Dynamic Regions & Linear Types

Wednesday, June 25, 2014 10:27 AM

As demonstrated in [Linear Types, Refs and Usage Polymorphism] wiki, adopting pure linear type system for a language with mutable state incurs prohibitive cost of expressivity. We need mechanisms, such as ownership types, for controlled aliasing to allow common idioms like:

1. Doubly linked lists, where there is aliasing that never escapes module boundaries.
2. Temporary aliasing, for example when swapping elements while reversing a linked list.

Intuitively, there is a synergy between data abstraction and parametric polymorphism on one hand, and linear typing on the other. Parametric polymorphism tells us that if a generic linked list is specialized as linked list of regions, then it is guaranteed that a region is not freed as a side-effect of calling a linked list method, provided that data abstraction guarantees that no internal state is accessible to the outside world. Unfortunately, we haven't been able to capture this intuition formally. We will get back to this problem at a later stage.

We will currently concentrate on the problem of region type inference. We add dynamic regions, but without linear typing. This means that well-typed programs are not necessarily memory safe, but memory safety is violated only by trying to open a transferred/freed region or by trying to transfer/free an open region. Such violations are allowed by static type system, but are caught at run-time, which should respond by throwing an appropriate exception.

Tofte and Talpin's region inference tries to localize regions as much as possible. Localizing regions is one of the goals for our region inference algorithm, but simplicity of inference algorithm and predictability of region annotations are also important.

The Language:

$\rho \in$ *region names*
$\alpha \in$ *type variables*
$\epsilon \in$ *effect variables*
$a \in$ *addresses*
$v ::= get(\rho) \mid set(\rho)$
$\varphi ::= \{\} \mid \{v\} \cup \varphi \mid \epsilon \cup \varphi$
$\tau ::= \alpha@\rho \mid int@\rho \mid unit \mid \tau\ ref@\rho \mid \tau_1 \rightarrow^{\varphi} \tau_2\ @\rho$
$\sigma ::= \forall\alpha.\sigma \mid \forall\epsilon.\sigma \mid \forall\rho.^{\varphi}\sigma \mid \tau$
$e ::= x \mid e\ e \mid n\ at\ e \mid \lambda(x{:}\tau).e\ \ at\ e \mid letregion\langle\rho\rangle\ x\ in\ e \mid e \otimes e\ at\ e \mid let\ x = e\ in\ e$
$\quad \mid ref\ e\ at\ e \mid !\,e \mid e := e \mid \lambda\rho.e \mid \Lambda\epsilon.e \mid \Lambda\alpha.e \mid e\ \rho \mid e\ \varphi \mid e\ \tau \mid Nil\ T\ at\ e \mid Cons\ T\ e\ e\ \ at\ e$
$\quad \mid new\langle\rho\rangle\langle T\rangle\ at\ e \mid transfer\ e \mid open\ e\ in\ e \mid valueOf\ e \mid Fix\ e$
$\quad match\ e\ with\ Nil \Rightarrow e \mid Cons\ x\ y \Rightarrow e$
$sv ::= n \mid Nil\ T \mid Cons\ T\ v\ v \mid region\langle\rho, o\rangle \mid ref\ v \mid \langle x, e, VE\rangle$ // Storable Values
$v ::= a$ //values

Type Rules

$\Delta;\ \gamma;\ \Gamma \vdash e :^{\varphi} \sigma$

$$\frac{\Gamma(x)=\sigma}{\Delta;\ \gamma;\ \Gamma \vdash x :^{\emptyset} \sigma}\ \text{[T-Var]}$$

$$\frac{\Delta;\ \gamma;\ \Gamma \vdash e_1 :^{\varphi_1} \tau \rightarrow^{\varphi_{12}} \tau_2\ @\ \rho \qquad \Delta;\ \gamma;\ \Gamma \vdash e_2 :^{\varphi_2} \tau \qquad \varphi = \varphi_1 \cup \varphi_{12} \cup \varphi_2 \cup \{get(\rho)\}}{\Delta;\ \gamma;\ \Gamma \vdash e_1\ e_2 :^{\varphi} \tau_2}\ \text{[T-App]}$$

$$\frac{\Delta \vdash \tau_1 \quad \rho \in \Delta \quad \Delta;\gamma;\Gamma,x{:}\tau_1 \vdash e : \varphi_{12}\tau_2 \quad \varphi = \{set(\rho)\}}{\Delta;\gamma;\Gamma \vdash \lambda(x{:}\tau_1).e \text{ at } \rho : \varphi\ \tau_1 \to^{\varphi_{12}} \tau_2 @ \rho} \text{ [T-Abs]}$$

$$\frac{\rho \in \Delta \quad \varphi = \{set(\rho)\}}{\Delta;\gamma;\Gamma \vdash n \text{ at } \rho : int@\rho} \text{ [T-Int]}$$

$$\frac{\Delta;\gamma;\Gamma \vdash e_1 : \varphi_1\, int@\rho_1 \ \Delta; \quad \rho \in \Delta \quad \Delta;\gamma;\Gamma \vdash e_2 : \varphi_2\, int@\rho_2 \quad \varphi = \varphi_1 \cup \varphi_2 \cup \{set(\rho)\}}{\Delta;\gamma;\Gamma \vdash e \otimes e \text{ at } \rho : int@\rho} \text{ [T-IntOp]}$$

$$\frac{\rho \notin \Delta \quad \Delta \cup \{\rho\};\gamma,\rho;\Gamma \vdash e : \varphi\sigma \quad \Delta \vdash \sigma}{\Delta;\gamma;\Gamma \vdash \text{letregion } \rho \text{ in } e : \varphi\sigma} \text{ [T-Reg]}$$

$$\frac{\Delta;\gamma;\Gamma \vdash e : \varphi_1 \tau \quad \rho \in \Delta \quad \varphi = \{set(\rho)\} \cup \varphi_1}{\Delta;\gamma;\Gamma \vdash (ref\ e)\ at\ \rho : \varphi\ \tau\ ref@\rho} \text{ [T-Ref]}$$

$$\frac{\Delta;\gamma;\Gamma \vdash e : \varphi_1\ \tau\ ref\ @\ \rho \quad \varphi = \{get(\rho)\} \cup \varphi_1}{\Delta;\gamma;\Gamma \vdash !e : \varphi\tau} \text{ [T-DeRef]}$$

$$\frac{\varphi = \varphi_1 \cup \varphi_2 \cup \{set(\rho)\} \quad \Delta;\gamma;\Gamma \vdash e_1 : \varphi_1\ \tau\ ref\ @\ \rho \quad \Delta;\gamma;\Gamma \vdash e_2 : \varphi_2 \tau}{\Delta;\gamma;\Gamma \vdash e_1 := e_2 : \varphi\ unit} \text{ [T-Assn]}$$

$$\frac{\varphi = \varphi_1 \cup \varphi_2 \quad \Delta;\gamma;\Gamma \vdash e_1 : \varphi_1 \sigma_1 \quad \Delta;\gamma;\Gamma,x{:}\sigma_1 \vdash e : \varphi_2 \sigma_2}{\Delta;\gamma;\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \varphi\ \sigma_2} \text{ [T-Let]}$$

$$\frac{\rho \notin \Delta \quad \Delta \cup \{\rho\};\gamma;\Gamma \vdash e : \varphi\ \sigma}{\Delta;\gamma;\Gamma \vdash \lambda\rho.e : \emptyset\ \forall\rho.\varphi\sigma} \text{ [T-RegGen]}$$

$$\frac{\varphi = \varphi_1 - \{\epsilon\} \quad \epsilon \notin \text{fev}(\Gamma) \quad \Delta;\gamma;\Gamma,\epsilon \vdash e : \varphi_1\ \sigma}{\Delta;\gamma;\Gamma \vdash \Lambda\epsilon.e : \varphi\ \forall\epsilon.\sigma} \text{ [T-EffGen]}$$

$$\frac{\alpha \notin \text{ftv}(\Gamma) \quad \Delta;\gamma;\Gamma,\alpha \vdash e : \varphi\ \sigma}{\Delta;\gamma;\Gamma \vdash \Lambda\alpha.e : \varphi\ \forall\alpha.\sigma} \text{ [T-TypGen]}$$

$$\frac{\Gamma \vdash e : \varphi \tau \to^{\emptyset} \tau}{\Gamma \vdash Fix\ e : \varphi\ \tau} \text{ [T-Fix]}$$

- γ denotes an ordered set that maintains $outlives$ relation among concrete regions.
- Δ denotes a (unordered) set of region names in scope.
- Region variables introduced by $\lambda\rho.e$ are not in $outlives$ relation with existing regions; neither are they in $outlives$ relation with other region variables. Therefore, they are not added to γ.
- $fev(\Gamma)$ and $ftv(\Gamma)$ denote free region variables and free type variables in Γ, respectively.
- Regions have operational significance. Hence $\lambda\rho.e$, and not $\Lambda\rho.e$.
- Fix point means that polymorphic recursion is not allowed

Examples

Example 1 :

$$\Lambda\rho_{ctx}.\Lambda\rho_1.\Lambda\rho_2.\lambda(rin : Rgn^1\langle\rho_{in}\rangle\langle record\ list\rangle, udf : record@\rho_1 \to outrec@\rho_2).$$
$$let\ rtrans = new\langle\rho_{tr}\rangle\langle outrec\ list\rangle\ at\ e$$

```
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {

   var rtrans = new Region<Batch<OutRecord>>();
   using (RegionContext.Open(rin)) {
         var batch = rin.GetData();
         using (RegionContext.OpenAlloc(rtrans)) {
           rtrans.SetData(new Batch<OutRecord>());
         }
         using (RegionContext.OpenAlloc()) {
           Iterator ix = new Iterator(batch);
            foreach Record x in ix {
                  var y = udf(x);
                  using (RegionContext.OpenAlloc(rtrans)) {
                    var z = rtrans.SuckObject(y);
```

```
                    rtrans.GetData().Append(z);
                }
            }
        }
    }
    rin.GiveUp();
    rtrans.Transfer(this.getDownStreamActor());
}
```

Linear Types, Refs & Usage Polymorphism

Wednesday, June 25, 2014 11:34 AM

We drop regions to concentrate on linear types in presence of references. The aim is to construct a type system that can quantify over linearity, such that functions like $append$ can be assigned general types. Furthermore, we want to explore how linearity interacts with mutable references and polymorphism.

We start with a monomorphic language with references and lists. The language has both linear and non-linear types, but has no parameterization over usage qualifiers; i.e., we cannot define a function that can be used over both linear data and non-linear data. Usual linearity guarantees are expected:

1. A linearly typed value has to be used exactly once. It can neither be discarded, nor be duplicated
2. A non-linear data structure (lists, closures or references) should not contain linear values. Closures should not trap linear values.

It can be noted that this language is essentially Wadler90's language extended with references and lists (let! construct will be added subsequently). However, Wadler90 gives type rules for linear and non-linear types. Our immediate goal is to parameterize type rules (not language) over usage qualifiers.

The Monomorphic Language

$u \ ::= 0 \mid 1$ // Usage qualifiers. 0 denotes non-linear and 1 denotes linear.
$\tau \ ::= \ (T, u)$ // Usage decorated type
$T \ ::= unit \mid int \mid \tau \ list \mid \tau \ ref \mid \tau \to \tau$ // Pre-types
$e \ ::= \ x \mid n \mid ¡n \mid \lambda(x{:}\tau).e \mid e \ e \mid let \ x = e \ in \ e \mid Ref \ e \mid e := e \mid !\, e \mid Fix \ e$
$\quad \mid Cons \ e \ e \mid Nil \ \tau \mid \ match \ e \ with \ Nil \Rightarrow e \mid Cons \ x \ y \Rightarrow e$

Syntactic Conventions

$T^u \stackrel{\text{def}}{=} (T, u)$
$(\tau_1 \to \tau_2)^u \stackrel{\text{def}}{=} \tau_1 \to^u \tau_2 \stackrel{\text{def}}{=} (\tau_1 \to \tau_2, u)$

Type Rules for Monomorphic Language

$\Gamma \vdash e{:}\tau$

$$\frac{}{\Gamma^0, x{:}\tau \vdash x{:}\tau} \qquad \frac{}{\cdot \vdash n{:}int^0} \qquad \frac{}{\cdot \vdash ¡n{:}int^1} \qquad \frac{\Gamma^u, x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma^u \vdash \tau_1 \to^u \tau_2} \qquad \frac{\Gamma_1 \vdash e_1{:}\tau_1 \to^u \tau_2 \quad \Gamma_2 \vdash e_2{:}\tau_1}{\Gamma_1 \oplus \Gamma_2 \vdash e_1 \ e_2{:}\ \tau_2}$$

$$\frac{\Gamma \vdash e{:}\tau_1 \to^0 \tau_2}{\Gamma \vdash Fix \ e : \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1{:}\tau_1 \quad \Gamma_2, x{:}\tau_1 \vdash e_2{:}\tau_2}{\Gamma_1 \oplus \Gamma_2 \vdash let \ x{=}e_1 \ in \ e_2{:}\ \tau_2} \qquad \frac{\Gamma \vdash e{:}T^u}{\Gamma \vdash Ref \ e : T^u \ ref^u} \qquad \frac{}{\Gamma \vdash Nil \ T^u : T^u \ list^u}$$

$$\frac{\Gamma_1 \vdash e_1{:}T^u \quad \Gamma_2 \vdash e_2{:}\ T^u \ list^u}{\Gamma_1 \oplus \Gamma_2 \vdash Cons \ e_1 \ e_2 : T^u \ list^u}$$

$$\frac{\Gamma_1 \vdash e_1{:}\ T^u \ ref^u \quad \Gamma_2 \vdash e_2{:}\ T^u}{\Gamma_1 \oplus \Gamma_2 \vdash e_1 := e_2 :\ unit^0} \qquad \frac{\Gamma \vdash e{:}\ T^u \ ref^u}{\Gamma \vdash !e : T^u}$$

$$\frac{\Gamma_1 \vdash e_1{:}\ T^u \ list^u \quad \Gamma_2 \vdash e_2{:}\tau \quad \Gamma_2, x{:}T^u, y{:}T^u \ list^u \vdash e_3{:}\tau}{\Gamma_1 \oplus \Gamma_2 \vdash match \ e_1 \ with \ Nil \Rightarrow e_2 \mid Cons \ x \ y \Rightarrow e_3{:}\tau}$$

Notes:

1. There is no subtype rule that allows coercions from non-linear types to linear types. Consequently, linearity is equivalent to *alias-freedom* in our system.
2. $\Gamma = \Gamma^0$, iff every binding in Γ is of form $x{:}\, T^0$. In this case, Γ is said to be non-linear environment. Otherwise, $\Gamma = \Gamma^1$, and Γ is linear environment.
3. Exchange, contraction and weakening are allowed on Γ^0. On the other hand, only exchange is allowed on Γ^1.
4. For a function to have non-linear type, it must be typable under non-linear environment. In other words, its closure must not trap any linear values. However, its argument can be linear.
5. Since $Fix\ f\ =\ f\ (f\ (f\ \ldots\ ))$, we require that f has non-linear type.
6. References to linear values are linear themselves. Likewise, a list of linear values is itself linear. Our language does not allow constructing linear list of non-linear values, or a linear reference to a non-linear value, although such notion is theoretically sensible. Further, it is not possible to ascribe a linear type to a non-linear function, although doing so is perfectly safe. This is a design choice for the sake of simplicity. Our philosophy is to avoid linearity unless it is absolutely necessary.
7. One could write a lambda that expects a value of type $int\ list^1$, or even $int^1\ list$. We don't prevent such cases in the calculus, as they do not effect the soundness of the system; it is anyway impossible to construct values of such type. Once we get to inference, our inference engine might fail to infer types for lambdas which expect such types. If we are interested in completeness of inference, we can add a well-formedness condition to the premise of type checking rule of lambdas so that functions which expect such types cannot be type checked in the first place.

Examples

In examples, we drop the superscript 0 over non-linear arrows. So, an arrow without any superscript should be treated as a non-linear arrow.

Example 1

$$
\begin{aligned}
&\lambda(x{:}\, int^1).\,(y{:}\, int^1)\\
&\quad let\ k = 2\ in\\
&\quad let\ n0 = ¡\,0\ in\\
&\quad let\ n1 = ¡\,1\ in\\
&\quad let\ l1 =\ Cons\ n0\ (Nil\ int^1)\ in\\
&\quad let\ f = \lambda(n3 :\ int^1).\,Cons\ n3\ (Cons\ n1\ l1)\ in\\
&\quad let\ g = \lambda(n4 : int^1).\,Cons\ n4\ (Nil\ int^1) in\\
&\quad\ \ append\ (f\ x)\ (g\ y)
\end{aligned}
$$

- The top-level lambda is type-checked under empty environment, which can be considered non-linear.
- The top-level let expression is type checked under linear $\Gamma = \cdot, x{:}\, int^1, y{:}\, int^1$.
- While type-checking let expression introducing $l1$, $\Gamma = \cdot, k{:}\, int, x{:}\, int^1, y{:}\, int^1, n0 :\ int^1, n1 :\ int^1$, which is split as $\Gamma_1 = \cdot, k{:}\, int, n0 :\ int^1$ and $\Gamma_2 = \cdot, k{:}\, int, x{:}\, int^1, y{:}\, int^1, n1 :\ int^1$. Observe that non-linear resource k is duplicated. k is eventually discarded, which is ok as it is non-linear.
- $f :\ int^1\ \rightarrow^1\ int^1\ list^1$, as f can only be typed under a linear environment. On the other hand, $g :\ int^1 \rightarrow\ int^1\ list^1$, as g can be typed under empty environment, which can be considered non-linear.

Example 2 (Linear Append):

$$
\begin{aligned}
&Fix\ (\lambda(f{:}\, int^1\ list^1 \rightarrow\ int^1\ list^1\ \rightarrow^1 int^1\ list^1).\\
&\quad \lambda(xs : int^1\ list^1).\lambda(ys : int^1\ list^1).
\end{aligned}
$$

$$match\ xs\ with\ Nil \Rightarrow ys \mid Cons\ \ x\ xs' \Rightarrow Cons\ x\ (f\ xs'\ ys)$$

- The type of the function is $int^1\ list^1 \rightarrow\ int^1\ list^1\ \rightarrow^1 int^1\ list^1$.
- Second arrow cannot be non-linear as partially applied $append$ contains a linear value in its closure. Our type rules enforce this

Linearity Relaxation (Pointer Borrowing)

We now add Wadler90's $let!$ construct that allows limited relaxation of aliasing constraints over linear resources to allow read-only accesses:

$e\ ::= \cdots \mid let!\ (x)\ y = e\ in\ e$

Variable x, which denotes a linear resource, is treated non-linearly in the expression bound to y. This construct is useful in following ways:

1. It forms theoretical base for $open$ expression to open transferable regions, as region handler is a linear resource.
2. When we have linear data structures, such as list of transferable regions, it allows us to use functions such as $length$, which treat the list in read-only way; so, the list need not be consumed.
    a. A note on inference : $length$ will have usage-parametric type. It is the responsibility of the inference mechanism to to introduce $let!$ around calls to length, and instantiate $length$ with non-linear type. We cannot expect programmers to use explicitly use $let!$.

Given a base type τ, its non-linear projection (¡ τ) is defined as following:

$$\begin{aligned}
¡\ T^0 &= T^0 \\
¡\ int^1 &= int^0 \\
¡\ unit^1 &= unit^0 \\
¡\ (\tau\ list^1) &= (¡\ \tau)\ list^{\ 0} \\
¡\ (\tau\ ref^1) &= (¡\ \tau)\ ref^{\ 0}
\end{aligned}$$

We give typing rule for $let!$ as following:

$$\frac{\Gamma_1 \vdash x{:}\tau \quad ¡\tau = \tau' \quad \Gamma_2, x{:}\tau' \vdash e_1{:}\tau_1 \qquad \Gamma_3, x{:}\tau, y{:}\tau_1 \vdash e_2{:}\tau_2}{\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3 \vdash let!\ (x)\ y{=}e_1\ in\ e_2{:}\ \tau_2}\ (\tau_1\ not\ arrow\ \&\ \tau_1\ safe\ for\ \tau)$$

Observe that

- e_1 cannot be a lambda. If we allow e_1 to be a lambda, then our lambda typing rule allows e_1 to trap references to (non-linear) x in its closure, which escape the scope of x through y. Moreover, we have no practical motivation to enable e_1 be a lambda. HMN01 (see related work) allow e_1 to return a lambda as the only linear resource in their language is a region handler allowing them to prevent closure trapping by simply disallowing non-linear projection of region handlers from being present in Γ when typing a closure.
- τ_1 should be safe for τ. This means that no linear component of τ should be a non-linear component of τ_1(i.e., no common components between τ and τ_1). This is again to ensure that no component of x escapes the scope of x.
- However, due to the presence of references, components can still escape via side-effects (via assignments to instance variables in C#). To prevent this, we need some sort of dataflow tracking to enable us to detect when a value originated from (non-linear) x is escaping its lifetime (its scope) via assignments to longer-living references. Fortunately, we already have a framework in place to perform this task - the region type system! When we combine the linear type system with

the region type system, **we reuse the region type checking to enfore the safety property required for $let!$ construct**. This observation is elaborated in subsequent paragraphs.

- When a linear resource is *opened* via $let!$, we conceptually create a new static region whose lifetime is same as that of the the scope of non-linear variable introduced via $let!$. Note that the new static region does not outlive any other existing regions. Now, if we assert that the non-linear value, and all its components, are stored in the new static region, the region type system will ensure that no reference to the non-linear value, or its components, escape the lifetime of newly created static region. This is enough to ensure the safety invariant of $let!$ construct.
- For eg, in the combination type system, let $x = (int^1@\rho_H)\ list^1@\rho_H$. Its non-linear projection is $(int@\rho_H)\ list@\rho_H$. By introducing a new static region ρ_s,we now localize it: $(int@\rho_s)\ list@\rho_s$. Now, following examples are ill typed :
  Eg 1:
  $let!\ (x)\ y = match\ x\ with\ Nil \Rightarrow 2@\rho_H \mid Cons\ h\ t \Rightarrow h\ in$
  …
  Eg 2:
  $let\ l = ref\ (Nil\ int\ @\rho_H)\ @\rho_H\ in$
  $let!\ (x)\ y = match\ x\ with\ Nil \Rightarrow\ () \mid Cons\ h\ t \Rightarrow l := t\ in$
  …

- Considering the synergy between region type system and $let!$ construct, we defer its full formal treatment to the time we merge the linear type system with region type system.

Usage Polymorphism

It can be noted that there is no reason for $append$ to operate only over linear lists. It can also be safely applied over non-linear lists, in which case the partially applied $append$ need not be a linear function. Our next aim is to generalize our type system such that it is possible to ascribe a type to $append$ that is parameterized over usage mode (linear/non-linear) of its arguments.

The Usage Polymorphic Language

$\upsilon \in usage\ variables$ //ranges over u
u ::= $0 \mid 1 \mid \upsilon \mid u \vee u$ // Usage qualifiers. 0 denotes non-linear and 1 denotes linear.
τ ::= (T, u) // Usage decorated type
T ::= $unit \mid int \mid \tau\ list \mid \tau\ ref \mid \tau \to \tau$ // Pre-types
σ ::= $\forall \upsilon.\sigma \mid \tau$
e ::= $x \mid n \mid ¡n \mid \lambda(x{:}\tau).e \mid e\ e \mid let\ x = e\ in\ e \mid Ref\ e \mid e := e \mid !e \mid Fix\ e$
$\mid Cons\ e\ e \mid Nil\ \tau \mid\ match\ e\ with\ Nil \Rightarrow e \mid Cons\ x\ y \Rightarrow e$
$\mid \Lambda\upsilon.e \mid e\ u$

Type Rules for Usage Polymorphic Language

$\Gamma \vdash e{:}\sigma$

$$\frac{}{\Gamma^0, x{:}\tau \vdash x{:}\tau} \qquad \frac{}{\cdot \vdash n{:}int^0} \qquad \frac{}{\cdot \vdash ¡n{:}int^1} \qquad \frac{\Gamma^u \vdash \tau_1 \quad \Gamma^u, x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma^u \vdash \tau_1 \to^u \tau_2} \qquad \frac{\Gamma_1 \vdash e_1{:}\tau_1 \to^u \tau_2 \quad \Gamma_2 \vdash e_2{:}\tau_1}{\Gamma_1 \oplus \Gamma_2 \vdash e_1\ e_2{:}\ \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1{:}\tau_1 \quad \Gamma_2, x{:}\tau_1 \vdash e_2{:}\tau_2}{\Gamma_1 \oplus \Gamma_2 \vdash let\ x{=}e_1\ in\ e_2{:}\ \tau_2} \qquad \frac{\Gamma \vdash e{:}T^u}{\Gamma \vdash Ref\ e : T^u\ ref^u} \qquad \frac{}{\Gamma \vdash Nil\ T^u : T^u\ list^u}$$

$$\frac{\Gamma_1 \vdash e_1{:}T^u \quad \Gamma_2 \vdash e_2{:}\ T^u\ list^u}{\Gamma_1 \oplus \Gamma_2 \vdash Cons\ e_1\ e_2 : T^u\ list^u}$$

$$\frac{\Gamma_1 \vdash e_1 : T^u\ ref^u \quad \Gamma_2 \vdash e_2 : T^u}{\Gamma_1 \oplus \Gamma_2 \vdash e_1 := e_2 : unit^0} \qquad \frac{\Gamma \vdash e : T^u\ ref^u}{\Gamma \vdash !e : T^u}$$

$$\frac{\Gamma_1 \vdash e_1 : T^u\ list^u \quad \Gamma_2 \vdash e_2 : \tau \quad \Gamma_2, x : T^u, y : T^u\ list^u \vdash e_3 : \tau}{\Gamma_1 \oplus \Gamma_2 \vdash match\ e_1\ with\ Nil \Rightarrow e_2 \mid Cons\ x\ y \Rightarrow e_3 : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \rightarrow^{\infty} \tau_2}{\Gamma \vdash Fix\ e : \tau_2} \qquad \frac{\upsilon \notin fuv(\Gamma) \quad \Gamma, \upsilon \vdash e : \sigma}{\Gamma \vdash \Lambda \upsilon . e : \forall \upsilon . \sigma} \qquad \frac{\Gamma \vdash e : \forall \upsilon . \sigma \quad \Gamma \vdash u}{\Gamma \vdash e\ u : [u/\upsilon].\sigma}$$

Notes:

1. $\Gamma = \Gamma^u$, where u is the disjunction of usage tags of all type bindings in Γ. For eg, if $\Gamma = \cdot, x : int^0, \upsilon_0, int^{\upsilon_0}\ list^{\upsilon_0}$, then $\Gamma = \Gamma^{0 \vee \upsilon_0} = \Gamma^{\upsilon_0}$. This definition generalizes the definitions of Γ^0 and Γ^1 given previously.
2. $fuv(A)$ denotes free usage variables in the semantic object A.
3. Well-formedness: semantic object A is well-formed under Γ (i.e., $\Gamma \vdash A$) iff $fuv(A) \subseteq fuv(\Gamma)$. The definition of well-formedness can be extended to specify that linear types contain only linear components, as described previously (Point (6) of notes below previous rules).
4. When the environment Γ is split into Γ_1 and Γ_2 (i.e., $\Gamma = \Gamma_1 \oplus \Gamma_2$), usage variables ($\upsilon$) are copied.

Examples

Example 1 (Usage Polymorphic Addition)

$$\Lambda \upsilon_0 . \Lambda \upsilon_1 . \lambda (x : int^{\upsilon_0}) . \lambda (y : int^{\upsilon_1}) . x + y$$

- The type is $\forall \upsilon_0 . \forall \upsilon_1 . int^{\upsilon_0} \rightarrow int^{\upsilon_1} \rightarrow^{\upsilon_0} int^0$.
- When typing the outermost lambda, there are no type bindings in Γ. Hence, first arrow gets 0 superscript (i.e., no superscript, as per our convention).
- When typing the inner lambda, Γ contains $x : int^{\upsilon_0}$ as the only type binding. Hence, second arrow gets υ_0 as superscript.

Example 2 (Usage Polymorphic Append)

$$\Lambda \upsilon . Fix\ (\lambda (f : int^{\upsilon}\ list^{\upsilon} \rightarrow int^{\upsilon}\ list^{\upsilon} \rightarrow^{\upsilon} int^{\upsilon}\ list^{\upsilon}) .$$
$$\lambda (xs : int^{\upsilon}\ list^{\upsilon}) . \lambda (ys : int^{\upsilon}\ list^{\upsilon}) .$$
$$match\ xs\ with\ Nil \Rightarrow ys \mid Cons\ x\ xs' \Rightarrow Cons\ x\ (f\ xs'\ ys)$$

- Arrow is right associative. So, $f : int^{\upsilon}\ list^{\upsilon} \rightarrow (int^{\upsilon}\ list^{\upsilon} \rightarrow^{\upsilon} int^{\upsilon}\ list^{\upsilon})$ is a non-linear function.
- Similar to the case of addition example, only second arrow needs to be linear.

Example 3 (ill-Typed Usage Polymorphic Append)

$$\Lambda \upsilon_0 . \Lambda \upsilon_1 . Fix\ (\lambda (f : int^{\upsilon_0}\ list^{\upsilon_0} \rightarrow int^{\upsilon_1}\ list^{\upsilon_1} \rightarrow^{\upsilon_0} int^{\upsilon_0 \vee \upsilon_1}\ list^{\upsilon_0 \vee \upsilon_1}) .$$
$$\lambda (xs : int^{\upsilon_0}\ list^{\upsilon_0}) . \lambda (ys : int^{\upsilon_1}\ list^{\upsilon_1}) .$$
$$match\ xs\ with\ Nil \Rightarrow ys \mid Cons\ x\ xs' \Rightarrow Cons\ x\ (f\ xs'\ ys)$$

- A linear list can only be appended to a linear list. Therefore, usage tags for both arguments of append need to be the same.
- ys has type $int^{\upsilon_1}\ list^{\upsilon_1}$, which is not (subtype of)/(equivalent to) $int^{\upsilon_0 \vee \upsilon_1}\ list^{\upsilon_0 \vee \upsilon_1}$. Further, $Cons$ application doesn't type check.
- Our type rules allow disjunctive superscripts to appear only over arrows. Disjunction originates within superscript of the environment (Γ) and gets propagated to superscript of the arrow via the the typing rule for lambdas.

