Region Type System + Exists

Tuesday, August 12, 2014 11:32 AM

In this wiki, we extend the region type system with explicit existential typing for transferable region handlers. The motivation is to allow data structures (eg: linked list) of transferable regions, while making best effort to disallow dangling references that result from freeing/transferring a transferable region.

This wiki reproduces contents from previous wiki for the sake of completeness.

FJ with Regions

The Language

$\rho, \pi, p \in \ region\ names$
$cn \ \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle p^a \bar{p} \rangle\langle \bar{\tau} \rangle$ //Instantiated class type
$C \ ::=$ class $cn\langle \rho^a \bar{\rho} \mid \phi \rangle\langle \overline{\alpha \lhd N} \rangle \lhd N \{\overline{\tau\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{\tau\ x})\{$ super $(\bar{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d \ ::= \ \tau\ mn\langle \rho^a \bar{\rho} \mid \phi \rangle\ (\overline{\tau\ x})\ \{s;$ return $e;\}$ //Methods
$\phi \ ::= true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ // constraints on region params
$\tau_{\lhd} ::= \ \alpha \mid N$ //Types that admit subtyping (subclassing)
$T ::= \ int \mid bool \mid unit$//Unboxed Types
$\tau \ ::= \ \tau_{\lhd} \mid Object\langle p^a \rangle \mid Region[\rho]\langle p^a \rangle\langle \tau \rangle \mid int \mid bool \mid unit \mid \exists \rho.\tau$
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid \ letregion\langle \rho \rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \mid e.suck(e)$
$\quad \mid \ let\ (\rho, \tau\ x) = unpack\ e$
$e \ ::= \ c \mid x \mid e.f \mid e.mn\langle p^a \bar{p} \rangle(\bar{e}) \mid$ new $N(\bar{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle \rho \rangle\langle \tau \rangle()$
$\quad \mid pack[\rho, e]\ as\ \exists \rho.\tau$ //Expressions

Notes

1. A note on notation: We write $\bar{a}$ to denote sequence of a's (i.e., $a_0, a_1, \ldots, a_i, \ldots, a_n$). Identifier a (without numberic subscript) is different from any a's in $\bar{a}$. When b is an identifier and $\odot$ denotes a binary relation, we write $b \odot \bar{a}$ to denote either $b \odot a_0 \wedge \cdots \wedge\ b \odot a_i\ \wedge \cdots$ or the set $\{b \odot a_0, \ldots, b \odot a_i, \ldots\}$, depending on the context. Similarlly, $\bar{b} \odot \bar{a}$ or $\overline{(b \odot a)}$ denotes point-wise $\odot$ relation between b's and a's.
2. We write $\alpha \lhd N$ to denote that α is subclass of N. We call the instantiated class N as the bound of the type variable α.
3. A region name (ρ, p or π) is a static identifier for a region created by either a $letregion$ expression or a $new\ Region$ expression. If such an expression is inside a loop or a recursive function, it creates multiple regions at run-time, all of which have same static identifier. Any outlives relation that holds on two static identifiers also holds between corresponding regions at run-time. For eg, consider the following code:

```
let var x = …;
while (.…){
```

```
  letregion<R0> {
    letregion<R1> {
      …
    }
  }
}
```

The outlives relation (R0⩾R1) that holds between static identifiers R0 and R1 inside while loop, also holds between run-time regions denoted by by R0 and R1 in every iteration of the while loop. It is possible to create an object in R1 that holds a reference to an object in R0. However, the outlives relation does not hold between regions across iterations. Accordingly, it is not possible to create an object inside R0 or R1 in one iteration, and use its reference in later iterations. The only way this could have been done is by assigning the object reference to a variable (x) that is declared above while statement, and dereference x in later iterations, but this is disallowed by region type system as variable x has longer life time than the objects in R0 or R1. It should be observed that region type system did not allow incorrect outlives relation between run-time regions identified by R0 and R1 across multiple iterations, while still allowing sound outlives relation between regions created in a single iteration.

