

Instance Pointcuts

An Aspect-Oriented Mechanism for Maintaining Object Sets

Kardelen Hatun, Christoph Bockisch, and Mehmet Akşit

TRESE, University of Twente
7500AE Enschede

The Netherlands

<http://www.utwente.nl/ewi/trese/>
{hatunk,c.m.bockisch,aksit}@ewi.utwente.nl

Abstract. In the life-cycle of objects there are different phases. The phase in which an object currently is, affects how it is handled in an application; however these phase shifts are typically implicit. In this study we propose a new language mechanism, called instance pointcuts, based on aspect-orientation. They maintain sets of objects according to events in their life-cycle and create notifications when new objects are added or removed from the set. The selection criteria of instance pointcuts can be refined, e.g., to define a subset or super-set of an existing instance pointcut; and they can be composed, e.g., by set operations. Our approach improves modularity by providing a fine-grained mechanism and a declarative syntax to maintain a set of objects.

1 Introduction

In object-oriented programming (OOP), objects encapsulate state and behavior; objects also have a life-cycle, which means that the same object can play different roles at different times. And which role an object is currently playing is important as it can affect the object's own behavior or how objects are handled. Typically the shift from one life-cycle phase to another is implicit, e.g., determined by passing an object from one client to another. In this paper, we propose a new language mechanism for declaratively specifying life-cycle phases and for exposing the set of objects which are currently in a specific phase. This declarativity allows us to give guarantees about these sets like subset relationships, as well as to perform compile-time checks like warning about sets that will always be empty.

As an example of different relevant phases in the life-cycle of objects, consider an online store application. Assume that the offered products are objects in the program and that specific budget plans should be offered for products depending on their life-cycle phase. A first example of such a phase is a period during which a product is in the wish-list of any customer; this phase begins when “product” object is added to the “wish-list” property of a “customer” object and it ends when it is removed from this property. A second example is the phase when an

object is out of stock. Thus, we defined two sets of objects: the set of objects into which at least one customer is interested and the set of objects that are sold out. Finally, the shop owner may be interested in the intersection of these sets and give priority to reordering the corresponding products.

Furthermore, in OOP objects are categorized according to their *types*. This is a structural categorization and it does not give any information about the events that object participates in. However, grouping objects according to another criteria such as, the class they were initialized in or being passed as an argument to a certain method would only be possible by inserting code at those particular points, which would litter the code.

To be able to process the objects which are currently in a relevant life-cycle phase (like having been added to a wish-list), bookkeeping is required to keep track of the set. To separate this bookkeeping code from the business logic of the program, aspect-oriented programming (AOP) is a well known technique. But in AOP, *pointcuts* select sets of so-called *join points* which are points in time during the execution of the program; current aspect-oriented languages do not offer dedicated mechanisms for selecting *sets of objects*.

These languages do not support a *declarative specification* of the objects belonging to a life-cycle phases; instead an *imperative implementation*, always following the same pattern, is required for collecting those objects. A consequence of such an imperative solution, besides all the negative effects of hand-writing boilerplate code, is that automatic reasoning becomes practically impossible. In addition, a declarative implementation makes the relevant information explicit, which reduces checking efforts as well as making the code readable.

To offer better support for processing objects according to their life-cycle phase, we propose a language construct that builds on the technology of aspect-orientation. We borrow the terminology and provide *instance pointcuts* to select sets of objects based on the execution history.

An instance pointcut definition consists of three parts: an identifier, a type which is the upper bound for all objects in the selected set, and a specification of relevant objects. The specification consists of two *pointcut expressions* that select relevant join points in the objects' life-cycle and expose the object: the mark the beginning and end of a life-cycle phase; i.e., the object is added to or removed from the set.

New instance pointcuts can be derived from existing ones. One instance pointcut can be declared to be a *subset* or a *super-set* of an existing one. In this case, the specification of the life-cycle phase is narrowed down or broadened, respectively. Composition of existing instance pointcut is also supported in terms of set operations: *intersection*, *union* and *set difference*.

