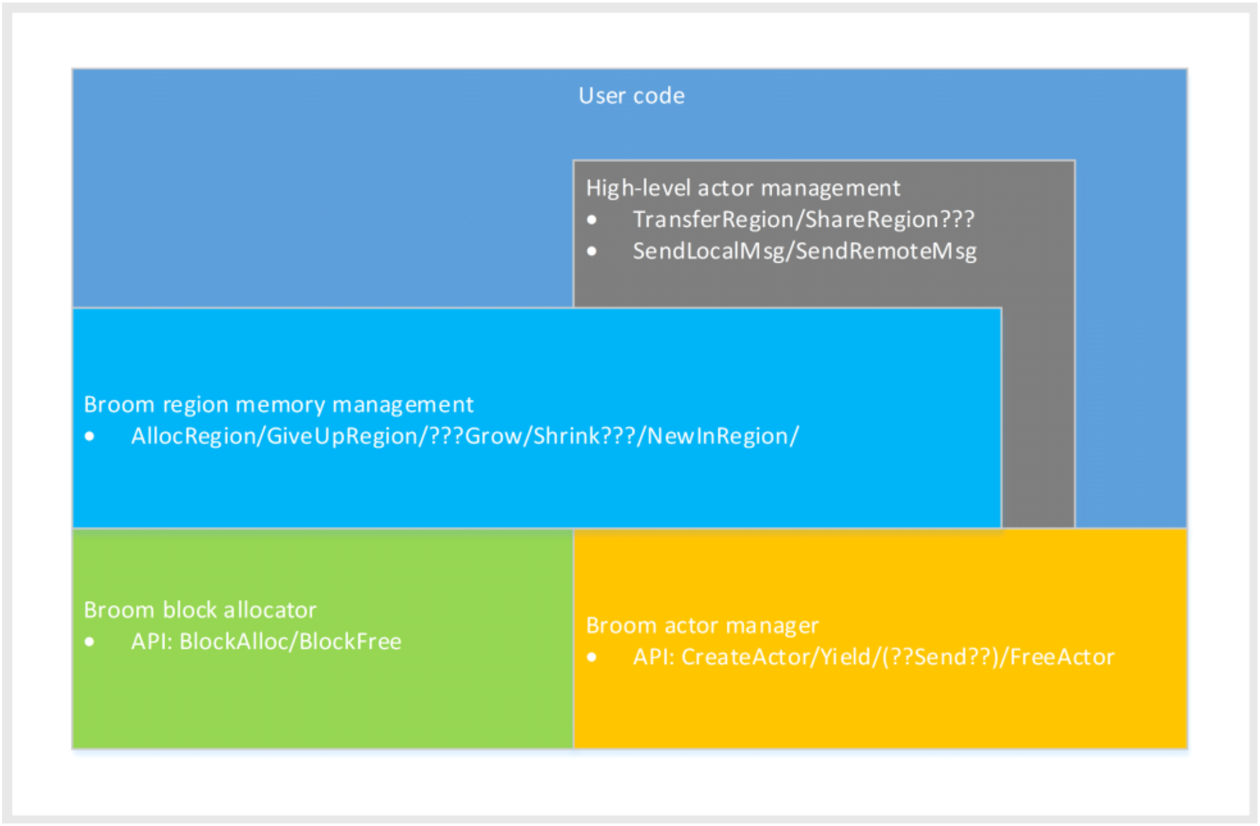
Architecture

Tuesday, November 19, 2013 2:03 PM

Data parallel languages/runtimes (DryadLINQ, Naiad, CScale)	
C# + Region annotations	
Bytecode + region instructions newobj(Region, Type) -> Object, free?	
CLR + region-based allocator Broom API + Alloc(Region, size) -> Object, Free(Object)	
Broom low-level allocator	Broom low-level actor management

Architecture - Drawing


User code
High-level actor management
• TransferRegion/ShareRegion???
• SendLocalMsg/SendRemoteMsg
Broom region memory management
• AllocRegion/GiveUpRegion/???Grow/Shrink???/NewInRegion/
Broom block allocator
• API: BlockAlloc/BlockFree
Broom actor manager
• API: CreateActor/Yield/(??Send??)/FreeActor

Broom API

Monday, October 21, 2013 9:26 PM

The purpose of this document is to define an API for a runtime that can be used to execute distributed event-based applications on LAN-connected cluster. The API is low-level, and design choices are intended to support two main goals: high performance both for scheduling and network transfer; and strict resource isolation.

