*restricted* and *shared*/*local* in the following rather abstract example. We show both a textual representation and a corresponding visual illustration of a composition of layers of partial object definitions.

The values assigned to fields and returned from methods are strings encoded as follows: 'L' stands for layer, 'O' for object, 'f' for field, and 'm' for method. And the digits following the latter further qualify the respective layer, object, field, or method. For example, 'L3O1f2' was assigned to field `f2` of object `O1` in layer `L3` and 'L4O1m3' is returned from method `m3` of object `O1` in layer `L4`.

Layer `L1` (Listing 5) provides partial definitions for object `O1`. There are four variables `f1`, `f2`, `f3`, and `f4`, that can be set via the public method `setVars`, and three more methods `m2`, `m3`, and `m4`. All fields are shared fields so they can be altered not only from within `O1` of `L1`, but also by other partial definitions of `O1`, assuming that `f1` is declared there to be a public field also. All methods are public methods and with that contribute to `O1`'s public interface—that is, the interface that can be accessed from outside of `L1` if `L1` is activated.

Layer `L2` (Listing 6) defines only local fields (`f2`, `f3`, and `f4`) and restricted methods (`m2`, `m3`, and `m4`) for object `O1` so that `L2`'s partial definition of `O1` cannot directly cause side effects with other partial definitions of `O1` and does not add to `O1`'s public interface.

Layers `L3` and `L4` (Listing 7) provide another mix of shared and local fields and public and restricted methods to make the composition discussed more interesting.

While the assignment of fields and the implementation of methods follows the simple pattern mentioned above, method `m2` of object `O1` in layer `L3` (`L3.O1.m2`) is different in that it returns a string

```
layer L1 {
  object O1 {
    shared var f1, f2, f3, f4;
    public setVars(_f1, _f2, _f3, _f4) {
      f1 = _f1;
      f2 = _f2;
      f3 = _f3;
      f4 = _f4;
    }
    public m1() { ↑ 'L1O1m1'; }
    public m2() { ↑ 'L1O1m2'; }
    public m3() { ↑ 'L1O1m3'; }
    public m4() { ↑ 'L1O1m4'; }
  }
}
```

Listing 5 Layer L1.

```
layer L2 {
  object O1 {
    local var f2, f3, f4;
    public setVars(_f2, _f3, _f4) {
      f2 = _f2;
      f3 = _f3;
      f4 = _f4;
    }
    restricted m2() { ↑ 'L2O1m2'; }
    restricted m3() { ↑ 'L2O1m3'; }
    restricted m4() { ↑ 'L2O1m4'; }
  }
}
```

Listing 6 Layer L2.

that assembles the values of all of the fields accessible from that method and all of the return values of the methods that can be called from partial method `m2` defined in layer `L3` for object `O1`. Also, `m2` is declared public and so can be called also from code outside of `L1`.

We compose the layers described above using a sequence of `with` statements with interspersed method calls for setting newly introduced instance variables or fields (Listing 8).

We now explain the visibility of fields and partial methods as the system composition evolves. After the activation of layer `L1` and the initialization of the instance variables accessible from object `O1`'s definition in `L1` (`<1>`), the values that can be retrieved from fields `f1`, `f2`, `f3`, and `f4` are the ones set via the preceeding invocation of `setVars`.

The situation after activating layer `L2` `<2>` is similar. Here it is important to note that partial definitions introduced by `L2` for object `O1` can only access fields `f2`, `f3`, and `f4` but not `f1` since `f1` is not declared for `O1` in `L2`.

With the activation of layer `L3` and the initialization of the fields declared for object `O1` (`<4>`), we can again only access fields explicitly declared

```
layer L3 {
  object O1 {
    shared var f2;
    local var f1, f3;
    public setVars(_f1, _f2, _f3) {
      f1 = _f1;
      f2 = _f2;
      f3 = _f3;
    }
    restricted m1() { ↑ 'L3O1m1'; }
    // *** point of interest ***
    public m2() {
      ↑ '*' + f1 + '_' + f2 + '_' + f3
        + '=' + m1() + '_' + next() + '_'
        + m3() + '_' + m4() + '*';
    }
    restricted m3() { ↑ 'L3O1m3'; }
  }
}

layer L4 {
  object O1 {
    shared var f1, f2;
    local var f3;
    public setVars(_f1, _f2, _f3) {
      f1 = _f1;
      f2 = _f2;
      f3 = _f3;
    }
    public m1() { ↑ 'L4O1m1'; }
    restricted m3() { ↑ 'L4O1m3'; }
  }
}
```

