Advanced Programming Methods
Seminar 14

1. Type-based analyses: a compile-time analyses that use type annotations to trace some program properties
   - Type annotations: information available at compile time, ex: size types with append example:

List^r append(List^m a, List^n b) where r=m+n {....}
r- nr of the result elements
m- nr of the list a elements
n - nr of the list b elements

Size informations are traced during a symbolic execution in order to determine some useful properties like indexes for array access, memory size, etc.

2. Heap management:
   - garbage collector: unpredictable in time and space, not suitable for real-time and embedded systems
   - manual memory management: error prone (too early deallocation), efficiency (too late deallocation)
   - automatic compile-time management: new/delete safely introduced by the compiler based on a compile-time analysis

Region-based memory management:

- a region is a memory space with designed lifetime
- objects having the same lifetime are allocated in the same region
- entire region is deallocated

Region Settings

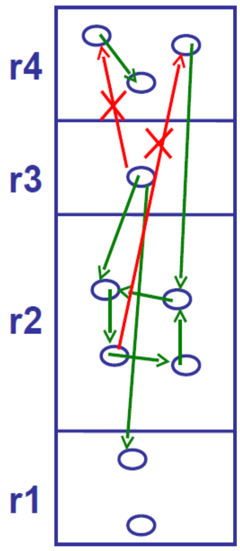
1. Lexically-Scoped Regions:
   - region allocation/deallocation :

     letreg r in e

   - stack discipline
   - outlive relation: r2 > r3

2. No Possible Dangling References: no references from older regions to younger regions

top (younger regions)



bottom (older regions)

Class Declaration – Region Types

First region **(r1)** is to store the receiver object (this).

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩
  where r2⪰r1 ∧ r3⪰r1 {
  Object⟨r2⟩ fst
  Object⟨r3⟩ snd
  ...
}
```

Next regions **(r2,r3)** are to store the fields.

No dangling references property as class invariant.

Subtyping

```
Pair⟨r1,r2,r3⟩ p1;
Pair⟨r1',r2',r3'⟩ p2;
  p2=p1;
```

Region Subtyping Principle:

An object stored in an older region can be passed to a location that expects a younger region.

$$\frac{r1 \succeq r1' \ \wedge \ r2 = r2' \ \wedge \ r3 = r3'}{\mathsf{Pair}\langle r1, r2, r3\rangle <: \mathsf{Pair}\langle r1', r2', r3'\rangle}$$

Method Declaration

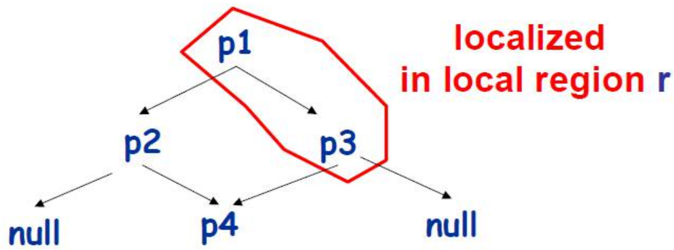
```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1 {
  ...
  void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o) where r4⪰r3
   {snd=o}
}
```

Method Region parameters (regions of the method arguments, receiver and result)

Method precondition (region constraints from method body)

Region Allocation/Deallocation

```
Pair⟨r5,r6,r7⟩ exalloc⟨r5,r6,r7⟩() where r7⪰r5 ∧ r6⪰r5
{
 letreg r in {
 Pair⟨r7,r7,r7⟩ p4 =  new Pair⟨r7,r7,r7⟩(null,null);
 Pair⟨r,r,r⟩ p3    =  new Pair⟨r,r,r⟩(p4,null);
 Pair⟨r5,r6,r7⟩ p2 =  new Pair⟨r5,r6,r7⟩(null,p4);
 Pair⟨r,r,r⟩ p1    =  new Pair⟨r,r,r⟩(p2,null);
 p1.setSnd⟨r,r,r,r⟩(p3);
 return p2
  }
}
```




```
{Pair p4=new Pair(null,null);
 Pair p3=new Pair(p4,null);
 Pair p2=new Pair(null,p4);
 Pair p1=new Pair(p2,null);
 p1.setSnd(p3);
 p2
}
```

(b) Source Program

```
{Pair⟨r4,r4a,r4b⟩ p4 = new Pair⟨r4,r4a,r4b⟩(null,null);  //r4a⪰r4∧r4b⪰r4
 Pair⟨r3,r3a,r3b⟩ p3 = new Pair⟨r3,r3a,r3b⟩(p4,null);  // r3a⪰r3∧r3b⪰r3∧r3a=r4
 Pair⟨r2,r2a,r2b⟩ p2 = new Pair⟨r2,r2a,r2b⟩(null,p4);  // r2a⪰r2∧r2b⪰r2∧r2b=r4
 Pair⟨r1,r1a,r1b⟩ p1 = new Pair⟨r1,r1a,r1b⟩(p2,null);  // r1a⪰r1∧r1b⪰r1∧r1a=r2
 p1.setSnd⟨r3⟩(p3); // r1b=r3
 p2
} ::  Pair⟨r2,r2a,r2b⟩
```

(c) Region-annotated target program.