Cluster-wide concepts

There are cluster-wide principals called users which have quotas, and various levels of permissions. The interface to manage users and possibly groups will be worked out later. There are cluster-wide principals called resources (roughly corresponding to the computers in the cluster) which have associated metadata that reveals hardware characteristics, some kind of network coordinates, probably some kind of status revealing instantaneous load, etc. The interface to manage resources will be worked out later.

Application model

There are cluster-wide principals called applications. Each application operates on behalf of a user. Applications have quotas and permissions, originally inherited from their user. The executable code of an application is a fixed collection of classes. Each class can be instantiated one or more times and an instantiation is called an actor. An actor is bound to a resource at the time it is instantiated. Actors are single-threaded, meaning that at most one thread of control can be active in a given actor at any time.

The executable code in a class is structured into private methods which can only be called by other methods in the same actor. Each class may include distinguished methods called actions which may be internal or public. An action has no return value and takes a single argument that is referred to as a message. An action may be queued by sending a message to the specified action on a named actor; a queued action will eventually be called, with some constraints on ordering between actions described below. Internal actions may only be queued by other actors in the same application, and public actions may be queued by any other actor or external client with the right permissions. Actors have opaque identifiers that can be serialized and used as handles when queuing their actions, and the identifier encodes which resource the actor resides on.

An action either runs to completion or yields. When an action yields, it returns the actor's thread of control to the resource where it is running. An action a of actor o may yield in one of three states:

1. **Non-interruptible**: the resource must return to action a at the point where it has yielded before executing any other code from o.
2. **Non-reentrant**: the resource must return to action a at the point where it has yielded before executing any other code from action a in o, but it may execute code from any other action in o in the meantime.
3. **Reentrant**: the resource may execute code from any other actions in o, including a, before returning control to the yield point.

An action has a static contract supplied by the compiler. Its exact contents are to be determined, but it might for example guarantee a maximum number of cycles that the action will take before yielding. (This might imply that contracts are resource-specific). There is a default contract that gives no guarantees.

When an action yields it specifies a yield record which includes the type of yield (which of the three above states it yields in) and optionally an updated contract with new guarantees for the fragment that will execute when the remainder of the action is rescheduled. An action might for example have an initial contract that guarantees a quick yield, and use the time before that yield to estimate the true contract for the action in a data-dependent way.

Actions have simple integer priorities and higher-priority runnable actions within an application will be more likely to be executed at a given resource. There is no hard guarantee, to cut down on synchronization at the resource. An action of actor o can obtain and modify the data structure containing the pending actions for o, and can thus cancel pending actions or modify their priorities. *Should we make it possible for an action to obtain and modify the data structure containing the priorities for its entire parent application, not just its parent actor, to allow flexible application-level scheduling? This would need to have relaxed semantics otherwise per-core scheduling would have to take locks everywhere. Probably leave it out, unless there's a compelling scenario.*

Local resource runtime

The runtime on a single resource is responsible for hosting actors, scheduling the execution of their actions, supplying them with memory, and sending messages between actors, both locally and via a network interconnect. As far as possible the runtime is assumed to be a pure, cooperatively scheduled operating system with no pre-emption, using polling to manage its network connections. It has no access to local storage or devices other than the network, and it has a dedicated pool of physical memory and processor cores. For sanity, we assume that the computer where the runtime is executing may optionally host a traditional operating system with its own private core(s) and memory, and its own network interface, and that this traditional OS can be used for debugging and monitoring. Even without a monitoring OS, probably the runtime will at least have an occasional timer interrupt that checks for application deadlocks so that buggy user code can be killed.

Memory is allocated in blocks called regions. Each region has a type that determines how the memory in the region will be shared:

- **Private**: the region will only ever be accessed by the actor that created it.
- **Transferable**: the region will only ever be accessed by one actor at a time, but it may be transferred between actors within the same application, or to and from the network interface.
- **Shareable**: the region may be accessed by more than one actor in the same application and/or the network, simultaneously.