Example 4 (Usage Polymorphic Map)

$$\Lambda \upsilon_0 . \Lambda \upsilon_1 . Fix\ (\lambda (map : int^{\upsilon_0}\ list^{\upsilon_0} \rightarrow (int^{\upsilon_0} \rightarrow int^{\upsilon_1}) \rightarrow^{\upsilon_0} int^{\upsilon_1}\ list^{\upsilon_1}) .$$
$$\lambda (xs : int^{\upsilon_0}\ list^{\upsilon_0}) . \lambda (f : int^{\upsilon_0} \rightarrow int^{\upsilon_1}) .$$
$$match\ xs\ with\ Nil \Rightarrow Nil\ int^{\upsilon_1} \mid Cons\ x\ xs' \Rightarrow Cons\ (f\ x)\ (map\ xs'\ f)$$

Example 5 (Usage Polymorphic Map2)

$$\Lambda\upsilon_0.\Lambda\upsilon_1.\Lambda\upsilon_2.Fix\ (\lambda(map2{:}\ int^{\upsilon_0}\ list^{\upsilon_0} \to int^{\upsilon_1}\ list^{\upsilon_1}\ \to^{\upsilon_0 \vee \upsilon_1}\ (int^{\upsilon_0} \to int^{\upsilon_1} \to^{\upsilon_0 \vee \upsilon_1}\ int^{\upsilon_2})\ \to^{\upsilon_0 \vee \upsilon_1}\ int^{\upsilon_2}\ list^{\upsilon_2}).$$
$$\lambda(xs : int^{\upsilon_0}\ list^{\upsilon_0}).\lambda(ys : int^{\upsilon_1}\ list^{\upsilon_1}).\lambda(f : int^{\upsilon_0} \to int^{\upsilon_1} \to^{\upsilon_0 \vee \upsilon_1} int^{\upsilon_2}).$$
$$match\ (xs, ys) with\ (Nil, Nil) \Rightarrow Nil\ int^{\upsilon_1}$$
$$|\ (Cons\ \ x\ xs', Cons\ y\ ys') \Rightarrow Cons\ (f\ x\ y)\ (map2\ xs'\ ys' f)$$

Example 6 (Usage Polymorphic Linked List Add)

$$\Lambda\upsilon.\lambda(l : int^{\upsilon}\ list^{\upsilon}\ ref^{\upsilon}).\lambda(x : int^{\upsilon}).$$
$$l := Cons\ \ x\ !\ l$$

- The type is $\forall\upsilon.\ int^{\upsilon}\ list^{\upsilon}\ ref^{\upsilon} \to\ int^{\upsilon}\ \to^{\upsilon}\ unit^{0}$.

Type Polymorphism

We now trivially extend the usage-polymorphic language by letting type variables (α) range over pre-types (undecorated types). The language is shown below:

Fully Polymorphic Language

$\upsilon \in usage\ variables$ //ranges over u
$\alpha \in type\ variables$ //ranges over T
u ::= 0 | 1 | υ | $u \vee u$ // Usage qualifiers. 0 denotes non-linear and 1 denotes linear.
τ ::= (T, u) // Usage decorated type
T ::= $unit$ | int | α | $\tau\ list$ | $\tau\ ref$ | $\tau \to \tau$ // Pre-types
σ ::= $\forall\upsilon.\sigma$ | $\forall\alpha.\sigma$ | τ
e ::= x | n | ¡n | $\lambda(x{:}\ \tau).e$ | $e\ e$ | $let\ x = e\ in\ e$ | $Ref\ e$ | $e := e$ | $!\ e$ | $Fix\ e$
| $Cons\ e\ e$ | $Nil\ \tau$ | $match\ e\ with\ Nil \Rightarrow e\ |\ Cons\ x\ y \Rightarrow e$
| $\Lambda\upsilon.e$ | $e\ u$ | $\Lambda\alpha.e$ | $e\ T$

Type Variable Generalization and Instantiation

$$\frac{\alpha \notin ftv(\Gamma) \quad \Gamma, \alpha \vdash e{:}\sigma}{\Gamma \vdash \Lambda\alpha.e\ {:}\forall\alpha.\sigma} \qquad \frac{\Gamma \vdash e{:}\forall\alpha.\sigma \quad \Gamma \vdash T}{\Gamma \vdash e\ T\ {:}[T/\alpha].\sigma}$$

Notes:

1. $ftv(A)$ denotes free type variables in the semantic object A.
2. Well-formedness: We extend the definition of well-formedness from the usage polymorphic language. A semantic object A is well-formed under Γ (i.e., $\Gamma \vdash A$) iff $fuv(A) \subseteq fuv(\Gamma) \wedge ftv(A) \subseteq ftv(\Gamma)$.
3. When the environment Γ is split into Γ_1 and Γ_2 (i.e., $\Gamma = \Gamma_1 \oplus \Gamma_2$), type variables ($\alpha$), along with usage variables (υ) are copied.

Examples:

Examples given in the usage polymorphic language can be trivially generalized for type polymorphism. We do not reproduce them here.

Some Useful Observations

- Any linear resource that is non-linearized via $let!$ cannot be destructively updated. Linear references are no exception. They should be treated as constant references within the $let!$ block.
- Let us say we introduce a type for constant reference : $\alpha\ cref$. The only operation that we can allow on values of type $\alpha\ cref$ is defereferencing:

$$\frac{\Gamma \vdash e : T^u \; cref^u}{\Gamma \vdash !e : T^u}$$

We cannot allow the downcast $\alpha \; cref <: \alpha \; ref$, for it makes updating constant references sound. However, we can allow $\alpha \; ref <: \alpha \; cref$. When $a : \alpha \; ref$ is is cast to $a : \alpha \; cref$, it essentially means that thenceforth a should be treated as a constant reference. This is similar to the cast that allows non-linear types to be treated as linear types (Type system allowing such cast does not guarantee *alias-freedom*, but nonetheless are linear in the sense that *a linear resource is consumed only once*. See Zdancewic et al's Linear System F, TLDI).

- To summarize the above discussion, a linear reference should be treated as a non-linear constant reference within a $let!$ block.
- How is the non-linear constant reference useful? Consider a linear linked list of transferable region handlers. Within a let! block, following idioms are supported:
  - Invoking $length$ function to get the length of the list, without losing the list
  - Traversing the list to (read from)/(make allocations to) transferable regions, without consuming the list.
- How is the list prevented from not being consumed? Linked list methods, such as $length$, $getFirst$ etc., will have usage polymorphic type. Instantiating the usage variable with 0 will let us pass a non-linear list reference, which is what we have inside the $let!$ block.
- But, even the types of add and pop will be usage polymorphic! Instantiating the usage variable to 0 here will let us update the non-linear list reference, which is supposed to be constant! Violation leads to losing alias-freedom as demonstrated by the following example:

  $let \; r1 = <..new \; transferable \; region \; ..>$
  $let \; l = <..reference \; to \; empty \; list \; of \; regions \; ..>$
  $let! \; (r1) \; \_ =$
  $\quad let! \; (l) \; \_ = add \; l \; r1 \; (* \; using \; add \; over \; non\text{-}linear \; list \; *) \; in$
  $\quad\quad () \; in$
  $\quad () \; in$
  ...

  Observe that the region root pointer has two alaises - one via r1, and other via the list.
- A straightforward way to avoid scenarios such as above is to require the type of linked list methods to clearly indicate whether they need an updateable reference to the linked list, or does constant reference suffice. For eg, the type of $length$ and $getFirst$ should be $\alpha \; list \; cref \rightarrow \cdots$, whereas the type of add and pop should be $\alpha \; list \; ref \rightarrow \cdots$.
- However, this approach is unsettling for we now have two notions of usage - linearity vs non-linearity, and updateable vs constant. Can these two notions be merged? One observation is that it makes no sense to call a method like $length$, which treats its argument as a constant, with a non-linear list ref. Although at the type system level, it simply means consuming the linear list ref, in practice we are allowing the program to simply give up a list of regions without freeing them. Conversely, as observed previously, it is unsound to call a method like add, which destructively updates its argument list ref, with a *non-linearized* list ref (within a $let!$ block). So, instead of giving them usage polymorphic types, why not require that $length$ be used only with non-linear lists, and add only with linear lists?
- Requiring that only a non-linear list ref be used is not restrictive; we can always temporarily convert linear list ref to non-linear list ref using $let!$. However, requiring only a linear list ref be used is very restrictive; we have effectively blocked linked list methods from being used on any linked list that is not a list of region handlers. What we actually need to do is to prevent linked list updating methods from being used on a *linked list containing non-linear references to region handlers*.
- Evidently, a general $let!$ construct is making things very complicated. One alternative is to discard the $let!$ expression, which relaxes linearity for all linear resources, in favour of $open$ expression, which only relaxes linearity for region handlers. This way, we don't have to deal with non-linear list of regions as there will never be one. However, adopting this solution incurs considerable loss of expressivity. For eg, the (presumably) common programming idiom of using an iterator to traverse the list of regions, so as to allocate some data in each region, is not possible with linear

list of region handlers, as it creates aliases to region handlers. One has to resort to an unintuitive solution of removing regions from the list one-by-one, performing allocations/deallocations, and then reconstructing the list. Furthermore, since we can never obtain a non-linear list of regions, type system treats every linked list method, even innocuous ones like $length$, as consuming the list (or its ref) passed. We therefore need to convey, via the type of $length$, that the linked list passed to $length$ is not consumed. However, while we are comparing $let!$ to $open$, it is also worth pointing out that even if we use $let!$, we still need to capture such information in types. For example, the type of linked list add method should convey that while it takes two linear arguments - a list and a region handler, only the region handler is consumed. The linear linked list is left unconsumed.

- Considering these observations, perhaps it is wise to leave out $let!$ in favour of $open$, and instead work on how to capture consumption information in types.

Notes:

1. Latent effects specifying which references are updated
2. Ownership types for imperative data structures such as doubly linked lists (dll):
    a. A dll of linear objects is linear
    b. Any number of references "owned" by the dll to the linear object are possible
    c. No reference owned by the dll escapes its module boundary

Example

Monday, June 30, 2014 3:50 PM

```
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {

   var rtrans = new Region<Batch<OutRecord>>();
   using (RegionContext.Open(rin)) {
          var batch = rin.GetData();
          using (RegionContext.OpenAlloc(rtrans)) {
            rtrans.SetData(new Batch<OutRecord>());
          }
          using (RegionContext.OpenAlloc()) {
            Iterator ix = new Iterator(batch);
             foreach Record x in ix {
                    var y = udf(x);
                    using (RegionContext.OpenAlloc(rtrans)) {
                      var z = rtrans.SuckObject(y);
                     rtrans.GetData().Append(z);
                   }
            }
      }
   }
   rin.GiveUp();
   rtrans.Transfer(this.getDownStreamActor());
}
```

In Allocation Context

Tuesday, July 1, 2014 3:54 PM

In this wiki, our aim is to adapt the region type system that we have previously developed to the concept of *inAllocationContext*. We can take two approaches:

1. We formalize the source language, which relies on the concept of *inAllocationContext*, and elaborate it to the standard calculus, for which we already have a type system
2. Independently develop a type system for the source language, basing our efforts on the region type system we already have.

The source language

- To model *inAllocationContext*, every function should take a region handler as an argument, which serves as its allocation context. If the topmost expression of the function body is $letregion$, then the allocation context argument is simply discarded. For a function which is inherently region polymorphic, we adopt the convention that region handler serving as its allocation context should be its first region argument. Therefore, in $\Lambda(\rho_0, \rho_1, \rho_2).\lambda x.e$, we consider ρ_0 as the inallocation context argument for the lambda that follows. For clarity, we add a superscript "a" to the region handler denoting *inAllocationContext*. Therefore: $\Lambda(\rho_0^a, \rho_1, \rho_2).\lambda x.e$.
- We assume that arguments of the function are allocated in regions that outlive the *inAllocationContext*. Therefore, $\rho_1 \succcurlyeq \rho_0^a$ and $\rho_2 \succcurlyeq \rho_0^a$, where $\succcurlyeq$ means "$outlives$". This assumption is what allows us to have local references (stored in *inAllocationContext*) to formal arguments within a function. This constraint is trivially satisfied at all call-sites, as region corresponding to the *inAllocationContext* is the latest among all the live regions.
- Consider the udf function, which is a higher-order argument to the $select$ function in Dimitrios's example. Function udf can be called from any *inAllocationContext*, even from within a static region created inside the $select$ function. Therefore, udf itself needs to be region polymorphic, which means that type of $select$ is rank-2 region polymorphic type.
- The need for higher-rank region polymorphism is not new. Without higher-rank polymorphism, it is impossible for a function to pass a locally stored value to its higher-order argument. Current systems live with this restriction in order for region inference to be possible. The inference mechanism would simply move the local value to upper (longer-living) regions, so that higher-order argument can be assigned a monomorphic type. Unfortunately, this solution does not work in the case of udf, because it allocates its return value in its *inAllocationContext*, which is a region local to the $select$ function.
- Besides solving the problem of *inAllocationContext*, Rank-2 polymorphism also lets functions pass locally allocated values to their higher-order arguments.
- Higher-rank polymorphism is strictly more general than polymorphic recursion. So, allowing higher-rank polymorphism gives polymorphic recursion for free. (Holdermans, Hage, Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators.)

$\rho, p \in \ region\ names$
$\alpha \ \in \ type\ variables$
$\epsilon \ \in \ effect\ variables$
$a \ \in \ addresses$
$C \ ::= \alpha \mid int \mid C\ ref \mid C\ list$
$T \ ::= \alpha \mid int \mid \tau\ ref \mid \tau\ list \mid rgn^1\langle\rho\rangle\langle C\rangle \mid rgn\langle\rho\rangle\langle C\rangle$
$\tau \ ::= T@\rho \mid unit \mid \tau \to \tau \mid \forall\{\rho\vec{\rho} \mid \phi\}.\tau \to \tau$
$\phi \ ::= true \mid \rho \succcurlyeq \rho \mid \phi \wedge \phi$
$\sigma ::= \forall\vec{\alpha}.\tau \mid \tau$
$e \ ::= x \mid e\ e \mid n\ at\ e \mid \lambda(x{:}\tau).e \mid letregion\langle\rho\rangle\ x\ in\ e \mid e \otimes e \mid let\ x = e\ in\ e$
$\quad \mid ref\ e \mid !\,e \mid e := e \mid \lambda\{\rho\vec{\rho} \mid \phi\}.e \mid \Lambda\vec{\alpha}.e \mid e\ \rho\vec{\rho} \mid e\ \vec{\tau} \mid Nil\ \tau \mid Cons\ e\ e$
$\quad \mid new\langle\rho\rangle\langle T\rangle \mid transfer\ e \mid open\ e\ as\ x\ in\ e \mid open^a\ e\ as\ x\ in\ e \mid valueOf\ e \mid Fix\ e$

$match\ e\ with\ Nil \Rightarrow e \mid Cons\ x\ y \Rightarrow e$

$sv ::= n \mid Nil\ \tau \mid Cons\ \tau\ v\ v \mid region\langle\rho, o\rangle \mid ref\ v \mid \langle x, e, VE\rangle$ // Storable Values

$v ::= a$ //values

- $\vec{A}$ denotes following sequence of A's : $A_0 A_1 \ldots A_n$, for some n.
- Observe that no expression has "$at\ e$" annotation. It is not possible to explicitly specify where a value needs to be stored. By default a value is stored in current *inAllocationContext*.
- There are no latent effects (i.e., effect annotation on an arrow). This simplifies type system. To ensure safety, we do not allow closures to be returned from within a $letregion$.
- Region polymorphism is higher-rank. A type of form $((\forall \rho\vec{\rho}.\tau_1 \rightarrow \tau_2) \rightarrow \tau_3$ is a valid type.
- In $\rho\vec{\rho}$, letter ρ denotes *inAllocationContext* being generalized/instantiated.
- No existential types. This means that functions which create a new transferable region, and return its handler are not possible.

Type Rules

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

$$\frac{\rho \notin \Delta \qquad \Delta\cup\{\rho\};\ \gamma\cup\{\Delta \geqslant \rho\};\ \rho;\ \Gamma \vdash e{:}\tau \quad \Delta;\ \Gamma \vdash \tau}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash letregion\langle\rho\rangle\ \text{in}\ e{:}\tau}\ \text{base}(\tau)\ \text{[T-Reg]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_1{:}\ T@\rho_1\ ref@\rho \qquad \Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_2{:}\ T@\rho_0 \quad \gamma \vdash \rho_0 \geqslant \rho_1}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_1 := e_2{:}\ unit}\ \text{[T-Assn]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}T@\rho \quad \gamma \vdash \rho \geqslant p^a}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash (ref\ e) : T@\rho\ ref@p^a}\ \text{[T-Ref]} \qquad \frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}\tau\ ref@\rho}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash !e{:}\tau}\ \text{[T-DeRef]}$$

$$\frac{\varphi_1 = \text{frv}(\vec{\rho}) \cup \{\rho\} \qquad \varphi_1 \cap \Delta = \emptyset \qquad \Delta\cup\varphi_1; \Gamma \vdash \phi \qquad \Delta\cup\varphi_1;\ \gamma\cup\{\phi\};\ \rho;\ \Gamma \vdash e{:}\ \tau_1 \rightarrow \tau_2}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash \lambda\{\rho\vec{\rho} \mid \phi\}.e{:}\ \forall\{\rho\vec{\rho} \mid \phi\}.\tau_1 \rightarrow \tau_2}\ \text{[T-RegGen]}$$

$$\frac{\text{frv}(\vec{p}) \subseteq \Delta \qquad \Delta;\ \gamma;\ p^a;\ \Gamma \vdash e : \forall\{\rho\vec{\rho} \mid \phi\}.\tau_1 \rightarrow \tau_2 \qquad \gamma \vdash [\vec{p}/\vec{\rho}][p^a/\rho]\ \phi}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e\ \rho^a\vec{p} : [\vec{p}/\vec{\rho}][\rho^a/\rho]\ \tau_1 \rightarrow \tau_2}\ \text{[T-RegInst]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}rgn^1\langle\rho\rangle\langle T\rangle \qquad \Delta\cup\{\rho\};\ \gamma;\ p^a;\ \Gamma, x{:}rgn\langle\rho\rangle\langle T\rangle \vdash e{:}\tau \quad \Delta;\ \Gamma \vdash \tau}{\Delta;\ \gamma;\ \Gamma \vdash open\ e\ as\ x\ \text{in}\ e{:}\tau}\ base(\tau)\ \&\ closed_\rho(\tau)\ \text{[T-Open]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}rgn^1\langle\rho\rangle\langle T\rangle \qquad \Delta\cup\{\rho\};\ \gamma;\ \rho;\ \Gamma, x{:}rgn\langle\rho\rangle\langle T\rangle \vdash e{:}\tau \quad \Delta;\ \Gamma \vdash \tau}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash open^a\ e\ as\ x\ in\ e{:}\tau}\ base(\tau)\ \&\ closed_\rho(\tau)\ \text{[T-OpenAlloc]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}rgn^1\langle\rho\rangle\langle T\rangle}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash transfer\ e : unit}\ \text{[T-Transfer]} \qquad \frac{\|T@\rho\| = \tau \qquad \Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}rgn\langle\rho\rangle\langle T\rangle}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash valueOf\ e{:}\tau}\ \text{[T-ValOf]}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}\ \tau_1 \quad \Delta;\ \gamma;\ \Gamma \vdash \tau_1 <: \tau_2}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e{:}\tau_2}\ \text{[T-Sub]}$$

- The context for the typing judgment consists of:
  - Δ - A set of live region names
  - γ - A context recording $outlives$ constraints of form $\rho_1 \succcurlyeq \rho_2$, where ρ_1 and ρ_2 are live regions. γ defines a partial order, whose properties are:
    - There is no least element; i.e., there need not necessarily exist a region, which is outlived by every other region. For example, region parameters of a region polymorphic function are not in $outlives$ relationship with existing concrete regions (see rule T-RegGen).
    - There is no maximum element; i.e., there need not necessarily exist a region, which outlives every other region. A heap region outlives every static private region, but does not outlive transferable regions. This is because a transferable region, whose ownership is transferred, can stay alive even after current actor is finished executing. Consequently, it is possible that a transferable region outlives the heap region of current actor, instead of being other way round.
  - ρ^a- Region handler for *inAllocationContext*.
  - Γ - Variable typing context. Also stores type variables. Γ is ordered. Any variable bound only once in Γ.
- Some auxiliary definitions:
  - $frv(A)$ and $ftv(A)$ denote free region variables and free type variables in the semantic object A, respectively.
  - A type is well-formed ($\Delta;\ \Gamma \vdash \tau$) if and only if all its free region names and free type variables are contained in Δ and Γ, respectively.
  - $base(\tau)$ Is $true$ if and only if τ is a base type (i.e., its top-level constructor is not $\forall$ or $\rightarrow$).
  - The side condition $closed_\rho(\tau)$ is actually not needed, as it is subsumed by $base(\tau)$ and well-formedness condition ($\Delta;\ \Gamma \vdash \tau$). Its indended definition is as following:
    - $closed_\rho(rgn\langle\rho'\rangle\langle T\rangle) \Leftrightarrow \rho = \rho'$
    - $closed_\rho(rgn^1\langle\rho'\rangle\langle T\rangle) \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\tau \rightarrow \tau) = false$
    - $closed_\rho\big(\forall\rho'\overrightarrow{\rho'}.\tau \rightarrow \tau\big) = false$
    - $closed_\rho(unit) = true$
    - $closed_\rho(int@\rho') \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\alpha@\rho') \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\tau\ ref@\rho') \Leftrightarrow closed_\rho(\tau) \wedge \rho = \rho'$
    - $closed_\rho(\tau\ list@\rho') \Leftrightarrow closed_\rho(\tau) \wedge \rho = \rho'$
  - If C denotes unannotated type of an object, then $\|C@\rho\|$ denotes its region-annotated type when the object is owned by (i.e., completely allocated within) region ρ. It is defined recursively:
    - $\|C@\rho\| = C@\rho,$ when $C \in \{\alpha, int\}$
    - $\|C\ ref@\rho\| = \|C@\rho\|\ ref@\rho$
    - $\|C\ list@\rho\| = \|C@\rho\|\ list@\rho$
- Rules T-Int, T-IntOp, and T-Ref capture the fact that storage for newly created value is allocated in the current *inAllocationContext* (ρ^a).
- In the T-Reg rule, observe that
  - The name of the new static region (ρ) introduced is added to Δ, and
  - The new static region (ρ) is outlived by every existing live region, even by a currently open transferable region.
- In the T-RegGen rule, observe that
  - Region names (ρ and $\mathrm{frv}(\vec{\rho})$) generalized by the expression are added to Δ before type checking bound e.
  - ρ denotes generalized inAllocationContext and $\vec{\rho}$ denotes a sequence of region names that occur free in type annotations within *e*. As descibed in the introduction section of the source language, inAllocationContext must be outlived by all other region parameters. This ensures that any values that are created in the function body can safely refer to actual parameters of the function.
  - Recall that no region outlives an open transferable region. Consequently, if the *inAllocationContext* at the time of calling a function is an open transferable region, then all parameters to the function must come from the same transferable region.

- The type system does not ensure linearity for transferable region handlers. The superscript 1 on transferable region handler ($rgn^1\langle\rho\rangle\langle T\rangle$) simply indicates that it ought to be treated linearly. Type system does not arrest aliasing. However, if a region handler is not aliased, then type system guarantees that a region cannot be freed while it is open.
- There is a subtle difference bewteen rules T-Open and T-OpenAlloc. In T-OpenAlloc, newly opened region is made *inAllocationContext*, whereas it is not the case with T-Open.

Implications of outlives relation on region params

- Consider a function that is well-typed under an empty environment. There are no references to env variables trapped in the closure. Therefore, no references to any region that 1. is not the inAllocationContext (ρ^a), and 2. does not store any of the arguments ($\rho_0, \rho_1, \ldots$).
  - The function is type checked under assumptions ($\rho_0 \succcurlyeq \rho^a, \rho_1 \succcurlyeq \rho^a, \ldots$). For type checking, regions $\rho_0, \rho_1, \ldots$ can be currently live static or transferable regions and assumptions need to checked for validity before type checking the body of the function. For the inference, however, we assume that $\rho^a, \rho_0, \rho_1, \ldots$ are all region parameters, which satisfy $outlives$ assumptions.
  - If the function creates an object in ρ^a and stores its reference in ρ_0, then it means that $\rho_0 = \rho^a$. Unification results in one less region param. This happens, for eg, while inferring the type for linked list add method.
  - $Outlives$ assumptions become $outlives$ constraints at call-sites. Therefore, if the *inAllocationContext* at the call-site is a transferable region, then all parameters have to come from the transferable region
  - Outlives assumption may not always be needed while type checking the function. Conversely, outlives constraints may be too restrictive at call-sites. This is particularly problem when the *inAllocationContext* is a transferable region. Consider a function $lenAsBoxedInt$, that takes a linkedlist allocated in any region, calculates its length as an integer, and stores the boxed integer in its *inAllocationContext*. If this function is called with a linkedlist allocated in some static region and when the *inAllocCtxt* is a transferable region, then the call does not type check. There is no sensible reason for the call to fail the type check. The root of this problem is assuming $outlives$ relation among region params by default. It can be solved by assuming $outlives$ only when it is needed to type check the function, resulting in region params explicitly annotated with constraints. We should add this functionality in next iteration.
- Now, consider the case where a function closure refers to an environment variable stored in a region (ρ_k). Assume that the function adds the variable to a list (provided as argument to the function) stored in some region ρ_1 (given as region parameter to the function). Now, it must be the case that $\rho_k \succcurlyeq \rho_1$. This constraint must be captured explicitly, as ρ_k is not an argument to the function.
- Why don't we drop subtyping for now, and simply unify instead?

Type Inference

- Region params are generalized when:
  - I am not making an assumption about this region param
  - I do not want the higher-order argument to make any assumptions about this region param.
- The first kind of params are generalized at the prenex position. The second kind are generalized at the rank-2 position.
- Most general type when we do not make assumptions, but we let higher-order argument make assumptions. So, for most general types, generalize as many region params as possible at the prenex position while trying to monomorphize the higher-order argument.
- Collect all type checking constraints (equalities). Possible types of constraints:
  - Constraints equating two region params
  - Constraints equating a region param to a concrete region in Δ.
  - Constraints equating a region param to a region created inside the function (region that $\notin \Delta$).
- Collect first two kinds of constraints (constraints that are well-formed under Δ). Solve them.
  - If solved, move further.

- Else, find pairs of non-unifiable equivalence classes (call each pair as S1 and S2). S1 and S2 must contain a param that can be generalized at rank-2 position. Find such params, and remove dependent constraints such that remaining constraints are solvable. If this is not possible, then constraints are not solvable….

- For the second kind of constraints,
- Find all free region variables that are unconstrained. Generalize them at the prenex position.

- We now demonstrate some ideas on type inference with help of the $select$ example reproduced below:

```
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {
   var rtrans = new Region<Batch<OutRecord>>();
   using (RegionContext.Open(rin)) {
          var batch = rin.GetData();
          using (RegionContext.OpenAlloc(rtrans)) {
            rtrans.SetData(new Batch<OutRecord>());
          }
          using (RegionContext.OpenAlloc()) {
            Iterator ix = new Iterator(batch);
             foreach Record x in ix {
                    var y = udf(x);
                    using (RegionContext.OpenAlloc(rtrans)) {
                      var z = rtrans.SuckObject(y);
                      rtrans.GetData().Append(z);
                   }
            }
      }
   }
   rin.GiveUp();
   rtrans.Transfer(this.getDownStreamActor());
}
```

- Assume ρ_0^a to be inAllocationContext of select, and ρ_1^a that of udf. We start with following template for the type of select:
  `void select<`ρ_0^a`>(` $rgn^1\langle\rho_1\rangle$`<Batch<Record>>@`$\rho_2$ `rin, Func<`ρ_1^a`><Record@`ρ_3`,OutRec@`ρ_4`> udf)`
- No static region is open when new transferable region is created. Its handler (`rtrans`) is stored in ρ_0^a. Assuming ρ_{tr} to be the name of the new region, the type of `rtrans` is $rgn^1\langle\rho_{tr}\rangle$`⟨Batch < OutRecord >⟩@`$\rho_0^a$.
- `batch` has type $\|Batch < Record > @\rho_1\|$ = `Batch<Record@`ρ_1`>@`ρ_1
- New static region (call is ρ_k) becomes the *inAllocationContext* when the iterator is created. The old allocation context outlives the new one, so $\rho_0^a \succcurlyeq \rho_k$. Since `rin` is already open when new static region is created, we also have that $\rho_1 \succcurlyeq \rho_k$.
- When the higher-order argument udf is called, *inAllocationContext* is ρ_1^a. We instantiate ρ_1^a in the type of udf to ρ_k, and collect the constraint $\rho_1^a \geq \rho_k$.
- The input x has type $Record@\rho_2$. Unifying formal and actual args, we have $\rho_3 \geq \rho_2$.
- y: $OutRecord@\rho_4$. Y is eventually sucked into `rtrans,` which imposes no constraints on ρ_4.
- Now, region vars generalized in non-function arguments are $\{\rho_0^a, \rho_1, \rho_2,\}$. There are no constraints on them. So, generalize them:
  `void select<`ρ_0^a, ρ_1, ρ_2`>(` $rgn^1\langle\rho_1\rangle$`<Batch<Record>>@`$\rho_2$ `rin, Func<`ρ_0^a`> <Record@`ρ_3`,OutRec@`ρ_4`> udf)`
- Now, region vars generalized in argument positions of function arguments are $\{\rho_0^a, \rho_3\}$. The constraints on them are
  $\{\rho_0^a = \rho_k, \rho_3 = \rho_1\}$.
- Since there are no regions that $outlive$ the definition of $select$, the type of $select$ has to be well-formed

under $\Delta = \emptyset$.

- Since there are no constraints on ρ_1 and ρ_2, we simply generalize ρ_1 and ρ_2 in the type $rgn^1\langle\rho_1\rangle$<Batch<Record>>@$\rho_2$, so that type type of select is:
  void select $\forall\rho_0^a\rho_1\rho_2.($ $rgn^1\langle\rho_1\rangle$<Batch<Record>>@$\rho_2$ rin, $\forall\rho_1^a$.Func<Record@ρ_3,OutRec@ρ_4> udf)
- The polymorphic type of udf (i.e., $\forall\rho_1^a$.Func<Record@ρ_3,OutRec@ρ_4>), needs to be well-formed under $\Delta = \{\rho_0^a, \rho_1, \rho_2\}$.
- The monomorphic type of udf (i.e., Func<Record@ρ_3,OutRec@ρ_4>) needs to be well-formed under $\Delta = \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$.
- ρ_3 should be such that $\rho_3 \in \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$ and $\rho_1 \succcurlyeq \rho_3$ and $\rho_3 \succcurlyeq \rho_1^a$. Therefore, $\rho_3 = \rho_1$. Applying this solution, the type of udf is $\forall\rho_1^a$. Func<Record@ρ_1,OutRec@ρ_4>.
- ρ_4 Should be such that $\rho_4 \in \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$ and $\rho_4 \succcurlyeq \rho_1^a$. Both ρ_1^a and ρ_3 satisfy the constraint of outliving ρ_1^a. However, among both the solutions, ρ_1^a is the least solution. So we conclude that $\rho_4 = \rho_1^a$.
- $\Rightarrow$ The polymorphic type of udf is: $\forall\rho_1^a\rho_3$. Func<Record@ρ_3,OutRec@ρ_1^a>

Trash:

- Assume ρ_0^a to be inAllocationContext of select. We start with following template for the type of select: void select $\forall\rho_0^a.($ $rgn^1\langle\rho_1\rangle$<Batch<Record>>@$\rho_2$ rin, $\forall\rho_1^a$.Func<Record@ρ_3,OutRec@ρ_4> udf)
- We have an assumption that $\rho_2 \succcurlyeq \rho_0^a$. (Todo: T-RegGen needs correction)
- We have a constraint that $\rho_3 \succcurlyeq \rho_1^a \wedge \rho_4 \succcurlyeq \rho_1^a$,
- Observe that ρ_0^a and ρ_1^a are already generalized.
- No static region is open when new transferable region is created. Its handler (rtrans) is stored in ρ_0^a. Assuming ρ_{tr} to be the name of the new region, the type of rtrans is $rgn^1\langle\rho_{tr}\rangle\langle$Batch < OutRecord >$\rangle@\rho_0^a$.
- batch has type $\|Batch < Record > @\rho_0\|$ = Batch<Record@ρ_0>@ρ_0
- New static region (call is ρ_k) becomes the *inAllocationContext* when the iterator is created. The old allocation context outlives the new one, so $\rho_0^a \succcurlyeq \rho_k$. Since rin is already open when new static region is created, we also have that $\rho_1 \succcurlyeq \rho_k$.
- When the higher-order argument udf is called, *inAllocationContext* is ρ_1^a. We instantiate ρ_1^a in the type of udf to ρ_k.
- x: $Record@\rho_1$ is the actual argument to udf. For the application of udf to type check, constraint $\rho_1 \succcurlyeq \rho_k$ needs to be satisfied, which is indeed satisfied.
- By unifying actual and formal arguments to udf, we have the constraint that $\rho_3 \geq \rho_1$, which means that either $\rho_1 \succcurlyeq \rho_3$ or ρ_3 can be instantiated to ρ_1.
- y: $OutRecord@\rho_4$. Y is eventually sucked into rtrans, which imposes no constraints on ρ_4.
- Since there are no regions that $outlive$ the definition of $select$, the type of $select$ has to be well-formed under $\Delta = \emptyset$.
  - Since there are no constraints on ρ_1 and ρ_2, we simply generalize ρ_1 and ρ_2 in the type $rgn^1\langle\rho_1\rangle$<Batch<Record>>@$\rho_2$, so that type type of select is:
    void select $\forall\rho_0^a\rho_1\rho_2.($ $rgn^1\langle\rho_1\rangle$<Batch<Record>>@$\rho_2$ rin, $\forall\rho_1^a$.Func<Record@ρ_3,OutRec@ρ_4> udf)
  - The polymorphic type of udf (i.e., $\forall\rho_1^a$.Func<Record@ρ_3,OutRec@ρ_4>), needs to be well-formed under $\Delta = \{\rho_0^a, \rho_1, \rho_2\}$.
  - The monomorphic type of udf (i.e., Func<Record@ρ_3,OutRec@ρ_4>) needs to be well-formed under $\Delta = \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$.
  - ρ_3 should be such that $\rho_3 \in \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$ and $\rho_1 \succcurlyeq \rho_3$ and $\rho_3 \succcurlyeq \rho_1^a$. Therefore, $\rho_3 = \rho_1$. Applying this solution, the type of udf is $\forall\rho_1^a$. Func<Record@ρ_1,OutRec@ρ_4>.
  - ρ_4 Should be such that $\rho_4 \in \{\rho_0^a, \rho_1, \rho_2, \rho_1^a\}$ and $\rho_4 \succcurlyeq \rho_1^a$. Both ρ_1^a and ρ_3 satisfy the constraint of outliving ρ_1^a. However, among both the solutions, ρ_1^a is the least solution. So we conclude that $\rho_4 = \rho_1^a$.