2 Motivation - Example Section

Objects can be categorized by how they are used (passed as arguments to method calls, act as receiver or sender for method calls, etc.). Concerns may be applicable to objects used in the same way. Therefore we must be able to identify and select


```

1 Set<Item> happyItems = createSet();
2 public boolean addToShoppingBag(Product p, int amount)
3 {
4     Item item = ItemFactory.createItem(p, amount);
5
6     if('timing condition')
7         items.put(item)
8
9     this.shoppingBag.add(item);
10 }
11 public boolean removeFromShoppingBag(Item item, int amount)
12 {
13     ...
14     items.remove(item);
15 }
16 public checkOut()
17 {
18     foreach(i in happyItems)
19         ProductManager.applyDiscount(i, DiscountRule.get(HAPPY_HOUR)
20             );
21 }

```

Listing 1: A Java implementation of Happy-hour discount rule

those items will be applied a discount in check out. In order to realize this in an OO-approach, one needs to insert the code regarding the discount concern, to the place where the event of interest occurs. With an OO-approach, to satisfy the new requirement we need to invasively change the system code. First we need to keep a list of items that were added to shopping bag inside the timing condition. Then when the user finally checks out, we should apply the discount to the items in the list. Listing 1 shows that we need to insert code in multiple places to satisfy the new requirement. First a set, called `happyItems`, is created (Line 1), to keep track of items that are added during the happy-hour. In `addToShoppingBag` method a if statement checking the timing condition is defined, when the condition is true then the created item is added to `happyItems` (Line 6-7). Finally in the `checkOut` method the discounts are applied to the items in `happyItems` (Line 18-19).

Even for a single discount rule, the code for the discount concern and the book-keeping that comes with it creates cluttering. If we would like to apply multiple discount rules for the *check out* event, this way of implementing is clearly not suitable. Also this event may not be the only event we want detect for applying discount rules and the OO implementation will eventually be scattered and tangled. An aspect-oriented implementation can offer a better solution, by encapsulating the concern in an aspect. In Listing 2 the `newItem` pointcut (Line 2-4) selects join-points where after customer adds a product to his bag, a new `Item` is created in `addToShoppingBag` method. The advice is executed when `newItem` is matched; after returning the new item, the timing condition

```

1 Set<Item> happyItems = createSet();
2 pointcut newItem(Item item):
3     call(* ItemFactory.createItem(..)) &&
4     withincode(* Customer.addToShoppingBag(..));
5 pointcut removeItem(Item item):
6     call(* Customer.removeFromShoppingBag(..)) &&
7     args(item);
8 pointcut checkOut(): call(* Customer.checkOut());

10 after() returning(Item item): newItem(item)
11 {
12     if('timing condition')
13         happyItems.put(item);
14 }
15 after(Item item): removeItem(item)
16 {
17     happyItems.remove(item);
18 }
19 before():checkOut()
20 {
21     foreach(i in happyItems)
22         ProductManager.applyDiscount(i, DiscountRule.get(HAPPY_HOUR)
23             );
24 }

```

Listing 2: An Aspectj implementation of Happy-hour discount rule

is checked and if it holds then the item is added to `happyItems`. We also define a `removeItem` pointcut (Line 5-7) to remove an item from `happyItems` if it is removed from shopping bag. Finally, we define a pointcut for selecting the `Customer.checkOut()` join-point (Line 8) and before the method is executed we apply the discount to items in `happyItems`. AO implementation, although offers a non-invasive solution, also suffers from boilerplate code written for book-keeping.

Let's look at another new requirement, which is as follows: Keep a set of products that are currently out of stock, once they become available raise an event and remove them from the set. This scenario also requires when a condition holds and the set is changed there should be an event raised which means the set should be monitored. This creates an additional concern.

Reusing existing sets by finding their subsets or composing them with set operations is also impractical with imperative programming. If we want to find the subset of `BeautyProducts` in a set of `Product` objects, we have to iterate over it and check instance types to create a new set. Implementing set operations such as intersection also results in further boilerplate code.