4. We have region-polymorphic classes and region-polymorphic methods. Examples are given at the end of this wiki (LinkedListNode and LinkedList). The syntax of a region-polymorphic class is given below:
   class $B\langle \rho^a \bar{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{\tau\ f}; k\ ; \bar{d} \right\}$
   ρ^a denotes the allocation context parameter of class B. $\bar{\rho}$ is the sequence of free region identifiers (also referred to as "region variables") in types ($\bar{\tau}$) of instance variables ($\bar{f}$) of B. To ensure the safety invariant that longer-living objects cannot contain references to shorter-living objects, we require that $\bar{\rho} \geqslant \rho^a$. This constraint is implicit for all class definitions, and is not recorded anywhere. The constraint is checked when region parameters are instantiated with concrete region identifiers whenever a new object is created. For example, assuming π^a and $\overline{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\overline{\pi} \geqslant \pi^a$ holds.
5. The syntax of a region-polymorphic method (m) is given below:
   $\tau\ m\langle \rho^a \bar{\rho} \mid \phi \rangle\ (\overline{\tau\ x})\ \{s;\ \text{return}\ e;\}$
   As per our convention, ρ^a denotes allocation context of the method, and $\bar{\rho}$ denotes sequence of free region variables in argument and return types ($\bar{\tau}$ & τ). Note that ρ^a can also occur free in $\bar{\tau}$ and τ. Unlike region parameters of a class, we do not have any implicit "outlives" constraints on region parameters of a method. Instead, we explicitly maintain a conjunction of constraints as a refinement predicate (ϕ) over set of region parameters. The predicate ϕ constraints the set of concrete regions that can be used to instantiate the region parameters of the method, and should be viewed as the precondition for the method call to be region-safe.
6. All methods are parameterized over their allocation context; so, all methods are region-polymorphic w.r.t their allocation region. If a method requires its allocation context to be a specific region (eg: same region as the allocation context of its class), then this constraint needs to be stated as an explicit equality constraint in the refinement. For eg, method `refresh` in the following defn of class `Foo` requires its inAllocationContext to be a particular region:

```
class Foo<Ra0,R0> extends Object<Ra0> {
  Object<R0> x;
  unit refresh<Ra1 | Ra1 = R0>(unit u) {
    this.x = new Object<Ra1>();
    return ();
  }
}
```

7. Consider the following recursive method:

```
unit foo<Ra>(Object<Ra> x) {
  letregion<R0> {
```

```
    …
    Object<R0> y = new Object<R0>();
    foo<R0>(y);
    …
  }
}
```

The method `foo` is region-polymorphic with its inAllocationContext (`Ra`) as the only region parameter. The outlives relation (Ra⩾R0) holds between foo's inAllocationContext (Ra) and newly created static region (R0). We allow region-polymorphic recursion, letting foo to pass region R0 as inAllocationContext to its recursive call. Since R0 is Ra for the recursive call, there exists outlives relation between different run-time regions with same static identifier (R0) across recursive calls. This outlives relation is captured statically via the relation Ra⩾R0. It should be observed that region type system allowed sound outlives relation among regions created across recursive calls.

8. Base types, such as int and bool are unboxed and does not have region annotations. A Type variable itself does not have region annotations, but its bound class (the class it extends) tells us what region its values are allocated in .
9. We have existential types that are tailor-made only for transferable region handlers. These existential types bound region names (ρ) of transferable region handlers (eg: $\exists\rho.\, Region[\rho]\langle p^a\rangle\langle\tau\rangle$), allowing them to be stored in data structures such as linked lists. There is a $pack$ expression and an $unpack$ statement to introduce and eliminate existential typing for region handlers.
10. We have $newRgn$ construct to create new dynamic transferable regions. $newRgn$ returns a packed (i.e., existentially typed) transferable region handler. The construct is parameterized over the existentially bound name (ρ) of the new region and region type (τ) of the root object. Since transferable region should not refer to objects in other regions, the type system requires that τ refer only to ρ (i.e., $frv(\tau) = \rho$). Note that newRgn is only there in the target (elaborated) language. The source language (C#) uses the new keyword to create new regions
11. When a transferable region is unpacked, we introduce its name into the context. For eg, let ρ^a be the allocation context in:

    $let\ (R, Region[R]\langle\rho^a\rangle\langle\tau\rangle\ x) = unpack\ (newRgn\langle\rho\rangle\langle\tau\rangle());$

    Newly created transferable region is named R, and the name is in scope for subsequent statements. We use Σ to denote the set of region names currently in scope. Note that a region whose name is currently in scope need not necessarily be live. A transferable region is live only when it is open. We use Δ to denote the set of names of currently live regions.
12. Classes and region polymorphic methods are typed under the assumption that all region parameters are live. That is, while checking the body of $foo : \langle\rho^a\bar{\rho}\rangle\tau_1 \rightarrow \tau_2$, we assume that $\rho^a, \bar{\rho} \in \Delta$. This assumption captures the fact that we can't pass references to objects which are not live. When region parameters are instantiated at call-sites, we check if the instantiation arguments are live regions. A consequence of this assumption is that if a function foo expects a handler to a transferable region ρ, and generalizes ρ at the beginning of its type, then we cannot pass the handler unless the transferable region open. To get around this, we need to use existential type in the type of foo:

    $foo : \langle Ra, R0\rangle\ \exists\rho.\, Region[\rho]\langle R0\rangle\langle Object\langle\rho\rangle\rangle \rightarrow \cdots$

Auxiliary Definitions

Alloc Region: Allocation region argument of a class

$allocRgn(A\langle\rho^a\bar{\rho}\rangle) = \rho^a$

Fields: Type substitutions carried out in types of fields in current class, and types of fields in super class.

$$fields(\mathrm{T}) = \cdot \qquad \frac{CT(B) = class\ B\langle\rho^a\bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\{\overline{\tau_B\ f}; \ldots\} \quad fields([\bar{\rho}/\bar{\rho}][\bar{\tau}/\bar{\alpha}\ ]\ N) = \overline{\tau_A\ g}}{fields(B\langle p^a\bar{\rho}\rangle\langle\bar{\tau}\rangle) = \overline{[\bar{\rho}/\bar{\rho}][p^a/p][\bar{\tau}/\bar{\alpha}\ ]\tau_B\ f}, \overline{\tau_A\ g}}$$

ctype: Types of constructor arguments of a class.

$$ctype(\mathrm{T}) = \cdot \qquad \frac{CT(B) = class\ B\langle \rho^a \overline{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \{\ldots; k_B; \ldots\} \quad k_B = B(\overline{\tau\ x})\{\ldots\}}{ctype(B\langle p^a \bar{p} \rangle \langle \bar{T} \rangle) = \overline{[\bar{p}/\bar{\rho}][p^a/p][\bar{T}/\bar{\alpha}]\ \tau}}$$

Method Type Lookup: We need type of method mn in class B. Methods of same name can exist in various classes, so we need class name.

$$\frac{CT(B) = class\ B\langle \rho^a \overline{\rho} \mid \phi_B \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \ \{\overline{\tau_B\ f}; k_B; d_B\} \quad \tau^2\ mn\langle \rho_m^a \overline{\rho_m} \mid \phi \rangle\ (\overline{\tau^1\ x})\ \{return\ e;\} \in d_B}{mtype(mn, B\langle p^a \bar{p} \rangle \langle \bar{T} \rangle) = [\bar{p}/\bar{\rho}][p^a/p][\bar{T}/\bar{\alpha}]\ \langle \rho_m^a \overline{\rho_m} \mid \phi \rangle\ \overline{\tau^1} \to \tau^2}$$

$$\frac{CT(B) = class\ B\langle \rho^a \overline{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \ \{\overline{\tau_B\ f}; k_B; d_B\} \quad mn \notin FMN(d_B)}{mtype(mn, B\langle p^a \bar{p} \rangle \langle \bar{T} \rangle) = mtype(mn, [\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{T}/\bar{\alpha}]N)}$$

Method Def Lookup: We need the definition of mn in class B.

$$\frac{CT(B) = class\ B\langle \rho^a \overline{\rho} \mid \phi_B \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \ \{\overline{\tau_B\ f}; k_B; d_B\} \quad \tau^2\ mn\langle \rho_m^a \overline{\rho_m} \mid \phi \rangle\ (\overline{\tau^1\ x})\ \{return\ e;\} \in d_B}{mdef(mn, B\langle p^a \bar{p} \rangle \langle \bar{T} \rangle) = (\{\rho_m^a \overline{\rho_m} \mid \phi\}, \overline{x}, [\bar{p}/\bar{\rho}][p^a/p]\ [\bar{T}/\bar{\alpha}]e)}$$

$$\frac{CT(B) = class\ B\langle \rho^a \overline{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \ \{\overline{\tau_B\ f}; k_B; d_B\} \quad mn \notin FMN(d_B)}{mdef(mn, B\langle p^a \bar{p} \rangle \langle \bar{T} \rangle) = mdef(mn, [\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{T}/\bar{\alpha}]\ N)}$$