Each region is private to an application within a resource. When a message is sent between applications that run on the same resource it is copied. This simplifies the semantics of quotas. We assume that in this day and age code is small enough relative to main memory that sharing it between applications is not necessary. We do not support the case where multiple applications need to share large (readonly) data: they will have to make multiple copies of the same data.

The runtime does not guarantee memory protection, so applications should be written defensively using a language that guarantees memory safety (in the absence of compiler and runtime bugs). *The only purpose of the region types is to let the runtime make NUMA and cache-aware allocation decisions, e.g. that there is some portion of physical memory that is preferentially handed out to private regions on a particular core. In principle there could be a "debug" mode that enforces region types via memory protection, but we expect the performance overhead of doing that to be prohibitive since page tables would need to be reprogrammed every time an action from a different actor is executed. If we can't think of measurable performance improvements that come from region types, maybe we should get rid of them.*

The runtime will try to use the largest appropriate page size (4 KB, 2MB or 1GB assuming we are using x64 processors) when allocating a region, so actors' heap managers should prefer a small number of large allocations where possible. *It is an open question whether we can do a better job than Windows of avoiding long-term fragmentation of physical memory that destroys the availability of large pages. It seems plausible that no more than 1GB of physical memory should need to be set aside for small allocations as long as actors don't needlessly allocate a lot of small but long-lived regions. It would be great if we could say something intelligent about this.*

Shared regions are reference counted, and are reclaimed only when all actors have given up references to them. Memory freed by an application is guaranteed to be zeroed before being given to another application, however it may be returned without being zeroed in a subsequent allocation by the application that freed it. *We might save a little bit of time over traditional OSes by not zeroing memory if we believe the runtime is memory safe.*

Whenever an action makes a runtime API call it also implicitly yields, so all runtime calls include a yield record. Some calls also includes a timeout parameter t. The runtime will wait at least t ticks before returning an error code if the request (e.g. a memory allocation) cannot be satisfied immediately due to resource scarcity. It is an error to request an infinite timeout with a non-interruptible yield type, but otherwise the runtime will not attempt to detect actors that have deadlocked themselves, since in general an actor could always receive a message from another actor that would free resources and allow a blocked action to continue.

Local message sends (between actors belonging to the same application, running on the same resource) are handled with a different API to remote message sends. This allows local messages to contain unserialized actors that hold pointers to other actors. Local sends don't have an effect on application memory quotas since regions are always shared or transferred, not copied, during a local send. Local sends between a pair of actors are guaranteed to be delivered in FIFO order: since actors are single-threaded this seems like a straightforward requirement for the implementation (it does not add any synchronization) and means that o1 can share a region with, or transfer a region to, o2 using a message before subsequent messages that refer to the data in the region, without worrying about potential consequences of out of order delivery. A local send either fails immediately, or is guaranteed to be delivered unless the resource fails (actually, the guarantee is that the sender and receiver share fate; all actors belonging to the application at the resource could be killed). Before deleting an actor, the actor must be closed, after which no local sends to it will succeed. Once all queued messages have been delivered to a closed actor it is informed, and can release its resources. This is intended to make actor migration feasible in a sane way.

Remote message sends (between applications or resources) always transfer buffers of bytes that will be copied by the system, so the receiver must have enough memory quota to receive the message before it is sent. Remote sends have no ordering guarantees, and applications will use a standard library that adds sequence numbers and re-orders deliveries if ordered delivery is required. *Depending on the type of network hardware we expect to use, we need to decide whether the system promises to detect and/or correct bit corruption, or the application should checksum where required.* An application may dedicate some of its memory quota to use as receive buffers, in which case we assume the network application has an efficient high-throughput mechanism for implementing back-pressure that allows sends to take place with low overhead as long as there is plenty of free space in the receive buffer. If the application does not set aside enough (or any) quota for dedicated receive buffers then the sender must negotiate with the receiver before each send and the receiver may allocate enough space to receive the message from the receiver's quota only on a per-send basis. The intention is to support high-throughput applications, but also to leave open the possibility of running very large numbers of `dormant' applications on a resource without dedicating receive buffer space to them.