In order to overcome the shortcomings of existing approaches, we need a way to declaratively select objects based their life-cycle phases, where the beginning

and the end of a phase is marked by events. From the scenarios described above, we conclude the following requirements:

- Requirement1: Provide a declarative way of reifying a set of objects by defining add/remove conditions
- Requirement2: Support multi-sets, an object can be added multiple times
- Requirement3: Add/Remove expressions select events and the object is exposed from the context of this event
- Requirement4: Access the set of objects which is currently selected and be notified when the set changes
- Requirement5: Provide a declarative way to refine existing sets to their subsets
- Requirement6: Compose sets via set operations (intersection, union, set difference)
- Requirement7: Provide checks for validity of composition, empty sets and subset-superset compatibility

3 Instance Pointcuts

Instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specific type, with the ability to select them over a period marked by events in their life-cycle, modularizing the object selection concern. The purpose of instance pointcuts is to let the user to make focused selections, therefore offering a different level of modularity.

In the remainder of this section, we will explain instance pointcuts in detail. We have implemented a prototype by extending AspectJ, throughout the section code examples of instance pointcuts will be in AspectJ.

3.1 Features

A concrete instance pointcut definition consists of a left hand-side and a right-hand side. In the left hand side the pointcut's name and the instance type of interest is declared. An instance pointcuts do not declare pointcut parameters, it has a single implicit parameter called `instance` of the declared instance type. In the right hand side a pointcut expression selects the desired events from join-points and then binds the exposed object as a member of the instance pointcut's set, which is represented by `instance` parameter. An instance pointcut is a *static aspect member* (Listing 3).

```

1 aspect MyAspect {
2   static instance pointcut foo<Object> ...
3 }

```

Listing 3: An instance pointcut declaration in an aspect

Add/Remove Expressions Join points mark *sites* of execution; a join-point by itself does not define an event. In AspectJ pointcuts, pointcut expressions select joinpoints and pointcuts are used with advice specifications to select a particular event in that join-point. Instance pointcuts are not used with advices; since they serve the specific purpose of reifying and maintaining object sets, they implicitly perform add to set and remove from set operations. Therefore instance pointcut expressions are used to select specific events rather than joinpoints. To be able to define events with instance pointcut expressions, it is necessary to use advice specification clauses *before* or *after* in the expression.

The *add expression* select events which contain the instances to be added to the instance pointcut set; an event selected by add expression marks the beginning of the life-cycle phase of interest. Instance pointcuts can also include an optional *remove expression*, which specifies when to remove which object, selects the event that marks the end of the life-cycle phase of interest. Both add and remove expressions contains at least one *expression element* and at most two. The expression elements can be one of two types, *before element* and an *after element*. Each element contains a pointcut expression and they can select more than one event. The before element selects the before events and the after element, selects after events.

The grammar definition below shows part of the instance pointcut syntax. An event definition is comprised of an event keyword, *before* or *after*, and a pointcut expression. The pointcut expression selects a set of join points and the event keyword specifies which event is selected from a join-point. *(pointcut – expression)* rule represents an AspectJ pointcut expression with the restriction that every pointcut expression there should be a binding predicate that bind the instance parameter and an extension that allows the use of *returning* clause as a binding predicate. In an instance pointcut expression, it is only possible to OR a before event with an after event. The *before* clause selects the start of executing an operation (i.e., the start of a join point in AspectJ terminology) and the *after* clause selects the end of such an execution. For two operations that are executed sequentially, the end of the first and the start of the second operation are treated as two different events. Thus, the before and after clauses select from two disjoint groups of events and the conjunction of a before and an after clause will always be empty.

```
1 instance pointcut customers<Customer>: after(call(SessionManager
    .createCustomer(..)) && returning(instance)) UNTIL before(
    call(SessionManager.destroyCustomer(..)) && args(instance))
```