Listing 7 Layers L3 and L4.

```
local var o = new O1();
with (L1) {
  o.setVars(
    'L1O1f1', 'L1O1f2', 'L1O1f3', 'L1O1f4');
  // <1>
  with (L2) {
    o.setVars(
      'L2O1f2', 'L2O1f3', 'L2O1f4');
    // <2>
    wiht (L3) {
      o.setVars(
        'L3O1f1', 'L3O1f2', 'L3O1f3');
      // <3>
      o.m2();
      // => '*L3O1f1_L3O1f2_L3O1f3=\
      // L3O1m1_L1O1m2_L3O1m3_L1O1m4*'
      wiht (L4) {
        o.setVars(
          'L4O1f1', 'L4O1f2', 'L4O1f3');
        // <4>
        o.m2();
        // => '*L3O1f1_L4O1f2_L3O1f3=\
        // L3O1m1_L1O1m2_L3O1m3_L1O1m4*'
      }
      // <5>
      o.m2();
      // => '*L3O1f1_L4O1f2_L3O1f3=\
      // L3O1m1_L1O1m2_L3O1m3_L1O1m4*'
    }
    // <6>
  }
  // <7>
}
```

Listing 8 Composition.

in that partial definition (L3 and O1). But here field f2 is marked shared and so shares its value of other public fields of the same name declared in other partial definitions of O1—currently (<4>) in L1 and later (<5>) L4.

When calling m2 at <4>, we activate m2 of O1 in L3 (L3.O1.m2), which will construct a string showing the content the fields are referring to and the return values provided by the other methods callable from there. (Even though m2 could call m2, it does not in our example to avoid infinite recursion.)

After adding layers L3 and L4 to our composition (see Figure 1 for a more visual illustration), calls to method m2 show the values of all accessible fields and the return values of all methods callable from m2 with L3 and L4 activated. As with the previous compositions, fields f1, f2, and f3 hold the content just assigned via setVars (all starting with 'L3'). For the methods, calling m1 and m3 from L3.O1.m2 invokes restricted methods m1 and m3 from the same layer m2's definition is located. The invocation of next from L3.O1.m2 will skip L2.O1.m2 since it is restricted to L2.O1 (!) and proceed to the next public version of m2, which is the one defined in L1.O1.

With the activation of layer L4 (<4>), the value returned by m2 is based on the current values of L3.O1.f1, L3.O1.f2, and L3.O1.f3. This is because the m2 executed is that found in L3.O1 and so m2 has only access to fields defined by L3.O1. Here, 'L4O1f2' is the value set for L4.O1.f2 after activating L4 (<4>), thus 'L4' at the beginning of the string. Since L4.O1.f2 and L3.O1.f2 (and also L1.O1.f1) are all declared shared, assignments to any one of them will also be an assignment to all of them!

After leaving the scope of L4—now with L3 being the outermost layer of our composition (<5>) again—the result of calling m2 from here shows that the side effect caused by L4 via the assignmet to f2 shared between L4.O1, L3.O1, and L1.O1 was preserved across layer activation/deactivation.

At <6> the values of f2, f3, and f4 were not affected by side effects since all of L2.O1's fields are local. However, at <7> with all fields of L1.O1 being shared, previous assignmets initiated from partial definitions located in other layers but L1 show also in L1.O1.

5 Lookup

One of the core mechanisms of COP language extensions is the method lookup in layer compositions. This lookup mechanism corresponds roughly to the one employed by plain object-oriented programming languages such as Smalltalk [7]. Here, the system starts its search for a method to be executed in response to a message received in the class of the receiver object.

The mechanism employed in most of the COP systems including ours work informally as follows: For each message received by an object the lookup tries to find a matching method implementation starting from the outermost layer of the current layer composition. If such method is found, it is invoked. With the exception of next, which proceeds to the next layers closer to the object to find a partial method with the same name and invokes that implementation if found, all subsequent message sends (including the ones sent to the current receiver object itself) cause the lookup mechanism to start over from the outermost to the innermost layer until a corresponding method implementation is found or the end of the lookup chain is reached (which usually leads to a run-time error to be dealt with by the system).

If employed as described above, our lookup leaves us with a high risk of name captures of restricted methods by other public methods with the same name introduced by other layers that contribute to the same object and composed after.

