Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 13

Readonly Types and Ownership Types

December 18, 2020

Task 1 (from a previous exam)

In the `readonly/readwrite` type system, which of the following assignments is not type correct?

1. `x = y;` where `x` is `readonly` and `y` is `readwrite`
2. `x = y.f;` where `x` is `readwrite`, variable `y` is `readonly` and field `f` is `readwrite`
3. `x = y.f;` where `x` is `readwrite`, variable `y` is `readwrite` and field `f` is `readwrite`
4. `x = y.f;` where `x` is `readonly`, variable `y` is `readwrite` and field `f` is `readwrite`

Task 2

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;
   }
}
```

Note that the `readwrite` annotations could have been omitted, since `readwrite` is the default; they are written explicitly here for clarity.

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 3

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 the 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] = ...;   // is this allowed?
y[1].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 `readonly readwrite 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 4

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. Assume that overriding methods can have covariant return types and contravariant parameter types.

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?

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`.

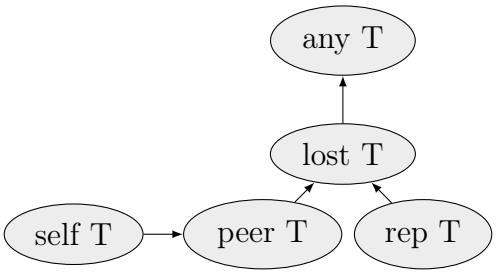
Note that:

- 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.

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 viewpoint adaptation function ▶, such that it is the most specific in terms of the context information it conveys (i.e. it conveys as much context information as possible), 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) 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?

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?