Listing 4: A basic instance pointcut declaration with add and remove expressions

The instance pointcut in Listing 4 shows a basic example. The left-hand side of the instance pointcut indicates the pointcut is called *customers* and it is interested in selecting the *Customer* objects. On the right hand side, there are two expression separated by the *UNTIL* keyword. The first one is the add expression.

```

<instance pointcut> ::= 'instance pointcut' <name> '<' <instance-type> '>' ':'
    <ip-expr> ('UNTIL' <ip-expr>)?

<ip-expr> ::= <after-event> '||' <before-event>
    | <before-event> '||' <after-event>
    | <after-event>
    | <before-event>

<after-event> ::= 'after' '(' <pointcut-expression> ')'
<before-event> ::= 'before' '(' <pointcut-expression> ')'

```

Fig. 2: Grammar definition for instance pointcuts

It selects the return event of the method `createCustomer` and from the context of this event it exposes the returned `Customer` object with the returning clause and binds it to the `instance` parameter. If the `instance` parameter is bound by the returning clause then it has to be in an after event. The second one is the remove expression and it selects the before event `destroyCustomer` call and exposes the `Customer` instance in the method arguments and binds it to the `instance` parameter with `args pointcut`.

Note that instance pointcuts do not keep objects alive. Because instance pointcuts are non-invasive constructs, which do not affect the program execution in any way. So even if the remove expression was not defined for `customers pointcut`, if the `Customer` instances were collected by the garbage collector, they would still be removed from the set.

Multisets An instance pointcut reifies an object set as a *multiset*. Multisets are sets which allow multiple appearances of a an object. The instance pointcut shown in In Listing 5 selects `Product` instances, which are added to the wishlist of a `Customer`. The remove expression removes a `Product` instance if it is removed from the wish-list of a `Customer`. With this pointcut we would like to count the currently wish-listed products. If instance pointcuts onyle supported a set then as soon as a customer removes a product from his wish-list, its only copy will be removed and it will appear removed from all customers' wish-lists, therefore a multi-set is used.

```

1 instance pointcut multi_wish<Product>:
2 before(call(Customer.addToWishList(..)) && args(instance))
3   UNTIL before(call(Customer.removeFromWishList(..)) && args(
    instance))

```

Listing 5: An instance pointcut utilizing multiset property

3.2 Refinement and Composition

Instance pointcuts can be referenced by other instance pointcuts, they can be refined to create a subset or broadened to create a superset.

Referencing and Dynamic Refinement Normally instance pointcuts are referenced by their name, however they can also take an additional statement for *type refinement*, which selects a subset of the instance pointcut *dynamically*. Type refinements require that, the refined type is a subtype of the original instance type. For example the instance pointcut `lowSelling` (Listing 6) can be refined with the following syntax: **instance pointcut** `lowSellingBeauty<BeautyProduct>: lowSelling<BeautyProduct>`. This indicates we would like to select the subset of `BeautyProduct` instances from the set of `Product` instances selected by `lowSelling`. Note that this notation will also select subtypes of `BeautyProduct`. The effect of a refinement subset being empty is equivalent to that of an unmatched pointcut, meaning it will not result in any changes on the set.

```
1 instance pointcut lowSelling<Product>: after(call(* OnlineShop.  
    setLowSelling(..)) && args(instance));
```

Listing 6: Two instance pointcuts composed to obtain a new one

Instance Pointcut Composition It is also possible to compose instance pointcuts to obtain a new pointcut based on others. In Figure 3, an extended version of the grammar definition is shown. A new pointcut can be defined either by referencing another pointcut and appending additional predicates to the pointcut expressions in before and after blocks or it can be defined as the composition of other pointcuts or both. In order to obtain a more refined set `&&` operation is used and for a broader set `||` is used.

The composition also has an effect on the types of the instances that are added to the set. To better explain this effect we have created a small example, where a members of a simple type hierarchy (Figure 4a) are used in instance pointcuts. Table 4b shows four distinct cases. In the first case we compose two pointcuts `pc1` and `pc2<C>` with an `&&` operation, the resulting pointcut from this composition can only select the narrower type `C` since the composed expression is stricter and the type restriction comes from `pc2`. However with `||` operation we can capture `B` types as well as `C` types, so the resulting composition will select a mixture. In the second row a case with sibling types are shown. For the `&&` operation the resulting composition cannot select any types since the types `A` and `B` cannot have a common instance. The `||` operation will again select a mix of instances of type `A` and `B`, then the instance type for the composed pointcut should be `R` to contain the subtypes.