Notes on Typing Rules

- We define an environment Γ to map variables to types, and environment Δ to map type variables to their bounds. We abuse Δ and use it as a set of currently live region names. For clarity, Δ can be considered as a pair of contexts Δ_α and Δ_ρ, such that
  - Δ_α maps type variables to their bounds, and
  - Δ_ρ is a set of live region names.
- Subtyping is reflexive transitive closure of subclass relationship. Subclass relation is defined by the user over region-annotated classes. The only restriction we impose is that a class and its super class must have same allocation parameter (ρ^a). For a type variable, subclass relation is between the variable and its bound, as declared in the class definition.
- As usual (with Featherweight Java based systems), subtyping, typing and well-formedness judgments are parameterized over the class table (*CT*).
- We define $bound_\Delta$ function over types (τ). For a given type, the $bound_\Delta$ function identifies the class where we need to look for fields or methods.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
  $bound_\Delta(T) = T$
- We need well-formedness judgment to check:
  - All type variables are in scope
  - All region names are live
  - Type instantiation satisfies stated subclass constraints.
  - Region parameter instantiation satisfies stated outlives constraints.
-

$\Sigma; \Delta; \gamma \vdash \tau_1 <: \tau_2$

$$\frac{\square}{\Sigma;\Delta;\gamma\vdash\tau<:\tau} \qquad \frac{\square}{\Sigma;\Delta;\gamma\vdash\alpha <:\Delta(\alpha)} \qquad \frac{\begin{array}{c}CT(B)=class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N\ \{\ldots\}\\ \Sigma;\Delta;\gamma\vdash B\langle p^a\overline{p}\rangle\langle\overline{\tau}\rangle\ OK \qquad \Sigma;\Delta;\gamma\vdash[\overline{p}/\overline{p}][p^a/\rho^a][\overline{\tau}/\overline{\alpha}\ ]\ N\ OK\end{array}}{\Sigma;\Delta;\gamma\vdash B\langle p^a\overline{p}\rangle\langle\overline{\tau}\rangle <:\ [\overline{p}/\overline{p}][p^a/\rho^a][\overline{\tau}/\overline{\alpha}\ ]\ N}$$

$$\frac{\Sigma;\Delta;\gamma\vdash\tau_1<:\tau_2 \quad \Sigma;\Delta;\gamma\vdash\tau_2<:\tau_3}{\Sigma;\Delta;\gamma\vdash\tau_1<:\tau_3} \qquad \frac{\square}{\Sigma;\Delta;\gamma\vdash\exists\rho.\tau<:\exists\rho'.[\rho'/\rho]\tau}$$

$\Sigma;\ \Delta;\ \gamma \vdash \tau\ OK \quad \Delta \vdash \phi\ OK$

$$\frac{p^a\in\Delta}{\Sigma;\ \Delta;\gamma\vdash\text{Object}\langle p^a\rangle\ OK} \qquad \frac{\alpha\in dom(\Delta)}{\Sigma;\ \Delta;\gamma\vdash\alpha\ OK} \qquad \frac{\rho_0,\rho_1\in\Delta}{\Delta\vdash\rho_0\geqslant\rho_1\ OK} \qquad \frac{\Delta\vdash\phi_0 OK \quad \Delta\vdash\phi_1 OK}{\Delta\vdash\phi_0\wedge\phi_1\ OK}$$

$$\frac{\begin{array}{c}CT(B)=\ class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N\ \{\ldots\}\\ p^a,\overline{p}\in\Delta \quad \gamma\vdash\overline{p}\geqslant p^a \quad S=[\overline{p}/\overline{p}][p^a/\rho^a][\overline{\tau}/\overline{\alpha}\ ] \quad \gamma\vdash S(\phi)\\ \Sigma;\Delta;\gamma\vdash\overline{\tau}\ OK \qquad \Sigma;\Delta;\gamma\vdash S(\overline{N})\ \text{OK} \qquad \Sigma;\Delta;\gamma\vdash\overline{\tau}<:S(\overline{N})\end{array}}{\Sigma;\ \Delta;\gamma\vdash B\langle p^a\overline{p}\rangle\langle\overline{\tau}\rangle\ OK}$$

$$\frac{p^a\in\Delta \quad \rho\in\Sigma \quad \Sigma;\{\rho\};\emptyset\vdash\tau\ OK}{\Sigma;\Delta;\gamma\vdash Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK} \qquad \frac{\rho\notin\Sigma \quad \Sigma\cup\{\rho\};\Delta;\gamma\vdash Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK}{\Sigma;\Delta;\gamma\vdash\exists\rho.Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK}$$