- ⇒ The polymorphic type of udf is: $\forall \rho_1^a \rho_3$. `Func<Record@`ρ_3`,OutRec@`ρ_1^a`>`
- ρ_4 can be any region that outlives the *inAllocationContext* of udf, namely ρ_. h

Full type rules with subtyping

Wednesday, July 9, 2014 1:25 PM

$\Delta, \gamma, \Gamma \vdash \tau_1 <: \tau_2$

$$\frac{\Delta; \Gamma \vdash T \quad \gamma \vdash \rho_1 \geqslant \rho_2}{\Delta; \gamma; \Gamma \vdash T@\rho_1 <: T@\rho_2} \text{ [ST-Base]} \qquad \frac{\Delta; \gamma; \Gamma \vdash \tau_{21} <: \tau_{11} \quad \Delta; \gamma; \Gamma \vdash \tau_{12} <: \tau_{22}}{\Delta; \gamma; \Gamma \vdash \tau_{11} \to \tau_{12} <: \tau_{21} \to \tau_{22}} \text{ [ST-Fun]}$$

$\Delta; \gamma; \rho^a; \Gamma \vdash e : \sigma$

$$\frac{\rho \notin \Delta \quad \Delta \cup \{\rho\}; \gamma \cup \{\Delta \geqslant \rho\}; \rho; \Gamma \vdash e : \tau \quad \Delta; \Gamma \vdash \tau}{\Delta; \gamma; \rho^a; \Gamma \vdash letregion\langle\rho\rangle \text{ in } e : \tau} \text{ base}(\tau) \text{ [T-Reg]} \qquad \frac{\Delta; \gamma; \rho^a; \Gamma \vdash e_1 : T@\rho_0 \; ref@\rho_1 \quad \Delta; \gamma; \rho^a; \Gamma \vdash e_2 : T@\rho_0}{\Delta; \gamma; \rho^a; \Gamma \vdash e_1 := e_2 : unit} \text{ [T-Assn]}$$

$$\frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : T@\rho^a}{\Delta; \gamma; \rho^a; \Gamma \vdash (ref\ e) : T@\rho^a \; ref@\rho^a} \text{ [T-Ref]} \qquad \frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : \tau \; ref@\rho}{\Delta; \gamma; \rho^a; \Gamma \vdash !e : \tau} \text{ [T-DeRef]}$$

$$\frac{\varphi_1 = frv(\vec{\rho}) \cup \{\rho\} \quad \varphi_1 \notin \Delta \quad \varphi_2 = frv(\vec{\rho}) \cup frv(\tau_1 \to \tau_2) \quad \Delta \cup \varphi_1; \gamma \cup \{\varphi_2 \geqslant \rho\}; \rho; \Gamma \vdash e : \tau_1 \to \tau_2}{\Delta; \gamma; \rho^a; \Gamma \vdash \lambda \rho \vec{\rho}.e : \forall \rho \vec{\rho}.\tau_1 \to \tau_2} \text{ [T-RegGen]}$$

$$\frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : \forall \rho \vec{\rho}.\tau \quad frv(\vec{p}) \subseteq \Delta \quad \gamma \vdash frv([\vec{p}/\vec{\rho}][\rho^a/\rho]\,\tau_1 \to \tau_2) \geqslant \rho^a}{\Delta; \gamma; \rho^a; \Gamma \vdash e\ \rho^a \vec{p} : [\vec{p}/\vec{\rho}][\rho^a/\rho]\,\tau_1 \to \tau_2} \text{ [T-RegInst]}$$

$$\frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle \quad \Delta \cup \{\rho\}; \gamma; \rho^a; \Gamma, x : rgn\langle\rho\rangle\langle T\rangle \vdash e : \tau \quad \Delta; \Gamma \vdash \tau}{\Delta; \gamma; \Gamma \vdash open\ e\ as\ x \text{ in } e : \tau} \; base(\tau)\ \&\ closed_\rho(\tau) \text{ [T-Open]}$$

$$\frac{\Delta; \gamma; \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle \quad \Delta \cup \{\rho\}; \gamma; \rho; \Gamma, x : rgn\langle\rho\rangle\langle T\rangle \vdash e : \tau \quad \Delta; \Gamma \vdash \tau}{\Delta; \gamma; \rho^a; \Gamma \vdash open^a\ e\ as\ x\ in\ e : \tau} \; base(\tau)\ \&\ closed_\rho(\tau) \text{ [T-OpenAlloc]}$$

$$\frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle}{\Delta; \gamma; \rho^a; \Gamma \vdash transfer\ e : unit} \text{ [T-Transfer]} \qquad \frac{\|T@\rho\| = \tau \quad \Delta; \gamma; \rho^a; \Gamma \vdash e : rgn\langle\rho\rangle\langle T\rangle}{\Delta; \gamma; \rho^a; \Gamma \vdash valueOf\ e : \tau} \text{ [T-ValOf]}$$

$$\frac{\Delta; \gamma; \rho^a; \Gamma \vdash e : \tau_1 \quad \Delta; \gamma; \Gamma \vdash \tau_1 <: \tau_2}{\Delta; \gamma; \rho^a; \Gamma \vdash e : \tau_2} \text{ [T-Sub]}$$

$\Delta; \gamma; \rho^a; \Gamma \vdash e : \sigma$

$$\frac{\Gamma(x)=\sigma}{\Delta;\gamma;\rho^a;\Gamma\vdash x:\sigma}\ [\text{T-Var}] \qquad \frac{\Delta;\gamma;\rho^a;\Gamma\vdash e_2:\tau \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_1:\tau\to\tau_2}{\Delta;\gamma;\rho^a;\Gamma\vdash e_1\ e_2:\tau_2}\ [\text{T-App}] \qquad \frac{\square}{\Delta;\gamma;\rho^a;\Gamma\vdash n:int@\rho^a}\ [\text{T-Int}]$$

$$\frac{\Delta;\gamma;\rho^a;\Gamma,x:\tau_1\vdash e:\tau_2 \quad \Delta;\Gamma\vdash\tau_1}{\Delta;\gamma;\rho^a;\Gamma\vdash\lambda(x:\tau_1).e:\tau_1\to\tau_2}\ [\text{T-Abs}] \qquad \frac{\Delta;\gamma;\rho^a;\Gamma\vdash e_1:int@\rho_1 \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_2:int@\rho_2}{\Delta;\gamma;\rho^a;\Gamma\vdash e\otimes e:int@\rho^a}\ [\text{T-IntOp}]$$

$$\frac{\rho\notin\Delta \quad \Delta\cup\{\rho\};\gamma\cup\{\Delta\geqslant\rho\};\rho;\Gamma\vdash e:\tau \quad \Delta;\Gamma\vdash\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash letregion\langle\rho\rangle\ \text{in}\ e:\tau}\ base(\tau)\ [\text{T-Reg}]$$

$$\frac{\Delta;\gamma;\rho^a;\Gamma\vdash e:T@\rho^a}{\Delta;\gamma;\ \rho^a;\Gamma\vdash(ref\ e):T@\rho^a\ ref@\rho^a}\ [\text{T-Ref}]$$

$$\frac{\Delta;\gamma;\rho^a;\Gamma\vdash e:\tau\ ref@\rho}{\Delta;\gamma;\rho^a;\Gamma\vdash !e:\tau}\ [\text{T-DeRef}] \qquad \frac{\Delta;\gamma;\rho^a;\Gamma\vdash e_1:T@\rho_0\ ref@\rho_1 \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_2:T@\rho_0}{\Delta;\gamma;\rho^a;\Gamma\vdash e_1:=e_2:unit}\ [\text{T-Assn}]$$

$$\frac{\Delta;\gamma;\ \rho^a;\Gamma\vdash e_1:\sigma_1 \quad \Delta;\gamma;\ \rho^a;\Gamma,x:\tau_1\vdash e_2:\sigma_2}{\Delta;\gamma;\rho^a;\Gamma\vdash\text{let}\ x=e_1\ \text{in}\ e_2:\sigma_2}\ [\text{T-Let}] \qquad \frac{\Delta'=\Delta\cup frv(\vec{\rho})\cup\{\rho\} \quad \Delta';\gamma;\rho;\Gamma\vdash e:\tau_1\to\tau_2}{\Delta;\gamma;\rho^a;\Gamma\vdash\lambda\rho\vec{\rho}.e:\forall\rho\vec{\rho}.\tau_1\to\tau_2}\ [\text{T-RegGen}]$$

$$\frac{frv(\vec{\rho})\subseteq\Delta \quad \Delta;\gamma;\rho^a;\Gamma\vdash e:\forall\rho\vec{\rho}.\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash e\ \rho^a\vec{p}:[\vec{p}/\vec{\rho}][\rho^a/\rho]\ \tau}\ [\text{T-RegInst}] \qquad \frac{\alpha\notin ftv(\Gamma) \quad \Delta;\gamma;\rho^a;\Gamma,\alpha\vdash e:\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash\Lambda\vec{\alpha}.e:\forall\vec{\alpha}.\tau}\ [\text{T-TypGen}]$$

$$\frac{\Delta;\gamma;\rho^a;\Gamma\vdash e:\forall\vec{\alpha}.\tau_e \quad \Delta;\Gamma\vdash\vec{\tau}}{\Delta;\gamma;\rho^a;\Gamma\vdash e\ \vec{\tau}:[\vec{\tau}/\vec{\alpha}].\tau_e}\ [\text{T-TypInst}] \qquad \frac{\rho\notin\Delta}{\Delta;\gamma;\rho^a;\Gamma\vdash new\langle\rho\rangle\langle T\rangle:\ rgn^1\langle\rho\rangle\langle T\rangle}\ [\text{T-NewRgn}]$$

$$\frac{\Delta;\gamma;\rho^a;\Gamma\vdash e:rgn^1\langle\rho\rangle\langle C\rangle \quad \Delta\cup\{\rho\};\gamma;\rho^a;\Gamma,x:rgn\langle\rho\rangle\langle C\rangle\vdash e:\tau \quad \Delta;\Gamma\vdash\tau}{\Delta;\gamma;\Gamma\vdash open\ e\ as\ x\ \text{in}\ e:\tau}\ base(\tau)\ \&\ closed_\rho(\tau)\ [\text{T-Open }]$$

$$\frac{\Delta;\gamma;\Gamma\vdash e:rgn^1\langle\rho\rangle\langle C\rangle \quad \Delta\cup\{\rho\};\gamma;\rho;\Gamma,x:rgn\langle\rho\rangle\langle C\rangle\vdash e:\tau \quad \Delta;\Gamma\vdash\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash open^a\ e\ as\ x\ in\ e:\tau}\ base(\tau)\&\ closed_\rho(\tau)\ [\text{T-OpenAlloc }]$$

$$\frac{\Delta;\gamma;\rho^{a};\Gamma\vdash e:\tau\rightarrow\tau}{\Delta;\gamma;\rho^{a};\Gamma\vdash \mathit{Fix}\ e:\tau}\ \text{[T-Fix]}\qquad \frac{\Delta;\gamma;\rho^{a};\Gamma\vdash e{:}rgn^{1}\langle\rho\rangle\langle C\rangle}{\Delta;\gamma;\rho^{a};\Gamma\vdash \mathit{transfer}\ e:\mathit{unit}}\ \text{[T-Transfer]}$$

$$\frac{\|C@\rho\|=\tau \quad \Delta;\gamma;\rho^{a};\Gamma\vdash e{:}rgn\langle\rho\rangle\langle C\rangle}{\Delta;\gamma;\rho^{a};\Gamma\vdash \mathit{valueOf}\ e{:}\tau}\ \text{[T-ValOf]}\qquad \frac{\Delta;\gamma;\rho^{a};\Gamma\vdash e{:}\ \tau_1 \quad \Delta;\gamma;\Gamma\vdash\tau_1<:\tau_2}{\Delta;\gamma;\rho^{a};\Gamma\vdash e{:}\tau_2}\ \text{[T-Sub]}$$

Talk

Friday, July 4, 2014 3:21 PM

There is a need to annotate types with the information about their regions in order to impose region constraints through types.

For example, consider a case when the root object of a transferable region (ρ_{tr}) is a linked list (l). When an object (o) is added to the linked list:

```
l.add(o);
```

O cannot come from any arbitrary region, lest it might create a reference from transferable region to some other region. We already know that ensuring safety with such inter-region pointers is very tricky. In order to prevent such pointers, type system has to ensure that O comes from the same region as l. Therefore, it is necessary that type of O be tagged with the region where the space for it is allocated.

As another example, consider inPlaceMap function that takes a linked list and an udf, and modifies linked list of objects:

```
void inPlaceMap (LinkedList<Node<T1>> ll, Func<Node<T1>,void> udf) {
        foreach(Node<T1> n in ll) {
           udf(n);
        }
}
```

Consider a case when map is called in some context with following udf:

```
void dup (Node<T> n) {
   var x = new Date();
l.add(o);

   n.timeStamp = x;
}
```

Is the call to map safe always? Or is it ever going to violate region constraints? The call is safe if and only if the linked list is allocated in the same inAllocationContext, when the call to map is made. However, this safety constraint originates at udf, and has to be propagated all the way to call-site of map. How can it be done? The answer is region type system.

How do these region annotations on types look like?

We simply decorate C# types (which we refer to as pre-types) with region identifiers. For eg, here is the type of an integer allocated in region ρ_0:

int@ρ_0

here is the region-annotated type a linked list of integers, which is allocated in a region with identifier ρ_0:

LinkedList<Node<T@ρ_0>@ρ_0>@ρ_0

The type clearly indicates a linked list allocated in ρ_0, with nodes allocates in ρ_0, where nodes store values of type T allocated in ρ_0. In practice, to avoid verbose types, we can simply write LinkedList<Node<T>>@ρ_0 to denote a linked list that is entirely allocate in ρ_0, but lets defer those conveniences to later time. Let us stick with elaborate types for time being.

With such region annotations, it can be seen that l.add(o) example can be type checked.

Similarly, a function (oneOf) that takes a linked list allocated entirely inside ρ_0 and returns one of stored values has the

following region-annotated type:

T@ρ_0 oneOf (LinkedList<Node<T@ρ_0>@ρ_0>@ρ_0 theList)

But, why restrict oneOf to lists stored in ρ_0? So, we make it parametric w.r.t the region in which the list is allocated:

T@ρ oneOf<ρ> (LinkedList<Node<T@ρ>@ρ>@ρ theList)

Similarly, the type of the canonical map function, which maps a linked list to a new list is shown below:

LinkedList<Node<T_2@ρ_1>@ρ_1>@ρ_1 map<ρ_0,ρ_1> (LinkedList<Node<T_1@ρ_0>@ρ_0>@ρ_0 theList,
Func<T_1,T_2> udf)

inAllocationContext

inAllocationContext (or as I sometimes call it, the current region) at any given program location is the youngest live region at that location.

The primary purpose of inAllocationContext is to interface with library code, which has no region annotations whatsoever. Objects created by such code has to be stored in some region, and inAllocationContext conveniently provides such region. For eg, here is a linkedlist reverse function that produces a new linked list that shares nodes with the original list:

```
LinkedList<Node<T1>> reverse(LinkedList<Node<T1>> theList) {
        var newList = new LinkedList<Node<T1>> ();
        var iter = new LLIterator(theList);
        for (iter.hasNext()){
                newList.addAtHead (iter.getNext());
        }
        return newList;
}
```

Where are newly created objects newList and iter stored? Here is where the inAllocationContext becomes very useful. Objects are simply allocated in inAllocationContext.

inAllocationContext, as the name indicates, is context sensitive. It differs from call-site to call-site. In this sense, inAllocationContext can be treated similar to dynamic scope of a function. Therefore, it must be treated as its explicit argument. We parameterize reverse w.r.t to its inAllocationContext. The type should convey this fact:

LinkedList<Node<T_2@ρ_1>@ρ_1>@ρ^a reverse<ρ^a,ρ_1> (LinkedList<Node<T_1@ρ_0>@ρ_0>@ρ_0 theList)

In case of multiple region parameters, we adopt following conventions:

- First region parameter is always context parameter
- Context is always outlived by every other region parameter. That is, in this case, $\rho_1 \succcurlyeq \rho^a$.

So far so good. Now consider the select function:

```
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {
   var rtrans = new Region<Batch<OutRecord>>();
   using (RegionContext.Open(rin)) {
          var batch = rin.GetData();
         using (RegionContext.OpenAlloc(rtrans)) {
           rtrans.SetData(new Batch<OutRecord>());
          }
         using (RegionContext.OpenAlloc()) {
           Iterator ix = new Iterator(batch);
            foreach Record x in ix {
                  var y = udf(x);
                   using (RegionContext.OpenAlloc(rtrans)) {
                     var z = rtrans.SuckObject(y);
                    rtrans.GetData().Append(z);
                  }
           }
      }
   }
   rin.GiveUp();
   rtrans.Transfer(this.getDownStreamActor());
}
```

In the absence of a concept like inAllocationContext, select cannot even be typed.

- Where is rtrans allocated?
- Where is the result of udf allocated?

Select takes a higher-order argument udf. As with any other function, udf is also parameterized over its inAllocationContext

```
void select<ρ0^a ρ1,ρ2>(Region< ρ1><Batch<Record>>@ρ2 rin,
                        Func<ρ1^a>(<Record@ρ1,OutRec@ρ1^a> udf)
```

In Allocation Context 2

Monday, July 7, 2014 9:07 PM

$\rho, p \in$ *region names*
$\alpha \in$ *type variables*
$\epsilon \in$ *effect variables*
$a \in$ *addresses*
$T ::= \alpha \mid int \mid T\ ref \mid T\ list$
$\tau ::= \alpha@\rho \mid int@\rho \mid unit \mid \tau\ ref@\rho \mid \tau\ list@\rho \mid \langle \rho^a \bar{\rho} \rangle \tau \xrightarrow{\rho^a} \tau \mid rgn^1\langle\rho\rangle\langle T\rangle@\rho$
$\mid rgn\langle\rho\rangle\langle T\rangle@\rho \mid \forall \rho\vec{\rho}.\tau \to \tau \mid \exists \rho.\tau$
$\sigma ::= \forall\vec{\alpha}.\tau \mid \tau$
$e ::= x \mid e\ e \mid n\ at\ e \mid \lambda(x{:}\tau).e \mid letregion\langle\rho\rangle\ x\ in\ e \mid e \otimes e \mid let\ x = e\ in\ e$
$\mid ref\ e \mid !\,e \mid e := e \mid \lambda\rho\vec{\rho}.e \mid \Lambda\vec{\alpha}.e \mid e\ \rho\vec{\rho} \mid e\ \vec{\tau} \mid Nil\ T \mid Cons\ e\ e$
$\mid new\langle\rho\rangle\langle T\rangle \mid transfer\ e \mid open\ e\ as\ x\ in\ e \mid open^a\ e\ as\ x\ in\ e \mid valueOf\ e \mid Fix\ e$
$\mid pack[\rho_0, e]\ as\ \exists\rho.\tau \mid unpack\ \rho, x = e\ in\ e \mid$
$match\ e\ with\ Nil \Rightarrow e \mid Cons\ x\ y \Rightarrow e$
$sv ::= n \mid Nil\ T \mid Cons\ T\ v\ v \mid region\langle\rho, o\rangle \mid ref\ v \mid \langle x, e, VE\rangle$ // Storable Values
$v ::= a$ //values

Type Rules

$\Delta;\ \rho^a; \Gamma \vdash e : \sigma$

$$\frac{\Gamma(x)=\sigma}{\Delta;\ \rho^a; \Gamma \vdash x : \sigma}\ [\text{T-Var}] \qquad \frac{\Delta;\rho^a; \Gamma \vdash e_2 : \tau \quad \Delta;\ \rho^a; \Gamma \vdash e_1 : \tau \to \tau_2}{\Delta;\ \rho^a; \Gamma \vdash e_1\ e_2 : \tau_2}\ [\text{T-App}] \qquad \frac{\square}{\Delta;\ \rho^a; \Gamma \vdash n : int@\rho^a}\ [\text{T-Int}]$$

$$\frac{\Delta;\ \rho^a; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \quad \Delta; \Gamma \vdash \tau_1}{\Delta;\ \rho^a; \Gamma \vdash \lambda(x{:}\tau_1).e : \tau_1 \to \tau_2}\ [\text{T-Abs}] \qquad \frac{\Delta;\ \rho^a; \Gamma \vdash e_1 : int@\rho_1 \quad \Delta;\ \rho^a; \Gamma \vdash e_2 : int@\rho_2}{\Delta;\ \rho^a; \Gamma \vdash e \otimes e : int@\rho^a}\ [\text{T-IntOp}]$$

$$\frac{\rho \notin \Delta \quad \Delta \cup \{\rho\};\ \rho; \Gamma \vdash e : \tau \quad \Delta; \Gamma \vdash \tau}{\Delta;\ \rho^a; \Gamma \vdash letregion\langle\rho\rangle\ \text{in}\ e : \tau}\ base(\tau)\ [\text{T-Reg}] \qquad \frac{\Delta;\ \rho^a; \Gamma \vdash e : \tau}{\Delta;\ \rho^a; \Gamma \vdash (ref\ e) : \tau\ ref@\rho^a}\ [\text{T-Ref}]$$

$$\frac{\Delta;\ \rho^a; \Gamma \vdash e : \tau\ ref@\rho}{\Delta;\ \rho^a; \Gamma \vdash !e : \tau}\ [\text{T-DeRef}] \qquad \frac{\Delta;\ \rho^a; \Gamma \vdash e_1 : \tau\ ref@\rho \quad \Delta;\ \rho^a; \Gamma \vdash e_2 : \tau}{\Delta;\ \rho^a; \Gamma \vdash e_1 := e_2 : unit}\ [\text{T-Assn}]$$

$$\frac{\Delta;\ \rho^a;\ \Gamma \vdash e_1 : \sigma_1 \qquad \Delta;\ \rho^a;\ \Gamma, x{:}\tau_1 \vdash e_2 : \sigma_2}{\Delta;\ \rho^a;\ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2}\ \text{[T-Let]} \qquad \frac{\Delta' = \Delta \cup \text{frv}(\vec{\rho}) \cup \{\rho\} \quad \Delta';\ \rho;\ \Gamma \vdash e : \tau_1 \to \tau_2}{\Delta;\ \rho^a;\ \Gamma \vdash \lambda \rho \vec{\rho}.e : \forall \rho \vec{\rho}.\tau_1 \to \tau_2}\ \text{[T-RegGen]}$$

$$\frac{\text{frv}(\vec{\rho}) \subseteq \Delta \quad \Delta;\ \rho^a;\ \Gamma \vdash e : \forall \rho \vec{\rho}.\tau}{\Delta;\ \rho^a;\ \Gamma \vdash e\ \rho^a \vec{p} : [\vec{p}/\vec{\rho}][\rho^a/\rho]\ \tau}\ \text{[T-RegInst]} \qquad \frac{\alpha \notin \text{ftv}(\Gamma) \quad \Delta;\ \rho^a;\ \Gamma, \alpha \vdash e : \tau}{\Delta;\ \rho^a;\ \Gamma \vdash \Lambda \vec{\alpha}.e : \forall \vec{\alpha}.\tau}\ \text{[T-TypGen]}$$

$$\frac{\Delta;\ \rho^a;\ \Gamma \vdash e : \forall \vec{\alpha}.\tau_e \quad \Delta;\ \Gamma \vdash \vec{\tau}}{\Delta;\ \rho^a;\ \Gamma \vdash e\ \vec{\tau} : [\vec{\tau}\ /\vec{\alpha}].\tau_e}\ \text{[T-TypInst]} \qquad \frac{\rho \notin \Delta}{\Delta;\ \rho^a;\ \Gamma \vdash new\langle\rho\rangle\langle T\rangle :\ rgn^1\langle\rho\rangle\langle T\rangle@\rho^a}\ \text{[T-NewRgn]}$$

$$\frac{\Delta;\ \rho^a;\ \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle@\rho_0 \qquad \Delta \cup \{\rho\};\ \rho^a;\ \Gamma, x{:}rgn\langle\rho\rangle\langle T\rangle@\rho_0 \vdash e : \tau \qquad \Delta;\ \Gamma \vdash \tau}{\Delta;\ \rho^a;\ \Gamma \vdash open\ e\ as\ x\ \text{in}\ e : \tau}\ base(\tau)\ \&\ closed_\rho(\tau)\ \text{[T-Open]}$$

$$\frac{\Delta;\ \gamma;\ \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle@\rho_0 \qquad \Delta \cup \{\rho\};\ \rho;\ \Gamma, x{:}rgn\langle\rho\rangle\langle T\rangle@\rho_0 \vdash e : \tau \qquad \Delta;\ \Gamma \vdash \tau}{\Delta;\ \rho^a;\ \Gamma \vdash open^a\ e\ as\ x\ in\ e : \tau}\ base(\tau)\ \&\ closed_\rho(\tau)\ \text{[T-OpenAlloc]}$$

$$\frac{\Delta;\ \rho^a;\ \Gamma \vdash e : \tau \to \tau}{\Delta;\ \rho^a;\ \Gamma \vdash Fix\ e : \tau}\ \text{[T-Fix]} \qquad \frac{\Delta;\ \rho^a;\ \Gamma \vdash e : rgn^1\langle\rho\rangle\langle T\rangle@\rho_0}{\Delta;\ \rho^a;\ \Gamma \vdash transfer\ e : unit}\ \text{[T-Trans]}$$

$$\frac{\|T@\rho\| = \tau \qquad \Delta;\ \rho^a;\ \Gamma \vdash e : rgn\langle\rho\rangle\langle T\rangle@\rho_0}{\Delta;\ \rho^a;\ \Gamma \vdash valueOf\ e : \tau}\ \text{[T-ValOf]} \qquad \frac{\rho_0 \in \Delta \qquad \Delta;\ \rho^a;\ \Gamma \vdash e : \tau}{\Delta;\ \rho^a;\ \Gamma \vdash\ pack[\rho_0, e]\ as\ \exists \rho.\tau : \exists \rho.\tau}\ \text{[T-Pack]}$$

$$\frac{\Delta;\ \rho^a;\ \Gamma \vdash e_1 : \exists \rho.\tau \qquad \Delta \cup \rho;\ \rho^a;\ \Gamma, x{:}\tau \vdash e_2 : \tau_2}{\Delta;\ \rho^a;\ \Gamma \vdash\ unpack\ \rho, x = e_1\ in\ e_2 : \tau_2}\ \text{[T-Unpack]}$$

Featherweight Java

Tuesday, July 8, 2014 1:10 PM

We formalize featherweight java with

1. Assignments
2. Parametric Polymorphism (Generics)
3. Higher-Order functions

The Language

$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$c \in Integers$
$Program = (CT, e)$
C ::= class cn extends $cn\ \{\overline{T\ f}; k\ ; \bar{d}\}$ //Class Definitions
k ::= $cn\ (\overline{T\ x})\{$ super $(\bar{x});\ \overline{\text{this}.f = x};\}$ //Constructors
d ::= $T\ mn\ (\overline{T\ x})\ \{\text{return } e;\}$ //Methods
T ::= $cn \mid int$ //Types
e ::= $c \mid x \mid e.f \mid e.mn(\bar{e}) \mid$ new $cn(\bar{e}) \mid (cn)\ e$ //Expressions

Note:

$Program = (CT, e)$
Where, $CT : cn \xrightarrow{Fin} C$ is a well-formed class table. Well-formedness of CT is point-wise extension of well-formedness of constituent classes definitions (defined below). e is the top-level expression (eg: main).