Runtime API

The API below is written according to some interim design decisions that can be revisited after discussion. In particular, it uses return values rather than exceptions to report error conditions, and it handles memory ownership via convention rather than using the type system. These are `C-like' conventions and may be changed if we end up with a more `managed' feel to the API. Code running in an action has access to the following API:

- AllocateRegion(ShareType s, UInt64 size, YieldRecord y, Interval timeout) → Region | E_NOMEMORY: try to allocate a region of the specified size and type. If the application can't find the required amount of memory within its quota after timeout seconds, the call returns E_NOMEMORY. The returned Region contains a field indicating whether or not the memory has been zeroed.

- FreeRegion(Region r, YieldRecord y) → void: give up the actor's reference to the region. The actor pledges not to dereference memory in the region after this call.

- TransferRegion(Region r, Actor o, Action<Region> a, Yield Record y)→ SUCCESS | E_INVALIDREGION | E_INVALIDACTOR: give up the actor's reference to r and transfer it to o, telling o about the transfer by queueing action a. The caller pledges not to dereference memory in the region after this call, though it may still have pointers in scope, e.g. that it will use in a subsequent call to SendLocalMessage. E_INVALIDREGION is returned if r is private. E_INVALIDACTOR is returned if o is nonexistent or closed.

- ShareRegion(Region r, Actor o, Action<Region> a, YieldRecord y) → SUCCESS | E_INVALIDREGION | E_INVALIDACTOR: give o a reference to r, telling o about the reference by queueing action a. E_INVALIDREGION is returned if r is not shareable. E_INVALIDACTOR is returned if o is nonexistent or closed.

- SendLocalMessage(Actor o, Action<MessageType> a, MessageType m, YieldRecord y) → SUCCESS | E_INVALIDACTOR: send a typed message to another action in the same application, running on the same resource. The sender promises that the memory holding the message is contained in, and contain pointers to, only regions that the remote actor has access to on delivery. The message is guaranteed to be delivered (as long as the resource doesn't fail) and the call returns SUCCESS unless o is nonexistent or closed, in which case the call returns E_INVALIDACTOR.

- SendRemoteMessage(Application a, Actor o, Action<SerializedMessage> a, Region r, UInt64 offset, UInt64 length, Action<SendStatus> onSend, Action<DeliveryStatus> onRecv, YieldRecord y, Interval timeout) → Success | E_INVALIDREGION: send a serialized message to another action. The message is serialized in a subregion of r. The call returns E_INVALIDREGION if r is private. When the send completes (is copied to the network or another local application) or fails, onSend is queued, and is passed the status code of the send, and r. The send status is one of SUCCESS, meaning the message was successfully sent to a remote application, E_INVALIDAPPLICATION meaning the application could not be contacted on the remote resource specified in o, or E_NOMEMORY meaning the receiving application has insuffcient quota to receive the message. The send may keep trying until timeout seconds have elapsed. The caller (say, c) must have access to r when it calls SendRemoteMessage. If r is transferable the caller promises not to dereference any memory in r before onSend is called. After onSend is called, c owns r again. If r is shareable and c freed r before onSend was called, then c acquires a new reference to r when onSend is called. This mechanism allows the caller to recycle network buffers effciently, and also perform appropriate actions if a send fails without storing additional copies of the message. *We assume that the application makes sure it can figure out whether or not the remote actor exists in some out of band manner; if it doesn't exist the send fails silently. If the application wants to send to a remote actor that might migrate, it should use a proxy actor at the destination that is guaranteed not to migrate. If the remote resource fails after delivery, the application will hear about it using some other mechanism and must do recovery to figure out whether the message got delivered or not. Instead of calling OnSend we could have this call "block" until send is complete, since we already implemented yielding. Not sure which is better.*

- Close(YieldRecord y, Action<void> onClose) → void: mark the caller as closed, and accepting no more message deliveries. When all pending messages have been delivered, queue action onClose. After onClose exits the caller will be garbage-collected and relinquish all its region references. An actor can only close itself.

- AllocateMessageReceiveQuota(UInt64 size)→ SUCCESS | E_NOMEMORY: set the application's dedicated message receive buffers to have access to size bytes at the local resource. Initially (and if size is set to 0) the application has no dedicated receive buffers and communication will be slower because senders will have to negotiate with the receiver to allocate receive buffers before every send.

