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 [4]. 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 an 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 [6]. 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 deactivate 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.

In a more dynamic execution environment where composition units can be added to and withdrawn from the system at run-time, it seems desirable to assist developers with additional visibility constructs that allow them to describe which units of behavior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 describe our proposal in more detail by discussing an example and describing a revised lookup mechanism in its support.

2 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.)

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.

2.1 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 only activated if the corresponding message received originated from the same layer that defines that method. It is important to not that *restricted* methods cannot be called via `next`, or put differently, methods from one layer can only proceed to *public* methods if available. (`next` [7] Is similar to CLOS `call-next-method` [9] or ContextFJs `proceed`) [6] 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.

2.2 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

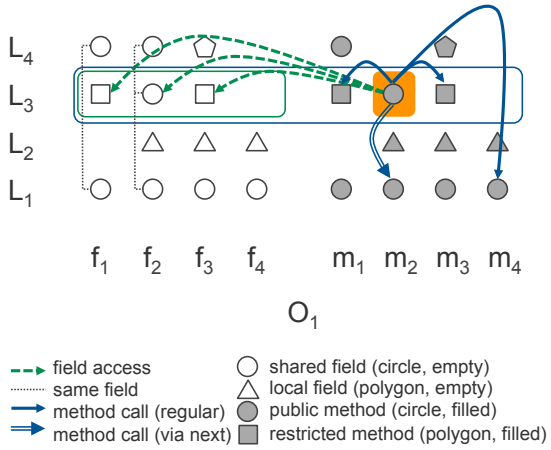

Fig. 1 Composition `with (L4 L3 L2 L1)`.

common practice in object-oriented programming from its inception.

But we also want to confine state so that is 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.

```
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 1 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 2 Layer L2.

3 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 1) 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 2) 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 3) provide another mix of shared and local fields and public and restricted methods to make the composition discussed more

```
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() {
      return
        '*' + 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 3 Layers L3 and L4.

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 that assembles the values of all of the fields accessible from that method and of 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

```
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 4 Composition.

variables or fields 4.

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 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`.

4 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 [3]. 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 4 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.m2` to be restricted is to make sure that, while callable from `L3.O1.m1` (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 in the same layer as the method from which the message was sent. (2) If there is such a method, continue execution with that method, otherwise (3) 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

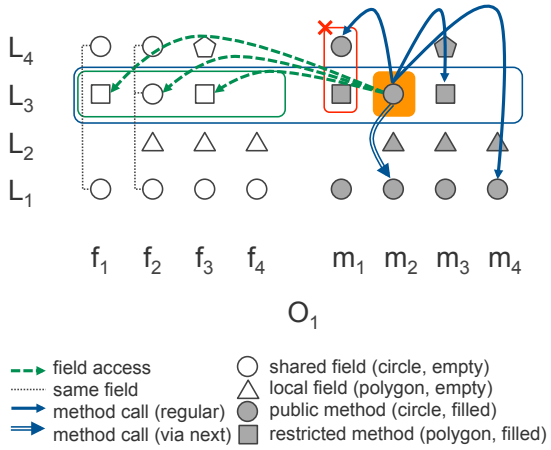

Fig. 2 Old Lookup.

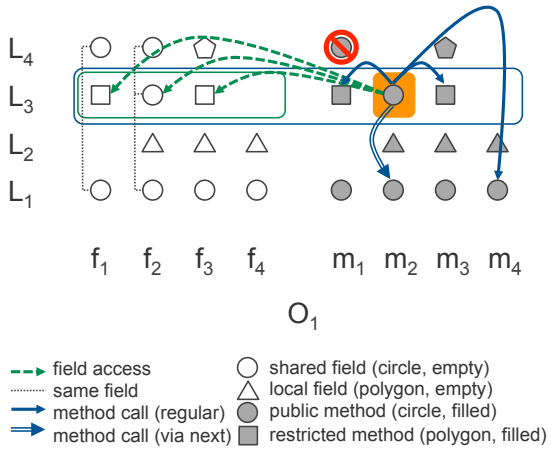

Fig. 3 New Lookup.

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).

5 Related Work

Scala's traits [8] can be considered partial object definitions without dynamic activation (that is, the application of a trait must be specified statically before an instantiation). Even though a trait can override a public method in the base class, it can only do so when this trait and the base class ex-

tend 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 their common interfaces.

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 [2] 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 [11] 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. Double underscores make the name unique to class, which effectively makes it restricted. Otherwise, instance variables are basically shared.

6 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 *restricted* as an access modifier in languages like Java [1] or C++ [10], 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 [7].

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 both work on both *L*'s foundations [5] 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).

References

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

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

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

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

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

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

[7] Hirschfeld, R., Masuhara, H., and Igarashi, A.:

Layer and Object Refinement for Context-oriented Programming in L, *Proceedings of 95th IPSJ Workshop on Programming*, IPSJ, 2013.

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

[9] Steele Jr., G. L.(ed.): *Common Lisp: The Language, 2nd Edition*, Digital Press, 1990.

[10] Stroustrup, B.: *The C++ Programming Language, 4th Edition*, Addison-Wesley, 2013.

[11] van Rossum, G. and Drake Jr., F. L.: The Python Language Reference, Technical report, Python Software Foundation, 2014.