ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 11

Ownership Types and Non-null Types

December 8, 2017

Task 1

Consider the following method signatures:

```
peer Object foo(any String el);
peer Object foo(rep String el);
rep Object foo(any String el);
any Object foo(peer String el);
rep Object foo(peer String el);
```

Find all the valid pairs of signatures such that one overrides the other.

solution

The general typing rules are `peer <: any` and `rep <: any` since `any` is less restrictive than `rep` and `peer`. Following these rules, we obtain that

- `peer Object foo(any String el)` overrides
  `any Object foo(peer String el)`
- `rep Object foo(any String el)` overrides
  `rep Object foo(peer String el)`, that overrides
  `any Object foo(peer String el)`
- `peer Object foo(any String el)` overrides
  `peer Object foo(rep String el)`

Task 2

[*From a previous exam*]

Consider the following declarations:

```
class A
{
   rep B first;
   rep B second;
}
class B
{
   any A obj;
   peer B sibling;
}
```

Which of the following programs are allowed in the topological ownership system? For any program that is accepted in the topological system, is it also accepted in the owner-as-modifier

system? Assume that none of the objects involved are `null`. Briefly explain each of your answers.

Program 1	Program 2	Program 3	Program 4

`rep B b;`
`...`
`b = b.sibling;`
	`peer A a; rep B b;`
	`...`
	`a = b.obj;`
		`any A a;`
		`...`
		`a.first.obj = a;`
			`peer A a;`
			`...`
			`a.first = a.first;`

solution

- **Program 1** is accepted in both systems.
- **Program 2** is not accepted in the topological system (and neither in the owner-as-modifier system). It attempts the assignment of an `any` reference to a `peer` reference. `peer` is not a super-type of `any`.
- **Program 3** is accepted in the topological system (it assigns `any` to `any`). However, it assigns to the field of a `lost` reference, which means that it is not accepted in the owner-as-modifier system.
- **Program 4** is not accepted in the topological system (and neither in the owner-as-modifier system), because it assigns to a `lost` location.

Task 3

Annotate the following program with appropriate ownership type modifiers (according to the topological ownership system) in order to maximize the buffer, the producer, and the consumer *encapsulation*. This means that the modifiers you choose should increase the depth of nested ownership context and reduce the number of (non-rep) edges/pointers between different contexts.

```
class Producer {
 int[] buf;
 int n;
 Consumer con;
 Producer()
 {
  buf = new int[10];
 }
 void produce(int x)
 {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 int[] buf;
 int n;
 Producer pro;
 Consumer(Producer p)
 {
  buf = p.buf;
  pro = p;
  p.con = this;
 }
 int consume()
 {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 Producer p;
 Consumer c;

 Context() {
  p = new Producer();
  c = new Consumer(p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i) {
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

solution

You might be tempted to annotate `con` in `Producer` and `pro` in `Consumer` as `any` — in general, this would even allow one modification less (in the topological system): of an `any` receiver, only an `any` field can be modified, whereas of a `peer` receiver, both a `peer` and an `any` field can be modified.
However, our goal is to maximize encapsulation, and therefore `peer` is the best choice here.

```
class Producer {
 rep int[] buf;
 int n;
 peer Consumer con;
 Producer()
 {
  buf = new rep int
      [10];
 }
 void produce(int x)
 {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 any int[] buf;
 int n;
 peer Producer pro;
 Consumer(peer
     Producer p)
 {
  buf = p.buf;
  pro = p;
  p.con = this;
 }
 int consume()
 {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 rep Producer p;
 rep Consumer c;

 Context() {
  p = new rep Producer
      ();
  c = new rep Consumer
      (p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i) {
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

Task 4

Assume the topological ownership framework taught in the course. Suppose that we want to create a class `SortedLinkedList`, with the internal invariant that the values stored in the nodes are sorted in ascending order.

```
package SortedLinkedList;
public class SortedLinkedList {
  private rep Node head;

  /// invariant head != null ==> head.sorted()
  ...
}
private class Node {
  protected peer Node next;
  protected int value;

  /// pure
  boolean sorted() {
    return next!=null ==> value < next.value && next.sorted()
  }
}
```

Suppose that all the methods in `SortedLinkedList` are guaranteed to preserve the invariant of the class.

Furthermore, suppose that we want to create iterators for such lists (defined in the same package):

```
public class LinkedListIterator { private any Node current_item; ... }
```

A) Why is the field `current_item` annotated as `any`? What drawbacks would the other possible annotations have?

solution

If `current_item` were annotated as `rep`, then the owner of the node it refers to is the iterator itself. In this case, the iterator cannot iterate over a `SortedLinkedList` object l, because l also owns its nodes. The ownership topology allows at most one owner per object.

If `current_item` were annotated as `peer`, then, assuming that `current_item` has a list owner l, the owner of the iterator must also be l. This may be OK in topological ownership. However, if we add "owners as modifiers", the iterator's methods that traverse l cannot be called directly from an object outside l, which defeats the purpose of iterators.