- CreateActor(Class c, Resource r, YieldRecord y, Action<CreateStatus> onCreate, Interval timeout) → void: create a new actor of class c on resource r. onCreate is queued eventually with status SUCCESS if the actor was created, E_NOMEMORY if there was insuffcient quota at r, or E_FAILURE if the remote application stopped communicating and it is unknown whether or not the message was delivered. *As for remote sends, maybe this should be a blocking call.*

- QueryUsage(YieldRecord y) → ResourceUsage: get some kind of information, to be determined, about the resource usage of the application on the local resource. For example, memory allocated, CPU usage, etc.

- QueryScheduler(YieldRecord y) → ActionQueue: get some kind of information, to be determined, about queued actions. This is intended to allow user-level schedulers to introspect on queued actions and change their priorities.

- QueryCluster(YieldRecord y) → ClusterInfo: get some kind of information, to be determined, about the resources that exist in the cluster. This is intended to be used for actors that want to make decisions about scheduling, and may include load information for resources as well as their network coordinates.

Discussion

We need to decide what to do about failure detection. If a resource fails, perhaps it should be possible for applications to register actions so they hear about the failure. How would this work? Would exactly one action get queued at one resource on the cluster, or would every resource have an action that got queued, and they negotiate in an application-specific manner to decide what to do? In general, should the runtime fabric include things like reliable state machines, or let applications implement their own?

We can assume applications will need some kind of storage. I imagine storage controllers look to applications like other applications that can have remote messages sent to them, but in fact are running on some other OS that has things like disk drivers. We might want to supply log storage as well as block storage as \standard" libraries.

Applications will also need to communicate with the outside world. As for storage, I imagine gateways that act like remote actors but really run a general-purpose OS with the ability to talk standard network protocols to other computers.

We need to decide what to do if an actor's method throws an exception. The easy choice is to terminate the whole application at the resource, since then no invariants are broken.

We may want some form of constructor for actors. For example, each time an actor is created on a resource it might be helpful to queue, as its first action, an initialization message that includes information about the resource it is running on, the names of applications that supply local storage, etc. This doesn't work very well with the current actor creation model, since the remote resource that makes the CreateActor call may not have enough information to call the constructor. Also because remote sends are not ordered, the creator couldn't be sure it queued the actor's first action unless it waited for the onRecv callback before telling anyone else about the newly-created actor, which would seem to add unnecessary delays on creation.

We may want the runtime to implement extensible performance counters and/or logging, similar to ETW. It would be good to do this in a way that keeps resource allocation transparent.

What language is this supposed to be written in? Obviously not standard .NET, since it doesn't have the fine control over memory and scheduling that we want. Vanilla C++ is probably out of the question because we want yielding to have syntactic sugar the way it does in .NET 4.5, or programmers will go insane. Two possible choices are some hacked-up .NET with a different CLR, e.g. based on Midori or some other internal project; or some descendant of THC with yielding built in to a C-like language. Jumping on a bandwagon with more support rather than less seems appealing.

Semantics of regions

Monday, October 21, 2013 9:22 PM

Memory allocation in Broom is based on regions. There are three types of types of regions, private, transferable and shared. The type of a region determines owner and the lifetime of the region. For efficiency, Broom uses reference counting at the granularity of regions for garbage collection. Broom does not track references to objects within regions. A drawback of this approach is that it does not automatically guarantee memory safety. The application using Broom must observe some conventions if memory safety is desired. This documents describes the semantics and conventions associated with each type of regions.

Private regions

Private regions are meant for use by a single object. Typically, an object allocates a private region for temporary storage during the course of processing a single action, or for storage that lasts for the lifetime of the object (such as hash tables used in computing grouping and aggregation, or auxiliary state used in computing joins).

The runtime does not use reference counting for private regions. Since the object that creates a private region owns the region, the object is expected to free the region before it is closed. Regions that are not freed before the object is closed are assumed to be unreachable. Once a private region is freed (or is out of scope), the runtime assumes that there are no live objects in the region, and that the region can be collected.

As described above, memory safety may be violated due to references into private regions. One way to ensure memory safety is to *disallow references into a private region* (even from other private regions). Note that objects within a private region may reference other objects in the same region, or objects in other transferable or shared regions. For example, an object may use a private regions as a scratchpad while holding references to a transferable/shared region.