In our example in Listing 8 at <4>, if lookup would follow the algorithm described above, the call to m1 originating from m2 (L3.O1.m2) would start from the outermost layer, here L4, and find its public partial implementation of m1 of O1 in L4 (Figure 2). This might lead to surprises since the intent of declaring L3.O1.m1 to be restricted is to make sure that, while callable from L3.O1.m2 (same layer), it can neither be invoked directly from outside of L3 nor the invocation be taken over by outer code.

For L_{four} we changed partial method lookup as follows: (1) First check if the method to be called is implemented as a restricted method in the same layer as the method from which the message was sent. (2) If there is such a method, continue execu-

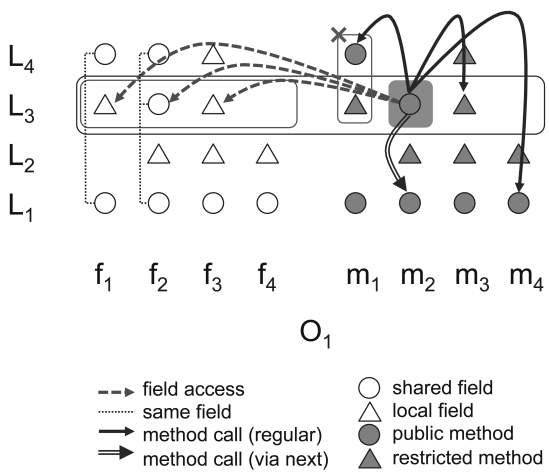

Fig. 2 Old Lookup.

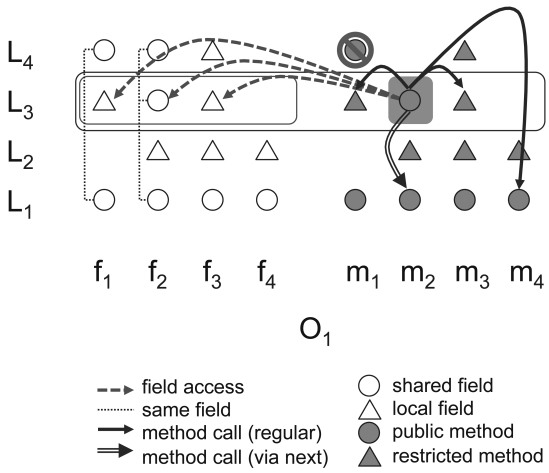

Fig. 3 New Lookup.

tion with that method, (3) otherwise proceed with the lookup starting with the outermost layer of the current composition.

This change was inspired again by the lookup of Smalltalk—in this case in the context of messages sent to `super` instead of `self`. As in many object-oriented languages, `self` (or `this`) and `super` refer to the receiver of a message. The difference with messages sent to `super` is that lookup does not start in the class of the receiver but in the superclass of the class that implements the method where the message send originates from, even if that particular method has been overridden in one or more subclasses.

With that new lookup in place, name captures like the one described above can be avoided (Figure 3).

6 Related Work

Scala's traits [12] can be considered partial object definitions without dynamic activation (that is, the application of a trait must be specified statically before an instantiation of an object). Even though a trait can override a public method of the base class, it can only do so when this trait and the base class extend the same interface that declares the method to be overridden. This means that overriding public methods is possible only for cases known and planned for in advance. In addition, a trait cannot declare a restricted method when a base class declares a public method with the same name, even if that name is not in a shared interface.

An instance variable or state declaration in Scala has two roles: creation of accessors and allocation of a store location. Since Scala treats accessors as regular methods, they follow the visibility rules for methods. Consequently, it is not possible in Scala for two modules to declare a shared instance variable without having a common interface in advance.

Ruby's modules [6] can be considered partial object definition without dynamic activation. Definitions in a module included later always precede the ones of modules included earlier; even if definitions are restricted. Surprisingly, restricted methods cannot be accessed even from the methods in the same module.

Instance variables in Ruby (those accessed with the `@` mark before their names) are always shared among modules. There is no visibility control mechanism except explicitly declared accessor methods.

Python's multiple inheritance mechanism [16] can be regarded a composition of partial class definitions. Method visibility in Python is achieved by naming conventions and renaming. When a class declares a method with a name beginning with double underscores, the name becomes unique to that class even if other classes used the same name including underscores. This makes name clashes between public and restricted methods impossible, at the cost of inconvenience to the programmers.

