ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 10

Readonly and Ownership Types

November 29, 2013

Task 1

The intuition behind a pure method is that its execution effects are not observable by the client. This essentially means that the result of any other method call or field read inside client code would not be affected by a pure method execution. One way to formalize this property is to require that the execution of a pure method does not change the program heap.

- Provide proof obligations that guarantee the purity of a method, according to this requirement. Can you define an analogous notion for constructors?
- Class `Set` represents a set of integers. Method `Set allLessThan(int bound)` (in class `Set`) returns a freshly-allocated instance of class `Set` that contains all elements of the original set that are smaller than `bound`.
  - Even though the method `allLessThan` does not change the behavior of other methods, it is not pure, according to our definition. Why?
  - How can the provided definition of purity be relaxed to allow declaration of the method `allLessThan` as pure, without violating the intuition above?
  - Provide proof obligations that guarantee purity of a method according to your relaxed definition.
  - Can you define an analogous notion for constructors?

Solution

- A method is pure if and only if:
  1. It does not contain field updates
  2. It does not invoke non-pure methods
  3. It does not create objects

  We cannot reasonably provide an analogous notion for constructors, since a constructor call is guaranteed to modify the heap.
-
  - Method `allLessThan` is not pure because it allocates new objects. Furthermore, it must either make field updates or call non-pure methods in order to add all the elements that are less than the given bound to the set returned by `allLessThan`. Nonetheless, it seems likely it does not change the behavior of other methods, and we would like to consider it as pure.
  - We need to allow "pure" methods to allocate new objects, and to perform modifications on those newly-allocated objects. In this case, we say that the method is "weakly pure"

– We shall use the readonly type system - a method is "weakly pure", if:

1. All its arguments are readonly
2. The receiver is treated as readonly - we can annotate the method in a similar way to C++ `const` methods:

```
Set allLessThan(int bound) readonly {
   Set result = new Set();
   for (readonly Node n = head; n!=null; n=n.next)
      if (n.val<bound)
         result.append(n.val);
   return result;
}
```

We assume the set is implemented as a linked list. We can access `this.head` as `readonly` as `this` is `readonly`. The disadvantage here is that we cannot store read/write references to our arguments (e.g. we could not have that the new `Set` includes a read/write pointer to the old one, or it has read/write access to members if it did not clone them).

– For constructors, we can make the same requirements except that the `this` pointer can be read/write (but again could not store read/write pointers to arguments).

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
```

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

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 `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?

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

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[]`.
- Considering `y[1].f` as an access which goes first via `y`, and then `y[1]`, we would obtain that:
  1. 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`.
  2. (a) is more restricted than (b), and (b) is more restricted than (c). So the reasonable subtyping relations are (a) `:>` (b) `:>` (c)
- Considering `y[1].f` as a direct access, we would obtain that:
  1. 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.
  2. The subtyping relations already pointed out still work. In addition we could have
     (a) `readonly readonly :> readonly readwrite`
     (b) `readonly readwrite :> readwrite readwrite`
- 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

In this question assume no type-casts or static variables or fields are used.

The C++ language supports the `const` modifier for types, which tries to model a weak readonly type system.

- The C++ type system does not ensure transitive readonly structures as the system shown in class. Show which typing rules could be changed and how to ensure transitivity (consider both pointers and references). Does this ensure that `x.f` is not modified in the method `m`?

```
class C{
```

```
    public: int f = 0;
}

void m(const C& x){...}
```

- Considering the changes in the previous section, show an example where the method `n` does modify `x.f`. Is this a problem?

```
void n(const C& x, C& y){...}
```

- The `mutable` modifier is used in C++ to denote a field that can be mutated also in `const` objects - meaning that its value does not affect the client visible behaviour of the object (such as caching the results of a time consuming calculation) - consider the following code:

```
class List{
    ...