We can use a type system to statically enforce this constraint. Enforcing this constraint statically has the added advantage that it makes garbage collection within the region easier. Alternatively, we can use remembered sets to ensure that accesses to objects in freed regions throw exceptions.

Transferable regions

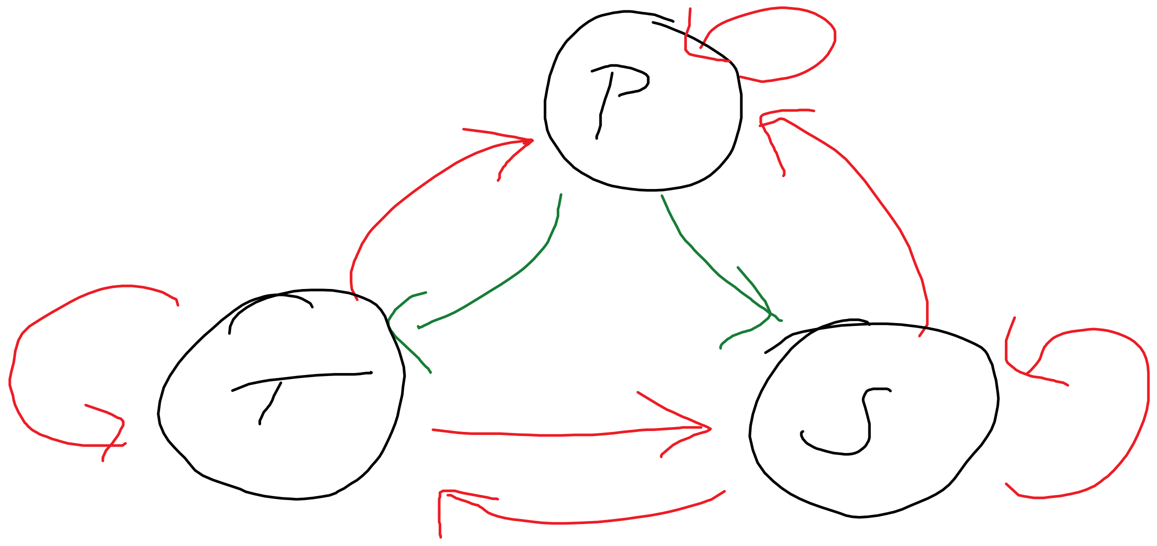
Transferable regions hold state that is eventually transferred to other objects (via messages). A transferrable region has a single owner at any point in time. Just like private regions, the runtime assumes that a transferable region can be collected and reused as soon as it is freed or when the object that owns the region is closed.

It is desirable to permit references from private regions into a transferable region. It is the obligation of the application to ensure that none of the objects in a transferable regions are accessed once the ownership of the region has been transferred to another object. *Again, we can use a type system to enforce this constraint, or detect such accesses at runtime and throw exceptions.*

Shared regions

Shared regions allow sharing of blocks of data between multiple objects without having to explicitly copy these blocks. At any point in time, multiple objects may hold references to a shared region. The typical use of shared regions is to store immutable data. *This is a constraint we could consider imposing statically. It might make sense for 'user' objects in shared regions to be immutable, but a middleware runtime like Naiad will no doubt need some mutable shared state for efficiency.* Shared regions are reference counted. The runtime can garbage collect a shared region as soon as all objects holding a reference to the region have freed the region or have been closed. It is the responsibility of the application to ensure that there are no dangling references into a shared region.

One way of ensuring memory safety is to disallow references from transferable or shared regions into a shared region i.e. only private regions can reference a shared region. This invariant ensures that a shared region can be safely collected once its reference count is zero.



Region Type System

Thursday, November 28, 2013 2:26 PM

Note: This is not a formal type-system, but an informal sketch of a proposal.

Region Classification

The current Broom API classifies regions into *Private*, *Transferable*, and *Shared* regions based on their ownership. We may further classify regions as follows, based on their lifetime/scope. Recall that an *action* refers to a single method/operation executed by an actor to process a single input message.