$\Sigma;\ \Delta;\gamma;p^a;\Gamma\vdash e:\tau$

$$\frac{x:\tau\in\Gamma}{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash x:\tau} \qquad \frac{\begin{array}{c}\Sigma;\ \Delta;\ \gamma;\ p^a;\ \Gamma\vdash e:\tau'\\ f:\tau\in fields\left(bound_\Delta(\tau')\right)\end{array}}{\Sigma;\ \Delta;\ \gamma;\ p^a;\ \Gamma\vdash e.f:\tau} \qquad \frac{\Sigma;\Delta;\gamma\vdash\tau_\lhd\ OK}{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash Null:\tau_\lhd}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\gamma\vdash N\ OK \quad allocRgn(N)=p \quad \gamma\vdash p^a\geqslant p\\ ctype(N)=\overline{\tau} \quad \Sigma;\ \Delta;\ \gamma;\ p^a;\ \Gamma\vdash\overline{e}:\overline{\tau_e} \quad \Sigma;\Delta;\gamma\vdash\overline{\tau_e}<:\overline{\tau}\end{array}}{\Sigma;\ \Delta;\ \gamma;\ p^a;\ \Gamma\vdash new\ N(\overline{e}):N}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\gamma;p^a;\Gamma\vdash e_0:\tau \qquad \overline{p}\in\Delta\\ mtype\left(m,bound_\Delta(\tau)\right)=\langle\rho^a\overline{\rho}\mid\phi\rangle\ \overline{\tau^1}\rightarrow\tau^2 \quad S=[\overline{p}/\overline{p}][p^a/\rho^a]\\ \Delta;\gamma\vdash S(\overline{\tau^1})\ OK \qquad \Delta;\gamma\vdash S(\tau^2)\ OK \qquad \Sigma;\Delta;\gamma;p^a;\Gamma\vdash\overline{e}:\overline{\tau_e} \qquad \Sigma;\Delta;\gamma\vdash\overline{\tau_e}<:S\left(\overline{\tau^1}\right) \quad \gamma\vdash S(\phi)\end{array}}{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash e_0.m\langle p^a\overline{p}\rangle(\overline{e}):S(\tau^2)}$$

$$\frac{\rho\in\Delta \quad \Sigma;\Delta;\gamma;\rho^a;\Gamma\vdash e:Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle}{\Sigma;\Delta;\gamma;\rho^a;\Gamma\vdash e.get():\tau} \qquad \frac{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash e:[\rho_0/\rho]\tau \quad \Sigma;\Delta;\gamma\vdash\exists\rho.\tau\ OK}{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash pack[\rho_0,e]\ as\ \exists\rho.\tau:\exists\rho.\tau}$$

$$\frac{\Sigma;\Delta;\gamma\vdash\exists\rho.Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK}{\Sigma;\Delta;\gamma;p^a;\Gamma\vdash newRgn\langle\rho\rangle\langle\tau\rangle():\exists\rho.Region[\rho]\langle p^a\rangle\langle\tau\rangle}$$

$\Sigma;\ \Delta;\ \gamma;\ \rho^a;\ \Gamma\vdash s\Rightarrow\Gamma';\Sigma'$

$$\frac{\Sigma;\Delta;\gamma \vdash \tau\ OK}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e{:}\tau_1 \quad \Sigma;\Delta;\gamma \vdash \tau_1 <: \tau}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash let\ \tau\ x{=}e \Rightarrow \Gamma, x{:}\tau;\ \Sigma}$$

$$\frac{e_1 \in \{x, e.f\} \quad \Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e_1{:}\tau_1}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e_2{:}\tau_2 \quad \Sigma;\Delta;\gamma \vdash \tau_2 <: \tau_1}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e_1{=}e_2 \Rightarrow \Gamma;\Sigma}$$

$$\frac{\rho \notin \Delta \quad \Sigma \cup \rho;\ \Delta \cup \rho;\ \gamma \cup \{\Delta \succcurlyeq \rho\},\ \rho,\ \Gamma \vdash s \Rightarrow \Gamma';\Sigma'}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash letregion\langle\rho\rangle\ \{\ s\ \} \Rightarrow \Gamma;\Sigma}$$

$$\frac{\Sigma;\ \Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash s_1 \Rightarrow \Gamma';\Sigma' \quad \Sigma';\ \Delta;\ \gamma;\ \rho^a;\ \Gamma' \vdash s_2 \Rightarrow \Gamma'';\Sigma''}{\Sigma;\ \Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash s_1;s_2 \Rightarrow \Gamma'';\Sigma''}$$