Examples

Example 1 (Point):

class Pt extends $Object$ {
  $int\ x;$
  $int\ y;$
  $Pt\ (int\ x, int\ y)\ \{$
    $super();\ this.x = x;\ this.y = y;$
  $\}$
  $int\ getx\ ()\ \{\ return\ this.x;\ \}$
  $int\ gety\ ()\{\ return\ this.y;\ \}$
  $int\ setx\ (int\ x)\{\ return\ (new\ Pt(x, this.y));\ \}$
  $int\ sety\ (int\ y)\{\ return\ (new\ Pt(this.x, y));\ \}$
}

Auxiliary Definitions

Fields: Fields of a class are those defined in the class, and those inherited from the super class. There are no fields in Object class.

$$fields(\text{Object}) = \cdot \qquad \frac{CT(B) = class\ B\ extends\ A\ \{\overline{T_B\ f}; \ldots\} \quad fields(A) = \overline{T_A\ g}}{fields(B) = \overline{T_B\ f}, \overline{T_A\ g}}$$

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.

$$\frac{CT(B)=class\ B\ extends\ A\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\} \in d_B}{mtype(mn,B) = \overline{T^1} \rightarrow T^2} \qquad \frac{CT(B)=class\ B\ extends\ A\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad mn \notin FMN(d_B)}{mtype(mn,B) = mtype(mn,A)}$$

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

$$\frac{CT(B)=class\ B\ extends\ A\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\} \in d_B}{mdef(mn,B) = (\bar{x},e)} \qquad \frac{CT(B)=class\ B\ extends\ A\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad mn \notin FMN(d_B)}{mdef(mn,B) = mdef(mn,A)}$$

Notes on CBV Operational Semantics

1. Dropping side effects has a pleasant side effect: evaluation can easily be formalized entirely within the syntax of FJ, with no additional mechanisms for modeling the heap.
2. Evaluation relation is implicitly indexed by a well-formed class table (CT).
3. Values are object creation expressions (i.e., "everything is an object"):
   $v ::= \text{new}\ cn(\bar{v})$
4. If f_i is in $fields(B)$, then $(new\ B(\bar{v})).f_i \longrightarrow v_i)$.
5. Method invocation is like function application, except that the receiver object is passed as the implicit "this" argument. That is, if $mdef(m,B) = (\bar{x},e)$ then $(new\ B(\bar{v})).m(\bar{u}) \longrightarrow [\bar{u}/\bar{x}][new\ B(\bar{v})/\text{this}]\ e$.
   a. Observe that the class of the receiver determines where to look for the body (supporting method override, dynamic dispatch)
   b. Substitution of receiver for "this" supports open recursion through self.
6. Cast expression is nop if the cast is an upcast. Evaluation gets stuck if the cast is a downcast or a stupid cast. It should be noted that this is a run-time rule that has subtyping judgment in its premise.
7. Almost all evaluation rules require class definitions. Therefore, class table should be retained at run-time.
8. Left-to-right order on function/constructor arguments
9. Number and order of arguments to constructor must match the number and order of fields in the class

Notes on Typing Rules

1. Typing judgment $\Gamma \vdash e: T$
2. Observe that type judgment is implicitly parameterized over CT.
3. Original FJ type rules are made to distinguish between typing derivations that include application of T-Dcast (rule to type check downcasts) and those that don't. Type safety is invalid for former. If we are not interested in precise statement for progress theorem, we can simplify type rules by
   a. Not distinguishing between different casts that depend on what subtype judgments are valid. Instead, we can have one rule for explicit casts: $\frac{\Gamma \vdash e:T_1}{\Gamma \vdash (T_2)\ e:T_2}$
   b. Factoring out subsumption rule that allows implicit upcasts: $\frac{\Gamma \vdash e:T_1 \quad T_1 <: T_2}{\Gamma \vdash e:T_2}$
4. Method definition well-formed: $d\ OK\ in\ B$. A method is well-formed in a class B iff
   a. it is well-typed under the assumption that "this" is an object of class B, and
   b. Its type is consistent with type of method with same name in superclass of B.
5. The implications of $d\ OK\ in\ B$ judgment:
   a. The method $int\ sum(int, int)$ is well-formed in any class
   b. Judgment emulates Fix typing rule for recursive definitions. Unlike the Fix rule, which adds the function type binding to Γ directly, the $d\ OK$ judgment extends the environment with type binding for "this", which enables application of T-Invoke-Method typing rule to type recursive occurances of method m via "this" (i.e., $this.m$).
6. Class definition well-formed: $B\ OK$. A Class B is well-formed iff:
   a. All its methods are well-formed in B, and
   b. Its constructor is well-formed - a) It has same name as that of the class (i.e., B here), b) It calls super with appropriate arguments, and c) It initializes all fields of B.
7. Subtype relation is reflexive transitive closure of subclass relation. Therefore, subtype relation is entirely defined by the programmer.

$\Gamma \vdash e : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \frac{\Gamma \vdash e{:}B \quad f{:}T \in fields(B)}{\Gamma \vdash e.f{:}T} \qquad \frac{\Gamma \vdash e_0{:}B \quad mtype(m,B) = \overline{T^1} \to T^2 \quad \Gamma \vdash \overline{e{:}T^1}}{\Gamma \vdash e_0.m(\bar{e}) : T^2} \qquad \frac{fields(B) = \overline{T\ f} \quad \Gamma \vdash \overline{e{:}T}}{\Gamma \vdash new\ B(\bar{e}){:}B} \qquad \frac{\Gamma \vdash e{:}T_1 \quad T_1 <: T_2}{\Gamma \vdash e{:}T_2}$$

$d\ OK\ in\ B$

$$\frac{\overline{x{:}T^1}, \text{this}{:}\ B \vdash e{:}T^2 \qquad CT(B) = \text{class } B \text{ extends } A\ \{\ \ldots\ \} \qquad override(m, A, \overline{T^1} \to T^2)}{T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\}\ \ OK\ in\ B}$$

$B\ OK$

$$\frac{k = B(\overline{T^A g}, \overline{T^B f})\{super(\bar{g}); \overline{this.f = f;}\} \qquad \bar{d}\ OK\ in\ B \qquad fields\ (A) = T^A g}{\text{class } B \text{ extends } A\ \{\ \overline{T^B\ f}; k\ ; \bar{d}\ \}\ \ OK}$$

Assignment

The Language

$cn \ \in Class\ Names\ (A, B, C\ \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid ()$ //Constants
$C \ ::= \ $ class cn extends $cn\ \{\overline{T\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{T\ x})\{$ super $(\bar{x})$; $\overline{\text{this}.f = x;}\}$ //Constructors
$d \ ::= \ T\ mn\ (\overline{T\ x})$ {return e;} //Methods
$T \ ::= \ cn \mid int \mid unit$ //Types
$e \ ::= \ c \mid x \mid e.f \mid e.mn(\bar{e}) \mid$ new $cn(\bar{e}) \mid (cn)\ e \mid e; e \mid e.f = e$ //Expressions

Examples

Example 1 (Point):

```
class Pt extends Object {
  int x;
  int y;
  Pt (int x, int y) {
    super(); this.x = x; this.y = y;
  }
  int getx () { return this.x; }
  int gety (){ return this.y; }
  unit setx (int x){ return this.x = x; }
  unit sety (int y){ return this.x = x; }
}
```

$\Gamma \vdash e : T$

$$\frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 ; e_2 : T} \qquad \frac{\Gamma \vdash e_1.f : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1.f = e_2 : unit}$$

$(\Sigma, e) \longrightarrow (\Sigma', e')$

Generics

The Language

$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
c ::= $n \mid ()$ //Constants
N ::= $cn\langle \bar{T} \rangle$ //Instantiated class type
C ::= class $cn\langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{T\ f}; k\ ; \bar{d} \right\}$ //Class Definitions
k ::= $cn\ (\overline{T\ x})\{$ super $(\bar{x})$; $\overline{\text{this}.f = x};\}$ //Constructors
d ::= $T\ mn\ (\overline{T\ x})\ \{\text{return } e;\}$ //Methods
$T_{\lhd}$ ::= $\alpha \mid N$
T ::= $T_{\lhd} \mid int \mid unit$ //Types
e ::= $c \mid x \mid e.f \mid e.mn(\bar{e}) \mid$ new $N(\bar{e}) \mid (N)\ e \mid e; e \mid e.f = e$ //Expressions

Notes

- When we say $\alpha \lhd N$, instantiated class N is the bound of the type variable α.

Examples

Example 1 (Pair):

```
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) {
    return this.fst = newfst;
  }
}
```

Auxiliary Definitions

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

$fields(\text{Object}) = \cdot$ $\qquad \dfrac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T_B\ f};\dots\} \quad fields([\overline{T}/\overline{\alpha}\,]N) = \overline{T_A\ g}}{fields(B\langle\overline{T}\rangle) = [\overline{T}/\overline{\alpha}\,]T_B\ f, \overline{T_A\ g}}$

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.

$$\frac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\} \in d_B}{mtype(mn, B\langle\overline{T}\rangle) = [\overline{T}/\overline{\alpha}\,]\ \overline{T^1} \to T^2}$$

$$\frac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad mn \notin FMN(d_B)}{mtype(mn, B\langle\overline{T}\rangle) = mtype(mn, [\overline{T}/\overline{\alpha}\,]N)}$$

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

$$\frac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\} \in d_B}{mdef(mn, B\langle\overline{T}\rangle) = (\bar{x}, [\overline{T}/\overline{\alpha}\,]e)}$$

$$\frac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T_B\ f};\ k_B;\ d_B\} \quad mn \notin FMN(d_B)}{mdef(mn, B\langle\overline{T}\rangle) = mdef(mn, [\overline{T}/\overline{\alpha}\,]N)}$$

Notes on Typing Rules

- We define an environment Γ to map variables to types, and environment Δ to map type variables to their bounds
- Subtyping is reflexive transitive closure of subclass relationship. Subclass relation is defined over instantiated classes. For a type variable, subclass relation is between the variable and its bound, as declared in the class definition.
- As usual, subtyping, typing and well-formedness judgments are parameterized over *CT*.
- We define $bound_\Delta$ function over $T_\lhd$. For a given $T_\lhd$, the function identifies the class where we need to look for fields or methods.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
- We need well-formedness judgment to check:
  - If a type variable is in scope, and
  - Type instantiation satisfies stated subclass constraints.
- Observe that in $class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\dots\}$, all type variables ($\bar{\alpha}$) are in scope when checking well-formedness of their bounds ($\overline{N}$). That means, $class\ B\langle\alpha \lhd LinkedList\langle\alpha\rangle\rangle \lhd Object$ is technically wellf-formed although there cannot exist any sensible instantiation for α. A more meaningful example is $class\ B\langle\alpha \lhd Object, \beta \lhd LinkedList\langle\alpha\rangle\rangle \lhd Object$.

$\Delta \vdash T_1 <: T_2$

$$\frac{}{\Delta \vdash T <: T} \qquad \frac{}{\Delta \vdash \alpha <: \Delta(\alpha)} \qquad \frac{CT(B) = class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\dots\}\ \ \Delta \vdash \overline{T}\ OK}{\Delta \vdash B\langle\overline{T}\rangle <:\ [\overline{T}/\overline{\alpha}\,]\ N} \qquad \frac{\Delta \vdash T_1 <: T_2 \quad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}$$

$\Delta; \Gamma \vdash e: T$

$$\frac{x{:}T \in \Gamma}{\Delta; \Gamma \vdash x{:}T} \qquad \frac{\Delta; \Gamma \vdash e{:}T_\lhd \quad bound_\Delta(T_\lhd) = N \quad f{:}T \in fields(N)}{\Delta; \Gamma \vdash e.f{:}T} \qquad \frac{\Delta; \Gamma \vdash e_0{:}T_\lhd \quad \Delta \vdash \overline{T^1}\ OK \quad \Delta \vdash T^2\ OK \quad mtype\big(m, bound_\Delta(T_\lhd)\big) = \overline{T^1} \to T^2 \quad \Delta; \Gamma \vdash \overline{e{:}T^1}}{\Delta; \Gamma \vdash e_0.m(\bar{e}) : T^2}$$

$$\frac{\Delta \vdash N\ OK \quad fields(N) = \overline{T\ f} \quad \Delta; \Gamma \vdash \overline{e{:}T}}{\Delta; \Gamma \vdash new\ N(\bar{e}){:}N} \qquad \frac{\Delta; \Gamma \vdash e{:}T_1 \quad T_1 <: T_2}{\Delta; \Gamma \vdash e{:}T_2} \qquad \frac{\Delta; \Gamma \vdash e_1{:}unit \quad \Delta; \Gamma \vdash e_2{:}T}{\Delta; \Gamma \vdash e_1; e_2 : T} \qquad \frac{\Delta; \Gamma \vdash e_1.f{:}T \quad \Delta; \Gamma \vdash e_2{:}T}{\Delta; \Gamma \vdash e_1.f = e_2{:}unit}$$

$\Delta \vdash T\ OK$

$$\frac{\square}{\Delta \vdash \text{Object } OK} \qquad \frac{\alpha \in dom(\Delta)}{\Delta \vdash \alpha \ OK} \qquad \frac{CT(B) = class\ B\langle \overline{\alpha \lhd N} \rangle \lhd N\ \{\ \ldots\ \} \quad \Delta \vdash \overline{T}\ OK \quad \Delta \vdash \overline{T} <: [\overline{T}/\overline{\alpha}]\ \overline{N}}{\Delta \vdash B\langle \overline{T} \rangle\ OK}$$

$d\ OK\ in\ B$

$$\frac{\Delta = \overline{\alpha \lhd N} \quad \Delta \vdash \overline{T^1}\ OK \quad \Delta \vdash T^2\ OK \quad \Delta\ ;\ \overline{x{:}T^1}, \text{this}{:}\ B\langle \overline{\alpha} \rangle \vdash e{:}T^2 \quad CT(B) = class\ B\langle \overline{\alpha \lhd N} \rangle \lhd N\ \{\ \ldots\ \} \quad override(m, N, \overline{T^1} \to T^2)}{T^2\ mn\ (\overline{T^1\ x})\ \{return\ e;\}\ \ OK\ in\ B}$$

$B\ OK$

$$\frac{\Delta = \overline{\alpha \lhd N} \quad \Delta \vdash \overline{N}\ OK \quad \Delta \vdash N\ OK \quad \Delta \vdash \overline{T^B} \quad k = B(\overline{T^A g}, \overline{T^B f}\ )\{super(\bar{g});\ \overline{this.f = f;}\} \quad \bar{d}\ OK\ in\ B \quad fields\ (N) = T^A g}{class\ B\langle \overline{\alpha \lhd N} \rangle \lhd N\ \{\ \overline{T^B f}; k\ ; \bar{d}\ \}\ \ OK}$$

Featherweight Regions

Thursday, July 10, 2014 3:52 PM

FJ with Regions

The Language

$\rho, p \in \ region\ names$
$cn \ \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \ //$Constants
$N \ ::= \ cn\langle\bar{p}\rangle\langle\bar{T}\rangle \ //$Instantiated class type
$C \ ::= \ $ class $cn\langle\bar{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N \{\overline{\tau\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{\tau\ x})\{$ super $(\bar{x})$; $\overline{\text{this}.f = x;}\}$ //Constructors
$d \ ::= \ \tau\ mn\langle\rho^a\bar{\rho} \mid \phi\rangle\ (\overline{\tau\ x})\ \{$return $e;\}$ //Methods
$\phi \ ::= true \mid \rho \geqslant \rho \mid \phi \wedge \phi$ //Outlives constraints on region params
$T_{\lhd} ::= \ \alpha \mid N$
$T \ ::= \ T_{\lhd} \mid int \mid unit \ $ //Pre-Types
$\tau \ ::= \ T@\rho$
$e \ ::= \ c \mid x \mid e.f \mid e.mn\langle p^a\bar{p}\rangle(\bar{e}) \mid$ new $N(\bar{e}) \mid (N)\ e \mid e; e \mid e.f = e$ //Expressions
$\quad \mid \ letregion\langle\rho\rangle\ \{\ e\ \}$

Notes

- When we say $\alpha \lhd N$, instantiated class N is the bound of the type variable α.

Examples

Example 1 (Pair):

```
class A extends Object {
  A() { super(); }
}
class B extends Object {
  B() { super(); }
}
class Pair⟨ρ0, ρ1⟩⟨X extends Object, Y extends Object⟩ extends Object {
  X@ρ0 fst;
  Y@ρ1 snd;
  Pair⟨ρa⟩(X@ρ0 fst, Y@ρ1 snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  X@ρ0 getfst⟨ρa⟩() {
    return this.fst;
  }
  unit setfst⟨ρa⟩(X@ρ0 newfst) {
    return this.fst = newfst;
  }
}
```

Auxiliary Definitions

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

$$fields(\text{Object}) = \cdot \qquad \frac{CT(B) = class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau_B\ f};\dots\} \quad fields([\overline{p}/\overline{\rho}][\overline{T}/\overline{\alpha}\,]N) = \overline{\tau_A\ g}}{fields(B\langle\overline{\rho}\rangle\langle\overline{T}\rangle) = \overline{[\overline{p}/\overline{\rho}][\overline{T}/\overline{\alpha}\,]\tau_B\ f}, \overline{\tau_A\ g}}$$

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.

$$\frac{CT(B) = class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau_B\ f};\ k_B;\ d_B\} \quad \tau^2\ mn\langle\rho^a\overline{\rho'} \mid \phi\rangle\ (\overline{\tau^1\ x})\ \{return\ e;\} \in d_B}{mtype(mn, B\langle\overline{p}\rangle\langle\overline{T}\rangle) = [\overline{p}/\overline{\rho}][\overline{T}/\overline{\alpha}\,]\ \langle\rho^a\overline{\rho'} \mid \phi\rangle\ \overline{\tau^1} \to \tau^2}$$

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

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

$$\frac{CT(B) = class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau_B\ f};\ k_B;\ d_B\} \quad \tau^2\ mn\langle\rho^a\overline{\rho'} \mid \phi\rangle\ (\overline{\tau^1\ x})\ \{return\ e;\} \in d_B}{mdef(mn, B\langle\overline{p}\rangle\langle\overline{T}\rangle) = (\{\rho^a\overline{\rho'} \mid \phi\}, \overline{x}, [\overline{p}/\overline{\rho}]\ [\overline{T}/\overline{\alpha}\,]e)}$$

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

Notes on Typing Rules

- We define an environment Γ to map variables to types, and environment Δ to map type variables to their bounds
- Subtyping is reflexive transitive closure of subclass relationship. Subclass relation is defined over instantiated classes. For a type variable, subclass relation is between the variable and its bound, as declared in the class definition.
- As usual, subtyping, typing and well-formedness judgments are parameterized over *CT*.
- We define $bound_\Delta$ function over $T_\lhd$. For a given $T_\lhd$, the function identifies the class where we need to look for fields or methods.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
- We need well-formedness judgment to check:
  - If a type variable is in scope, and
  - Type instantiation satisfies stated subclass constraints.
- Observe that in $class\ B\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\dots\}$, all type variables ($\overline{\alpha}$) are in scope when checking well-formedness of their bounds ($\overline{N}$). That means, $class\ B\langle\alpha \lhd LinkedList\langle\alpha\rangle\rangle \lhd Object$ is technically wellf-formed although there cannot exist any sensible instantiation for α. A more meaningful example is $class\ B\langle\alpha \lhd Object, \beta \lhd LinkedList\langle\alpha\rangle\rangle \lhd Object$.

$\Delta \vdash T_1 <: T_2$

$$\frac{\square}{\Delta \vdash T <: T} \qquad \frac{\square}{\Delta \vdash \alpha <: \Delta(\alpha)} \qquad \frac{CT(B) = class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\ \ldots\ \} \quad \Delta \vdash B\langle\overline{\rho}\rangle\langle\overline{T}\rangle\ OK}{\Delta \vdash B\langle\overline{\rho}\rangle\langle\overline{T}\rangle <:\ [\overline{\rho}/\overline{\rho}][\overline{T}/\overline{\alpha}\ ]\ N} \qquad \frac{\Delta \vdash T_1 <: T_2 \quad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}$$

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

$$\frac{\Delta \vdash T\ OK \quad \gamma \vdash \rho_1 \geqslant \rho_2}{\Delta;\ \gamma \vdash T@\rho_1 <: T@\rho_2}$$

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

$$\frac{x{:}\tau \in \Gamma}{\Delta;\gamma;p^a;\Gamma \vdash x{:}\tau} \qquad \frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash_T e{:}T_\lhd \quad bound_\Delta(T_\lhd) = N \quad f{:}\tau \in fields(N)}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e.f{:}\tau}$$

$$\frac{\Delta \vdash N\ OK \quad fields(N) = \overline{\tau\ f} \quad \Delta;\ \gamma;\ p^a;\ \Gamma \vdash \overline{e{:}\tau'} \quad \Delta;\ \gamma \vdash \overline{\tau' <: \tau}}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash new\ N(\bar{e}){:}\ N@p^a}$$

$$\frac{\begin{array}{c}\Delta;\gamma;p^a;\Gamma \vdash_T e_0{:}T_\lhd \\ mtype\big(m, bound_\Delta(T_\lhd)\big) = \langle\rho^a\overline{\rho} \mid \phi\rangle\ \overline{\tau^1} \to \tau^2 \quad \Delta;\gamma;p^a;\Gamma \vdash \overline{e{:}\tau^e} \\ S = [\overline{\rho}/\overline{\rho}][p^a/\rho^a] \quad \Delta \vdash S(\overline{\tau^1})\ OK \quad \Delta \vdash S(\tau^2)\ OK \quad \Delta;\ \gamma \vdash \overline{\tau^e <: S(\tau^1)} \quad \gamma \vdash S(\phi)\end{array}}{\Delta;\gamma;p^a;\Gamma \vdash e_0.m\langle p^a\overline{\rho}\rangle(\bar{e}) : S(\tau^2)}$$

$$\frac{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_1{:}unit \quad \Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_2{:}\tau}{\Delta;\ \gamma;\ p^a;\ \Gamma \vdash e_1; e_2{:}\ \tau} \qquad \frac{\Delta;\gamma;p^a;\Gamma \vdash e_1.f{:}\tau \quad \Delta;\gamma;p^a;\ \Gamma \vdash e_2{:}\tau' \quad \Delta;\ \gamma \vdash \tau' <: \tau}{\Delta;\gamma;p^a;\Gamma \vdash e_1.f = e_2{:}unit}$$

$\Delta \vdash T\ OK \quad \Delta \vdash \tau\ OK \quad \Delta \vdash \phi\ OK$

$$\frac{\square}{\Delta \vdash \text{Object}\ OK} \qquad \frac{\alpha \in dom(\Delta)}{\Delta \vdash \alpha\ OK} \qquad \frac{CT(B) = class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\ \ldots\ \} \quad \Delta \vdash \overline{T}\ OK \quad frv(\overline{\rho}) \in \Delta \quad \Delta \vdash \overline{T} <: [\overline{\rho}/\overline{\rho}][\overline{T}/\overline{\alpha}\ ]\ \overline{N}}{\Delta \vdash B\langle\overline{\rho}\rangle\langle\overline{T}\rangle\ OK}$$

$$\frac{\Delta \vdash T\ \text{OK} \quad \rho \in frv(\Delta)}{\Delta \vdash T@\rho\ OK} \qquad \frac{\rho_0, \rho_1 \in \Delta}{\Delta \vdash \rho_0 \geqslant \rho_1\ OK} \qquad \frac{\Delta \vdash \phi_0 OK \quad \Delta \vdash \phi_1 OK}{\Delta \vdash \phi_0 \wedge \phi_1\ OK}$$

$d\ OK\ in\ B$

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

$B\ OK$

$$\frac{\Delta=(\overline{\alpha \lhd \overline{N}};\overline{\rho}) \quad \Delta\vdash\overline{N}\ OK \quad \Delta\vdash N\ OK \quad \Delta\vdash\overline{\tau^B} \quad k=B\left(\overline{\tau^A g},\overline{\tau^B f}\right)\{super(\bar{g});\ \overline{this.f=f;}\} \quad \bar{d}\ OK\ in\ B \quad fields\ (N)=\overline{\tau^A g}}{class\ B\langle\overline{\rho}\rangle\langle\overline{\alpha \lhd \overline{N}}\rangle \lhd N\left\{\ \overline{\tau^B f};k\ ;\bar{d}\ \right\}\ OK}$$

Examples

Example 2 (Bicycle example) This one comes from Oracle's official java tutorial:

```
class Bicycle⟨R0, R1, R2⟩ extends Object {

  int@R0 cadence;
  int@R1 gear;
  int@R2 speed;

  Bicycle⟨R^a⟩ (int@R0 startCadence, int@R1 startSpeed, int@R2 startGear) {
    super();
    this.gear = startGear;
    this.cadence = startCadence;
    this.speed = startSpeed;
  }

  unit setCadence⟨R^a, R3 | R3 ⩾ R0⟩ (int@R3 newValue) {
    return this.cadence = newValue;
  }

  unit setGear⟨R^a, R4 | R4 ⩾ R1⟩(int@R4 newValue) {
    return this.gear = newValue;
  }

  int@R1 getGear⟨R^a⟩ (unit u) {
    return this.gear;
  }
  unit applyBrake⟨R^a, R5 | R5 ⩾ R2⟩ (int@R5 decrement) {
    return this.speed −= decrement;
  }

  unit speedUp⟨R^a, R5 | R5 ⩾ R2⟩ (int@R5 increment) {
    return this.speed += increment;
  }

}

class MountainBike⟨R0, R1, R2, R3⟩ extends Bicycle⟨R0, R1, R2⟩ {

  int seatHeight@R3;
  MountainBike⟨R^a⟩(int@R3 startHeight,
            int@R0 startCadence,
            int@R1 startSpeed,
            int@R2 startGear) {
    super(startCadence, startSpeed, startGear);
    this.seatHeight = startHeight;
  }
```

```
  unit setHeight⟨R^a, R6 | R6 ⩾ R3⟩(int@R6 newValue) {
    return this.seatHeight = newValue;
  }
}
```

Notes:

- Constructors in FGJR do not create objects. This needs to be addressed.
- How do we say that "all fields in this class are allocated in same region as the object itself is allocated"? Currently, we cannot say that. One way is to add a region param that denotes the owner of the object itself. In this case, inAllocCtxt parameter on constructor needs to go.

Example 3

Featherweight Regions V2

Thursday, July 10, 2014 3:52 PM

FJ with Regions

The Language

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

Notes

- When we say $\alpha \lhd N$, instantiated class N is the bound of the type variable α.
- We write $N[p^a]$ as a shorthand for $[p^a/allocRgn(N)]N$. The definition of $allocRgn$ is given below.
- $Region[\rho]\langle p^a\rangle\langle\tau\rangle$ to be treated as $\exists\rho.\ Region[\rho]\langle p^a\rangle\langle\tau\rangle$, with automatic packing and unpacking when opening and closing the region, respectively. Further, τ should be well-formed under $\Delta = \{\rho\}$.
- Although we have a variable binding like r: $Region[\rho_0]\langle p^a\rangle\langle\tau\rangle$ in the context, the type $Pair\langle\rho_0\rangle\langle\alpha@\rho_0, \beta@\rho_0\rangle \rightarrow \ \alpha@\rho_0$ is still ill-formed, as $\rho_0 \notin \Delta$. The region name cannot be used to write types, or in region param instantiations until the region is open.
- It is alright for two transferable regions to have same name (ρ_0 above). Our static semantics prevent both from being open at the same time, as a given ρ_0 cannot be bound twice in Δ (see rules for $open$ and $open^a$).

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.

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

ctype: Types of constructor arguments of a class.

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

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.

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

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

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

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

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

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, 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.

-

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

$$\frac{}{\Delta;\gamma\vdash\tau<:\tau} \qquad \frac{}{\Delta;\gamma\vdash\alpha <:\Delta(\alpha)} \qquad \frac{CT(B)=class\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha\lhd N}\rangle \lhd N\ \{\ldots\} \quad \Delta;\gamma\vdash B\langle p^a\overline{p}\rangle\langle\overline{\tau}\rangle\ OK \quad \Delta;\gamma\vdash[\overline{p}/\overline{\rho}][\overline{\tau}/\overline{\alpha}\,]\,N[p^a]\ OK}{\Delta;\gamma\vdash B\langle p^a\overline{p}\rangle\langle\overline{\tau}\rangle <:\ [\overline{p}/\overline{\rho}][\overline{\tau}/\overline{\alpha}\,]\,N[p^a]}$$

$$\frac{\Delta;\gamma\vdash\tau_1<:\tau_2 \quad \Delta;\gamma\vdash\tau_2<:\tau_3}{\Delta;\gamma\vdash\tau_1<:\tau_3}$$

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

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

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

$$\frac{p^a\in\Delta \qquad \{\rho\};\emptyset\vdash\tau\ OK}{\Delta;\gamma\vdash Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK}$$

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

$$\frac{x:\tau\in\Gamma}{\Delta;\gamma;p^a;\Gamma\vdash x:\tau} \qquad \frac{\Delta;\gamma;p^a;\Gamma\vdash e:\tau' \quad f:\tau\in fields\left(bound_\Delta(\tau')\right)}{\Delta;\gamma;p^a;\Gamma\vdash e.f:\tau} \qquad \frac{\Delta;\gamma\vdash N\ OK \quad allocRgn(N)=p \quad \gamma\vdash p^a\geqslant p \quad ctype(N)=\overline{\tau} \quad \Delta;\gamma;p^a;\Gamma\vdash\overline{e:\tau}}{\Delta;\gamma;p^a;\Gamma\vdash new\ N(\bar{e}):N}$$

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

$$\frac{\Delta;\gamma;p^a;\Gamma\vdash e:\tau_1 \quad \Delta;\gamma\vdash\tau_1<:\tau_2}{\Delta;\gamma;p^a;\Gamma\vdash e:\tau_2} \qquad \frac{\Delta;\gamma\vdash\tau_\lhd\ OK}{\Delta;\gamma;p^a;\Gamma\vdash Null:\tau_\lhd} \qquad \frac{\rho\in\Delta \quad \Delta;\gamma;\rho^a;\Gamma\vdash e:Region[\rho]\langle\rho_k^a\rangle\langle\tau\rangle}{\Delta;\gamma;\rho^a;\Gamma\vdash e.get():\tau}$$

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

$$\frac{\Delta;\gamma\vdash\tau\ OK \qquad \Delta;\gamma;\rho^a;\Gamma\vdash e:\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash let\ \tau\ x=e\Rightarrow\Gamma,x:\tau} \qquad \frac{e_1\in\{x,e.f\} \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_1:\tau \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_2:\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash e_1=e_2\Rightarrow\Gamma}$$

$$\frac{\rho\notin\Delta \quad \Delta\cup\rho;\ \gamma\cup\{\Delta\geqslant\rho\},\ \rho,\ \Gamma\vdash s\Rightarrow\Gamma'}{\Delta;\gamma;\rho^{a};\Gamma\vdash letregion\langle\rho\rangle\{\ s\ \}\Rightarrow\Gamma} \qquad \frac{\Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1\Rightarrow\Gamma' \quad \Delta;\ \gamma;\ \rho^{a};\ \Gamma'\vdash s_2\Rightarrow\Gamma''}{\Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1;s_2\Rightarrow\Gamma''}$$

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

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

$d\ OK\ in\ B$

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

$B\ OK$

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

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,Y> extends Object {
    X fst;
    Y snd;
    Pair(X fst, Y snd) {
        super(); this.fst=fst; this.snd=snd;
    }
    X getfst() {
        return this.fst;
    }
    Pair setfst(Object 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)

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

```
  T head⟨R_m^a⟩(unit u) {
    return this.head.val;
  }
  LinkedList⟨R_m^a, R1⟩⟨T⟩ reverse⟨R_m^a | R1 ⩾ R_m^a⟩(unit u) {
    LinkedList⟨R_m^a, R1⟩⟨T⟩ xs = new LinkedList⟨R_m^a, R1⟩⟨T⟩ (this.head.val);
    LinkedListNode⟨R^a, R1⟩⟨T⟩ cur = this.head.next;
      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[Rin]<LinkedList<Object>> rin) {
        let Region[Rout]<Object> rout = new Region[Rout]<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.