Instance variables in Python are treated the same

as methods. Leading double underscores make the name unique to a class, which effectively makes it restricted. Otherwise, instance variables are basically shared.

A comparison of access policies supported by Scala, Ruby, and Python is provided in Table 1 in Appendix 7.

Stateful traits [3] [4] are an extension to stateless traits [13]. Instance variables introduced by a stateful trait are considered to be private to that trait by default but can be made accessible to an importing client. Compared to *L*, this mechanism is static, whereas visibility constrains on state in L_{four} are considered when composing layers and with that state access at run-time.

7 Discussion and Outlook

For explaining the application of *public* and *restricted* for methods and *shared* and *local* for fields, we decided to always use these keywords everywhere in our code examples. We are aware of the verbosity such keywords introduce and so suggest as the *default* to assume methods to be restricted and fields to be local if not marked otherwise.

To avoid confusion with other established uses of *private* as an access modifier in languages like Java [2] or C++ [15], we are reviewing alternative names, but so far have not decided yet.

To reduce verbosity even more, we are considering the following replacements in future versions of *L*: `+` for *public*, `-` for *restricted*, `*` for *shared*, and `/` for *local*.

Another simplification we are contemplating is the use of *shared* (`+`) and *local* (`-`) not only for fields but also for methods.

Furthermore, the mechanisms to control visibility need to be integrated with our proposal on layer and object refinement [11]. Also, adding visibility control to objects and layers and its interaction with that of behavior and state needs to be investigated in future versions of *L*.

To allow for partial definitions to provide an interface that cannot be layered any further once activated, we are investigating some form of *final* to allow for that at the level of methods, objects, and layers.

After our rather informal investigation of possible language designs, we need to work on both *L*'s foundations [9] for clarifying some of our ideas and on implementations to better understand their applicability.

Acknowledgments

This paper is based upon work supported in part by the Hasso Plattner Design Thinking Research Program (HPDTRP) and SAP's Communications Design Group (CDG).

References

[1] Appeltauer, M., Hirschfeld, R. and Lincke, J.: Declarative Layer Composition With the JCop Programming Language, *Journal of Object Technology*, Vol. 12, No. 2(2013), pp. 4:1–37.

[2] Arnold, K., Gosling, J. and Holmes, D.: *The Java Programming Language, 4th Edition*, Addison-Wesley, 2005.

[3] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R.: Stateful Traits, in *Proceedings of IWST'07*, Lecture Notes in Computer Science, Vol. 4406, Springer, 2007, pp. 66–90.

[4] Bergel, A., Ducasse, S., Nierstrasz, O. and Wuyts, R.: Stateful Traits and Their Formalization, *Computer Languages, Systems and Structures*, Vol. 34, No. 2–3(2008), pp. 83–108.

[5] Bracha, G.: *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*, PhD Thesis, University of Utah of Utah, 1982.

[6] Flanagan, D. and Matsumoto, Y.: *The Ruby Programming Language*, O'Reilly, 2008.

[7] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[8] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-oriented Programming, *Journal of Object Technology*, Vol. 7, No. 3(2008), pp. 125–151.

[9] Hirschfeld, R., Igarashi, A. and Masuhara, H.: ContextFJ: A Minimal Core Calculus for Context-oriented Programming, in *Proceedings of FOAL'11*, ACM, 2011.

[10] Hirschfeld, R., Masuhara, H. and Igarashi, A.: L—Context-oriented Programming With Only Layers, in *Proceedings of COP'13*, ACM, 2013.

[11] Hirschfeld, R., Masuhara, H. and Igarashi, A.: Layer and Object Refinement for Context-oriented Programming in L, in *Proceedings of 95th IPSJ Workshop on Programming*, IPSJ, 2013.

[12] Odersky, M.: The Scala Language Specification Version 2.9, Technical report, Programming Methods Laboratory LAMP, EPFL, 2014.

[13] Schaerli, N., Ducasse, S., Nierstrasz, O. and Black, A. P.: Traits: Composable Units of Behaviour, in *Proceedings of ECOOP'03*, Lecture Notes in Computer Science, Vol. 2743, Springer,

2003, pp. 248–274.
[14] Steele Jr., G. L.(ed.): *Common Lisp: The Language, 2nd Edition*, Digital Press, 1990.
[15] Stroustrup, B.: *The C++ Programming Language, 4th Edition*, Addison-Wesley, 2013.
[16] van Rossum, G. and Drake Jr., F. L.: The Python Language Reference, Technical report, Python Software Foundation, 2014.