$\langle \text{instance-pointcut} \rangle ::= \text{'instace pointcut' } \langle \text{name} \rangle \text{'<'} \langle \text{instance-type} \rangle \text{'>' ' : '}$
 $\quad (\langle \text{ip-ref-expr} \rangle \langle \text{logical-operator} \rangle)? (\langle \text{ip-expr} \rangle \text{'UNTIL'} \langle \text{ip-expr} \rangle)?$
 $\langle \text{logical-operator} \rangle ::= \text{'\&\&'}$
 $\quad | \text{'||'}$
 $\langle \text{ip-ref-expr} \rangle ::= \langle \text{ip-ref} \rangle$
 $\quad | \langle \text{ip-ref-expr} \rangle \text{'\&\&'} \langle \text{ip-ref-expr} \rangle$
 $\quad | \langle \text{ip-ref-expr} \rangle \text{'||'} \langle \text{ip-ref-expr} \rangle$
 $\langle \text{ip-ref} \rangle = \langle \text{name} \rangle$
 $\quad | \langle \text{name} \rangle (\text{'<'} \langle \text{refined-instance-type} \rangle \text{'>'})?$
 \dots

Fig. 3: Syntax for instance pointcut composition

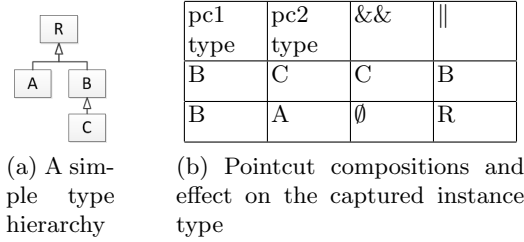


Fig. 4: An example to illustrate composition's effect on types

Semantics The composition of two instance pointcuts is computed by composing their instance pointcut expressions. An instance pointcut can have at most four expression elements; before (part of) add (*ba*) and after add (*aa*) form the add expression and, before remove (*br*) and after remove (*ar*) form the remove expression. Let us define an instance pointcut as $IP = ba_{IP} \parallel aa_{IP} \text{ UNTIL } br_{IP} \parallel ar_{IP}$. Each element contains a pointcut expression referred with the following notation; $ba_{IP_expression}$.

$$IP1 = ba_{IP1} \parallel aa_{IP1} \text{ UNTIL } br_{IP1} \parallel ar_{IP1} \quad (1)$$

$$IP2 = ba_{IP2} \parallel aa_{IP2} \text{ UNTIL } br_{IP2} \parallel ar_{IP2} \quad (2)$$

Let us define a new pointcut as; $IP3 = IP1 \&\& IP2$. By distributivity this operation is results in the following formulation:

$$\begin{aligned}
IP3 \rightarrow \text{add expression} = & (ba_{IP1} \&\& ba_{IP2}) \\
& \parallel (ba_{IP1} \&\& aa_{IP2}) \\
& \parallel (aa_{IP1} \&\& aa_{IP2}) \\
& \parallel (aa_{IP1} \&\& ba_{IP2})
\end{aligned} \tag{3}$$

$$\begin{aligned}
IP3 \rightarrow \text{remove expression} = & (br_{IP1} \parallel br_{IP2}) \\
& \parallel (br_{IP1} \parallel ar_{IP2}) \\
& \parallel (ar_{IP1} \parallel ar_{IP2}) \\
& \parallel (ar_{IP1} \parallel br_{IP2})
\end{aligned} \tag{4}$$

Notice that the remove expressions are applied \parallel instead of $\&\&$. If we narrow down removal criteria by $\&\&$ then it is possible that some objects which are added will not be removed. By keeping the removal criteria for both of the composed pointcuts, we ensure that all the added object can be removed. So we can say that the $\&\&$ and the \parallel operations only effect the expression elements in the add expression.