```
class AnActor⟨R^a⟩ extends Object⟨R^a⟩ {
  AnActor () {
    super();
  }
  unit onRecv⟨R_m^a⟩(Region[Rin]⟨R_0^a⟩⟨LinkedList⟨Rin, Rin⟩⟨Object⟨Rin⟩⟩⟩ rin) {
    let Region[Rout]⟨R_m^a⟩⟨Object⟨Rout⟩⟩ rout = new Region[Rout]⟨Object⟨Rout⟩⟩;
    open rin {
      let LinkedList⟨Rin, Rin⟩⟨Object⟨Rin⟩⟩ xs = rin.get();
      letregion⟨Rs0⟩ {
        // Rs0 is the new inAllocationContext
        // below call to reverse type checks because Rin ⩾ Rs0
        let LinkedList⟨Rs0, Rin⟩⟨Object⟨Rin⟩⟩ sx = xs.reverse⟨Rs0⟩();
```

```
            let Object⟨Rin⟩ v = sx.head⟨Rs0⟩();
            open^a rout {
              rout.suck(v);
            }
          }
        }
        rin.giveUp();
        rout.transfer();
        return ();
      }
    }
```

- Immediately needed extensions:
  - Transferable regions
  - Null value (Primarily needed to avoid initializing all instance variables in constructor)
  - Ability to leave some instance variables uninitialized in the constructor.
  - Ability to state outlives constraints explicitly over region params of class (this requirement goes hand-in-hand with above two)
  - Local variables
  - Interfaces
  - Way to call super class methods, when they are overridden in subclass
  - Iteration.
  - Polymorphic methods
- A fallout of only allowing $\bar{\rho} \succcurlyeq \rho^a$, 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.

Middleweight Regions

Thursday, July 10, 2014 3:52 PM

FJ with Regions

The Language

$\rho, p \in \ region\ names$
$cn \ \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle p^a \bar{p}\rangle\langle\bar{\tau}\rangle$ //Instantiated class type
$C \ ::= \ \text{class}\ cn\langle \rho^a \bar{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{\tau\ x})\{\ \text{super}\ (\bar{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d \ ::= \ \tau\ mn\langle \rho^a \bar{\rho} \mid \phi\rangle\ (\overline{\tau\ x})\ \{s;\ \text{return}\ e;\}$ //Methods
$\phi \ ::= true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ //Outlives constraints on region params
$\tau_{\lhd} ::= \ \alpha \mid N$
$T ::= \ Object\langle p^a\rangle \mid Region[\rho]\langle p^a\rangle\langle\tau\rangle \mid int \mid bool \mid unit$ //Base Types
$\tau \ ::= \ \tau_{\lhd} \mid T$
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid \ letregion\langle\rho\rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e \ ::= \ c \mid x \mid e.f \mid e.mn\langle p^a \bar{p}\rangle(\bar{e}) \mid \text{new}\ N(\bar{e}) \mid (N)\ e \mid e.get()$ //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. When we say $\alpha \lhd N$, instantiated class N is 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 ($R0 \geqslant R1$) 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:
   $\text{class } B\langle \rho^{a}\bar{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\ ; \bar{d}\}$
   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 $\bar{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\bar{\pi} \geqslant \pi^{a}$ 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. $Region[\rho]\langle p^a\rangle\langle\tau\rangle$ to be treated as $\exists\rho.\, Region[\rho]\langle p^a\rangle\langle\tau\rangle$, with automatic packing and unpacking when opening and closing the region, respectively. Further, τ should be well-formed under $\Delta = \{\rho\}$.
10. Although we have a variable binding like r: $Region[\rho_0]\langle p^a\rangle\langle\tau\rangle$ in the context, the type $Pair\langle\rho_0\rangle\langle\alpha@\rho_0, \beta@\rho_0\rangle \rightarrow \alpha@\rho_0$ is still ill-formed, as $\rho_0 \notin \Delta$. The region name cannot be used to write types, or in region param instantiations until the region is open.
11. It is alright for two transferable regions to have same name (ρ_0 above). Our static semantics prevent both from being open at the same time, as a given ρ_0 cannot be bound twice in Δ (see rules for $open$ and $open^a$).

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.

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

ctype: Types of constructor arguments of a class.

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

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.

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

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

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

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

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

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.
-

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

$$\frac{}{\Delta;\gamma\vdash\tau<:\tau} \qquad \frac{}{\Delta;\gamma\vdash\alpha <:\Delta(\alpha)} \qquad \frac{CT(B)=class\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha\lhd N}\rangle \lhd N\ \{\ \ldots\}}{\Delta;\gamma\vdash B\langle p^a\bar{p}\rangle\langle\bar{\tau}\rangle\ OK \qquad \Delta;\gamma\vdash[\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{\tau}/\bar{\alpha}\ ]\ N\ OK}{\Delta;\gamma\vdash B\langle p^a\bar{p}\rangle\langle\bar{\tau}\rangle <:\ [\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{\tau}/\bar{\alpha}\ ]\ N}$$

$$\frac{\Delta;\gamma\vdash\tau_1<:\tau_2 \quad \Delta;\gamma\vdash\tau_2<:\tau_3}{\Delta;\gamma\vdash\tau_1<:\tau_3}$$

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

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

$$\frac{CT(B)= class\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha\lhd N}\rangle \lhd N\ \{\ \ldots\} \quad p^a\cup\bar{p}\subseteq\Delta \quad \gamma\vdash\bar{p}\geqslant p^a \quad \Delta;\gamma\vdash\bar{\tau}\ OK \quad \Delta;\gamma\vdash[\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{\tau}/\bar{\alpha}\ ]\ \bar{N}\ \mathrm{OK} \qquad \Delta;\gamma\vdash\bar{\tau}<:[\bar{p}/\bar{\rho}][p^a/\rho^a][\bar{\tau}/\bar{\alpha}\ ]\ \bar{N}}{\Delta;\gamma\vdash B\langle p^a\bar{p}\rangle\langle\bar{\tau}\rangle\ OK}$$

$$\frac{p^a\in\Delta \qquad \{\rho\};\emptyset\vdash\tau\ OK}{\Delta;\gamma\vdash Region[\rho]\langle p^a\rangle\langle\tau\rangle\ OK}$$

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

$$\frac{x:\tau\in\Gamma}{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash x:\tau} \qquad \frac{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash e:\tau' \quad f:\tau\in fields\left(bound_{\Delta}(\tau')\right)}{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash e.f:\tau} \qquad \frac{\Delta;\gamma\vdash N\ OK \quad allocRgn(N)=p \quad \gamma\vdash p^{a}\geqslant p \quad ctype(N)=\overline{\tau} \quad \Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash \overline{e:\tau}}{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash new\ N(\bar{e}):N}$$

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

$$\frac{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash e:\tau_1 \quad \Delta;\gamma\vdash\tau_1<:\tau_2}{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash e:\tau_2} \qquad \frac{\Delta;\gamma\vdash\tau_{\triangleleft}\ OK}{\Delta;\gamma;\mathrm{p}^{a};\Gamma\vdash Null:\tau_{\triangleleft}} \qquad \frac{\rho\in\Delta \quad \Delta;\gamma;\rho^{a};\Gamma\vdash e:Region[\rho]\langle\rho_k^{a}\rangle\langle\tau\rangle}{\Delta;\gamma;\rho^{a};\Gamma\vdash e.get():\tau}$$

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

$$\frac{\Delta;\gamma\vdash\tau\ OK \quad \Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash e:\tau}{\Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash let\ \tau\ x=e\Rightarrow\Gamma,x:\tau} \qquad \frac{e_1\in\{x,\ e.f\} \quad \Delta;\gamma;\rho^{a};\Gamma\vdash e_1:\tau \quad \Delta;\gamma;\rho^{a};\Gamma\vdash e_2:\tau}{\Delta;\gamma;\rho^{a};\Gamma\vdash e_1=e_2\Rightarrow\Gamma}$$

$$\frac{\rho\notin\Delta \quad \Delta\cup\rho;\ \gamma\cup\{\Delta\geqslant\rho\},\ \rho,\ \Gamma\vdash s\Rightarrow\Gamma'}{\Delta;\gamma;\rho^{a};\Gamma\vdash letregion\langle\rho\rangle\ \{\ s\ \}\Rightarrow\Gamma} \qquad \frac{\Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1\Rightarrow\Gamma' \quad \Delta;\ \gamma;\ \rho^{a};\ \Gamma'\vdash s_2\Rightarrow\Gamma''}{\Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1;s_2\Rightarrow\Gamma''}$$

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

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

$d\ OK\ in\ B$

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

$B\ OK$

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

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:

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

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.

$$
\begin{array}{l}
class\ LinkedListNode\langle R^a, R0\rangle\langle T\ extends\ Object\langle R0\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad T\ val; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ prev; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ next; \\
\quad LinkedListNode(T\ val)\ \{ \\
\quad\quad super(); \\
\quad\quad this.val\ =\ val; \\
\quad\quad this.prev\ =\ Null; \\
\quad\quad this.head\ =\ Null; \\
\quad \} \\
\}
\end{array}
$$

```
class LinkedList⟨R^a, R1⟩⟨T extends Object⟨R1⟩⟩ extends Object⟨R^a⟩ {
  LinkedListNode⟨R^a, R1⟩⟨T⟩ head;
  int count;
  LinkedList(T v) {
    super();
    this.head = new LinkedListNode⟨R^a, R1⟩⟨T⟩(v);
   this.count = 1;
  }
  unit add⟨R_m^a | R_m^a ⩾ R^a⟩(T v) {
    let LinkedListNode⟨R^a, R1⟩⟨T⟩ n = new LinkedListNode⟨R^a, R1⟩⟨T⟩(v);
    n.next = this.head;
    this.head.prev = n;
    this.head = n;
    this.count = this.count + 1;
    return ();
  }
  T head⟨R_m^a⟩(unit u) {
    return this.head.val;
  }
  LinkedList⟨R_m^a, R1⟩⟨T⟩ reverse⟨R_m^a | R1 ⩾ R_m^a⟩(unit u) {
    let LinkedList⟨R_m^a, R1⟩⟨T⟩ xs = new LinkedList⟨R_m^a, R1⟩⟨T⟩ (this.head.val);
    let LinkedListNode⟨R^a, R1⟩⟨T⟩ cur = this.head.next;
     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[Rin]<LinkedList<Object>> rin) {
        let Region[Rout]<Object> rout = new Region[Rout]<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.

```
class AnActor⟨R^a⟩ extends Object⟨R^a⟩ {
  AnActor () {
    super();
  }
  unit onRecv⟨R_m^a, R_0^a⟩(Region[Rin]⟨R_0^a⟩ ⟨LinkedList⟨Rin, Rin⟩⟨Object⟨Rin⟩⟩⟩ rin) {
    let Region[Rout]⟨R_m^a⟩⟨Object⟨Rout⟩⟩ rout = new Region[Rout]⟨Object⟨Rout⟩⟩;
    open rin {
      let LinkedList⟨Rin, Rin⟩⟨Object⟨Rin⟩⟩ xs = rin.get();
      letregion⟨Rs0⟩ {
        // Rs0 is the new inAllocationContext
        // below call to reverse type checks because Rin ⩾ Rs0
        let LinkedList⟨Rs0, Rin⟩⟨Object⟨Rin⟩⟩ sx = xs.reverse⟨Rs0⟩();
        let Object⟨Rin⟩ v = sx.head⟨Rs0⟩();
        open^a rout {
          rout.suck(v);
        }
      }
    }
    rin.giveUp();
    rout.transfer();
    return ();
  }
}
```

-

- Immediately needed extensions:
  - Ability to state outlives constraints explicitly over region params of class (this requirement goes hand-in-hand with above two)
  - 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.

ML Type Inference

Wednesday, July 16, 2014 9:46 PM

HM(X) Type inference:

$$\frac{x{:}(\forall\overline{\alpha}.D{\Rightarrow}\tau){\in}\Gamma \quad \overline{\beta}\ new \quad (C,\psi){=}normalize(D,[\overline{\beta}/\overline{\alpha}])}{\psi_{|ftv(\Gamma)},C,\Gamma{\vdash}x{:}\psi(\tau)} \qquad \frac{\psi,C,\Gamma[x{\mapsto}\alpha]{\vdash}e{:}\tau \quad \alpha\ new}{\psi{-}\{\alpha\},C,\Gamma{\vdash}\lambda x.e{:}\psi(\alpha)\to\tau} \qquad \frac{\begin{array}{c}\psi_1,C_1,\Gamma{\vdash}e_1{:}\tau_1 \quad \psi_2,C_2,\Gamma{\vdash}e_2{:}\tau_2 \\ \psi'=\psi_1 o\psi_2 \\ D{=}C_1{\wedge}C_2{\wedge}(\tau_1{\preccurlyeq}\tau_2{\to}\alpha) \\ (C,\psi){=}normalize(D,\psi')\end{array}}{\psi_{|ftv(\Gamma)},C,\Gamma{\vdash}e_1e_2{:}\psi(\alpha)}$$

$$\frac{\begin{array}{c}\psi_1,C_1,\Gamma{\vdash}e_1{:}\tau_1 \quad (C_2,\sigma){=}gen(C_1,\psi_1(\Gamma),\tau_1) \\ \psi_2,C_3,\Gamma[x{\mapsto}\sigma]{\vdash}e_2{:}\tau \quad \psi'=\psi_1 o\psi_2 \\ D{=}C_2{\wedge}C_3 \quad (C,\psi){=}normalize(D,\psi')\end{array}}{\psi_{|ftv(\Gamma)},C,\Gamma\vdash let\ x{=}e_1\ in\ e_2{:}\ \psi(\tau)}$$

- A constraint domain SC needs to specify a set S of "solved forms". For first-order term algebra, $x = c$, $x = f(\dots)$ and $x = y$ are all solved forms, but $f(\overline{e_1}) = f(\overline{e_2})$ is not.
- Function normalize accepts a constraint D, a substitution ϕ, and produces a solved form constraint C, and reduced substition ψ. (C, ψ) is called normal form of (D, ϕ).
  - $\phi \leq \psi$
  - $C \vDash \psi(D)$
  - $\psi(C) = C$
- The generalization procedure should generalize free variables in the type that do not occur inside Γ, and isolate constraints over generalized variables as refinement of generalization.
  - $gen(C, \Gamma, \sigma) = (D, \forall\bar{\alpha}.C' \Rightarrow \sigma)$, where $C = C' \wedge D$ and $\bar{\alpha} = ftv(\sigma) \cup ftv(C) - ftv(\Gamma)$ and $\bar{\alpha} \notin ftv(D)$.

Region Inference

Monday, July 21, 2014 11:09 AM

In this document, we describe an algorithm to infer region types for OO programs using Broom regions. The input to the algorithm is a standard OO program (with Java/C# types) that uses Broom regions for memory management. Provided that the program is provably memory safe (i.e., does not create dangling references), the output of the algorithm is the same program (i.e., program with same run-time semantics) with fully annotated region types. The elaborated program should successfully type check as per the rules described in previous wiki. If the program commits a potentially memory-unsafe operation type inference fails indicating that there does not exist a region-type assignment to the program, such that it is safe as per type checking rules.

The source language of input programs and target language of output programs are shown below. Note that the target language is different from language of previous wiki in following ways:
1. Refinement predicates are now allowed over region parameters of classes, along with methods.

The Source Language

$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c ::= n \mid () \mid true \mid false \mid Null$ //Constants
$N ::= cn\langle \overline{T} \rangle$ //Instantiated class type
$C ::=$ class $cn\langle \overline{\alpha \lhd N} \rangle \lhd N \{\overline{T\ f}; k\ ; \overline{d}\}$ //Class Definitions
$k ::= cn\ (\overline{T\ x})\{$ super $(\overline{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d ::= \tau\ mn\ (\overline{\tau\ x})\ \{s;$ return $e;\}$ //Methods
$T ::= \alpha \mid N \mid Object \mid Region\langle\tau\rangle \mid int \mid bool \mid unit$ // Types
$v ::= c \mid x \mid new\ N(\overline{v})$
$s ::= \cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid letregion\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e ::= c \mid x \mid e.f \mid e.mn(\overline{e}) \mid$ new $N(\overline{e}) \mid (N)\ e \mid e.get()$ //Expressions

The Target Language

$\rho, p \in \ region\ names$
$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c ::= n \mid () \mid true \mid false \mid Null$ //Constants
$N ::= cn\langle p^a \overline{p} | \phi \rangle \langle \overline{\tau} \rangle$ //Instantiated class type
$C ::=$ class $cn\langle \rho^a \overline{\rho} \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \{\overline{\tau\ f}; k\ ; \overline{d}\}$ //Class Definitions
$k ::= cn\ (\overline{\tau\ x})\{$ super $(\overline{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d ::= \tau\ mn\langle \rho^a \overline{\rho} \mid \phi \rangle\ (\overline{\tau\ x})\ \{s;$ return $e;\}$ //Methods
$\phi ::= true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ //Outlives constraints on region params
$\tau_{\lhd} ::= \alpha \mid N$
$T ::= Object\langle p^a \rangle \mid Region[\rho]\langle p^a \rangle \langle \tau \rangle \mid int \mid bool \mid unit$ //Base Types
$\tau ::= \tau_{\lhd} \mid T$

v ::= $c \mid x \mid new\ N(\bar{v})$
s ::= $\cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid\ letregion\langle\rho\rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
e ::= $c \mid x \mid e.f \mid e.mn\langle p^a \bar{p}\rangle(\bar{e}) \mid$ new $N(\bar{e}) \mid (N)\ e \mid e.get()$ //Expressions

Inference Rules (Algorithm HM(ρ))

- Inference rules are given as syntax-directed judgments. They describe an algorithm ($HM(\rho)$) to elaborate basic class definition to a class definition with region-annotated types (hereafter called as the elaborated definition). The algorithm generates constraints over region variables such that the elaborated definition is well-formed if and only if constraints are satisfiable. HM(ρ) uses a separate constraint solving algorithm (accessible through $normalize$ function) to solve constraints. The nature of constraints and constraint solving is described later in this wiki.
- At top-level, we have the judgment $B\ OK$, which judges when the elaborated definition of class B is well-formed. As a side-effect, the judgment also populates the class table (CT') with the elaborated definition of B. Judment $B\ OK$ relies on $B\ header\_OK$ and $B\ cons\_OK$ judgments, which elaborate header (signature and instance variables) of B, and constructor of B, respectively. The three judgments represent three kinds of occassions on which constraints are solved and solution is applied - after elaborating the header, after elaborating the constructor, and each time a method is elaborated.
- Rules make use of an environment Γ to map variables to their region-annotated types, an environment Δ to map type variables to their bounds, and a set Σ of region variables in scope.
- 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.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
  $bound_\Delta(T) = T$

$B\ OK, B\ header\_OK, B\ cons\_OK$

Note: Rules in this section are rather large and difficult to read. I believe they can be simplified.

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f}; k_S\ ; \overline{d_S}\} \\ \overline{X_N}=templateTy(\overline{N_S}) \quad {}^{X}N=templateTy(N_S) \quad \overline{X_\tau}=templateTy(\bar{T}) \\ \rho^a=allocRgn({}^{X}N)\ \ \bar{\rho}=frv(\overline{X_N}, {}^{X}N, \overline{X_\tau})-\{\rho^a\} \quad \psi_i=[B\langle\rho^a\bar{\rho}\rangle\langle\bar{\alpha}\rangle/B\langle\bar{\alpha}\rangle] \\ \bar{N}=\psi_i(\overline{X_N}) \quad N=\psi_i({}^{X}N) \quad \bar{\tau}=\psi_i(\overline{X_\tau}) \end{array}}{CT'[B \mapsto class\ B\langle\rho^a\bar{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}\}]}$$

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f}; k_S\ ; \overline{d_S}\} \\ CT'(B)=class\ B\langle\rho^a\bar{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}\} \\ C_{\bar{N}} \vdash \bar{N}\ OK \quad C_N \vdash N\ OK \quad C_{\bar{\tau}} \vdash \bar{\tau}\ OK \\ C=\ C_{\bar{N}} \wedge C_N \wedge C_{\bar{\tau}} \wedge \bar{\rho} \geqslant \rho^a \quad (D, \psi_i)=normalize(C) \\ \overline{N_T}=\psi_i(\overline{X_N}) \quad N_T=\psi_i({}^{X}N) \quad \overline{\tau_T}=\psi_i(\overline{X_\tau}) \quad \rho_T^a=\psi_i(\rho^a) \quad \overline{\rho_T}=frv(\overline{N_T}, N_T, \overline{\tau_T})-\{\rho_T^a\} \\ \phi=D-\wedge(\overline{\rho_T} \geqslant \rho_T^a) \end{array}}{B\ header\_OK \dashv CT'[B \mapsto class\ B\langle\rho_T^a\overline{\rho_T} \mid \phi\rangle\langle\overline{\alpha \lhd N_T}\rangle \lhd N_T\ \{\overline{\tau_T\ f}\}]}$$

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T_S\ f};k_s\ ;\overline{d_s}\} \\ CT'(B)=\ class\ B\langle\rho_B^a\overline{\rho_B}\mid\phi_B\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f}\} \quad B\ header\_OK \\ ctype(N_B)=\overline{\tau_A} \quad k_s=B(\overline{T_x\ x})\{super(\upsilon_g);\overline{this.f=\upsilon_f;}\} \quad \overline{\tau_a}=templateTy(\overline{T_x}) \quad C_a\vdash\overline{\tau_a}\ OK \\ \Gamma=\cdot,this{:}B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle,x{:}\overline{\tau_x} \quad \Sigma=\rho^a\cup\overline{\rho} \quad \Delta=\overline{\alpha \lhd N_B} \quad \Sigma;\Delta;\Gamma;\rho_B^a;C_g\vdash\overline{\upsilon_g}\hookrightarrow\overline{\upsilon_g'{:}\tau_g} \\ \Sigma;\Delta;\Gamma;\rho_B^a;C_f\vdash\overline{this.f=\upsilon_f};\hookrightarrow\overline{this.f=\upsilon_f'}; \quad C=\ C_a\wedge C_g\wedge C_f\wedge(\overline{\rho_B}\geqslant\rho_B^a)\wedge\phi_B \quad (D,\psi_i)=normalize(C) \\ \overline{N}=\psi_i(\overline{N_B}) \quad N=\psi_i(N_B) \quad \overline{\tau}=\psi_i(\overline{\tau_B}) \quad \overline{\tau_x}=\psi_i(\overline{\tau_a}) \quad \rho^a=\psi_i(\rho_B^a) \quad \overline{\rho}=frv(\overline{N},N,\overline{\tau})-\{\rho^a\} \\ \phi=D|_{\rho^a\overline{\rho}}-\wedge\overline{\rho}\geqslant\rho^a \quad \upsilon_g''=\psi_i(\upsilon_g') \quad \upsilon_f''=\psi_i(\upsilon_f') \quad k=B(\overline{\tau_x\ x})\{super(\upsilon_g'');\overline{this.f=\upsilon_f''};\} \end{array}}{B\ cons\_OK\dashv CT'[B\mapsto class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k\}]}$$

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}\} \\ class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}\}\hookrightarrow class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k;\overline{d}\} \end{array}}{B\ OK\dashv CT'[B\mapsto class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k;\overline{d}\}]}$$

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}\} \\ CT'(B)=\ class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k\} \\ B\ cons\_OK \end{array}}{class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\}\hookrightarrow class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k\}}$$

$$\frac{\begin{array}{c} CT(B)=class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}\} \\ class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}\}\hookrightarrow class\ B\langle\rho_B^a\overline{\rho_B}\mid\phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};k;\overline{d_B}\} \\ d_s=T_r\ m(\overline{T_x\ x})\{s;return\ e;\} \quad \tau_p=templateTy(T_r) \quad \overline{\tau_a}=templateTy(\overline{T_x}) \quad new(\pi^a) \\ \overline{\pi}=frv(\tau_p,\overline{\tau_a}) \quad d_t=\tau_p\ m\langle\rho_m^a|\rho_m^a=\pi^a\rangle(\overline{\tau_a\ x})\{\cdot\} \quad \Sigma_\rho=\rho_B^a\cup\overline{\rho_B} \quad \Sigma_\pi=\pi^a\cup\overline{\pi} \\ CT'\left[B\mapsto class\ B\langle\rho_B^a\overline{\rho_B}\mid\phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};k;\overline{d_B}d_t\}\right] \quad \Gamma=\cdot,this{:}B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle,\overline{x{:}\tau_a} \quad \Sigma=\Sigma_\rho\cup\Sigma_\pi \quad \Delta=\overline{a} \\ \Sigma;\Delta;\Gamma;\pi^a;C_s\vdash s\hookrightarrow s'\dashv\Gamma' \quad \Sigma;\Delta;\Gamma';\pi^a;C_e\vdash e\hookrightarrow e'{:}\tau_q \quad C=C_s\wedge C_e\wedge(\tau_q<:\tau_p)\wedge(\overline{\rho_B}\geqslant\rho_B^a)\wedge\phi_B \quad (D,\psi_i)=norma \\ \overline{N}=\psi_i(\overline{N_B}) \quad N=\psi_i(N_B) \quad \overline{\tau}=\psi_i(\overline{\tau_B}) \quad \overline{\tau_x}=\psi_i(\overline{\tau_a}) \quad \rho^a=\psi_i(\rho_B^a) \quad \overline{\rho}=frv(\overline{N},N,\overline{\tau})-\{\rho^a\} \\ \Sigma_\rho'=\rho^a\cup\overline{\rho} \quad (\rho_m^a,\phi_m^a)=(\psi_i(\pi^a)\in\Sigma_\rho')?(\pi^a,\pi^a=\psi_i(\pi^a)):(\psi_i(\pi^a),\top) \quad \tau_r=\psi_i(\tau_p) \quad \overline{\tau_x}=\psi_i(\overline{\tau_a}) \\ \overline{\rho}=frv(\overline{\tau_x},\tau_r)-\{\rho_m^a\} \quad \Sigma_\pi'=\rho_m^a\cup\overline{\rho_m} \quad \phi=D|_{\Sigma_\rho'}-\wedge\overline{\rho}\geqslant\rho^a \quad \phi_m=D|_{\Sigma_\pi'\cup\Sigma_\rho'} \\ s''=\psi_i(s') \quad e''=\psi_i(e') \quad d=\tau_x\ m\langle\rho_m^a|\phi_m^a\wedge\phi_m\rangle(\overline{\tau_x\ x})\{s'';return\ e'';\} \end{array}}{class\ B\langle\overline{\alpha \lhd N_S}\rangle \lhd N_S\ \{\overline{T\ f};k_s\ ;\overline{d_s}d_s\}\hookrightarrow class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};k;\overline{d}d\}}$$

Auxiliary Definitions

```
fun templateTy(T) = case T of
  α|int|bool|unit => T
| Object => Object<ρ> where new(ρ)
| A(T̄) => if A ∈ dom(CT') ∧ CT'(A) = class A⟨ρ^a ρ̄ | φ⟩⟨α ◁ N̄⟩ ◁ N then
            then A<π^a π̄><τ̄> where new(π^a π̄) ∧ |π̄| = |ρ̄| ∧ τ̄ = templateTy(T̄)
            else T
| Region<T_root> =>let τ'= templateTy(T_root) in
                  let τ_root= [ρ/frv(τ')] τ' where new(ρ) in
                      Region[ρ]<π><τ_root> where new(π)
```

```
fun superClasses(B<π^a π̄><τ̄>) = case B of
  Object => {}
| Region[ρ] => Object<π^a>
| _ =>
    let class B⟨ρ^a ρ̄ | φ⟩⟨α ◁ N⟩ ◁ N = CT'(B)in
    let N' = [π̄/ρ̄][π^a/ρ^a] N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<π^a π̄><τ̄>)= π^a
  | _ => error()
```

Auxiliary judgments

We have several auxiliary judgments of form $Ctxt; C \vdash Q$ denoting that Q is derivable under context $Ctxt$, given that constraint C is satisfied.

- $C \vdash \tau\ OK$ denotes that type is well-formed subject to constraint C. This judgment is implicitly parameterized over class table CT'.
- $\Delta; C \vdash \tau_1 <: \tau_2$ collects constraints for type τ_1 to be subtype of τ_2. Δ provides bounds for free type variables in τ_1 and τ_2. Subtyping is reflexive transitive closure of subclass relationship. The only constraint 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.
- $\Sigma; \Delta; \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$ elaborates expression e containing no region variables to expression e' containing region variable annotatations. τ is the region-annotated type of elaborated expression.
- $\Sigma; \Delta; \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'$ elaborates statement s to s'. Since statements can introduce new local variables (eg: Object x = new Object()), environment containing new variable bindings is also returned.