Appendix A

Table 1 compares access control policies in L_{four}, Scala, Ruby, and Python, which are discussed in Section 6. When there are member (either M(ethod) or S(tate)) definitions of the same name in a base module and a overriding module at the same time with different access modifiers (either pub(lic), priv(ate), shar(ed), or local), each table entry indicates which definition or store-location is accessed from different methods. The three letters separated by a slash correspond to the module in which the accessing method is defined, namely the base module, the overriding module, or another module, in this order. The letters, either B(ase), O(verriding), U(nique), or – (prohibited) denote the module that the accessed member belongs to.

For example, the second column (M priv) at the first row (M pub) in the L_{four} table has "B/O/B", meaning "when base and overriding objects respectively define public and private method with the same name, a call on the method dispatches to the method defined in the base object unless the method call is performed by the overriding object."

base (overridden) member		overriding member: M pub	M priv	S shar	S local
L_{four}					
M	pub	O/O/O	B/O/B		
M	priv	B/O/O	B/O/–		
S	shar			U/U/–	U/O/–
S	local			B/U/–	B/O/–
Scala		pub	priv	pub	priv
M	pub	O/O/O†1	†2		
M	priv	B/O/O	B/O/–		
S	pub			†2	†2
S	priv			B/O/O	B/O/–
Ruby		pub	priv	(default)	
M	pub	O/O/O	–/–/–		
M	priv	O/O/O	–/–/–		
S	(def)			U/U/U†3	
Python		pub	priv	pub	priv
M	pub	O/O/O	B/O/O		
M	priv	O/B/B	B/O/–		
S	pub			O/O/O	B/O/O
S	priv			O/B/B	B/O/–

†1 The overriding module must be defined with the "override" modifier as well.
†2 Compile error.
†3 No access control modifiers for instance variables that are accessed through variables with an at-mark (@).

Table 1 Comparison of access policies.

Robert Hirschfeld

Robert Hirschfeld (hirschfeld@hpi.de) is a professor of Computer Science at the Hasso Plattner Institute at the University of Potsdam, Germany. He is interested in improving the comprehension and design of software systems. Robert enjoys explorative programming in interactive environments. He served as a visiting professor at the Tokyo Institute of Technology and The University of Tokyo, Japan. Robert was a senior researcher with DoCoMo Euro-Labs, the European research facility of NTT DoCoMo Japan, where he worked on infrastructure components for next generation mobile communication systems with a focus on dynamic service adaptation and context-oriented programming. Prior to joining DoCoMo Euro-Labs, he was a principal engineer at Windward Solutions in Sunnyvale, California, where he designed and implemented distributed object systems, consulted in the area of object database technologies, and developed innovative software products and applications. Robert studied engineering cybernetics and computer science at Ilmenau University of Technology, Germany. (See also http://hpi.de/swa/)

Hidehiko Masuhara

Hidehiko Masuhara is a Professor at the Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.Sc., M.Sc., and Ph.D degrees from Department of Information Science, University of Tokyo in 1992, 1994, and 1999, respectively. His research interest is in programming

languages and programming environments, especially advanced modularization mechanisms, optimization techniques, code recommendations, and debuggers.

Atsushi Igarashi

Atsushi Igarashi is a Professor at Dept. of Communication and Computer Engineering, Graduate School of Informatics, Kyoto University. He received his B.Sc., M.Sc., and Ph.D degrees from Department of Information Science, University of Tokyo in 1995, 1997, and 2000, respectively. His major research interest is in principles of programming languages. He received the 20th IBM Japan Science Prize in Computer Science in 2006 and Dahl–Nygaard Junior Prize in 2011. He is a member of ACM, IEEE Computer Society, JSSST, and IFIP TC2 WG2.11 and served as Chair of SIG-PPL, JSSST from 2009 to 2012.

Tim Felgentreff

Tim Felgentreff (tim.felgentreff@hpi.de) is a Ph.D student at the Software Architecture Group at the Hasso Plattner Institute (HPI) at the University of Potsdam and a member the HPI Research School for Service-Oriented Systems Engineering since. His research interests are around programming language constructs and virtual machines. (See also http://hpi.de/swa/people/felgentreff/.)