Composition of before and after event via $\&\&$ will be empty as explained at the beginning of this section. Composing two events of the same type with a logical operator is formulated as follows:

$$\begin{aligned}
event(exp_1) \&\& event(exp_2) &= event(exp_1 \&\& exp_2) \\
event(exp_1) \parallel event(exp_2) &= event(exp_1 \parallel exp_2)
\end{aligned}$$

Note that we do not evaluate expressions to verify their composition and leave that to the AspectJ compiler. For example AspectJ does not allow use of different binding predicates in the composed expression and this is also a limitation of instance pointcut composition.

Then IP3 becomes;

$$\begin{aligned}
IP3 = & \overbrace{(ba_{IP1} \&\& ba_{IP2})}^{ba_{IP3}} \parallel \overbrace{(aa_{IP1} \&\& aa_{IP2})}^{aa_{IP3}} \\
& \text{UNTIL} \\
& \underbrace{(br_{IP1} \parallel br_{IP2})}_{br_{IP3}} \parallel \underbrace{(ar_{IP1} \parallel ar_{IP2})}_{ar_{IP3}}
\end{aligned} \tag{5}$$

If we define another instance pointcut as $IP4 = IP1 \parallel IP2$, after performing the above formulations:

$$\begin{aligned}
 IP4 &= \overbrace{(ba_{IP1} \parallel ba_{IP2})}^{ba_{IP4}} \parallel \overbrace{(aa_{IP1} \parallel aa_{IP2})}^{aa_{IP4}} \\
 &\text{UNTIL} \\
 &\overbrace{(br_{IP1} \parallel br_{IP2})}^{br_{IP4}} \parallel \overbrace{(ar_{IP1} \parallel ar_{IP2})}^{ar_{IP4}}
 \end{aligned} \tag{6}$$

Using the same composition principles, instance pointcuts can also be composed with expression elements, to refine or broaden the pointcut scope. For the instance pointcut IP5 which is defined as:

$$\begin{aligned}
 IP5 &= IP1 \parallel (ba_{new} \text{ UNTIL } ar_{new}) \\
 &= \overbrace{(ba_{IP1} \parallel ba_{new})}^{ba_{IP5}} \parallel \overbrace{(aa_{IP1})}^{aa_{IP5}} \\
 &\text{UNTIL} \\
 &\overbrace{(br_{IP1})}^{br_{IP5}} \parallel \overbrace{(ar_{IP1} \parallel ar_{new})}^{ar_{IP5}}
 \end{aligned} \tag{7}$$

Let us illustrate these concepts with a concrete example. The first instance pointcut (`lowSelling`) shown in Listing 7 selects the `Products` which are marked as *low-selling* by the `OnlineShop` class and remove those products which are discontinued by the `Vendor`. The second pointcut (`wishList`) selects the `BookProducts` which are added to the wish-list of a `Customer`, removes them when they are out of the wishlist.

```

1 instance pointcut lowSelling<Product>: after(call(* OnlineShop.
   setLowSelling(..) && args(instance)) UNTIL before(call(*
   Vendor.discontinue(..) && args(instance));
2 instance pointcut wishList<BookProduct>: after(call(* Customer.
   addToWishList(..) && args(instance)) UNTIL before(call(*
   Customer.removeFromWishList(..) && args(instance)) ;
3 instance pointcut composition<Product>: lowSelling || purchase;

```

Listing 7: Two instance pointcuts composed to obtain a new one

The new instance pointcut `composition` is the result of an `||` operation between `lowSelling` and `wishList`. Then the add and remove expressions of composition is created by factoring out the `args` pointcut and is shown in Listing 8.

