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?

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?

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).

Task 4

Annotate the following program with appropriate ownership type modifiers 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;
 }
}
```

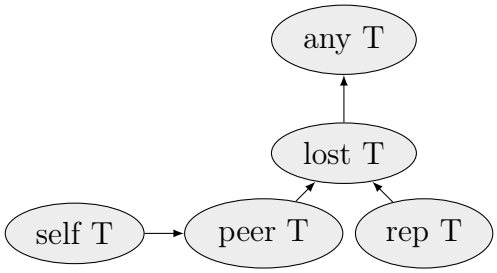
```
class Consumer {
 int[] buf;
 int n;
 Producer pro;
 Consumer(Producer p)
 {
  buf = p.buf;
  pro = p;
  p.con = this;
 }
 int consume()
 {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
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();
  }
 }
}
```

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?
   }
}
```