ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Readonly and Ownership Types

November 28, 2014

Task 1

Consider the following classes:

```
class A {
   readwrite StringBuffer n1=...;
   readonly StringBuffer n2=...;
}

class B {
   readwrite A x;
   readonly A y;
   public B(readwrite A x, readonly A y) {
      this.x=x;
      this.y=y;
   }
}
```

Check which programs typecheck and explain why they do or do not typecheck.

Program 1

```
readwrite A obj=new A();
readonly  B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.y.n1;
```

Program 2

```
readwrite A obj=new A();
readwrite B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.y.n1;
```

Program 3

```
readwrite A obj=new A();
readwrite B obj2=new B(obj, obj);
readwrite StringBuffer v=obj2.x.n1;
```

Program 4

```
readonly  A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readwrite StringBuffer v=obj3.y.n1;
```

Program 5

```
readwrite A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readonly  StringBuffer v=obj3.y.n1;
```

Program 6

```
readwrite A obj=new A();
readonly  A obj2=new A();
readwrite B obj3=new B(obj, obj2);
readonly  StringBuffer v=obj3.y.n2;
```

Task 2

Consider how we might extend readonly types to handle arrays. For an array of primitive types, one might want to declare that the array elements cannot be changed, using a declaration such as:

```
readonly int[] x;
```

which, similarly to the rules for accessing fields of objects, would forbid an update such as

```
x[2] = 2; // error - x is declared with a readonly type
```

A) Should there be a subtyping relationship (in either direction) between types `readwrite int[]` and `readonly int[]`?

B) For arrays of reference types, there are two reasonable questions to consider for readonly typing. Firstly, just as for an array of primitive types, whether or not the array reference can be used for modifications. Secondly, whether the array elements can be used for modifications.

```
y[1] = y[2];      // is this allowed?
y[1].f = y[2].f; // is this allowed?
```

In order to express all possibilities, consider having two `readonly`/`readwrite` modifiers for an array type - the first to denote access to the array reference itself, and the second to denote access to the objects stored in its elements, e.g., `readonly readonly T[] y;` is the most restrictive possibility.

Consider what the semantics of `readwrite readonly T[] y;` could be. There are two reasonable choices here, depending on whether we regard accesses of the form `y[1].f` as accesses which go first via the array `y`, and then to the element `y[1]`, or consider them as accesses directly to `y[1]` (in a sense "skipping" `y`, and hence ignoring the first modifier).

For each of these two possible semantics, consider the following:

- Do all four combinations of modifiers express something different from one another?
- What subtyping relationships (if any) would be reasonable between the four possible variants of a `T[]` type?

C) In the light of these questions, which of the two semantics seems the best choice?

Task 3

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.

Task 4

(The following question is taken 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;`

Task 5

Consider the typing rules for a field update $e_1.f = e_2$ (lecture 6, slide 64)

A) Consider two particular cases: e_2 is typed with the ownership modifier `any`, or e_2 is typed with the ownership modifier `lost`.

Suppose that e_2 refers to an object (i.e., not `null`). Is there a difference between the information that these two modifiers convey about where this object is located in the heap topology of ownership trees?

Can you find an example (by choosing the ownership modifiers for e_1 and `f`) when a field assignment would be typeable in one of the two cases (of e_2 being `any` or `lost`) but not the other? Explain briefly why this is the case.

B) Suppose instead that e_1 is typed with ownership modifier $\tau(e_1)$ and `f` has ownership modifier $\tau(f)$. We consider two different cases: $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `any`, or $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `lost`.

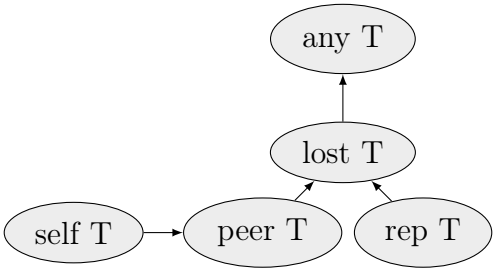
Is there a difference between the information that these two modifiers convey about *topological requirements* associated with the location $e_1.f$ (i.e., what needs to be guaranteed before an object can be validly assigned to this location)?

Can you find an example (by choosing the ownership modifier for e_2) when a field assignment would be typeable in one of the two cases (of $\tau(e_1) \blacktriangleright \tau(f)$ being `any` or `lost`) but not the other? Explain briefly why this is the case.

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.



B) Define the most specific (in terms of the context information it conveys) viewpoint adaptation function $\blacktriangleright$ by filling the table below (for a combination $T_e \blacktriangleright T_f$ 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 $T_e \blacktriangleright T_f$. Note that this applies to field updates as well as field reads.

▶	peer	rep	any	down
self				
peer				
rep				
lost				
any				
down				

C) 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?

D) In writing your answers for **B** and **C** 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?
   }
}
```

Do your solutions disallow all invalid assignments?