$$\frac{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e{:}Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle \quad \rho \notin \Delta \quad \Sigma;\ \Delta \cup \rho;\ \gamma;\ \rho^a;\ \Gamma \vdash s \Rightarrow \Gamma';\Sigma'}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash open\ e\ \{\ s\ \} \Rightarrow \Gamma;\Sigma}$$

$$\frac{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e{:}Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle \quad \rho \notin \Delta \quad \Sigma;\Delta \cup \rho;\ \gamma;\ \rho;\ \Gamma \vdash s \Rightarrow \Gamma';\Sigma'}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash open^a\ e\ \{\ s\ \} \Rightarrow \Gamma;\Sigma}$$

$$\frac{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e{:}Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle \quad \rho \in \Delta \quad \Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e_1{:}\tau}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e.set(e_1) \Rightarrow \Gamma;\Sigma}$$

$$\frac{a \in \{transfer, giveUp\} \quad \rho \notin \Delta \quad \Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e{:}Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle}{\Sigma;\Delta;\gamma;\rho^a;\Gamma \vdash e.a() \Rightarrow \Gamma;\Sigma}$$

$$\frac{\Sigma;\ \Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash e{:}\exists\rho.\tau \quad \rho_0 \notin \Sigma \quad \tau' = [\rho_0/\rho]\tau}{\Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash let\ (\rho_0, \tau' x) = unpack\ e \Rightarrow \Gamma, x{:}\tau';\ \Sigma \cup \rho_0}$$

$d\ OK\ in\ B$

$$\frac{\begin{array}{c}\Sigma{=}\rho^a \cup \overline{\rho} \cup \rho_m^a \cup \overline{\rho_m} \quad \Delta{=}(\overline{\alpha \lhd N};\Sigma) \quad \Delta \vdash \phi\ OK \\ \gamma{=}\phi \cup (\overline{\rho} \succcurlyeq \rho^a) \quad \Sigma;\Delta;\gamma \vdash \overline{\tau^1}\ OK \quad \Sigma;\Delta;\gamma \vdash \tau^2\ OK \\ CT(B){=}class\ B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\ \ldots\ \} \quad \Gamma{=}\cdot, \overline{x{:}\tau^1}, \text{this}{:}\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle \\ override\big(m, N, \langle\rho_m^a\overline{\rho_m} \mid \phi\rangle\overline{\tau^1} \to \tau^2\big) \quad \Sigma;\Delta;\ \gamma;\ \rho_m^a;\ \Gamma \vdash s \Rightarrow \Gamma';\Sigma' \quad \Sigma';\Delta;\gamma;\ \rho_m^a;\ \Gamma' \vdash e{:}\tau^2\end{array}}{\tau^2\ mn\langle\rho_m^a\overline{\rho_m} \mid \phi\rangle\ \big(\overline{\tau^1\ x}\big)\ \{s;\ return\ e;\}\ OK\ in\ B}$$

$B\ OK$

$$\frac{\begin{array}{c}\Delta{=}(\overline{\alpha \lhd N};\rho^a \cup \overline{\rho}) \quad \Sigma{=}\rho^a \cup \overline{\rho} \quad \Delta \vdash \phi\ \text{OK} \quad \gamma{=}\ \overline{\rho} \succcurlyeq \rho^a;\phi \\ \Sigma;\Delta;\gamma \vdash \overline{N}\ OK \quad allocRgn(N){=}\rho^a \quad \Sigma;\Delta;\gamma \vdash N\ OK \quad \Sigma;\Delta;\gamma \vdash \overline{\tau^B} OK \\ \bar{d}\ OK\ in\ B \quad ctype\ (N){=}\ \overline{\tau^A} \quad \Gamma{=}\ \cdot, \text{this}{:}\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle, \overline{x{:}\tau}\end{array}}{\Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash \overline{v_g{:}\tau^A} \quad \Delta;\ \gamma;\ \rho^a;\ \Gamma \vdash \overline{this.f{=}v_f{:}\ unit} \quad k{=}B(\overline{\tau\ x})\{super\big(\overline{v_g}\big);\ \overline{this.f{=}v_f;}\}}{class\ B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\ \overline{\tau^B\ f};k\ ;\bar{d}\ \}\ OK}$$