B) We would like to have the following features:

(i) the invariant of a `SortedLinkedList` object is guaranteed to hold in any program, except when one of its methods executes

(ii) `LinkedListIterator` is a *modifying* iterator, i.e., it may change the value of the object it is pointing to

We can't have both features. Depending on whether or not we impose the "owners as modifiers" discipline, we can have either (i) or (ii). Argue why this is the case.

solution

If we don't have "owners as modifiers", an object may get/hold an `any` reference to a node of the list, modify its `value` field, and break the invariant: (i) is not achieved.

If we do have "owners as modifiers", then the iterator may not modify the value of the node it is pointing at, because it holds an `any` reference to it: (ii) is not achieved.

C) The fact that (i) and (ii) cannot both hold together is not surprising. A modifying iterator can break the invariant of the list it is iterating over. However, the "owners as modifiers" discipline may disallow harmless designs. Write a benign class (perhaps a restricted modifying iterator), which would not break the invariant of any object of `SortedLinkedList`, but still does not compile under "owners-as modifiers".

solution

We could have an iterator that performs the requested modification iff this does not violate the invariant:

```
public class LinkedListIterator {
  private any Node f;

  ... // some non-modifying methods

  public void modifyCarefully(int x) {
    if(f.value <= x && (f.next == null || x < f.next.value))
      f.value = x;
    // benign but does not type check under "owners as modifiers"
  }
}
```

Task 5

[*From a previous exam*]

The topological ownership system guarantees the following property: If a reference `a.f` to an object `b` is of ownership type `rep C`, then the object `a` is the *owner* of `b`. Moreover, each object has at most one owner.

The topological ownership system has a weakness: it does not support ownership transfer, which is desirable in many situations. Let us try to remedy this situation. Consider the following incomplete definition of a class `T`:

```
class T {
  public rep U f, g;
  ...
}
```

and the following program P, which, in addition to the field assignments, *implicitly also changes the owner* of object `e2.g` from `e2` to `e1`:

```
// implicitly: e2.g.owner = e1;
e1.f = e2.g;
e2.g = null;
```

where `e1, e2` are two non-null objects of type `T`.

A) The code P is not allowed in the topological ownership system. Which rule disallows it?

solution

Assuming `e1` is not syntactically equal to `this`, then `e1.f` must be `lost` and can therefore not be assigned to.

B) Write a code snippet C, such that executing $C; P$ is *guaranteed* to break the property described in the first paragraph of this task, *after P has finished executing. Do not rely on any specific implementation of class* `U` (but you may assume the existence of a constructor without parameters). You may also add constructors to class `T`.

Note that:

- you can assume that P is accepted by the compiler.
- *all* the code that *you* write *must respect the topological ownership system.* P is the only code that breaks the rules.
- you may *not* use reflection in your solution.
- you may *not* use P anywhere in the code that you write.

solution

Add the following constructor to `T`:

```
T() {
  f = new rep U();
  g = f;
}
```

Now use the following code C:

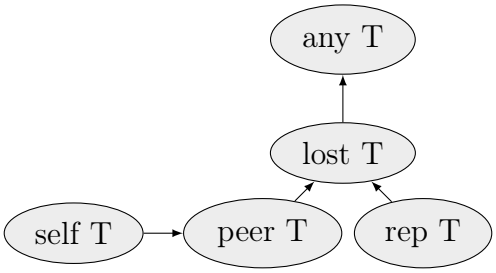
```
e1 = new peer T();
e2 = new peer T();
```

The invariant is broken after $C; P$, because `e1` is the owner of `e1.f`, but the `rep` field `f` of a different object (`e2`) points to it.

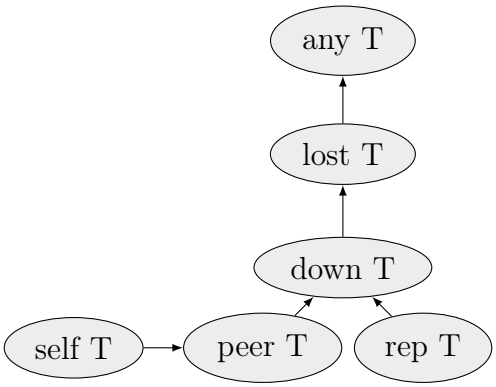
Task 6

The ownership type system allows the following ownership modifiers: `peer`, `rep`, `self`, `lost` and `any` - to structure the object store and to restrict how references can be passed and used. We want to extend the ownership type system by adding one more modifier `down`. This modifier is introduced to denote references to objects in the same context as `this` or in the context (transitively) owned by an object in the same context as `this`.

A) Redraw the subtype relation diagram below to include the newly introduced type of the universe type system.



solution



B) Consider the following example:

```
public class Node{
   rep Node c;
   down Node d;

   public void foo() {
      this.d.d = this;   // should this line typecheck?
      this.c.d = this.d; // should this line typecheck?
   }
}
```

Which of the assignments above should be allowed by the type system? Why?

solution

The example code shows two cases where the field updates should not be allowed, because we would allow a `down` field to point upwards (to `this`) in the ownership topology, and in the second, because we would allow a `down` field to point to some object which is considered `down` from the viewpoint of `this`, but not necessarily from the viewpoint of `this.c`.