Similarly we can define a new pointcut (Listing 10), this time by composing a pointcut with an expression element, then the `composition2` pointcut has the pointcut expression shown in Listing ??.

```

1 instance pointcut composition<Product>:
2 after((call(* OnlineShop.setLowSelling(..)) || call(* Customer.
   addToWishList(..))) && args(instance))
3 UNTIL
4 before((call(* Vendor.discontinue(..)) || call(* Customer.
   removeFromWishList(..))) && args(instance))

```

Listing 8: The new instance pointcut after composition

```

1 instance pointcut composition2<Product>: lowSelling && (after(if
   (instance.getPrice() > 10)) UNTIL before(call(* OnlineShop.
   removeFromShop(..)) && args(instance));

```

Listing 9: An instance pointcuts and an expression is composed

```

1 instance pointcut composition2<Product>:
2 (call(* OnlineShop.setLowSelling(..)) && args(instance) && if(
   instance.getPrice() > 10))
3 UNTIL
4 before((call(* Vendor.discontinue(..)) || call(* OnlineShop.
   removeFromShop(..))) && args(instance));

```

Listing 10: composition2 pointcut expression in full

Subset and Superset Relationships There are two mechanism to define relationships between instance pointcuts, *subsetting* and *supersetting*. The declaration of these relationships impose some restrictions on the *declaring* instance pointcut. Let us update the syntax definition to host relationship declarations.

For the following declaration subsetIP1 **subsetof** IP1, the subset relationship imposes that the instance set created by subsetIP1 has to be the subset of the instance set of IP1. Hence, if later in the implementation, the scope of subsetIP1 is broadened, it should still follow the subset restriction. When the *instance type* is not declared like in the example then it is inferred from the superset IP1. It is also possible to declare subsets by using type refinement. For example subsetIP1<SubType> **subsetof** IP1 = IP1<SubType>, creates a subset of IP1 by selecting the instances of type SubType. A subset must declare a co-variant type of its superset and superset must declare a contra-variant type. For the superset declaration supersetIP1 **supersetof** IP1, the superset relationship imposes that the instance set created by supersetIP1 has to contain the set of IP1. This rule should hold even if the scope of supersetIP1 is narrowed. When the instance type is not declared then the instance type of IP1 is copied.

The right-hand side of a relationship declaration can be a full instance pointcut expression, formed by add and remove pointcuts. For subsets, && instance pointcut composition rule is applied, composing this expression with the superset's and for supersets, || composition rule is applied to the subset's, composing the expression with the subset's.

```

<instance-pointcut> ::= ...
| 'instance pointcut' <name> ('<' <instance-type> '>')? <relationship> ':' <ip-expr>
  ('UNTIL' <ip-expr>)?
| 'instance pointcut' <name> ('<' <instance-type> '>')? <relationship> ':' <ip-ref>

<relationship> ::= 'subsetof' <ip-ref>
| 'supersetof' <ip-ref>

```

Fig. 5: Grammar definition for relationship declarations

Subset and superset relationships enforce consistency between instance pointcuts and reduce redundancy. Being able to select subsets/supersets of instance pointcuts, eliminates the need for defining separate pointcuts, which may result in erroneous selections. Since the subsets only supports the subtypes of the original instance type and superset support only the supertypes, it works with the system's type hierarchy in a natural manner.

3.3 Using Instance Pointcuts

Upto now we have explained the syntax and semantics for instance pointcuts. In this section we will explain how to use instance pointcut in the context of an AO language, namely, AspectJ. The instance pointcut defined in Listing 11 satisfies the following requirement; Maintain a set which contains products that are currently out of stock, once they become available remove them from the set.

```

1 aspect MyAspect{
2   static instance pointcut outOfStock<Product>: after(call(*
    Product.outOfStock(..)) && target(instance)) UNTIL after(
    call(* Vendor.stock(..)) && args(instance));
3   ...
4 }

```

Listing 11: An instance pointcut for out of stock products

Set Access Instance pointcuts reify a set and this set can be accessed through the pointcut. When a pointcut is referenced outside of a pointcut expression (composition, subset etc.), it means that its set is accessed. Then the *read* methods of the multiset(**google guava?**) interface can be used to retrieve objects from the set. Write methods are not allowed since, it creates data inconsistencies and in may result in concurrent modification exceptions. The set can also be referenced with a type refinement, which will return a subset of the original set with objects of refined type.