$C \vdash \tau\ OK$

$$\top \vdash \alpha\ OK, int\ OK, bool\ OK \qquad \top \vdash Object\langle\rho\rangle\ OK \qquad \frac{frv(\tau)=\{\rho\}}{\top \vdash Region[\rho]\langle\rho^a\rangle\langle\tau\rangle\ OK}$$

$$\frac{\begin{array}{c} CT'(B)=class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N\ \{\ldots\} \\ C_\tau\vdash\overline{\tau}\ OK \quad \Delta=\overline{\alpha\lhd N} \quad new(\pi^a,\overline{\pi}) \quad |\overline{\pi}|=|\overline{\rho}| \quad \psi=[\overline{\pi}/\overline{\rho}][\pi^a/\rho^a] \\ \psi'=\psi\ o\ [\overline{\tau}/\overline{\alpha}] \quad C_\phi=\psi(\phi) \quad \Delta;C_\lhd\vdash\overline{\tau}<:\psi'(\overline{N}) \quad C=C_\tau\wedge C_\lhd\wedge C_\phi\wedge(\overline{\pi}\geqslant\pi^a) \end{array}}{C\vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ OK}$$

$\Delta;\ C \vdash \tau_1 <: \tau_2$

$$\Delta; \top \vdash \tau <: \tau \qquad \Delta;\ \pi^a = \rho^a \vdash\ B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ <:\ Object\langle\pi^a\rangle$$

$$\frac{\Delta; C\vdash\Delta(\alpha)<:\tau_2}{\Delta; C\vdash\alpha<:\tau_2} \qquad \frac{A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle\in SuperClasses(B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle)}{\Delta;\left(A\langle\pi_1^a\overline{\pi}_1\rangle\langle\overline{\tau}_1\rangle=A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle\right)\vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle<:A\langle\pi_1^a\overline{\pi}_1\rangle\langle\overline{\tau}_1\rangle}$$

$\Sigma; \Delta; \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$

$$\frac{x{:}\tau \in \Gamma}{\Sigma;\Delta;\Gamma;\ \rho^a;\top \vdash x \hookrightarrow x{:}\tau} \qquad \frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e \hookrightarrow e' {:} \tau' \quad f{:}\tau \in fields(bound_\Delta(\tau'))}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e.f \hookrightarrow e'.f{:}\tau}$$

$$\frac{\rho \in \Sigma \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e \hookrightarrow e' {:} Region[\rho]\langle \pi^a \rangle \langle \tau \rangle}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e.get() \hookrightarrow e'.get(){:}\tau}$$

$$\frac{\begin{array}{c} N{=}templateTy(N_s) \quad C_N \vdash N\ OK \\ \pi^a{=}allocRgn(N) \quad \overline{\tau_B}{=}ctype(N) \quad \Sigma;\ \Delta;\ \Gamma;\rho^a;C_e \vdash \bar{e} \hookrightarrow \overline{e_1}{:}\overline{\tau_e} \\ \Delta;\ C_\lhd \vdash \overline{\tau_e} <: \overline{\tau_B} \quad C{=}C_N \wedge C_e \wedge C_\lhd \wedge (\rho^a \succcurlyeq \pi^a) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash new\ N(\bar{e}) \hookrightarrow new\ N(\overline{e_1}){:}N} \qquad \frac{eraseRgn\ (\tau) \lhd Object}{\Sigma;\Delta;\Gamma;\ \rho^a;\top \vdash Null{:}\tau}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_0 \vdash e_0 \hookrightarrow e_0' {:} \tau_0 \quad mtype\big(m, bound_\Delta(\tau_0)\big) = \langle \rho_m^a \overline{\rho_m} \mid \phi_m \rangle \overline{\tau_x} \to \tau \\ new(\overline{\pi}) \quad |\overline{\pi}| = |\overline{\rho_m}| \quad \psi = [\overline{\pi}/\overline{\rho_m}][\rho^a/\rho_m^a] \quad C_x \vdash \overline{\psi(\tau_x)}\ OK \quad C_r \vdash \psi(\tau)\ OK \\ \Sigma;\ \Delta;\ \Gamma;\rho^a;C_e \vdash \bar{e} \hookrightarrow \overline{e'}{:}\overline{\tau_e} \quad \Delta;\ C_\lhd \vdash \overline{\tau_e} <: \overline{\psi(\tau_x)} \quad C{=}C_x \wedge C_r \wedge C_e \wedge C_\lhd \wedge \psi(\phi_m) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e_0.m(\bar{e}) \hookrightarrow e_0'.m\langle \rho^a \overline{\pi} \rangle (\overline{e'}){:}\psi(\tau)}$$

$\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash s \hookrightarrow s' \dashv \Gamma'$

$$\frac{\begin{array}{c} \tau{=}templateTy(T) \quad C_T \vdash \tau\ OK \quad \Gamma'{=}\Gamma, x{:}\tau \\ \Sigma;\Delta;\Gamma;\rho^a;C \vdash e \hookrightarrow e' {:} \tau_e \quad \Delta;\ C_\lhd \vdash \tau_e <: \tau \quad C{=}C_T \wedge C_e \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash let\ T\ x{=}e \hookrightarrow let\ \tau\ x{=}e' \dashv \Gamma'}$$

$$\frac{\begin{array}{c} e_1 \in \{x, e.f\} \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_1 \vdash e_1 \hookrightarrow e_1' {:} \tau_1 \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_2 \vdash e_2 \hookrightarrow e_2' {:} \tau_2 \quad \Delta;\ C_\lhd \vdash \tau_2 <: \tau_1 \\ C{=}C_1 \wedge C_2 \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash e_1{=}e_2 \hookrightarrow e_1'{=}e_2' \dashv \Gamma}$$

$$\frac{new(\rho) \quad \Sigma \cup \{\rho\};\ \Delta;\ \Gamma;\ \rho;\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C{=}\rho \notin \Sigma \wedge \Sigma \succcurlyeq \rho \wedge C_s}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash letregion\ \{\ s\ \} \hookrightarrow letregion\langle \rho \rangle\ \{\ s'\ \} \dashv \Gamma} \qquad \frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_1 \vdash s_1 \hookrightarrow s_1' \dashv \Gamma_1 \quad \Sigma;\ \Delta;\ \Gamma_1;\ \rho^a;\ C_2 \vdash s_2 \hookrightarrow s_2' \dashv \Gamma' \quad C{=}C_1 \wedge C_2}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash s_1;s_2 \hookrightarrow s_1';s_2' \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e \hookrightarrow e' {:} Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \\ \Sigma \cup \{\rho\};\ \Delta;\ \Gamma;\ \rho^a;\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C{=}\rho \notin \Sigma \wedge C_R \wedge C_s \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash open\ e\ \{\ s\ \} \hookrightarrow open\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e \hookrightarrow e' {:} Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \\ \Sigma \cup \{\rho\};\ \Delta;\ \Gamma;\ \rho;\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C{=}\rho \notin \Sigma \wedge C_R \wedge C_s \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash open^a\ e\ \{\ s\ \} \hookrightarrow open^a\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e \hookrightarrow e' {:} Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \quad \rho \in \Sigma \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_2 \vdash e_2 \hookrightarrow e_2' {:} \tau' \quad C{=}C_R \wedge C_2 \wedge (\tau'{=}\tau) \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash e.set(e_2) \hookrightarrow e'.set\big(e_2'\big) \dashv \Gamma} \qquad \frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R \vdash e \hookrightarrow e' {:} Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \quad \rho \notin \Sigma \quad a \in \{transfer, giveUp\}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash e.a() \hookrightarrow e'.a() \dashv \Gamma}$$

Nature of constraints

Algorithm HM(ρ) generates four kinds of constraints:

1. Region variable equality constraints ($\rho_1 = \rho_2$)
2. Region variable outlives constraints ($\rho_1 \succcurlyeq \rho_2$)
3. Object type equality constraints ($A\langle \pi_1^a \overline{\pi}_1 \rangle \langle \overline{\tau}_1 \rangle = A\langle \pi_2^a \overline{\pi_2} \rangle \langle \overline{\tau_2} \rangle$).
4. Region variable uniqueness constraints ($\rho \notin \Sigma$)

Constraints of type 3 can be elaborated trivially to equality constraints by point-wise comparision. Similarly, constraints of type 4 can be elaborated to disequality constraints. (Eg: ρ ∉ {ρ1,ρ2} can be elaborated to ρ ≠ ρ1 ∧
ρ ≠ ρ2). After elaboration, we are left with 3 kinds of constraints:

1. Region variable equality constraints ($\rho_1 = \rho_2$)
2. Region variable disequality constraints ($\rho_1 \neq \rho_2$)
3. Region variable outlives constraints ($\rho_1 \succcurlyeq \rho_2$)

The normalize function, when successful, outputs a substitution and a set of residual constraints. Residual constraints are outlives constraints over free region variables. Recall that free region varaibles are generalized as parameters at the beginning of class or method definitions. Along with region parameters, any residual constraints are also recorded as refinements (ϕ) so that they can be checked when region parameters are instantiated with concrete regions. This happens when the class is instantiated, or the method is called.

If normalize function encounters a contradiction (eg: $\rho_1 = \rho_2 \wedge \rho_1 \neq \rho_2$), it fails. This means that the program is not region safe. To demonstrate this point, consider two nested static regions:

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

The type inference algorithm generates constraints that R1 ≠ R0 and R0 ≽ R1 (as R0 outlives R1). Now, assume that code pushes an object allocated in R1 into a list allocated in R0. For this operation to be safe, R1 must outlive R0; so, HM(ρ) generates constraint that R1 ≽ R0. Now, $normalize$ has to solve following constraint:

R1 ≠ R0 ∧ R0 ≽ R1 ∧ R1 ≽ R0

Since R0 and R1 can outlive each other only when R0=R1, the constraint is simplified to:

R1 ≠ R0 ∧ R0 = R1

which is a contradiction! Hence, HM(ρ) fails to infer the type for this program.

The exact definition of constraint solving function ($normalize$) is yet to be formulated.

Limitations

There are some limitations:

- Dependencies among class definitions is a partial order. That is, mutually recursive class definitions are not allowed.
- No region polymorphic method recursion. So, a recursive call needs to be in same allocation context as the method itself.
- No higher-order methods. So, code using UDFs can only be analyzed at link-time, when all higher-order functions can be inlined.
- A generic (polymorphic) class must be used monomorphically in its definition. For eg, the definition of class LinkedList<T> can contain references to LinkedList<T>, but not to LinkedList<int>.

Going forward, we plan to relax first three restrictions.

Examples:

Consider some examples to demonstrate the region type inference process.

Example 1 (LinkedListNode) The unelaborated version is given below:

```
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;
    }
}
```

In the very first step, we consider the definition of class with its instance variables, but not with its constructor and methods:

```
class LinkedListNode<T extends Object>  extends Object  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
}
```

Note that this definition is sufficient to determine the region parameters of the class, as only those region parameters that occur free in types of instance variables are generalized at the class level (along with inAllocationContext parameter, which is the always the first region variable generalized). We start with the following template:

```
class LinkedListNode<T extends Object<R1>>  extends Object<R0>  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
}
```

Type variables (like T above), and base types like int are not region annotated. But, class types are annotated. However, notice that we haven't annotated the occurances of LinkedListNode<T> in our template. This is because they are recursive occurances, and we still don't know how many region params are there for LinkedListNode<T>. We are in the process of figuring that out.

Now, we calculate the set of free region variables in our template definition. The set is {R0,R1}. Our constraints on subtype rules dictate that a class and its superclass have same inAllocationContext parameter. Since LinkedListNode<T> extends Object<R0>, the first region variable that needs to be generalized in LinkedListNode signature is R0. Hence, its signature is LinkedListNode<R0,R1 | true><T>. The "true" denotes that we have no explicit constraints over R0 and R1, except for the implicit constraint that R1>=R0. We substitute the annotated type of LinkedListNode for unannotated occurances of LinkedListNode inside the definition, yeilding:

```
class LinkedListNode<R0,R1 | true><T extends Object<R1>> extends
Object<R0> {
```

```
    T val;
    LinkedListNode<R0,R1><T>  prev;
    LinkedListNode<R0,R1><T>  next;
}
```

This concludes the first step (header_OK). In the second part, we elaborate the constructor signature:

```
LinkedListNode(T val) {
    super();
    this.val = val;
    this.prev = Null;
    this.head = Null;
}
```

Observe that the constructor has no arguments of class type, so no annotations need to be added. The argument var, and instance variable "this.var" have same type (T). Further, Null typechecks against any type. So no constraints are generated, trivially concluding the second step (cons_OK) of the elaboration. The elaborated constructor definition is same as the original definition.

Therefore, the fully elaborated definition of LinkedListNode class is:

```
class LinkedListNode<R0,R1 | true><T extends Object<R1>> extends
Object<R0> {
    T val;
    LinkedListNode<R0,R1><T>  prev;
    LinkedListNode<R0,R1><T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = new Null;
        this.head = new Null;
    }
}
```

Example 2 (LinkedList) Now, consider the LinkedList class that uses LinkedListNode:

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

In first pass, we consider the class definition with only instance variables:

```
class LinkedList<T extends Object>  extends Object  {
    LinkedListNode<T>  head;
    int count;
}
```

Here is its initial template:

```
class LinkedList<T extends Object<R1>>  extends Object<R0>  {
    LinkedListNode<R2,R3><T>  head;
    int count;
}
```

The set of free region vars is {R0,R1,R2,R3}. We generalize them in the class signature. Since LinkedListNode extends Object<R0>, its allocRgn needs to be R0; so, it is the first region var generalized. The generalized class type is LinkedListNode<R0,R1,R2,R3 | true><T>. There are no recursive occurances of LinkedList<T> to be substituted with new type. The final template is:

```
class LinkedList<R0,R1,R2,R3 | true><T extends Object<R1>>  extends
Object<R0>  {
    LinkedListNode<R2,R3><T>  head;
    int count;
}
```

The definition of LinkedListNode class dictates that for its instance LinkedListNode<R2,R3><T> to be OK, it better be the case that
a) R3>=R2, and
b) T <: Object<R3>.
But we know that:
c) T <: Object<R1>
Unifying b and c generates constraint that
d) R3=R1
We already have implicit constraints:
{R1>=R0, R2>=R0, R3>=R0}
So, total set of constraints is:
{R1>=R0, R2>=R0, R3>=R0, R3>=R2, R3=R1}
Solving constraints result in substitution: [R1/R3] and residual constraints:
{R1>=R0, R2>=R0, R1>=R2}
The first two constraints are implicit. The only explicit constraint is {R1>=R2}.
We perform the substitution, generalize free region vars, and record the explicit constraint as refinement of generalized region vars:

```
class LinkedList<R0,R1,R2 | R1>=R2><T extends Object<R1>>  extends
Object<R0>  {
    LinkedListNode<R2,R1><T>  head;
    int count;
}
```

This concludes header_OK pass. We now move to cons_OK pass. The constructor is:

```
LinkedList(T v) {
    super();
    this.head = new LinkedListNode<T>(v);
    this.count=1;
}
```

Arguments require no region annotations. The statement requiring elaboration is:

```
this.head = new LinkedListNode<T>(v);
```

We start with following template for LinkedListNode instance: LinkedListNode<R4,R5><T>. Constructor is executed in same allocation context for class. So, when new LinkedListNode is created, allocation context is R0. Our rules dictate that a) R0>=R4. Further, for LinkedListNode<R4,R5><T> to be OK requires that R5>=R4 and T<:Object<R5>. But we know that T<:Object<R1>. Unifying generates constraint that R5=R1. The type of LHS is LinkedListNode<R2,R1><T>, whereas on RHS is LinkedListNode<R4,R5><T>. Unifying requires that R4=R2 and R5=R1. So, the new set of constraints is:

{R0>=R4, R5>=R4, R5=R1, R4=R2}

We already have R1>=R2 as explicit constraint, and {R2>=R0, R1>=R0} as implicit constraints. Therefore, the final set of constraints is:

{R1>=R2, R2>=R0, R1>=R0, R0>=R4, R5>=R4, R5=R1, R4=R2}

Solving generates substitution [R0/R1,R1/R5,R0/R4], and residual constraint as R1>=R0. We apply the substitution to yeild the following version of elaborated class definition:

```
class LinkedList<R0,R1 | true><T extends Object<R1>>  extends
Object<R0>  {
    LinkedListNode<R0,R1><T>  head;
    int count;
    LinkedList(T v) {
        super();
        this.head = new LinkedListNode<R0,R1><T>(v);
        this.count=1;
    }
}
```

Observe that residual constraint (R1>=R0) need not be recorded as refinement, as it is implicitly guaranteed.

Similarly, we elaborate the add method to following elaborated version:

```
unit add<R2 | R2>=R0 )(T v) {
    let LinkedListNode<R0,R1><T>  n = new LinkedListNode<R0,R1><T>(v);
    n.next = this.head;
```

```
    this.head.prev = n;
    this.head = n;
    this.count=this.count+1;
    return ();
}
```

Observe the new allocation context parameter for add method. We parameterize every method over its allocation context.
Likewise, the reverse function shown below:

```
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;
}
```

Is elaborated to following version:

```
LinkedList<R2,R1><T>   reverse<R2 |R1>=R2> (unit u) {
    let LinkedList<R2,R1><T>  xs = new LinkedList<R2,R1><T>
(this.head.val);
    let LinkedListNode<R2,R1><T>   cur = this.head.next;
    while(not (cur== Null)) {
      xs.add(cur.val);
      cur = cur.next;
    }
    return xs;
}
```

Observe that elaborated version allows spine of the new list to be allocated in the current inAllocationContext as long as it is outlived by R1 (region where elements of the list are stored). This elaboration does not change the set of free region vars, nor does it impose any new constraints on free region vars. So, rest of the class defn remains the same. The defn of LinkedList with reverse method is:

```
class LinkedList<R0,R1 | true><T extends Object<R1>>  extends
Object<R0>  {
    LinkedListNode<R0,R1><T>  head;
    int count;
    LinkedList(T v) {
        super();
        this.head = new LinkedListNode<R0,R1><T>(v);
        this.count=1;
    }

  unit add<R2 | R2>=R0 )(T v) {
    let LinkedListNode<R0,R1><T>  n = new LinkedListNode<R0,R1><T>(v);
    n.next = this.head;
    this.head.prev = n;
    this.head = n;
    this.count=this.count+1;
    return ();
```

```
  }
  LinkedList<R2,R1><T>   reverse<R2 |R1>=R2> (unit u) {
    let LinkedList<R2,R1><T>  xs = new LinkedList<R2,R1><T>
(this.head.val);
    let LinkedListNode<R2,R1><T>   cur = this.head.next;
    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 {
                let LinkedList<Object> sx = xs.reverse();
                let Object v = sx.head;
                openAlloc rout {
                    rout.suck(v);
                }
            }
        }
        rin.giveUp();
        rout.transfer();
        return ();
    }
}
```

The Actor class has no instance variables, and trivial constructor. Therefore, the first two phases (header_OK and cons_OK) are trivial, and lead to following defn:

```
class AnActor<R0> extends Object<R0> {
    AnActor () {
        super();
    }
}
```

We now elaborate the onRecv function. We start with the following template for its type:

```
unit onRecv<R2,R3>(Region[Rin]<R3><LinkedList<Rin,Rin><Object<Rin>>>
rin);
```

The identifier Rin in square brackets statically identifies the input transferable region. We assume that every new region creation expression can be statically identified with a unique identifier. These unique identifiers can be introduced by the compiler during parsing.

R2 is the allocation context parameter for `onRecv` function, and R3 is the region where region root pointer for input transferable region (Rin) is allocated. Notice that the type of input region root object indicates that it is entirely allocated inside Rin, as required by our constraints on objects in transferable region.

First statement of `onRecv` creates a new transferable region with identifer as Rout. The template for the statement is:

```
let Region[Rout]<R4><Object<Rout>> rout = new Region[Rout]<R5>
<Object<Rout>>;
```

With constraints: {R2>=R4, R2>=R5, R5=R4}

Identifier Rout identifies newly created transferable region. The binding `rout : Region[Rout] <R4><Object<Rout>>` is added to current var env. The next statement is `open rin {…}`. Since `rin:Region[Rin]<R3><LinkedList<Rin,Rin><Object<Rin>>>` , we add Rin to the current set of live region identifiers before elaborating statements inside open.

The first inner statement is `let LinkedList<Object> xs = rin.get();`. Template is:

```
let LinkedList<R6,R7><Object<R7>> xs = rin.get();
```

Checking OK for type on LHS produces constraint that R7>=R6. The type of RHS expression is `LinkedList<Rin,Rin><Object<Rin>>`. Equating, we get constraints: {R6=Rin, R7=Rin}.

At this point, the complete constraint set is: {R2>=R4, R2>=R5, R5=R4, R6=Rin, R7=Rin, R7>=R6}. We add `xs : LinkedList<R6,R7><Object<R7>>` to var env (Γ).

Moving on, we next elaborate letregion {…} statement. We generate unique identifier Rs0 to identify this region creation expression, and add the constraint that 1) It is outlived by existing live regions {Rin>=Rs0}, and 2) It is different from any of the existing live regions: {Rs0 ≠ Rin}. Region Rso becomes new allocation context.

Next, we elaborate `let LinkedList<Object> sx = xs.reverse();`. The template is:

```
let LinkedList<R8,R9><Object<R9>> sx = xs.reverse<Rs0><R>();
```

Instantiating the constraints on reverse region params gives us constraint that {R7>=Rs0}. The return type of reverse is `LinkedList<Rs0,R7><Object<R7>>`.Checking OK on LHS gives us constraint that {R9 >= R8}. Equating type on LHS to return type of reverse on RHS gives us constraints: {R8=Rs0, R9 =R7}. So, the current constraint set is:

{R2>=R4, R2>=R5, R5=R4, R6=Rin, R7=Rin, R7>=R6, Rin>=Rs0, Rs0 ≠ Rin, R7>=Rs0, R9 >= R8, R8=Rs0, R9 =R7}.
We add `sx:LinkedList<R8,R9><Object<R9>>` to the env(Γ).

Likewise, we elaborate `let Object v = sx.head;` to `let Object<R10> v = sx.head;` along with constraints: {R10 = R8}.

Next, we open Rout for allocation making it live, and adding uniqueness constraints that state {Rout ≠ Rin, Rout ≠ Rs0 }. Function suck imposes no constraints.

Now, we come out of all three nested open and letregion blocks; so, we remove Rout, Rs0 and Rin from

our set of live region names.

Next, `rin.giveUp()` statement is OK as Rin is no longer live. Similarly, `rout.transfer()` is OK. With this, our constraint generation is complete. The elaborated onRecv function before constraint solving is:

```
unit onRecv<R2,R3>(Region[Rin]<R3><LinkedList<Rin,Rin><Object<Rin>>>
rin) {
  let Region[Rout]<R4><Object<Rout>> rout = new Region[Rout]<R5>
<Object<Rout>>;
        open rin {
            let LinkedList<R6,R7><Object<R7>> xs = rin.get();
            letregion<Rs0> {
                let LinkedList<R8,R9><Object<R9>> sx = xs.reverse<Rs0>
<R>();
                let Object<R10> v = sx.head;
                openAlloc rout {
                    rout.suck(v);
                }
            }
        }
        rin.giveUp();
        rout.transfer();
        return ();
    }
}
```

The set of constraints is: {R2>=R4, R2>=R5, R5=R4, R6=Rin, R7=Rin, R7>=R6, Rin>=Rs0, Rs0 ≠ Rin, R7>=Rs0, R9 >= R8, R8=Rs0, R9=R7, R10 = R8, Rout ≠ Rin, Rout ≠ Rs0 }.

Now, we solve constraints. The substitutions yeilded are: [R4/R5, Rin/R6, Rin/R7, Rs0/R8, Rin/R9, Rs0/R10].
The residual constraints are {R2>=R4, Rin>=Rs0, Rs0≠ Rin, Rout ≠ Rin, Rout ≠ Rs0 }.

We apply substitutions to the elaborated onRecv defn to yeild:

```
unit onRecv<R2,R3|true>(Region[Rin]<R3><LinkedList<Rin,Rin>
<Object<Rin>>> rin) {
  let Region[Rout]<R4><Object<Rout>> rout = new Region[Rout]<R4>
<Object<Rout>>;
        open rin {
            let LinkedList<Rin,Rin><Object<Rin>> xs = rin.get();
            letregion<Rs0> {
                let LinkedList<Rs0,Rin><Object<Rin>> sx =
xs.reverse<Rs0><R>();
                let Object<Rs0> v = sx.head;
                openAlloc rout {
                    rout.suck(v);
                }
            }
        }
        rin.giveUp();
        rout.transfer();
        return ();
    }
}
```

The refinement is for generalized region vars (R2,R3) is "true" because there are no residual constraints over R2 and R3. Among the residual constraints, Rin>=Rs0, Rs0≠ Rin, Rout ≠ Rin and Rout ≠ Rs0 are guaranteed to hold (they are implicit constraints); so, they need not be recorded in any case. The remaining constraint R2>=R4 essentially says that Region object for Rout transferable region can be assumed to be allocated in any region that is outlived by R2 (the inAllocationContext of onRecv function).

Example 4 (Map Actor) Let us consider the map actor that uses a higher-order udf:

```
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {

   var rtrans = new Region<Batch<OutRecord>>();
   using (RegionContext.Open(rin)) {
         var batch = rin.GetData();
         using (RegionContext.OpenAlloc(rtrans)) {
          rtrans.SetData(new Batch<OutRecord>());
         }
         using (RegionContext.OpenAlloc()) {
          Iterator ix = new Iterator(batch);
           foreach Record x in ix {
                 var y = udf(x);
                 using (RegionContext.OpenAlloc(rtrans)) {
                   var z = rtrans.SuckObject(y);
                   rtrans.GetData().Append(z);
                }
           }
      }
   }
   rin.GiveUp();
   rtrans.Transfer(this.getDownStreamActor());
}
```

This example is similar to simple actor example (3), except that this uses udf to map records in Rin to Rout. We start with the following template for method type :

```
void select<Ra0,R0>(Region[Rin]<R0><Batch<Rin,Rin><Record<Rin>>> rin,
                (Func<Ra1><Record<R1>,OutRec<R2>) udf);
```

Func is no ordinary type constructor. It denotes a function, and like all functions, it expects an inAllocationContext. So, Ra1 here denotes not the region where Func object itself is allocated, but the region which is the inAllocation context for the method. Notice that we haven't generalized R1 and R2. If we go by the usual rules, we will have to generalize Ra1, R1 and R2 along with Ra0 and R0 leading to:

```
void select<Ra0,Ra1,R0,R1,R2>(Region[Rin]<R0><Batch<Rin,Rin>
<Record<Rin>>> rin,
                    (Func<Ra1><Record<R1>,OutRec<R2>) udf);
```

This is clearly problematic, as we now require the caller of select to determine the inAllocationContext for any calls to udf inside select.

```
void select<Ra0,R0,R>(Region[Rin]<R0><Batch<Rin,Rin><Record<Rin>>>
rin,
                (Record<R1> -> OutRec<R2>) udf);
```

Middleweight Regions + Exists

Thursday, July 10, 2014 3:52 PM

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.

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 $cn\langle \rho^a \bar{\rho} \mid \phi \rangle\langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{\tau\ f}; k\ ; \bar{d} \right\}$ //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
$\tau \ ::= \ \tau_{\lhd} \mid Object\langle p^a \rangle \mid Region[\rho]\langle p^a \rangle\langle \tau \rangle \mid int \mid bool \mid unit \mid \exists \rho.\tau$
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid \ letregion\langle \rho \rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \mid e.suck(e)$
$\quad \mid \ let\ (\rho, \tau\ x) = unpack\ e$
$e \ ::= \ c \mid x \mid e.f \mid e.mn\langle p^a \bar{p} \rangle(\bar{e})$ | 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 ($R0 \geqslant R1$) 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:
   $\text{class } B\langle \rho^a \bar{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{\tau f}; k ; \bar{d} \right\}$
   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 $\bar{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\bar{\pi} \geqslant \pi^a$ 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.

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

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.

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

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

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

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

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

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.
-

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\rho\notin\Delta \quad \Sigma\cup\rho;\ \Delta\cup\rho;\ \gamma\cup\{\Delta\geqslant\rho\},\ \rho,\ \Gamma\vdash s\Rightarrow \Gamma';\Sigma'}{\Sigma;\Delta;\gamma;\rho^{a};\Gamma\vdash letregion\langle\rho\rangle\ \{\ s\ \}\Rightarrow\Gamma;\Sigma} \qquad \frac{\Sigma;\ \Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1\Rightarrow\Gamma';\Sigma' \quad \Sigma';\ \Delta;\ \gamma;\ \rho^{a};\ \Gamma'\vdash s_2\Rightarrow\Gamma'';\Sigma''}{\Sigma;\ \Delta;\ \gamma;\ \rho^{a};\ \Gamma\vdash s_1;s_2\Rightarrow\Gamma'';\Sigma''}$$

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

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

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

$d\ OK\ in\ B$

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

$B\ OK$

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

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.

$$
\begin{array}{l}
class\ LinkedListNode\langle R^a, R0\rangle\langle T\ extends\ Object\langle R0\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad T\ val; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ prev; \\
\quad LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ next; \\
\quad LinkedListNode(T\ val)\ \{ \\
\quad\quad super(); \\
\quad\quad this.val\ =\ val; \\
\quad\quad this.prev\ =\ Null; \\
\quad\quad this.head\ =\ Null; \\
\quad \} \\
\} \\
class\ LinkedList\langle R^a, R1\rangle\langle T\ extends\ Object\langle R1\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ head; \\
\quad int\ count; \\
\quad LinkedList(T\ v)\ \{ \\
\quad\quad super(); \\
\quad\quad this.head\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v); \\
\quad\quad this.count = 1; \\
\quad \} \\
\quad unit\ add\langle R^a_m |\ R^a_m \geqslant R^a\rangle(T\ v)\ \{ \\
\quad\quad let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ n\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v); \\
\quad\quad n.next\ =\ this.head; \\
\quad\quad this.head.prev\ =\ n; \\
\quad\quad this.head\ =\ n; \\
\quad\quad this.count = this.count + 1; \\
\quad\quad return\ (); \\
\quad \} \\
\quad T\ head\langle R^a_m\rangle(unit\ u)\ \{ \\
\quad\quad return\ this.head.val; \\
\quad \} \\
\quad LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ reverse\langle R^a_m |\ R1 \geqslant R^a_m\rangle(unit\ u)\ \{ \\
\quad\quad let\ LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ xs\ =\ new\ LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ (this.head.val); \\
\quad\quad let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ cur\ =\ this.head.next; \\
\quad\quad\quad while(not\ (cur ==\ Null))\ \{ \\
\quad\quad\quad\quad xs.add(cur.val);
\end{array}
$$

```
      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.

$$
\begin{array}{l}
class\ AnActor\langle R^a\rangle\ extends\ Object\langle R^a\rangle\ \{ \\
\quad AnActor\ ()\ \{ \\
\quad\quad super(); \\
\quad \} \\
\quad unit\ onRecv\langle R_m^a, R_0^a\rangle(\ \exists R.\ Region[R]\langle R_0^a\rangle \left\langle LinkedList\langle R, R\rangle\langle Object\langle R\rangle\rangle\right\rangle\ rin)\ \{ \\
\quad\quad let\ \left(Rin, Region[Rin]\langle R_0^a\rangle \left\langle LinkedList\langle Rin, Rin\rangle\langle Object\langle Rin\rangle\rangle\right\rangle\ rin\right) = unpack\ rin; \\
\quad\quad let\ \left(Rout, Region[Rout]\langle R_m^a\rangle\langle Object\langle Rout\rangle\rangle\ rout\right) = \\
\quad\quad\quad\quad\quad\quad unpack\ \ newReg\langle R\rangle\langle Object\langle R\rangle\rangle(); \\
\quad\quad open\ rin\ \{ \\
\quad\quad\quad let\ LinkedList\langle Rin, Rin\rangle\langle Object\langle Rin\rangle\rangle\ xs\ =\ rin.get(); \\
\quad\quad\quad letregion\langle Rs0\rangle\ \{ \\
\quad\quad\quad\quad //\ Rs0 \text{ is the new inAllocationContext} \\
\quad\quad\quad\quad //\ \text{below call to reverse type checks because } Rin \geqslant Rs0 \\
\quad\quad\quad\quad let\ LinkedList\langle Rs0, Rin\rangle\langle Object\langle Rin\rangle\rangle\ sx\ =\ xs.reverse\langle Rs0\rangle(); \\
\quad\quad\quad\quad let\ Object\langle Rin\rangle\ v\ =\ sx.head\langle Rs0\rangle(); \\
\quad\quad\quad\quad open^a\ rout\ \{ \\
\quad\quad\quad\quad\quad rout.suck(v); \\
\quad\quad\quad\quad \} \\
\quad\quad\quad \} \\
\quad\quad \}
\end{array}
$$

```
    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.

Middleweight Regions + UDFs

Thursday, July 10, 2014 3:52 PM

FJ with Regions

The Language

$\rho, p \in \ region\ names$
$cn \ \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle p^a \bar{p}\rangle\langle \bar{\tau}\rangle$ //Instantiated class type
$C \ ::= \ \text{class}\ cn\langle \rho^a \bar{\rho}\rangle\langle \overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{\tau\ x})\{\ \text{super}\ (\bar{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d \ ::= \ \tau\ mn\langle \rho^a \bar{\rho} \mid \phi\rangle\ (\overline{\tau\ x})\ \{s;\ \text{return}\ e;\}$ //Methods
$\phi \ ::= true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ //Outlives constraints on region params
$\tau_{\lhd} ::= \ \alpha \mid N$
$T ::= \ Object\langle p^a\rangle \mid Region[\rho]\langle p^a\rangle\langle \tau\rangle \mid int \mid bool \mid unit$ //Base Types
$\tau \ ::= \ \tau_{\lhd} \mid T \mid \langle \rho^a \bar{\rho} \mid \phi\rangle \bar{\tau} \xrightarrow{p^a} \tau \mid \exists \rho.\, \tau$
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid NOp \mid \ letregion\langle \rho\rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e \ ::= \ c \mid x \mid e.f \mid e\langle p^a \bar{p}\rangle(\bar{e}) \mid \lambda\langle \rho^a \bar{\rho} \mid \phi\rangle(\overline{\tau\ x}).\{s; return\ e{:}\ \tau;\} \mid \text{new}\ N(\bar{e}) \mid (N)\ e \mid e.get()$
//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. When we say $\alpha \lhd N$, instantiated class N is 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:
   $\text{class } B\langle \rho^a \bar{\rho} \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{\tau f}; k ; \bar{d} \right\}$
   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 $\bar{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\bar{\pi} \geqslant \pi^a$ 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. $Region[\rho]\langle p^a\rangle\langle\tau\rangle$ to be treated as $\exists\rho.\, Region[\rho]\langle p^a\rangle\langle\tau\rangle$, with automatic packing and unpacking when opening and closing the region, respectively. Further, τ should be well-formed under $\Delta = \{\rho\}$.
10. Although we have a variable binding like r: $Region[\rho_0]\langle p^a\rangle\langle\tau\rangle$ in the context, the type $Pair\langle\rho_0\rangle\langle\alpha@\rho_0, \beta@\rho_0\rangle \rightarrow\ \alpha@\rho_0$ is still ill-formed, as $\rho_0 \notin \Delta$. The region name cannot be used to write types, or in region param instantiations until the region is open.
11. It is alright for two transferable regions to have same name (ρ_0 above). Our static semantics prevent both from being open at the same time, as a given ρ_0 cannot be bound twice in Δ (see rules for $open$ and $open^a$).

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.

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

ctype: Types of constructor arguments of a class.

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

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.