```
r3a=r2b=r4 ∧ r1a=r2

rs = {r3,r3b,r1,r1b} does not
        outlive {r2,r2a,r2b}
```

```
letreg r in
 { Pair⟨r4,r4a,r4b⟩ p4 =  new Pair⟨r4,r4a,r4b⟩(null,null);
   Pair⟨r,r4,r⟩ p3     =  new Pair⟨r,r4,r⟩(p4,null);
   Pair⟨r2,r2a,r4⟩ p2  =  new Pair⟨r2,r2a,r4⟩(null,p4);
   Pair⟨r,r2,r⟩ p1     =  new Pair⟨r,r2,r⟩(p2,null);
   p1.setSnd⟨r⟩(p3);
   p2
 }
```

Inference Rules

$$\vdash \text{def} \Rrightarrow \text{def}' \; ; \; \text{Q}$$

$$\Gamma \vdash \text{meth} \Rrightarrow \text{meth}' \; ; \; \text{Q}$$

$$\Gamma \vdash \text{e} \Rrightarrow \text{e}' \; : \; t \; ; \; \phi$$

$$\frac{\begin{array}{c}\Gamma \;\vdash e_1 \Rrightarrow e_1' \;:\; t_1, \phi_1 \\ \Gamma \;\vdash e_2 \Rrightarrow e_2' \;:\; t_2, \phi_2\end{array}}{\Gamma \;\vdash e_1 \,;\, e_2 \;\Rrightarrow e_1' \,;\, e_2' \;:\; t_2, \phi_1 \wedge \phi_2}$$

Recursive Example

```
class List⟨r1,r2,r3⟩ extends Object⟨r1⟩
  where r2⪰r1 ∧ r3⪰r1 ∧ r2⪰r3
{
     Object⟨r2⟩ value
     List⟨r3,r2,r3⟩ next
     ...
}
```

```
List join(List xs, List ys)
{if isNull(xs) then
   if isNull(ys) then (List)null
   else join(ys,xs)
  else {
   x=xs.getValue();
   r=join(ys,xs.getNext());
   new List(x,r)
   }
}
```

The Recursive Method: `join`

```
List⟨r7,r8,r9⟩ join⟨r1,..,r9⟩
  (List⟨r1,r2,r3⟩ xs, List⟨r4,r5,r6⟩ ys)
   where pre.join⟨r1,..,r9⟩
     :

Q={pre.join⟨r1,..,r9⟩=(r2⪰r8)
   ∧pre.join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩
   ∧pre.join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩}
```

Inferred Method: `join`

$$
\begin{aligned}
\texttt{pre.join}_0\langle \texttt{r1},..,\texttt{r9}\rangle &= \texttt{True}\\
\texttt{pre.join}_1\langle \texttt{r1},..,\texttt{r9}\rangle &= \texttt{r2}\succeq\texttt{r8} \wedge \texttt{pre.join}_0\langle \texttt{r4},\texttt{r5},\texttt{r6},\texttt{r1},\texttt{r2},\texttt{r3},\texttt{r7},\texttt{r8},\texttt{r9}\rangle\\
&= \texttt{r2}\succeq\texttt{r8}\\
\texttt{pre.join}_2\langle \texttt{r1},..,\texttt{r9}\rangle &= \texttt{r2}\succeq\texttt{r8} \wedge \texttt{pre.join}_1\langle \texttt{r4},\texttt{r5},\texttt{r6},\texttt{r1},\texttt{r2},\texttt{r3},\texttt{r7},\texttt{r8},\texttt{r9}\rangle\\
&= \texttt{r2}\succeq\texttt{r8} \wedge \texttt{r5}\succeq\texttt{r8}\\
\texttt{pre.join}_3\langle \texttt{r1},..,\texttt{r9}\rangle &= \texttt{r2}\succeq\texttt{r8} \wedge \texttt{pre.join}_2\langle \texttt{r4},\texttt{r5},\texttt{r6},\texttt{r1},\texttt{r2},\texttt{r3},\texttt{r7},\texttt{r8},\texttt{r9}\rangle\\
&= \mathbf{r2}\succeq\mathbf{r8} \wedge \mathbf{r5}\succeq\mathbf{r8}
\end{aligned}
$$

Fixed-Point Analysis