Examples

Example 1 (Pair) Here is an implementation of generic pair class:

```
class A extends Object {
    A() { super(); }
}
class B extends Object {
    B() { super(); }
}
class Pair<X extends Object,Y extends Object> extends Object {
    X fst;
    Y snd;
    Pair(X fst, Y snd) {
```

```
        super(); this.fst=fst; this.snd=snd;
    }
    X getfst() {
        return this.fst;
    }
    unit setfst(X newfst) {
        this.fst = newfst;
        return ();
    }
}
```

And here is its region-annotated version:

$$
\begin{array}{l}
class\ A\langle R^a\rangle\ extends\ Object\ \{ \\
\quad A()\ \{\ super();\ \} \\
\} \\
class\ B\langle R^a\rangle\ extends\ Object\ \{ \\
\quad B()\ \{\ super();\ \} \\
\} \\
class\ Pair\langle R^a, R0, R1\rangle\big\langle X\ extends\ Object\langle R0\rangle, Y\ extends\ Object\langle R1\rangle\big\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad X\ fst; \\
\quad Y\ snd; \\
\quad Pair(X\ fst, Y\ snd)\ \{ \\
\quad\quad super();\ this.fst = fst;\ this.snd = snd; \\
\quad \} \\
\quad X\ getfst\langle R^a_m\rangle()\ \{ \\
\quad\quad return\ this.fst; \\
\quad \} \\
\quad unit\ setfst\langle R^a_m\rangle(X\ newfst)\ \{ \\
\quad\quad this.fst = newfst; \\
\quad\quad return\ (); \\
\quad \} \\
\}
\end{array}
$$

Example 2 (Linked List) Here is an implementation of linked list class:

```
class LinkedListNode<T extends Object>  extends Object  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = Null;
        this.head = Null;
    }
}
class LinkedList<T extends Object>  extends Object  {
    LinkedListNode<T>  head;
    int count;
    LinkedList(T v) {
        super();
        this.head = new LinkedListNode<T>(v);
        this.count=1;
    }
    unit add(T v) {
```

```
        let LinkedListNode<T>  n = new LinkedListNode<T>(v);
        n.next = this.head;
        this.head.prev = n;
        this.head = n;
        this.count=this.count+1;
        return ();
    }
    T head(unit u) {
        return this.head.val;
    }
    LinkedList<T>   reverse(unit u) {
        let LinkedList<T> xs = new LinkedList<T>  (this.head.val);
        let LinkedListNode<T> cur = this.head.next;
        while(not (cur== Null)) {
              xs.add(cur.val);
              cur = cur.next;
           }
        return xs;
    }
}
```

Its elaborated (region-annotated) version is given below.

$$
\begin{array}{l}
class\ LinkedListNode\langle R^a, R0\rangle\langle T\ extends\ Object\langle R0\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad T\ val; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ prev; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ next; \\
\quad LinkedListNode(T\ val)\ \{ \\
\quad\quad super(); \\
\quad\quad this.val\ =\ val; \\
\quad\quad this.prev\ =\ Null; \\
\quad\quad this.head\ =\ Null; \\
\quad \} \\
\} \\
class\ LinkedList\langle R^a, R1\rangle\langle T\ extends\ Object\langle R1\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ head; \\
\quad int\ count; \\
\quad LinkedList(T\ v)\ \{ \\
\quad\quad super(); \\
\quad\quad this.head\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v); \\
\quad\quad this.count = 1; \\
\quad \} \\
\quad unit\ add\langle R_m^a |\ R_m^a \geqslant R^a\rangle(T\ v)\ \{ \\
\quad\quad let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ n\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v); \\
\quad\quad n.next\ =\ this.head; \\
\quad\quad this.head.prev\ =\ n; \\
\quad\quad this.head\ =\ n; \\
\quad\quad this.count = this.count + 1; \\
\quad\quad return\ (); \\
\quad \} \\
\quad T\ head\langle R_m^a\rangle(unit\ u)\ \{ \\
\quad\quad return\ this.head.val; \\
\quad \} \\
\quad LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ reverse\langle R_m^a |\ R1 \geqslant R_m^a\rangle(unit\ u)\ \{ \\
\quad\quad let\ LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ xs\ =\ new\ LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ (this.head.val); \\
\quad\quad let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ cur\ =\ this.head.next;
\end{array}
$$

```
      while(not (cur == Null)) {
        xs.add(cur.val);
        cur = cur.next;
      }
    return xs;
  }
}
```