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

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

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

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

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

Method Override: If a method overrides a method in super class, does it have same type or not? We define equality of types modulo alpha renaming of bound region variables (allocation context variables need to be renamed independent of other region variables). Observe the implication in premise, which indicates that if the method with same name does not exist in the super class, then overriding is trivially sound.

$$\frac{\begin{array}{c}CT(B)=class\ B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha\lhd N}\rangle \lhd N\ \{\overline{\tau_B\ f};\ k_B;\ d_B\} \\ mtype(m,[\overline{p}/\overline{\rho}][p^a/\rho^a][\overline{T}/\overline{\alpha}\ ]\ N)= \langle\pi_1^a\overline{\pi_1}|\ \phi_1\rangle\overline{\tau^{11}}\rightarrow\tau^{12}\Rightarrow \\ \langle\pi_1^a\overline{\pi_1}|\ \phi_1\rangle\overline{\tau^{11}}\rightarrow\tau^{12}\equiv_\alpha\langle\pi_2^a\overline{\pi_2}\mid\phi_2\rangle\overline{\tau^{21}}\rightarrow\tau^{22}\end{array}}{override\big(m,B\langle p^a\overline{p}\rangle\langle\overline{T}\rangle,\langle\pi_2^a\overline{\pi_2}\mid\phi_2\rangle\overline{\tau^{21}}\rightarrow\tau^{22}\big)}$$

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.
-

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

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

$$\frac{\Delta;\gamma\vdash\tau_1<:\tau_2 \quad \Delta;\gamma\vdash\tau_2<:\tau_3}{\Delta;\gamma\vdash\tau_1<:\tau_3}$$

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

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

$$\frac{\begin{array}{c}CT(B)= class\ B\langle\rho^{a}\overline{p}\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N\ \{\ \dots\ \}\\ p^{a}\cup\overline{p}\subseteq\Delta \qquad \gamma\vdash\overline{p}\geqslant p^{a} \qquad \Delta;\gamma\vdash\overline{\tau}\ OK\\ \Delta;\gamma\vdash[\overline{p}/\overline{\rho}][p^{a}/\rho^{a}][\overline{\tau}/\overline{\alpha}\ ]\ \overline{N}\ \mathrm{OK} \qquad \Delta;\gamma\vdash\overline{\tau}<:[\overline{p}/\overline{\rho}][p^{a}/\rho^{a}][\overline{\tau}/\overline{\alpha}\ ]\ \overline{N}\end{array}}{\Delta;\gamma\vdash B\langle p^{a}\overline{p}\rangle\langle\overline{\tau}\rangle\ OK}$$

$$\frac{p^{\mathrm{a}}\in\Delta \qquad \{\rho\};\emptyset\vdash\tau\ OK}{\Delta;\gamma\vdash Region[\rho]\langle p^{\mathrm{a}}\rangle\langle\tau\rangle\ OK}$$

$\Delta;\gamma;p^{\mathrm{a}};\Gamma\vdash e:\tau$

$$\frac{x:\tau\in\Gamma}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash x:\tau} \qquad \frac{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e:\tau' \quad f:\tau\in fields\Big(bound_{\Delta}(\tau')\Big)}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e.f:\tau} \qquad \frac{\Delta;\gamma\vdash N\ OK \quad allocRgn(N)=p \quad \gamma\vdash p^{a}\geqslant p \quad ctype(N)=\overline{\tau} \quad \Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash\overline{e:\tau_e} \quad \Delta;\gamma\vdash\overline{\tau_e}<:\overline{\tau}}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash new\ N(\bar{e}):N}$$

$$\frac{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e:\tau \quad mtype\big(m,bound_{\Delta}(\tau)\big)=\langle\rho^{a}\overline{\rho}\mid\phi\rangle\ \overline{\tau^{1}}\rightarrow\tau^{2}}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e.m:\langle\rho^{a}\overline{\rho}\mid\phi\rangle\ \overline{\tau^{1}}\xrightarrow{p^{\mathrm{a}}}\tau^{2}}$$

$$\frac{\Delta'=\Delta\cup\rho^{a}\cup\overline{\rho} \quad \gamma'=\gamma\cup\phi \quad \Delta';\gamma'\vdash\overline{\tau^{1}}\ OK \quad \Delta';\gamma'\vdash\tau^{2}\ OK \quad \Delta';\gamma';\rho^{\mathrm{a}};\Gamma,\overline{x:\tau^{1}}\vdash s\Rightarrow\Gamma' \quad \Delta';\gamma';\rho^{\mathrm{a}};\Gamma'\vdash e:\tau \quad \Delta';\gamma'\vdash\tau<:\tau^{2}}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash\lambda\langle\rho^{a}\overline{\rho}\mid\phi\rangle\big(\overline{\tau^{1}\ x}\big).\{s;return\ e:\tau;\}:\langle\rho^{a}\overline{\rho}\mid\phi\rangle\ \overline{\tau^{1}}\xrightarrow{p^{\mathrm{a}}}\tau^{2}}$$

$$\frac{\begin{array}{c}\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e_0:\langle\rho^{a}\overline{\rho}\mid\phi\rangle\ \overline{\tau^{1}}\rightarrow\tau^{2}\\ |\overline{p}|=|\overline{\rho}| \qquad S=[\overline{p}/\overline{\rho}][p^{\mathrm{a}}/\rho^{a}] \quad \Delta;\gamma\vdash\mathrm{S}(\overline{\tau^{1}})\ OK\\ \Delta;\gamma\vdash\mathrm{S}(\tau^{2})\ OK \quad \Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash\overline{e:\tau_e} \quad \Delta;\gamma\vdash\overline{\tau_e}<:\mathrm{S}(\overline{\tau^{1}}) \quad \gamma\vdash S(\phi)\end{array}}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash e_0\langle p^{\mathrm{a}}\bar{p}\rangle(\bar{e}):S(\tau^{2})}$$

$$\frac{\Delta;\gamma\vdash\tau_{\lhd}\ OK}{\Delta;\gamma;\mathrm{p}^{\mathrm{a}};\Gamma\vdash Null:\tau_{\lhd}} \qquad \frac{\rho\in\Delta \quad \Delta;\gamma;\rho^{\mathrm{a}};\Gamma\vdash e:Region[\rho]\langle\rho_k^{\mathrm{a}}\rangle\langle\tau\rangle}{\Delta;\gamma;\rho^{\mathrm{a}};\Gamma\vdash e.get():\tau}$$

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

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

$$\frac{\rho\notin\Delta \quad \Delta\cup\rho;\gamma\cup\{\Delta\geqslant\rho\},\rho,\Gamma\vdash s\Rightarrow\Gamma'}{\Delta;\gamma;\rho^{\mathrm{a}};\Gamma\vdash letregion\langle\rho\rangle\ \{\ s\ \}\Rightarrow\Gamma} \qquad \frac{\Delta;\gamma;\rho^{\mathrm{a}};\Gamma\vdash s_1\Rightarrow\Gamma' \quad \Delta;\gamma;\rho^{\mathrm{a}};\Gamma'\vdash s_2\Rightarrow\Gamma''}{\Delta;\gamma;\rho^{\mathrm{a}};\Gamma\vdash s_1;s_2\Rightarrow\Gamma''}$$

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

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

$$\frac{\Delta;\gamma;\rho^a;\Gamma\vdash e_1{:}\tau_1 \quad f{:}\tau\in fields\big(bound_\Delta(\tau_1)\big) \quad \Delta;\gamma;\rho^a;\Gamma\vdash e_2{:}\tau_2 \quad \Delta;\gamma\vdash\tau_2<:\tau}{\Delta;\gamma;\rho^a;\Gamma\vdash e.f=e\Rightarrow\Gamma}$$

$d\ OK\ in\ B$

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

$B\ OK$

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

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:

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

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.

$class\ LinkedListNode\langle R^a, R0\rangle\langle T\ extends\ Object\langle R0\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{$
  $T\ val;$
  $LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ prev;$
  $LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ next;$
  $LinkedListNode(T\ val)\ \{$
    $super();$
    $this.val\ =\ val;$
    $this.prev\ =\ Null;$
    $this.head\ =\ Null;$
  $\}$
$\}$
$class\ LinkedList\langle R^a, R1\rangle\langle T\ extends\ Object\langle R1\rangle\rangle\ extends\ Object\langle R^a\rangle\ \{$
  $LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ head;$
  $int\ count;$
  $LinkedList(T\ v)\ \{$
    $super();$
    $this.head\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v);$
    $this.count = 1;$
  $\}$
  $unit\ add\langle R_m^a |\ R_m^a \succcurlyeq R^a\rangle(T\ v)\ \{$
    $let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ n\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v);$
    $n.next\ =\ this.head;$
    $this.head.prev\ =\ n;$
    $this.head\ =\ n;$
    $this.count = this.count + 1;$
    $return\ ();$
  $\}$
  $T\ head\langle R_m^a\rangle(unit\ u)\ \{$
    $return\ this.head.val;$
  $\}$
  $LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ reverse\langle R_m^a |\ R1 \succcurlyeq R_m^a\rangle(unit\ u)\ \{$
    $let\ LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ xs\ =\ new\ LinkedList\langle R_m^a, R1\rangle\langle T\rangle\ (this.head.val);$
    $let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ cur\ =\ this.head.next;$
      $while(not\ (cur\ ==\ Null))\ \{$
        $xs.add(cur.val);$
        $cur\ =\ cur.next;$
      $\}$
    $return\ xs;$
  $\}$
$\}$

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[Rin]<LinkedList<Object>> rin) {
        let Region[Rout]<Object> rout = new Region[Rout]<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.

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

-

- Immediately needed extensions:
  - Ability to state outlives constraints explicitly over region params of class (this requirement goes hand-in-hand with above two)
  - 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.

Simplified Region Inference

Monday, July 21, 2014 11:09 AM

In this document, we describe a simplified rendition of the region type inference algorithm from previous wiki. Basically, we describe top-level elaboration process (i.e., elaboration of class header, constructor and methods) in form of a functional program, instead of using inference rules to described the elaboration. For expression and statement elaboration however, we retain the inference rule approach as it is already simple and easy to understand. Also, the section on nature of generated constraints and examples section do not require any changes, and are not reproduced in this wiki.

The Source Language

$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c ::= n \mid () \mid true \mid false \mid Null$ //Constants
$N ::= cn\langle \overline{T} \rangle$ //Instantiated class type
$C ::=$ class $cn\langle \overline{\alpha \lhd N} \rangle \lhd N \{ \overline{T\ f}; k\ ; \overline{d} \}$ //Class Definitions
$k ::= cn\ (\overline{T\ x})\{$ super $(\overline{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d ::= \tau\ mn\ (\overline{\tau\ x})\ \{s;$ return $e;\}$ //Methods
$T ::= \alpha \mid N \mid Object \mid Region\langle \tau \rangle \mid int \mid bool \mid unit$ // Types
$v ::= c \mid x \mid new\ N(\overline{v})$
$s ::= \cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid letregion\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^{a}\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e ::= c \mid x \mid e.f \mid e.mn(\overline{e}) \mid$ new $N(\overline{e}) \mid (N)\ e \mid e.get()$ //Expressions

The Target Language

$\rho, p \in region\ names$
$cn \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c ::= n \mid () \mid true \mid false \mid Null$ //Constants
$N ::= cn\langle p^{a}\overline{p} \mid \phi \rangle \langle \overline{\tau} \rangle$ //Instantiated class type
$C ::=$ class $cn\langle \rho^{a}\overline{\rho} \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \{ \overline{\tau\ f}; k\ ; \overline{d} \}$ //Class Definitions
$k ::= cn\ (\overline{\tau\ x})\{$ super $(\overline{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d ::= \tau\ mn\langle \rho^{a}\overline{\rho} \mid \phi \rangle\ (\overline{\tau\ x})\ \{s;$ return $e;\}$ //Methods
$\phi ::= true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ //Outlives constraints on region params
$\tau_{\lhd} ::= \alpha \mid N$
$\tau ::= \tau_{\lhd} \mid Object\langle p^{a} \rangle \mid Region[\rho]\langle p^{a} \rangle \langle \tau \rangle \mid int \mid bool \mid unit$ //Last 3 are unboxed
$v ::= c \mid x \mid new\ N(\overline{v})$
$s ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid letregion\langle \rho \rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^{a}\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e ::= c \mid x \mid e.f \mid e.mn\langle p^{a}\overline{p} \rangle (\overline{e}) \mid$ new $N(\overline{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle \rho \rangle \langle \tau \rangle ()$ //Expressions

Elaboration (Algorithm HM(ρ))

- The function `elaborate` describes an algorithm ($HM(\rho)$) to elaborate basic class definition to a class definition with region-annotated types (hereafter called as the elaborated definition). The algorithm generates constraints over region variables such that the elaborated definition is well-formed if and only if constraints are satisfiable. HM(ρ) uses a separate constraint solving algorithm (accessible through $normalize$ function) to solve constraints. The nature of constraints and constraint solving is described later in this wiki.
- the top-level `elaborate` function populates the class table (CT') with the elaborated definition of B. It makes use of `elaborate-header`, `elaborate-cons`, and `elaborate-methods` functions which elaborate header (signature and instance variables) of B, the constructor of B, and methods of B respectively. The three functions represent three kinds of occassions on which constraints are solved and solution is applied - after elaborating the header, after elaborating the constructor, and each time a method is elaborated.
- Rules make use of an environment Γ to map variables to their region-annotated types, an environment Δ to map type variables to their bounds, and a set Σ of region variables in scope.
- 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.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
  $bound_\Delta(T) = T$

```
fun elaborate(B) =
  let
    hdB = elaborate-header(B)
    consB = elaborate-cons(B,hdB)
    fullB = elaborate-methods(B,consB)
  in
    CT'[B ↦ fullB]
  end

fun elaborate-header(B) =
  let
    class B⟨α⊲N_s⟩ ⊲ N_s {T f; k_s ; d_s} = CT(B)
    class B⟨ρ^a ρ | T⟩⟨α⊲N⟩ ⊲ N {τ f} = header-template(B)
    C1 = type-ok(N)
    C2 = type-ok(N)
    C3 = type-ok(τ)
    C = C1 ∧ C2 ∧ C3 ∧ ρ ⩾ ρ^a
    (D,ψ_i) = normalize(C)
    N_T = ψ_i(N)
    N_T = ψ_i(N)
    τ_T = ψ_i(τ)
    ρ^a_T = ψ_i(ρ^a)
    ρ_T = (frv(N_T, N_T, τ_T)) − {ρ^a_T}
    ϕ = D − {ρ_T ⩾ ρ^a_T} (* We need not record implicit constraints*)
  in
    class B⟨ρ^a_T ρ_T | ϕ⟩⟨α⊲N_T⟩ ⊲ N_T {τ_T f}
  end

fun header-template (B) =
  let
    class B⟨α⊲N_s⟩ ⊲ N_s {T f; k_s ; d_s} = CT(B)
    X N = templateTy(N_s) (* templateTy is an auxiliary fn defined at the end *)
```

```
    ^X N = templateTy(N_s)
    X τ̄ = templateTy(T̄)
    ρ^a = allocRgn(^X N)
    ρ̄ = (frv(X N̄, ^X N, X τ̄)) − {ρ^a}
    ψ_i = [B⟨ρ^a ρ̄⟩⟨ᾱ⟩/B⟨ᾱ⟩] (* templateTy does not templatize recursive occurances of B,
          because it doesn't know how many region params are there for B. But, now we know.
          We substitute the region annotated type of B for its simple type in the class defn. *)
    N̄ = ψ_i(^X N̄)
    N = ψ_i(^X N)
    τ̄ = ψ_i(^X τ̄)
  in
    class B⟨ρ^a ρ̄ | T⟩⟨α ◁ N̄⟩ ◁ N {τ̄ f}
  end

fun elaborate-cons(B, hdB) =
  let
    class B⟨α ◁ N_s⟩ ◁ N_s {T̄ f; k_s ; d̄_s} = CT(B)
    class B⟨ρ^a_B ρ̄_B | φ_B⟩⟨α ◁ N_B⟩ ◁ N_B {τ_B f} = hdB
    τ̄_A = ctype(N_B) (* Types of super class constructor args *)
    B(T̄_x x̄){super(ū_g); this.f = u_f;} = k_s
    τ̄_a = templateTy(T̄_x)
    Ca = type-ok(τ̄_a)
    _ = CT'[B ↦ class B⟨ρ^a_B ρ̄_B | φ⟩⟨α ◁ N_B⟩ ◁ N_B {τ_B f;}] (* temporarily update CT'
          so that "this.f" gives correct type for any field f of B*)
    Γ = ·, this: B⟨ρ^a_B ρ̄_B⟩⟨ᾱ⟩, x: τ̄_x
    Σ = ρ^a_B ∪ ρ̄_B
    Δ = α ◁ N_B
    (u′_g: τ_g, Cg) = elab-expr(Σ; Δ; Γ; ρ^a_B ⊢ ū_g)
    Csub = subtype-ok(Δ ⊢ τ̄_g <: τ̄_a) (* Actual types of args to super should be subtype of
          expected types. *)
    (this.f = u′_f, _, Cf) = elab-stmt(Σ; Δ; Γ; ρ^a_B ⊢ this.f = u_f)
    C = Ca ∧ Cg ∧ Csub ∧ Cf ∧ (ρ̄_B ⩾ ρ^a_B) ∧ φ_B
    (D, ψ_i) = normalize(C)
    (N̄, N, τ̄, τ̄_x, ρ^a) = (ψ_i(N̄_B), ψ_i(N_B), ψ_i(τ̄_B), ψ_i(τ̄_a), ψ_i(ρ^a_B))
    ρ̄ = (frv(N̄, N, τ̄)) − {ρ^a}
    φ = project-constraints(D, {ρ^a, ρ̄}) − (ρ̄ ⩾ ρ^a) (* Collect residual constraints
          over region params of class B that need to be recorded explicitly as refinement *)
    (ū″_g, ū″_f) = (ψ_i(ū′_g), ψ_i(ū′_f))
    k = B(τ̄_x x̄){super(ū″_g); this.f = u″_f;}
  in
    class B⟨ρ^a ρ̄ | φ⟩⟨α ◁ N⟩ ◁ N {τ̄ f; k}
  end

fun elaborate-methods (B, consB) =
    elaborate-methods-rec (CT(B), consB)

fun elaborate-methods-rec(Bdef, consB) = case Bdef of
  class B⟨α ◁ N_s⟩ ◁ N_s {T̄ f; k_s ;} => consB (* If there are no methods, we are done *)
| class B⟨α ◁ N_s⟩ ◁ N_s {T̄ f; k_s ; d̄_s d_s} =>
  let
    fullB' = elaborate-methods-rec (class B⟨α ◁ N_s⟩ ◁ N_s {T̄ f; k_s ; d̄_s}, consB)
```

class $B\langle\rho_B^a\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};\ k;\ \overline{d_B}\}$ = fullB'

(* Our task is to elaborate method d_s*)

$T_r\ m(\overline{T_x\ x})\{s; \text{return } e;\}$ = d_s

τ_p = templateTy(T_r)

$\overline{\tau_a}$ = templateTy(T_x)

$(\pi^a, \rho_m^a, \overline{\pi})$ = (new(), new(), frv(τ_p,$\overline{\tau_a}$)) (* π^a denotes allocation context param of "m". ρ_m^a is to be used as a dummy variable to facilitate the unification of allocation contexts for recursive calls of "m" with π^a. In other words, no region polymorphic recursion *)

$d_t = \tau_p\ m\langle\rho_m^a|\rho_m^a = \pi^a\rangle(\overline{\tau_a\ x})\{\cdot\}$ (* We use this type of "m" to typecheck recursive applications.

Body of "m" is insignificant; We denote it with a hole. *)

_ = CT'[B $\mapsto$ class $B\langle\rho_B^a\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};\ k;\ \overline{d_B}d_t\}$] (* temporarily update CT' so that "this.m" gives correct type*)

$\Gamma = \cdot, \text{this}: B\langle\rho_B^a\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle, x: \overline{\tau_a}$

$\Sigma = \rho_B^a \cup \overline{\rho_B} \cup \pi^a \cup \overline{\pi}$

$\Delta = \overline{\alpha \lhd N_B}$

(s', Γ', Cs) = elab-stmt(Σ; Δ; Γ; $\pi^a \vdash s$)

$(e' : \tau_q, Ce)$ = elab-expr(Σ; Δ; Γ'; $\pi^a \vdash e$)

Csub = subtype-ok($\Delta \vdash \tau_q <: \tau_p$) (* Actual return type must be subtype of expected return type *)

C = Cs $\wedge$ Ce $\wedge$ Csub $\wedge$ $(\overline{\rho_B} \geqslant \rho_B^a) \wedge \phi_B$ (* Set of all constraints *)

(D, ψ_i) = normalize(C)

$(\overline{N}, N, \overline{\tau}, \overline{\tau_x}, \tau_r, \rho^a) = \left(\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\tau_p), \psi_i(\rho_B^a)\right)$

$\overline{\rho}$ = (frv($\overline{N}$, N, $\overline{\tau}$)) $- \{\rho^a\}$

$\Sigma_\rho = \rho^a \cup \overline{\rho}$ (* ρ^a and $\overline{\rho}$ are new region vars that replace ρ_B^a and $\overline{\rho_B}$ as region params of class B *)

(ρ_m^a, ϕ_m^a) = if $\psi_i(\pi^a) \in \Sigma_\rho$

then $(\pi^a, \pi^a = \psi_i(\pi^a))$ (* If allocation ctxt for method is required to be one of the preexisting regions, then record it explicitly as an equality constraint over allocation context parameter. *)

else $(\psi_i(\pi^a), \top)$ (* Else, simply do the substitution *)

$\overline{\rho_m}$ = (frv($\overline{\tau_x}, \tau_r$)) $- \{\rho_m^a\}$

$\Sigma_\pi = \rho_m^a \cup \overline{\rho_m}$

ϕ = project-constraints(D, Σ_ρ) - ($\overline{\rho} \geqslant \rho^a$) (* Explicit constraints over region params of B *)

ϕ_m = project-constraints(D, $\Sigma_\rho \cup \Sigma_\pi$) (* Constraints over region params of method m *)

(s'', e'') = (ψ_i(s'), ψ_i(e'))

d = $\tau_x\ m\langle\rho_m^a\overline{\rho_m}|\phi_m^a \wedge \phi_m\rangle(\overline{\tau_x\ x})\{s''; \textit{return } e''\}$

in

class $B\langle\rho^a\bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};\ k;\ \bar{d}d\}$

end

(* AUXILIARY FUNCTIONS *)

fun templateTy(T) = case T of

α|int|bool|unit => T

| Object => Object<ρ> where new(ρ)

| A($\overline{T}$) => if A $\in$ dom(CT') $\wedge$ CT'(A) = $\textit{class } A\langle\rho^a\bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N$ then

```
            then A<π^a π̄><τ̄> where new(π^a π̄) ∧ |π̄| = |ρ̄| ∧ τ̄ = templateTy(T̄)
            else T
| Region<T_root> =>let τ'= templateTy(T_root) in
                  let τ_root= [ρ/frv(τ')] τ' where new(ρ) in
                      Region[ρ]<π><τ_root> where new(π)

fun superClasses(B<π^a π̄><τ̄>) = case B of
  Object => {}
| Region[ρ] => Object<π^a>
| _ =>
    let class B⟨ρ^a ρ̄ | φ⟩⟨α ◁ N⟩ ◁ N = CT'(B)in
    let N' = [π̄/ρ̄][π^a/ρ^a] N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<π^a π̄><τ̄>)= π^a
  | _ => error()

fun project-constraints (D,S) = case D of
  true => D
| φ ∧ D'=>
    let φ = project-constraints (D',S) in
      if frv(φ)⊆ S then φ ∧ φ else φ
```

Auxiliary judgments

For `type-ok`, `subtype-ok`, `elab-expr` and `elab-stmt`, we retain the judgment notation that we had in previous wiki, for they are already simple and easy to understand. As usual, the judgments are of form $Ctxt; C \vdash Q$ denoting that Q is derivable under context $Ctxt$, given that constraint C is satisfied.

- $C \vdash \tau\ OK$ is equivalent to saying that C = `type-ok(`τ`)`.
- $\Delta; C \vdash \tau_1 <: \tau_2$ is equivalent to saying that C = sub`type-ok(`$\Delta \vdash \tau_1 <: \tau_2$`)`
- $\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$ is equivalent to saying that $(e' : \tau, C)$ = `elab-expr(`$\Sigma;\ \Delta;\ \Gamma\ ; \rho^a \vdash$e`)`
- $\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'$ is equivalent to saying that (s', Γ', C) = `elab-expr(`$\Sigma;\ \Delta;\ \Gamma\ ; \rho^a \vdash$s`)`.

$C \vdash \tau\ OK$

$$\top \vdash \alpha\ OK, int\ OK, bool\ OK \qquad \top \vdash Object\langle\rho\rangle\ OK \qquad \frac{frv(\tau)=\{\rho\}}{\top \vdash Region[\rho]\langle\rho^a\rangle\langle\tau\rangle\ OK}$$

$$\frac{\begin{array}{c} CT'(B)=class\ B\langle\rho^a\bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\ldots\} \\ C_\tau \vdash \bar{\tau}\ OK \quad \Delta=\overline{\alpha \lhd N} \quad new(\pi^a,\bar{\pi}) \quad |\bar{\pi}|=|\bar{\rho}| \quad \psi=[\bar{\pi}/\bar{\rho}][\pi^a/\rho^a] \\ \psi'=\psi\ o\ [\bar{\tau}/\bar{\alpha}] \quad C_\phi=\psi(\phi) \quad \Delta; C_\lhd \vdash \bar{\tau} <: \psi'(\bar{N}) \quad C=C_\tau \wedge C_\lhd \wedge C_\phi \wedge (\bar{\pi} \geqslant \pi^a) \end{array}}{C \vdash B\langle\pi^a\bar{\pi}\rangle\langle\bar{\tau}\rangle\ OK}$$

$\Delta;\ C \vdash \tau_1 <: \tau_2$

$$\Delta; \top \vdash \tau <: \tau \qquad \Delta;\ \pi^a = \rho^a \vdash\ B\langle\pi^a\bar{\pi}\rangle\langle\bar{\tau}\rangle\ <:\ Object\langle\pi^a\rangle$$

$$\frac{\Delta;\, C \vdash \Delta(\alpha) <: \tau_2}{\Delta;\, C \vdash \alpha <: \tau_2} \qquad \frac{A\langle \pi_2^a \overline{\pi_2}\rangle\langle \overline{\tau_2}\rangle \in SuperClasses(B\langle \pi^a \overline{\pi}\rangle\langle \overline{\tau}\rangle)}{\Delta; \left(A\langle \pi_1^a \overline{\pi_1}\rangle\langle \overline{\tau_1}\rangle = A\langle \pi_2^a \overline{\pi_2}\rangle\langle \overline{\tau_2}\rangle\right) \vdash B\langle \pi^a \overline{\pi}\rangle\langle \overline{\tau}\rangle <: A\langle \pi_1^a \overline{\pi_1}\rangle\langle \overline{\tau_1}\rangle}$$

$\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash e \hookrightarrow e' : \tau$

$$\frac{x{:}\tau \in \Gamma}{\Sigma;\Delta;\Gamma;\, \rho^a;\, \top \vdash x \hookrightarrow x{:}\tau} \qquad \frac{\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash e \hookrightarrow e' : \tau' \quad f{:}\tau \in fields(bound_\Delta(\tau'))}{\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash e.f \hookrightarrow e'.f{:}\tau}$$

$$\frac{\rho \in \Sigma \quad \Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle \pi^a \rangle \langle \tau \rangle}{\Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e.get() \hookrightarrow e'.get() : \tau}$$

$$\frac{\begin{array}{c} N = templateTy(N_s) \quad C_N \vdash N\ OK \\ \pi^a = allocRgn(N) \quad \overline{\tau_B} = ctype(N) \quad \Sigma;\, \Delta;\, \Gamma; \rho^a; C_e \vdash \bar{e} \hookrightarrow \overline{e_1} : \overline{\tau_e} \\ \Delta;\, C_\lhd \vdash \overline{\tau_e} <: \overline{\tau_B} \quad C = C_N \wedge C_e \wedge C_\lhd \wedge (\rho^a \geqslant \pi^a) \end{array}}{\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash new\ N(\bar{e}) \hookrightarrow new\ N(\overline{e_1}) : N} \qquad \frac{eraseRgn\,(\tau) \lhd Object}{\Sigma;\Delta;\Gamma;\, \rho^a; \top \vdash Null : \tau}$$

$$\frac{\begin{array}{c} \Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C_0 \vdash e_0 \hookrightarrow e_0' : \tau_0 \quad mtype\big(m, bound_\Delta(\tau_0)\big) = \langle \rho_m^a \overline{\rho_m} \mid \phi_m \rangle \overline{\tau_x} \to \tau \\ new(\overline{\pi}) \quad |\overline{\pi}| = |\overline{\rho_m}| \quad \psi = [\overline{\pi}/\overline{\rho_m}][\rho^a/\rho_m^a] \quad C_x \vdash \overline{\psi(\tau_x)}\ OK \quad C_r \vdash \psi(\tau)\ OK \\ \Sigma;\, \Delta;\, \Gamma; \rho^a; C_e \vdash \bar{e} \hookrightarrow \overline{e'} : \overline{\tau_e} \quad \Delta;\, C_\lhd \vdash \overline{\tau_e} <: \overline{\psi(\tau_x)} \quad C = C_x \wedge C_r \wedge C_e \wedge C_\lhd \wedge \psi(\phi_m) \end{array}}{\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash e_0.m(\bar{e}) \hookrightarrow e_0'.m\langle \rho^a \overline{\pi} \rangle (\overline{e'}) : \psi(\tau)}$$

$\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C \vdash s \hookrightarrow s' \dashv \Gamma'$

$$\frac{\begin{array}{c} \tau = templateTy(T) \quad C_T \vdash \tau\ OK \quad \Gamma' = \Gamma, x{:}\tau \\ \Sigma;\Delta;\Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau_e \quad \Delta;\, C_\lhd \vdash \tau_e <: \tau \quad C = C_T \wedge C_e \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash let\ T\ x = e \hookrightarrow let\ \tau\ x = e' \dashv \Gamma'}$$

$$\frac{\begin{array}{c} e_1 \in \{x, e.f\} \quad \Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C_1 \vdash e_1 \hookrightarrow e_1' : \tau_1 \\ \Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C_2 \vdash e_2 \hookrightarrow e_2' : \tau_2 \quad \Delta;\, C_\lhd \vdash \tau_2 <: \tau_1 \\ C = C_1 \wedge C_2 \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash e_1 = e_2 \hookrightarrow e_1' = e_2' \dashv \Gamma}$$

$$\frac{new(\rho) \quad \Sigma \cup \{\rho\};\, \Delta;\, \Gamma;\, \rho;\, C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge \Sigma \geqslant \rho \wedge C_s}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash letregion\ \{\ s\ \} \hookrightarrow letregion\langle \rho \rangle\ \{\ s'\ \} \dashv \Gamma} \qquad \frac{\Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C_1 \vdash s_1 \hookrightarrow s_1' \dashv \Gamma_1 \quad \Sigma;\, \Delta;\, \Gamma_1;\, \rho^a;\, C_2 \vdash s_2 \hookrightarrow s_2' \dashv \Gamma' \quad C = C_1 \wedge C_2}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash s_1; s_2 \hookrightarrow s_1'; s_2' \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \\ \Sigma \cup \{\rho\};\, \Delta;\, \Gamma;\, \rho^a;\, C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge C_R \wedge C_s \end{array}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash open\ e\ \{\ s\ \} \hookrightarrow open\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \\ \Sigma \cup \{\rho\};\, \Delta;\, \Gamma;\, \rho;\, C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge C_R \wedge C_s \end{array}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash open^a\ e\ \{\ s\ \} \hookrightarrow open^a\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\begin{array}{c} \Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \quad \rho \in \Sigma \\ \Sigma;\, \Delta;\, \Gamma;\, \rho^a;\, C_2 \vdash e_2 \hookrightarrow e_2' : \tau' \quad C = C_R \wedge C_2 \wedge (\tau' = \tau) \end{array}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash e.set(e_2) \hookrightarrow e'.set(e_2') \dashv \Gamma} \qquad \frac{\Sigma;\, \Delta;\, \Gamma;\, \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle \pi^a \rangle \langle \tau \rangle \quad \rho \notin \Sigma \quad a \in \{transfer, giveUp\}}{\Sigma;\Delta;\Gamma;\, \rho^a; C \vdash e.a() \hookrightarrow e'.a() \dashv \Gamma}$$

Region Inference + Exists

Wednesday, August 13, 2014 12:42 PM

In this wiki, we extend type inference to existential types of transferable regions. The basic rule that guides inference is that the type of the transferable region is by default an existential with bound region name (eg: $\exists\rho_0.\,\mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\tau_0\rangle$). That is, when we generate a region type template for C# type $\mathrm{Region}\langle\dots\rangle$, we always generate $\exists\rho_0.\,\mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\dots\rangle$, where ρ_0 and π_0^a are both new. The default elaboration of transferable region type to an existential type simplfies inference, while not having a significant adverse impact on the expressivity. For example:

1. LinkedList<Region<...>> is by default elaborated to LinkedList<π_0^a,π_1^a><$\exists\rho\,.\,\mathrm{Region}[\rho\,]\langle\pi_1^a\rangle\langle\dots\rangle$>. Here, the elaboration has rightly decided to assign existential type to region handlers stored linked list.
2. *void foo(Region<…> r)* is appropriately elaborated to *void foo<…>(*$\exists\rho\,.\,Region[\rho\,]\langle\dots\rangle\langle\dots\rangle$ *r)* rather than elaborating it to *void foo<…,ρ>(*$Region[\rho\,]\langle\pi_1^a\rangle\langle\dots\rangle$ *r)*. The latter requires *ρ* to be live (open) when *foo* is called, an assumption which we don't want functions like *foo* to make. We insist that functions explictly open any transferable region handlers they receive, rather than assuming that they are already open and that they satisfy certain outlives relationships.
3. *Class B {Region<…> r; …}* is elaborated to *Class B*$\langle\dots\rangle$ *{*$\exists\rho\,.\,Region[\rho\,]\langle\dots\rangle\langle\dots\rangle$ *r; …}* rather than *Class B*$\langle\dots,\rho\rangle$ *{*$Region[\rho\,]\langle\dots\rangle\langle\dots\rangle$ *r; …}*. The latter requires *ρ* to be live (open) before any object of type *B* is created. Consequently, it disallows the common coding idiom where the constructor of a class creates and assigns a new transferable region to its instance variable.

until a method-local variable is initialized to refer the region handler, at which point the existential is *unpacked* and the name of transferable region is materialized. We constrain our source language such that only local variables of unpacked transferable region type (eg: $\mathrm{Region}[\rho_1]\langle\pi_1^a\rangle\langle\tau_1\rangle$, where ρ_1 is in scope) are allowed in *open* and *openAlloc* statements. Consequently, a transferable region *has* to be referred by a local variable before opening, and the corresponding variable declaration is elaborated by our algorithm to *unpack* statement for transferable region handler.