In Listing 12, a method called `calculateDamages` is defined in `MyAspect`. This method calculates a rough estimate of the shop's damages when beauty

```

1 public static double calculateDamages()
2 {
3     double damage = 0;
4     for(BeautyProduct p: MyAspect.outOfStock<BeautyProduct>)
5         damage = damage + p.getPrice();
6     return damage;
7 }

```

Listing 12: Calculate a damage estimate for out of stock beauty products

products are out of stock. On line 4, the for loop iterates over the `BeautyProduct` subset of `outOfStock`'s set, which is accessed as a static field and also with the refinement syntax.

Set Monitoring An instance pointcuts definition defines two set change events; add event and remove event. These events are interesting since they signify that the object has become relevant or irrelevant. In order to select the join-points of these events, every instance pointcut definition automatically has two implicit regular pointcuts. These implicit pointcuts has the following naming convention; $\langle name \rangle_add$, $\langle name \rangle_remove$, where $\langle name \rangle$ is the name of the instance pointcut. These pointcuts allow the user to access the set change event and the object to be added or removed. In Listing 13 a before advice using the `outOfStock_add` pointcut is shown. When a product is marked out of stock then it is added to the set, this advice uses the `Product` instance to be added and uses its information to notify the related `Vendor` that the product is out of stock.

```

1 before(Product p) : outOfStock_add(p)
2 {
3     OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
4 }

```

Listing 13: Set monitoring pointcut used to notify vendors

3.4 Compilation of Instance Pointcuts

In our current implementation, we compile instance pointcuts to AspectJ code. For the instance pointcut shown in Listing 11 and the code snippets shown in Listings 12 and 13, the AspectJ code shown in Listing 14 is generated.

The lines 2-27 are generated from the instance pointcut `outOfStock`. On line 2, a multiset, named `outOfStock_set` with the generics `Product` is instantiated. The generic type is inferred from the instance type declared in `outOfStock`. This particular instance pointcut has only two expression elements; an after element in the add expression and an after element in the remove expression. These expression elements result in two pointcuts shown in lines 4

and 5 and two advice bodies shown in lines 10-14 and 15-19 respectively. For adding elements to the set, `add_outOfStock_set` method (lines 20-23) and for removing `remove_outOfStock_set`(lines 24-27) are generated. The set monitoring pointcuts for these methods are `outOfStock_add(Product p)` and `outOfStock_remove(Product p)`, respectively.

4 Related Work

5 Discussion

6 Conclusion and Future Work


```

1 aspect MyAspect{
2     public static Multiset<Product> outOfStock_set =
3         new TreeMultiset<Product>();
4     pointcut outOfStock_aa(): call(* Product.outOfStock(..));
5     pointcut outOfStock_ar(): call(* Vendor.stock(..));
6     pointcut outOfStock_add(Product p):
7         call(* MyAspect.add_outOfStock_set(..) && within(MyAspect)
8             && args(p));
9     pointcut outOfStock_remove(Product p):
10        call(* MyAspect.remove_outOfStock_set(..) && within(
11            MyAspect) && args(p));
12    after():outOfStock_aa()
13    {
14        Product p = thisJoinPoint.getTarget();
15        add_outOfStock_set(p);
16    }
17    after():outOfStock_ar()
18    {
19        Product p = findArgument(thisJoinPoint.getArgs(),Product.
20            class);
21        remove_outOfStock_set(p);
22    }
23    public boolean add_outOfStock_set(Product p)
24    {
25        return outOfStock_set.put(thisJoinPoint.getTarget());
26    }
27    public boolean remove_outOfStock_set(Product p)
28    {
29        return outOfStock_set.remove(p);
30    }
31    public static double calculateDamages()
32    {
33        Multiset<BeautyProduct> outOfStock_BeautyProduct_set = new
34            TreeMultiset<BeautyProduct>();
35        for(Product p: outOfStock_set)
36            if(p instanceof BeautyProduct)
37                outOfStock_BeautyProduct_set.put(p)
38        double damage = 0;
39        for(BeautyProduct p:outOfStock_BeautyProduct_set)
40            damage = damage + p.getPrice();
41        return damage;
42    }
43    before(Product p):outOfStock_add(p)
44    {
45        OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
46    }
47 }

```

Listing 14: The generated AspectJ code from outOfStock instance pointcut