ETH
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`

solution

Number 2 is not allowed - it casts from a `readonly` reference to a `readwrite` reference.

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;
```

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 also `readwrite`, `n1` is `readwrite`, so `obj2.x.n1` is also `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 except 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 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[]`?

solution

`readonly int[]` 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] = ...;   // 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?

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` (equivalent to `readonly readwrite`)

  (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`.

- The reasonable subtyping relations are (b) `<:` (a) and (c) `<:` (a). The case (b) `<:` (a) corresponds to invariant array typing. The (c) `<:` (a) case corresponds to covariant array typing but it is sound since the array type in (a) is `readonly` and, thus, an array element type only appears in covariant position (e.g., `v := a[i]`).

  Note that the relation (c) `<:` (b) would also correspond to covariant array typing but it would not be sound since it would indirectly allow casting a `readonly` reference to a `readwrite` reference:

```
class P { String n; }

class C {
    void client(readonly P p) {
        readwrite readwrite P[] w = new P[1];
        readwrite readonly P[] r = w;
        r[0] = p;         // legal since r[0] and p are readonly
        w[0].n = "...";   // legal since w[0] is readwrite
    }
}
```

The assignment in the third line of `client` is legal since we have `readwrite` as the first modifier of `r`. Moreover, note that `p` should not be modifiable within the `client` method, as it is passed as `readonly`. However, by allowing the alias in the second line of the method, we enable a way to change `p`. This is undesirable and unsound. The implicit cast from `readonly` to `readwrite` is done on `p` here.

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 cannot assign elements in the array but we can write fields accessed via the array elements.
- The subtyping relations are:
  (a) `readwrite readonly <: readonly readonly`
  (b) `readonly readwrite <: readonly readonly`
  (c) `readwrite readwrite <: readonly readwrite`
  (d) `readwrite readwrite <: readonly readonly`

  Note that we still have that `readwrite readwrite` $\not<:$ `readwrite readonly`. This subtype relation is not reasonable; if it were allowed, then we could use the example from the first semantic to modify an object through a `readonly` reference.

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 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.

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 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

We can add the following constructor to `T`:

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

and use the following code C:

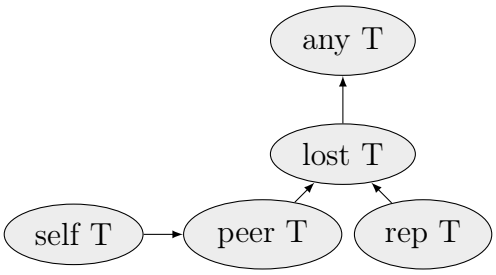
```
e1 = new peer T();
e2 = new peer T();
```

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.

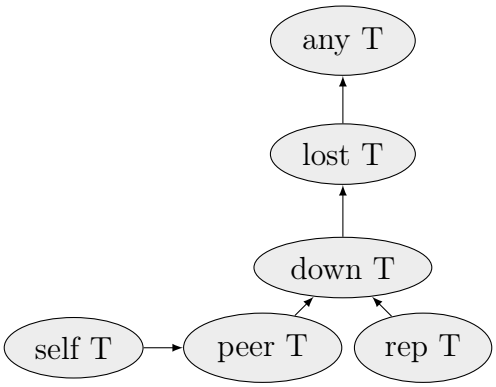
Task 6

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) 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				

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. 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 we need to add extra restrictions on field assignments in the cases where we use `down` to over-approximate in this way.

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 with 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. This second approach is not very flexible; only `rep` and `peer` objects can ever be typed as `down` (via subtyping).

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?

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`.

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 **B** we have to make sure that the examples from part **C** do not type-check, as those field updates are unsafe. Therefore, 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.