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;
```

solution

- **Program 1** does not compile since `obj2` is `readonly`, so `obj2.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
- **Program 2** does not compile since field `y` in `B` is `readonly`, so `obj2.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
- **Program 3** compiles! `obj2` is `readwrite`, `x` is `readwrite` (so `obj2.x` is `readwrite`), `n1` is `readwrite` (so `obj2.x.n1` is `readwrite`), and we assign `obj2.x.n1` to a `readwrite` variable.

- **Program 4** does not compile since `obj` is `readonly` and it is passed to the constructor of `B` as the first argument, while the constructor expects a `readwrite` variable.
- **Program 5** compiles! We can always assign something to a `readonly` variable.
- **Program 6** compiles! We can always assign something to a `readonly` variable.

In addition: for all the programs expect 4, the first argument passed to the constructor of `B` is `readwrite`, and the second argument can be `readwrite` or `readonly` since a `readonly` argument is expected.

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[]`?

solution

`int[] readonly` is more restrictive than `readwrite int[]` (fewer operations can be performed with such a reference) so we could have `readwrite int[] <: 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?

solution

Considering `y[1].f` as an access which goes first via `y`, and then `y[1]`, we would obtain that:

- If the first modifier is `readonly`, all the accesses to elements of the array will be treated as `readonly`, since the `readonly` modifier for the array will be considered first. Therefore, the only interesting combinations are:
  (a) `readonly readonly`
  (b) `readwrite readonly`
  (c) `readwrite readwrite`

  Note: The same approach is adopted when we have a `readonly` object variable and we access a `readwrite` field through it: the result would be `readonly`, since any access via a `readonly` reference is `readonly`.
- (a) is more restricted than (b), and (b) is more restricted than (c). So the reasonable subtyping relations are (c) `<:` (b) `<:` (a)

Considering `y[1].f` as a direct access, we would obtain that:

- All the four different combinations have different semantics. With respect to the previous example, we would have that `readonly readonly` will allow only read accesses both on the array and on the elements stored in it, while with `readonly readwrite` we have that we cannot assign elements in the array but we can write fields accessed via the array elements.
- The subtyping relations already pointed out still work. In addition we could have
  (a) `readonly readwrite <: readonly readonly`
  (b) `readwrite readwrite <: readonly readwrite`

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

solution

The second solution is more expressive than the first one, since it allows the developer to have more fine-grained control on the read and write accesses on arrays and on their elements. Thus, the second choice seems to be the best. However, it should be carefully considered whether such an approach (that would be different compared to the one adopted for objects and field accesses) may confuse the developers, and eventually create safety problems.

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.

solution

The general typing rules are `peer <: any` and `rep <: any` since `any` is more 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 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;`

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 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.

solution

There is no difference between the information that these two modifiers convey about where this object is located in the heap topology; something referred to by either `any` or `lost` could have any owner. There is no example where we could use `lost` and not `any` as the type for e_2 or vice versa; in fact, the only time either would be acceptable is if the field `f` was typed with the `any` modifier.

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.

solution

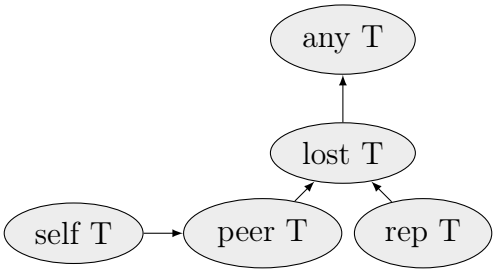
In the case where $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `any`, this indicates that there are *no* requirements that need to be satisfied in order for such a field update to preserve topological information. On the other hand, if $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `lost`, then this indicates that there *are* requirements that need to be satisfied, but that the type system is not able to describe them precisely (for example, we assign to the `rep` field of a `rep` reference; there is no ownership modifier to describe the requirements here). For this reason, such field updates are never allowed.

If e_2 is any reference with an appropriate class type (it doesn't make a difference what the ownership modifier is), then the field update e_1`.f = `e_2 will be allowed when $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `any`, and disallowed if $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier `lost`.

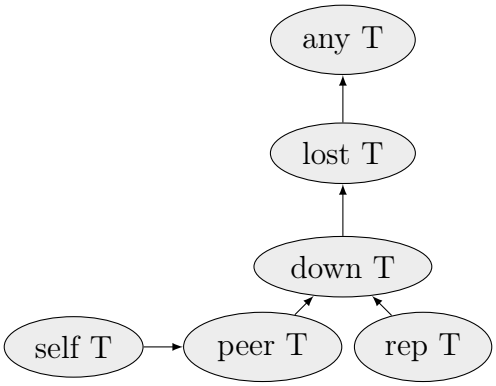
Task 6

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) Define the most specific (in terms of the context information it conveys) viewpoint adaptation function ▶ by filling the table below (for a combination $T_e \blacktriangleright T_f$ 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				

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 **C** 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 **D** 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).

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?

solution

With the first (most precise) variant of the viewpoint adaptation function from **B** 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 **B**, we do not need to make any changes to the field assignment rule, to guarantee the topological constraints of the type system.

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?

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`.

Both would be disallowed by the solutions above.