public:
    ///ensures result >= 0
    int length()const{...}

    ///requires index >= 0 && index <length()
    int at(int index) const {
        if (index == lastSearch)
            return lastSearchResult;
        else
        {
            int result = atHelper(index);
            lastSearch = index;
            lastSearchResult = result;
            return result;
        }
    }

private:
    int atHelper(int index) const{...} //Time consuming
    mutable int lastSearch=-1;
    mutable int lastSearchResult=0;
}
```

  In this section assume that the `const` modifier is transitive for both pointers and references. We try to prove correctness of the `at` method by showing that we get the same result regardless of the values of `lastSearch` and `lastSearchResult`. However, this requires a stronger class invariant - give such an invariant, assuming that `atHelper` is pure (and does not modify even `mutable` fields).

Solution

- The change that is needed is in the typing of field dereference - so that dereferencing a pointer/reference field of a `const` type gives a pointer/reference to `const`. This would prevent `m` from modifying `x.f` as transitive `const`ness is ensured.

- `n` can modify `x.f` through aliasing - for example:

```
void g()
{
    C& c = *new C();
    assert(c.f==0);
    n(c,c);
    assert(c.f==0); //fails
}
```

```
void n(const C& x, C& y){
   y.f=1;
}
```

This is not a problem, as the only guarantee the system gives is that no modification is done through `const` objects.

- We could add the class invariant: `lastSearch>=0 ==> (lastSearch<length()&& lastSearchResult == atHelper(lastSearch))`.

Task 4

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:

```
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

```
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];
 }
}
```

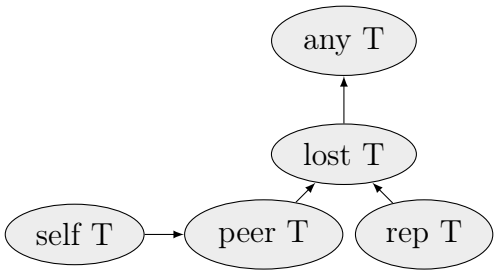
```
class Context {
 rep Producer p;
 rep Consumer c;

 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();
  }
 }
}
```

Note: You might be tempted to annotate `con` in `Producer` and `pro` in `Consumer` as `any`, but in the topological ownership system (without Owner-as-modifier) it is possible to also modify objects of type `any` and this will result in a less encapsulated design.

Task 5

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 ▶ 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				

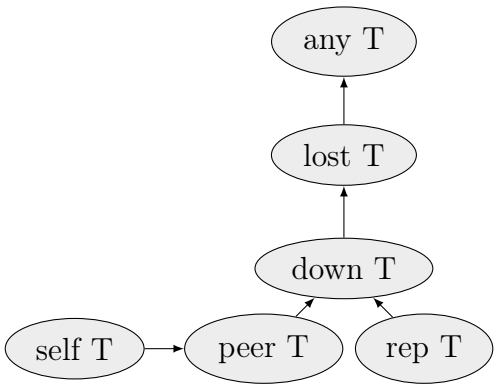
(c) Assuming that you only need to enforce the topological constraints of the type system , how should the field update rule from lecture 7 slide 40 be adapted to the system extended with the `down` modifier? Do you need to make any changes?

You might like to consider the following example code, in assessing your answers to (b) and (c):

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

   public void foo() {
      this.d.d = this;    // does this/should this type-check?
      this.c.d = this.d; // does this/should this type-check?
   }
}
```

Solution

(a)



(b) There are two reasonable approaches to defining viewpoint adaptation. One (and perhaps the most intuitive) is to define it as describing the most precise information possible about where such a reference may belong in the heap topology (possibly over-approximating, in cases where we cannot describe precisely what we want). For example, `rep▶rep` can be `down` in this approach, 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 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 (c) would type-check, which would not be safe. Here is the appropriate table, taking this approach:

▶	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

The alternative approach is not to allow this kind of over-approximation; the modifier chosen has to 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. See also the code in the next part. As you can see, this second approach is not very flexible; only `rep` and `peer` objects can ever be types as `down` (via subtyping).

(c) Depends on the answer to the previous part. In the first case, we need to require that the result of the viewpoint adaption 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 approach, we do not need to make any changes to the field assignment rule, to guarantee the topological constraints of the type system (as long as we define viewpoint adaptation as above).

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`.

These would be disallowed by both of our solutions above (check this!):

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

   public void foo() {
      this.d.d = this;    // does this/should this type-check?
      this.c.d = this.d; // does this/should this type-check?
   }
}
```