Example 3 (An Actor): Here is an implementation of an actor that accepts a region with list of objects, and transfers a new region with a single object, which is the last object of the list in input region:

```
class AnActor extends Object {
    AnActor () {
        super();
    }
    unit onRecv(Region<LinkedList<Object>> rin) {
        let Region<Object> rout = new Region<Object>;
        open rin {
            let LinkedList<Object> xs = rin.get();
            letregion<Rs0> {
                let LinkedList<Object> sx = xs.reverse();
                let Object v = sx.head;
                openAlloc rout {
                    rout.suck(v);
                }
            }
        }
        rin.giveUp();
        rout.transfer();
        return ();
    }
}
```

Its region-annotated version is given below. Reader has to be convinced that elaboration adheres to the rules given previously.

$$
\begin{array}{l}
class\ AnActor\langle R^a\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad AnActor\ ()\ \{ \\
\quad\quad super(); \\
\quad \} \\
\quad unit\ onRecv\langle R^a_m, R^a_0\rangle(\ \exists R.Region[R]\langle R^a_0\rangle \left\langle LinkedList\langle R,R\rangle\langle Object\langle R\rangle\rangle\right\rangle\ rin)\ \{ \\
\quad\quad let\ \left(Rin, Region[Rin]\langle R^a_0\rangle \left\langle LinkedList\langle Rin,Rin\rangle\langle Object\langle Rin\rangle\rangle\right\rangle\ rin\right) = unpack\ rin; \\
\quad\quad let\ \left(Rout, Region[Rout]\langle R^a_m\rangle\langle Object\langle Rout\rangle\rangle\ rout\right) = \\
\quad\quad\quad\quad\quad\quad unpack\ \ newReg\langle R\rangle\langle Object\langle R\rangle\rangle(); \\
\quad\quad open\ rin\ \{ \\
\quad\quad\quad let\ LinkedList\langle Rin,Rin\rangle\langle Object\langle Rin\rangle\rangle\ xs\ =\ rin.get(); \\
\quad\quad\quad letregion\langle Rs0\rangle\ \{ \\
\quad\quad\quad\quad \text{// } Rs0 \text{ is the new inAllocationContext} \\
\quad\quad\quad\quad \text{// below call to reverse type checks because } Rin \geqslant Rs0 \\
\quad\quad\quad\quad let\ LinkedList\langle Rs0,Rin\rangle\langle Object\langle Rin\rangle\rangle\ sx\ =\ xs.reverse\langle Rs0\rangle(); \\
\quad\quad\quad\quad let\ Object\langle Rin\rangle\ v\ =\ sx.head\langle Rs0\rangle(); \\
\quad\quad\quad\quad open^a\ rout\ \{ \\
\quad\quad\quad\quad\quad rout.suck(v); \\
\quad\quad\quad\quad \}
\end{array}
$$

```
      }
    }
    rin.giveUp();
    rout.transfer();
    return ();
  }
}
```

Example 4 (An Actor storing region handlers): Here is an implementation of an actor that accumulates regions, and transfers 10 of them at once.

```
class AnActor extends Object {
    LinkedList<Region<Object>> rs;
    AnActor () {
        super();
    }
    unit onRecv(Region<Object> rin) {
        this.rs.add(rin);
        if (this.rs.count == 10) {
            foreach (r in rs) r.transfer();
            this.rs.removeAll();
        }
        return ();
    }
}
```

Its region-annotated version is given below.

```
class AnActor<Ra0,R0> extends Object<Ra0> {
    LinkedList<R0,R1><∃R.Region[R]<R1><Object<R>>> rs;
    AnActor () {
        super();
    }
    unit onRecv<Ra1>(∃R.Region[R]<R1><Object<R>> rin) {
        let (R2,Region[R2]<R1><Object<R2>> rin) = unpack rin;
        this.rs.add(pack[R2,rin] as ∃R.Region[R]<R1><Object<R>>>);
        if (this.rs.count == 10) {
            foreach (r in rs) {
                Let (R3,Region[R3]<R1><Object<R3>> r') = unpack r;
                r'.transfer();
            }
            this.rs.removeAll();
        }
        return ();
    }
}
```

-

- Immediately needed extensions:
  - Interfaces
  - Way to call super class methods, when they are overridden in subclass
  - Iteration.
  - Polymorphic methods

- A fallout of only allowing $\bar{\rho} \geqslant \rho^{a}$, but not allowing to state outlives constraints within $\bar{\rho}$ is that an object contained in class that expects two or more region params cannot be allocated in a different region. Such an instance variable cannot even be declared in current setting.