C) Starting from the example from **Task B**, define the most specific (in terms of the context information it conveys) viewpoint adaptation function ▶ by filling the table below (for a combination T_e ▶ T_f the modifier T_e specifies the row, and the modifier T_f the column of the table used).

Recall that the viewpoint adaptation function ▶ is used, in particular, to determine the owner of an object referenced by a field access. More exactly, if the ownership modifier of `e` is T_e and the ownership modifier of a field `f` is T_f , then the ownership modifier assigned to the field access `e.f` is determined as T_e ▶ T_f. Note that this applies to field updates as well as field reads.

▶	peer	rep	any	down
self				
peer				
rep				
lost				
any				
down				

solution

Here is the table that defines the viewpoint adaptation as describing the most precise information possible about where such a reference may belong in the heap topology:

▶	peer	rep	any	down
self	peer	rep	any	down
peer	peer	down	any	down
rep	rep	down	any	down
lost	lost	lost	any	lost
any	lost	lost	any	lost
down	down	down	any	down

Note that in the table above we over-approximate entries, in cases where we cannot describe precisely what we want. For example, `rep`▶`rep` can be `down`, because `down` over-approximates the objects which can actually be stored in such a field. Note that this is a true approximation - `rep`▶`rep` is not allowed to store all objects which can be referred to via `down`, only some of them. This means that in **D** we need to add extra restrictions on field assignment in the cases where we use `down` to over-approximate in this way; otherwise the examples in part **B** would type-check, which would not be safe.

If we *relax the requirement to have a most specific viewpoint adaptation function*, we can take an alternative approach which does not allow this kind of over-approximation; the modifier chosen could reflect precisely the requirements for a reference to be allowed to be stored in such a location, and thus avoid the need for extra requirements on the field assignment rule. Here is the table in this approach:

▶	peer	rep	any	down
self	peer	rep	any	down
peer	peer	lost	any	down
rep	rep	lost	any	lost
lost	lost	lost	any	lost
any	lost	lost	any	lost
down	lost	lost	any	lost

In this case, perhaps surprisingly, cases such as `rep`▶`rep` and `down`▶`down` result in `lost`. This is because, choosing the answer `down` is not restrictive enough. In general, we have no way to express what is safe to assign to the `down` field of a `rep` receiver (`down` from our

viewpoint includes objects above the `rep`, which should not be included), and similarly for a `down` receiver. As you can see, this second approach is not very flexible; only `rep` and `peer` objects can ever be typed as `down` (via subtyping).

D) Assuming that you only need to enforce the topological constraints of the type system, how should the field update rule from lecture 6 slide 64 be adapted to the system extended with the `down` modifier? Do you need to make any changes?

solution

With the first (most precise) variant of the viewpoint adaptation function from **C** we need to require that the result of the viewpoint adaptation is not `down`, except in the special case of the receiver being `self` or `peer`, and the field type being `down` (in these cases, the `down` result expresses precisely what is safe to assign to the location; it is not an over-approximation).

With the second (avoiding over-approximation) variant of the viewpoint adaptation function from **C**, we do not need to make any changes to the field assignment rule, to guarantee the topological constraints of the type system.

Task 7

[*From a previous exam*]

This question is about extending the non-null type system to handle arrays (ignoring initialization). Array types can have two type modifiers, declaring independently the nullity expectations for the array itself and the array elements. For any array type `T[]` the corresponding variants are `T?[]?`, `T?[]!`, `T![]?`, `T![]!` (the first modifier applies to the type of objects stored in the array, while the second modifier concerns the reference to the array object itself).

Assuming that we want to guarantee a statically sound approach to subtyping (that is, we want to enforce safety at compile time, without using runtime checks), explain whether or not the following subtype relations are safe. For each relation you consider unsafe, provide a code snippet illustrating that allowing such a subtype relationship would break the safety guarantees of the type system. For these unsafe cases, explain also what runtime checks could be made to restore safety.

- `T?[]! <: T?[]?`
- `T![]! <: T![]?`
- `T![]? <: T?[]?`
- `T![]! <: T?[]!`

solution

- `T?[]! <: T?[]?` - Safe
- `T![]! <: T![]?` - Safe
- `T![]? <: T?[]?` - Unsafe

```
Object![]? x = new Object![1]?;
Object?[]? y = x;
if(y!=null) y[0]=null;
if(x!=null) x[0].toString();
```

- `T![]! <: T?[]!` - Unsafe

```
Object![]! x = new Object![1]!;
Object?[]! y = x;
y[0]=null;
x[0].toString();
```

In both the last two cases, we need to check at runtime if a value stored in an array with dynamic non-null type for the elements stored in the array is not the `null` value. Alternatively, we can check at runtime if a value read from an array with dynamic non-null type is not the `null` value.