The Source Language

$cn \in Class\ Names\ (A, B, C \dots)$
$mn \in Method\ Names\ (m, n, \dots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
c ::= $n \mid () \mid true \mid false \mid Null$ //Constants
N ::= $cn\langle\overline{T}\rangle$ //Instantiated class type
C ::= class $cn\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{T\ f}; k\ ; \overline{d}\}$ //Class Definitions
k ::= $cn\ (\overline{T\ x})\{\ \mathrm{super}\ (\overline{v});\ \overline{\mathrm{this}.f = v;}\}$ //Constructors
d ::= $T\ mn\ (\overline{T\ x})\ \{s;\ \mathrm{return}\ e;\}$ //Methods
T ::= $\alpha \mid N \mid Object \mid Region\langle T\rangle \mid int \mid bool \mid unit$ // Types
v ::= $c \mid x \mid new\ N(\overline{v})$
s ::= $\cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid letregion\ \{\ s\ \} \mid open\ x\ \{\ s\ \}$
$\mid open^a\ x\ \{\ s\ \} \mid s; s \mid x.set(e) \mid x.transfer() \mid x.giveUp()$
e ::= $c \mid x \mid e.f \mid e.mn(\overline{e}) \mid \mathrm{new}\ N(\overline{e}) \mid (N)\ e \mid x.get()$ //Expressions

The Target Language

$\rho, \pi, p \in\ region\ names$
$cn \in Class\ Names\ (A, B, C \dots)$
$mn \in Method\ Names\ (m, n, \dots)$

$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
c ::= n | () | $true$ | $false$ | $Null$ //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
ϕ ::= $true \mid \rho \geqslant \rho \mid \rho = \rho \mid \phi \wedge \phi$ // constraints on region params
$\tau_{\lhd}$ ::= $\alpha \mid N$ //Types that admit subtyping (subclassing)
τ ::= $\tau_{\lhd} \mid Object\langle p^a \rangle \mid Region[\rho]\langle p^a \rangle \langle \tau \rangle \mid int \mid bool \mid unit \mid \exists \rho.\tau$
v ::= $c \mid x \mid new\ N(\bar{v})$
s ::= $\cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid letregion\langle \rho \rangle \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \mid e.suck(e)$
$\mid let\ (\rho, \tau\ x) = unpack\ e$
e ::= $c \mid x \mid e.f \mid e.mn\langle p^a \bar{p} \rangle (\bar{e})$ | new $N(\bar{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle \rho \rangle \langle \tau \rangle ()$
$\mid pack[\rho, e]\ as\ \exists \rho.\tau$ //Expressions

Elaboration (Algorithm HM(ρ))

- The function `elaborate` describes an algorithm ($HM(\rho)$) to elaborate basic class definition to a class definition with region-annotated types (hereafter called as the elaborated definition). The algorithm generates constraints over region variables such that the elaborated definition is well-formed if and only if constraints are satisfiable. HM(ρ) uses a separate constraint solving algorithm (accessible through $normalize$ function) to solve constraints. The nature of constraints and constraint solving is described later in this wiki.
- the top-level `elaborate` function populates the class table (CT') with the elaborated definition of B. It makes use of `elaborate-header`, `elaborate-cons`, and `elaborate-methods` functions which elaborate header (signature and instance variables) of B, the constructor of B, and methods of B respectively. The three functions represent three kinds of occassions on which constraints are solved and solution is applied - after elaborating the header, after elaborating the constructor, and each time a method is elaborated.
- Rules make use of an environment Γ to map variables to their region-annotated types, an environment Δ to map type variables to their bounds, and a set Σ of region variables in scope.
- 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.
  $bound_\Delta(\alpha) = \Delta(\alpha)$
  $bound_\Delta(N) = N$
  $bound_\Delta(T) = T$

```
fun elaborate(B) =
  let
    hdB = elaborate-header(B)
    consB = elaborate-cons(B,hdB)
    fullB = elaborate-methods(B,consB)
  in
    CT'[B ↦ fullB]
  end

fun elaborate-header(B) =
  let
```

$\text{class } B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s; \overline{d_s}\}$ = CT(B)

$\text{class } B\langle\rho^a\bar{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}\}$ = header-template(B)

C1 = type-ok($\bar{N}$)

C2 = type-ok(N)

C3 = type-ok($\bar{\tau}$)

C = C1 $\wedge$ C2 $\wedge$ C3 $\wedge$ $\bar{\rho} \geqslant \rho^a$

(D, ψ_i) = normalize(C)

$\overline{N_T} = \psi_i(\overline{N})$

$N_T = \psi_i(N)$

$\overline{\tau_T} = \psi_i(\overline{\tau})$

$\rho^a_T = \psi_i(\rho^a)$

$\overline{\rho_T}$ = (frv($\overline{N_T}, N_T, \overline{\tau_T}$)) $- \{\rho^a_T\}$

$\phi = D - \{\overline{\rho_T} \geqslant \rho^a_T\}$ (* We need not record implicit constraints*)

in

$\text{class } B\langle\rho^a_T\overline{\rho_T} \mid \phi\rangle\langle\overline{\alpha \lhd N_T}\rangle \lhd N_T\ \{\overline{\tau_T\ f}\}$

end

fun header-template (B) =

let

$\text{class } B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s; \overline{d_s}\}$ = CT(B)

$\overline{{}^X N} =$ templateTy($\overline{N_s}$) (* templateTy is an auxiliary fn defined at the end *)

${}^X N =$ templateTy(N_s)

$\overline{{}^X\tau} =$ templateTy($\bar{T}$)

$\rho^a =$ allocRgn($^X N$)

$\bar{\rho}=$ (frv($\overline{{}^X N}, {}^X N, \overline{{}^X\tau}$)) $-\{\rho^a\}$

$\psi_i = [B\langle\rho^a\bar{\rho}\rangle\langle\bar{\alpha}\rangle / B\langle\bar{\alpha}\rangle]$ (* templateTy does not templatize recursive occurances of B, because it doesn't know how many region params are there for B. But, now we know. We substitute the region annotated type of B for its simple type in the class defn. *)

$\overline{N} = \psi_i(\overline{{}^X N})$

$N = \psi_i({}^X N)$

$\bar{\tau} = \psi_i(\overline{{}^X\tau})$

in

$\text{class } B\langle\rho^a\bar{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}\}$

end

fun elaborate-cons(B, hdB) =

let

$\text{class } B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s; \overline{d_s}\}$ = CT(B)

$\text{class } B\langle\rho^a_B\overline{\rho_B} \mid \phi_B\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f}\}$ = hdB

$\overline{\tau_A}$= ctype(N_B) (* Types of super class constructor args *)

$B(\overline{T_x\ x})\{\text{super}(\overline{v_g}); \overline{\text{this.f} = v_f;}\}$ = k_s

$\overline{\tau_a} =$ templateTy($\overline{T_x}$)

Ca = type-ok($\overline{\tau_a}$)

_ = CT'[B $\mapsto$ $\text{class } B\langle\rho^a_B\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};\}$] (* temporarily update CT' so that "this.f" gives correct type for any field f of B*)

Γ = $\cdot$, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\bar{\alpha}\rangle$, x: $\bar{\tau}_x$

$\Sigma = \rho^a_B \cup \overline{\rho_B}$

$\Delta = \overline{\alpha \lhd N_B}$

($\overline{v'_g}: \overline{\tau_g}$, Cg) = elab-expr($\Sigma$; Δ; Γ; $\rho^a_B \vdash \overline{v_g}$)

Csub = subtype-ok($\Delta \vdash \overline{\tau_g} <: \overline{\tau_a}$) (* Actual types of args to super should be subtype of expected types. *)

$(\overline{\text{this.f} = v'_f}, \_,$ Cf) = elab-stmt(Σ; Δ; Γ; $\rho^a_B \vdash \overline{\text{this.f} = v_f}$)

```
    C = Ca ∧ Cg ∧ Csub ∧ Cf ∧ (ρ̄_B ⋗ ρ^a_B) ∧ φ_B
    (D, ψ_i) = normalize(C)
    (N̄, N, τ̄, τ̄_x, ρ^a) = (ψ_i(N̄_B), ψ_i(N_B), ψ_i(τ̄_B), ψ_i(τ̄_a), ψ_i(ρ^a_B))
    ρ̄ = (frv(N̄, N, τ̄)) − {ρ^a}
    φ = project-constraints(D, {ρ^a, ρ̄}) − (ρ̄ ⋗ ρ^a)  (* Collect residual constraints
                over region params of class B that need to be recorded explicitly as refinement *)
    (ῡ''_g, ῡ''_f) = (ψ_i(ῡ'_g), ψ_i(ῡ'_f))
    k = B(τ̄_x x̄){super(ῡ''_g); this.f = ῡ''_f;}
  in
    class B⟨ρ^a ρ̄ | φ⟩⟨ᾱ ◁ N̄⟩ ◁ N {τ̄ f; k}
  end

fun elaborate-methods (B, consB) =
    elaborate-methods-rec (CT(B), consB)

fun elaborate-methods-rec(Bdef, consB) = case Bdef of
  class B⟨ᾱ ◁ N̄_s⟩ ◁ N_s {T̄ f; k_s ;} => consB  (* If there are no methods, we are done *)
| class B⟨ᾱ ◁ N̄_s⟩ ◁ N_s {T̄ f; k_s ; d̄_s d_s} =>
  let
    fullB' = elaborate-methods-rec (class B⟨ᾱ ◁ N̄_s⟩ ◁ N_s {T̄ f; k_s ; d̄_s}, consB)
    class B⟨ρ^a_B ρ̄_B | φ⟩⟨ᾱ ◁ N̄_B⟩ ◁ N_B {τ̄_B f; k; d̄_B} = fullB'
    (* Our task is to elaborate method d_s*)
    T_r m(T̄_x x̄){s; return e;} = d_s
    τ_p = templateTy(T_r)
    τ̄_a = templateTy(T̄_x)
    (π^a, ρ^a_m, π̄) = (new(), new(), frv(τ_p, τ̄_a)) (* π^a denotes allocation context param of "m".
                ρ^a_m is to be used as a dummy variable to facilitate the unification of allocation
                contexts for recursive calls of "m" with π^a. In other words, no region polymorphic
                recursion *)
    d_t = τ_p m⟨ρ^a_m | ρ^a_m = π^a⟩(τ̄_a x̄){·} (* We use this type of "m" to typecheck recursive
    applications.
                                        Body of "m" is insignificant; We denote it with a hole. *)
    _ = CT'[B ↦ class B⟨ρ^a_B ρ̄_B | φ⟩⟨ᾱ ◁ N̄_B⟩ ◁ N_B {τ̄_B f; k; d̄_B d_t}]  (* temporarily
                update CT' so that "this.m" gives correct type*)
    Γ = ·, this: B⟨ρ^a_B ρ̄_B⟩⟨ᾱ⟩, x: τ̄_a
    Σ = ρ^a_B ∪ ρ̄_B ∪ π^a ∪ π̄
    Δ = ᾱ ◁ N̄_B
    s0 = redec-rgn-handler(T̄_x x̄) (* Re-declare any arguments that are transferable
    region
                handlers. Elaboration ensures that region handlers are unpacked *)
    (s', Γ', Cs) = elab-stmt(Σ; Δ; Γ; π^a ⊢ s0;s)
    (e' : τ_q, Ce) = elab-expr(Σ; Δ; Γ'; π^a ⊢ e)
    Csub = subtype-ok(Δ ⊢ τ_q <: τ_p)  (* Actual return type must be subtype of expected
                                        return type *)
    C = Cs ∧ Ce ∧ Csub ∧ (ρ̄_B ⋗ ρ^a_B) ∧ φ_B  (* Set of all constraints *)
    (D, ψ_i) = normalize(C)
    (N̄, N, τ̄, τ̄_x, τ_r, ρ^a) = (ψ_i(N̄_B), ψ_i(N_B), ψ_i(τ̄_B), ψ_i(τ̄_a), ψ_i(τ_p), ψ_i(ρ^a_B))
    ρ̄ = (frv(N̄, N, τ̄)) − {ρ^a}
    Σ_ρ = ρ^a ∪ ρ̄  (* ρ^a and ρ̄ are new region vars that replace ρ^a_B and ρ̄_B as region params of class
    B *)
    (ρ^a_m, φ^a_m) = if ψ_i(π^a) ∈ Σ_ρ
                then (π^a, π^a = ψ_i(π^a))  (* If allocation ctxt for method is required to be one of
```

```
                the
                                                    preexisting regions, then record it explicitly as an
                                                  equality
                                                    constraint over allocation context parameter. *)
                  else (ψ_i(π^a), ⊤)  (* Else, simply do the substitution *)
    ρ_m^a ∪ ρ_m^a = (frv(τ_x, τ_r)) − {ρ_m^a}
    Σ_π = ρ_m^a ∪ ρ_m^a
    ϕ = project-constraints(D, Σ_ρ) - (ρ̄ ≽ ρ^a) (* Explicit constraints over region
    params of B *)
    ϕ_m = project-constraints(D, Σ_ρ ∪ Σ_π) (* Constraints over region params of method
    m *)
    (s'', e'') = (ψ_i(s'), ψ_i(e'))
    d = τ_x m⟨ρ_m^a ρ_m^a | ϕ_m^a ∧ ϕ_m⟩(τ_x x){s''; return e''}
  in
    class B⟨ρ^a ρ̄ | ϕ⟩⟨α ◁ N⟩ ◁ N {τ f; k; d}
  end

(* AUXILIARY FUNCTIONS *)

fun redec-rgn-handler(T x) =
  foldr (T x, nop, fn (T x, s) => case T of
    Region⟨T'⟩ => (let T x = x); s
  | _ => s)

fun templateTy(T) = case T of
  α|int|bool|unit => T
| Object => Object<ρ> where new(ρ)
| A(T̄) => if A ∈ dom(CT') ∧ CT'(A) = class A⟨ρ^a ρ̄ | ϕ⟩⟨α ◁ N⟩ ◁ N then
            then A<π^a π̄><τ̄> where new(π^a π̄) ∧ |π̄| = |ρ̄| ∧ τ̄ = templateTy(T̄)
            else T
| Region<T_root> =>let τ' = templateTy(T_root) in
                 let τ_root = [ρ/frv(τ')] τ' where new(ρ) in
                     ∃ρ. Region[ρ]<π><τ_root> where new(π)

fun superClasses(B<π^a π̄><τ̄>) = case B of
  Object => {}
| Region[ρ] => Object<π^a>
| _ =>
    let class B⟨ρ^a ρ̄ | ϕ⟩⟨α ◁ N⟩ ◁ N = CT'(B)in
    let N' = [π̄/ρ̄][π^a/ρ^a] N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<π^a π̄><τ̄>)= π^a
  | _ => error()

fun project-constraints (D,S) = case D of
  true => D
| φ ∧ D'=>
    let ϕ = project-constraints (D',S) in
      if frv(φ)⊆ S then φ ∧ ϕ else ϕ

fun elab-stmt(Σ; Δ; Γ ; ρ^a ⊢s) = case s of
```

```
  (let Region⟨T⟩ x = e) =>
  let
    ∃ρ0. Region[ρ0]⟨π0^a⟩⟨τ0⟩ = templateTy(Region⟨T⟩)
    Ct = type-ok(Region[ρ0]⟨π0^a⟩⟨τ0⟩)
    (e' : τe, Ce) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e)
    (τ2, s') = case τe of
        ∃ρ1. τ1 => ([ρ0/ρ1] τ1, let (ρ0, Region[ρ0]⟨π0^a⟩⟨τ0⟩ x) = unpack e')
      | _ => (τe, let Region[ρ0]⟨π0^a⟩⟨τ0⟩ x = e')
    Csub = subtype-ok(Δ ⊢ τ2 <: Region[ρ0]⟨π0^a⟩⟨τ0⟩ )
    C = Ct ∧ Ce ∧ Csub
    Γ' = Γ, x: Region[ρ0]⟨π0^a⟩⟨τ0⟩
  in
    (s', Γ', C)
  end
| (let T x = e) =>
  let
    τ = templateTy(T)
    Ct = type-ok(τ)
    (e' : τe, Ce) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e)
    Csub = subtype-ok(Δ ⊢ τe <: τ)
    C = Ct ∧ Ce ∧ Csub
    Γ' = Γ, x: τ
  in
    (let τ x = e', Γ', C)
  end
| (e1 = e2) where e1 ∈ {x, e.f} =>
  let
    (e1' : τ1, C1) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e1)
    (e2' : τ2, C2) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e2)
    (Csub, s') = case (τ1, τ2) of
        (∃ρ1. Region[ρ1]⟨π1^a⟩⟨τ1⟩, Region[ρ2]⟨π2^a⟩⟨τ2⟩) =>
          (subtype-ok(Δ ⊢ τ2 <: [ρ1/ρ0]τ1),
           e1 = pack[ρ2, e2] as ∃ρ1. Region[ρ1]⟨π1^a⟩⟨τ1⟩)
      | _ => (subtype-ok(Δ ⊢ τ2 <: τ1), e1' = e2')
    C = C1 ∧ C2 ∧ Csub
  in
    (s', Γ, C)
  end
| (open x { s }) =>
| (open^a x { s }) =>

fun elab-subtype-ok (Δ ⊢ e : τ1 <: τ2) = case (τ1, τ2) of
  (Region[ρ1]⟨π1^a⟩⟨τ1⟩, ∃ρ2. τ2') =>
  let
    Csub = subtype-ok(Δ ⊢ τ1 <: [ρ1/ρ2]τ2)
    e' = pack[ρ1, e] as ∃ρ2. τ2'
  in
    (e', Csub)
  end
| _ => (e, subtype-ok(Δ ⊢ τ1 <: τ2))

fun elab-expr (Σ; Δ; Γ ; ρ^a ⊢e) = case e of
  (x.get()) =>
```

```
  let
    (e′: Region[ρ0]⟨π0^a⟩⟨τ0⟩, Cr) = elab-expr (Σ; Δ; Γ; ρ^a ⊢ x)
    C = Cr ∧ (ρ0 ∈ Σ)
  in
    (e′.get(): τ0, C)
  end
| (new N(ē)) =>
  let
    N = templateTy (Ns)
    Cn = type-ok (N)
    π^a = allocRgn(N)
    τ̄B = ctype(N)
    (ē1 : τ̄e, Ce) = elab-expr (Σ; Δ; Γ; ρ^a ⊢ ē)
    (ē2, Csub) = elab-subtype-ok(Δ ⊢ ē1 : τ̄e <: τ̄B)
    C = Cn ∧ Ce ∧ Csub ∧ (ρ^a ≽ π^a)
  in
    (new N(ē2) : N, C)
  end
| (e0.m(ē)) =>
  let
    (e0′: τ0, C0) = elab-expr(Σ; Δ; Γ; ρ^a; C0 ⊢ e0)
    ⟨ρm^a ρ̄m | φm⟩τ̄x → τ = mtype(m, bound_Δ(τ0))
    π̄ = new(length(ρ̄m))
    ψ = [π̄/ρ̄m][ρ^a/ρm^a]
    Cx = type-ok(ψ(τ̄x))
    Cr = type-ok (ψ(τ))
    (ē1 : τ̄e, Ce) = elab-expr (Σ; Δ; Γ; ρ^a ⊢ ē)
    (ē′, Csub) = elab-subtype-ok(Δ ⊢ ē1 : τ̄e <: ψ(τ̄x))
    C = Cx ∧ Cr ∧ Ce ∧ Csub ∧ ψ(φm)
  in
    (e0′.m(ē′) : ψ(τ), C)
  end
|
```

```
(* 1. Uniqueness constraints on transferable regions - only in first
branch of case expression. That is, only when new region name is
introduced. The second branch allows aliasing of region handlers,
while unifying the region names for aliases.
   2. Handling uninitialized variables. Needed for loops.
   3. Subtyping only has to consider two packed types or two unpacked
types. It is equality for former and alpha-equivalence for later.*)
```

Auxiliary judgments

For `type-ok`, `subtype-ok`, `elab-expr` and `elab-stmt`, we retain the judgment notation that we had in previous wiki, for they are already simple and easy to understand. As usual, the judgments are of form $Ctxt; C \vdash Q$ denoting that Q is derivable under context $Ctxt$, given that constraint C is satisfied.

- $C \vdash \tau\ OK$ is equivalent to saying that C = `type-ok(`τ`)`.
- $\Delta; C \vdash \tau_1 <: \tau_2$ is equivalent to saying that C = sub`type-ok(`$\Delta \vdash \tau_1 <: \tau_2$`)`
- $\Sigma; \Delta; \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$ is equivalent to saying that $(e' : \tau, C)$ = `elab-expr(`$\Sigma; \Delta; \Gamma; \rho^a \vdash$`e)`

- $\Sigma; \Delta; \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'$ is equivalent to saying that $(s', \Gamma', C) =$ `elab-expr(`$\Sigma; \Delta; \Gamma; \rho^a \vdash s$`)`.

$C \vdash \tau\ OK$

$$\top \vdash \alpha\ OK, int\ OK, bool\ OK \qquad \top \vdash Object\langle\rho\rangle\ OK \qquad \frac{frv(\tau)=\{\rho\}}{\top \vdash Region[\rho]\langle\rho^a\rangle\langle\tau\rangle\ OK}$$

$$\frac{\begin{array}{c} CT'(B)=class\ B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\dots\} \\ C_\tau \vdash \overline{\tau}\ OK \quad \Delta=\overline{\alpha \lhd N} \quad new(\pi^a,\overline{\pi}) \quad |\overline{\pi}|=|\overline{\rho}| \quad \psi=[\overline{\pi}/\overline{\rho}][\pi^a/\rho^a] \\ \psi'=\psi\ o\ [\overline{\tau}/\overline{\alpha}] \quad C_\phi=\psi(\phi) \quad \Delta; C_\lhd \vdash \overline{\tau} <: \psi'(\overline{N}) \quad C=C_\tau \wedge C_\lhd \wedge C_\phi \wedge (\overline{\pi} \geqslant \pi^a) \end{array}}{C \vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ OK}$$

$\Delta;\ C \vdash \tau_1 <: \tau_2$

$$\Delta; \top \vdash \tau <: \tau \qquad \Delta;\ \pi^a = \rho^a \vdash\ B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ <:\ Object\langle\pi^a\rangle$$

$$\frac{\Delta; C \vdash \Delta(\alpha) <: \tau_2}{\Delta; C \vdash \alpha <: \tau_2} \qquad \frac{A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle \in SuperClasses(B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle)}{\Delta; \left(A\langle\pi_1^a\overline{\pi_1}\rangle\langle\overline{\tau_1}\rangle = A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle\right) \vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle <: A\langle\pi_1^a\overline{\pi_1}\rangle\langle\overline{\tau_1}\rangle}$$

$\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$

$$\frac{x:\tau \in \Gamma}{\Sigma;\Delta;\Gamma;\ \rho^a; \top \vdash x \hookrightarrow x:\tau} \qquad \frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e \hookrightarrow e' : \tau' \quad f:\tau \in fields(bound_\Delta(\tau'))}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e.f \hookrightarrow e'.f:\tau}$$

$$\frac{\rho \in \Sigma \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^a; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle\pi^a\rangle\langle\tau\rangle}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a; C_R \vdash e.get() \hookrightarrow e'.get() : \tau}$$

$$\frac{\begin{array}{c} N=templateTy(N_s) \quad C_N \vdash N\ OK \\ \pi^a=allocRgn(N) \quad \overline{\tau_B}=ctype(N) \quad \Sigma;\ \Delta;\ \Gamma; \rho^a; C_e \vdash \bar{e} \hookrightarrow \overline{e_1} : \overline{\tau_e} \\ \Delta;\ C_\lhd \vdash \overline{\tau_e} <: \overline{\tau_B} \quad C=C_N \wedge C_e \wedge C_\lhd \wedge (\rho^a \geqslant \pi^a) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash new\ N(\bar{e}) \hookrightarrow new\ N(\overline{e_1}) : N} \qquad \frac{eraseRgn\ (\tau) \lhd Object}{\Sigma;\Delta;\Gamma;\ \rho^a; \top \vdash Null:\tau}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_0 \vdash e_0 \hookrightarrow e_0' : \tau_0 \quad mtype\left(m, bound_\Delta(\tau_0)\right) = \langle\rho_m^a\overline{\rho_m} \mid \phi_m\rangle\overline{\tau_x} \to \tau \\ new(\overline{\pi}) \quad |\overline{\pi}|=|\overline{\rho_m}| \quad \psi=[\overline{\pi}/\overline{\rho_m}][\rho^a/\rho_m^a] \quad C_x \vdash \overline{\psi(\tau_x)}\ OK \quad C_r \vdash \psi(\tau)\ OK \\ \Sigma;\ \Delta;\ \Gamma; \rho^a; C_e \vdash \bar{e} \hookrightarrow \overline{e'} : \overline{\tau_e} \quad \Delta;\ C_\lhd \vdash \overline{\tau_e} <: \overline{\psi(\tau_x)} \quad C=C_x \wedge C_r \wedge C_e \wedge C_\lhd \wedge \psi(\phi_m) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C \vdash e_0.m(\bar{e}) \hookrightarrow e_0'.m\langle\rho^a\overline{\pi}\rangle(\overline{e'}) : \psi(\tau)}$$

$\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'$

$$\frac{\begin{array}{c} \tau=templateTy(T) \quad C_T \vdash \tau\ OK \quad \Gamma'=\Gamma,x:\tau \\ \Sigma;\Delta;\Gamma;\rho^a;C \vdash e \hookrightarrow e' : \tau_e \quad \Delta;\ C_\lhd \vdash \tau_e <: \tau \quad C=C_T \wedge C_e \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash let\ T\ x=e \hookrightarrow let\ \tau\ x=e' \dashv \Gamma'}$$

$$\frac{\begin{array}{c} e_1 \in \{x, e.f\} \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_1 \vdash e_1 \hookrightarrow e_1' : \tau_1 \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_2 \vdash e_2 \hookrightarrow e_2' : \tau_2 \quad \Delta;\ C_\lhd \vdash \tau_2 <: \tau_1 \\ C=C_1 \wedge C_2 \wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C \vdash e_1=e_2 \hookrightarrow e_1'=e_2' \dashv \Gamma}$$

$$\frac{new(\rho) \quad \Sigma\cup\{\rho\};\ \Delta;\ \Gamma;\ \rho;\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge \Sigma \not\ni \rho \wedge C_s}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash letregion\ \{\ s\ \} \hookrightarrow letregion\langle\rho\rangle\ \{\ s'\ \} \dashv \Gamma}$$

$$\frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_1 \vdash s_1 \hookrightarrow s_1' \dashv \Gamma_1 \quad \Sigma;\ \Delta;\ \Gamma_1;\ \rho^a;\ C_2 \vdash s_2 \hookrightarrow s_2' \dashv \Gamma' \quad C = C_1 \wedge C_2}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash s_1; s_2 \hookrightarrow s_1'; s_2' \dashv \Gamma'}$$

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}}; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \\ \Sigma\cup\{\rho\};\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}};\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge C_R \wedge C_s\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash open\ e\ \{\ s\ \} \hookrightarrow open\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}}; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \\ \Sigma\cup\{\rho\};\ \Delta;\ \Gamma;\ \rho;\ C_s \vdash s \hookrightarrow s' \dashv \Gamma' \quad C = \rho \notin \Sigma \wedge C_R \wedge C_s\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash open^a\ e\ \{\ s\ \} \hookrightarrow open^a\ e'\ \{\ s'\ \} \dashv \Gamma'}$$

$$\frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}}; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \rho \in \Sigma \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}};\ C_2 \vdash e_2 \hookrightarrow e_2' : \tau' \quad C = C_R \wedge C_2 \wedge (\tau' = \tau)}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash e.set(e_2) \hookrightarrow e'.set\left(e_2'\right) \dashv \Gamma}$$

$$\frac{\Sigma;\ \Delta;\ \Gamma;\ \rho^{\mathrm{a}}; C_R \vdash e \hookrightarrow e' : Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \rho \notin \Sigma \quad a \in \{transfer, giveUp\}}{\Sigma;\Delta;\Gamma;\ \rho^a; C \vdash e.a() \hookrightarrow e'.a() \dashv \Gamma}$$