Single-action-scope region: These regions have a lifetime that is restricted to a single action. These are used for temporary data used solely when processing a single input message. These are necessarily private regions.

Multiple-action-scope region: These are regions whose lifetime exceeds the lifetime of the action that creates it. These can be private, or transferable, or shared. Examples of private multiple-action-scope regions are those that store auxiliary state that is accumulated over multiple input messages (and, hopefully, released at some point).

Immortal regions: These are regions that live forever (the word "region" may be inappropriate for these). We can classify this further into two kinds: *Immutable* and *Mutable*. An immutable immortal region is reasonable (essentially static data). However, a mutable immortal region will most likely need a GC. Both of these kinds can be either private or shared. We can also distinguish *actor-scoped (private) regions* whose lifetime is the same as that of an actor from truly immortal (shared) regions.

Memory Safety

Our goal is a region type system that is expressive enough to permit the types of regions mentioned above (without affecting performance negatively) and guarantees that ***whenever a region is freed, none of the remaining regions or stack-variables point to objects inside the freed region***.

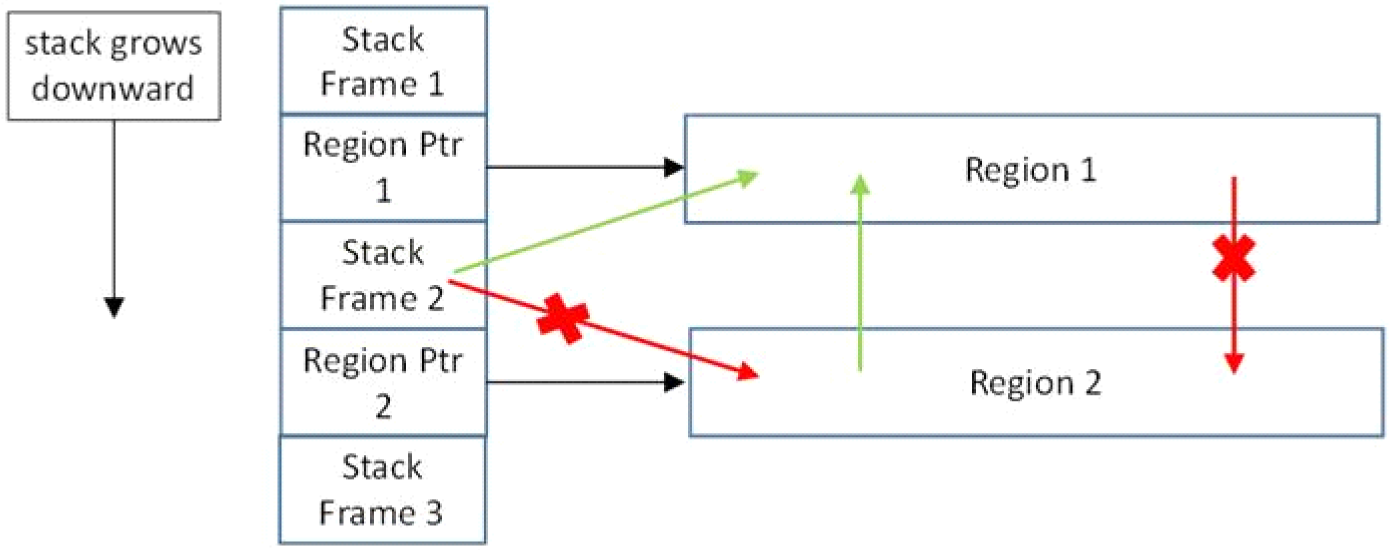
Statically-Scoped Regions

Most original region type systems utilize *statically-scoped regions*, illustrated below. These systems guarantee memory safety by ensuring that the program satisfies the following two constraints:

1. If an object in region R1 points to an object in region R2, then region R2's lifetime must contain the lifetime of region R1.
2. If a stack-allocated variable V points to an object in region R, then region R's lifetime must contain the lifetime of variable V.

Thus, at run-time, we have a stack that looks as follows: Green arrows show pointers that are allowed. Red arrows show pointers that are not allowed. The stack grows downward, so stack-frames and regions higher up have a longer lifetime.

Note: In the following diagrams, region-pointers are shown in the stack-frame for illustrative purpose. The implementation may or may not do exactly this.



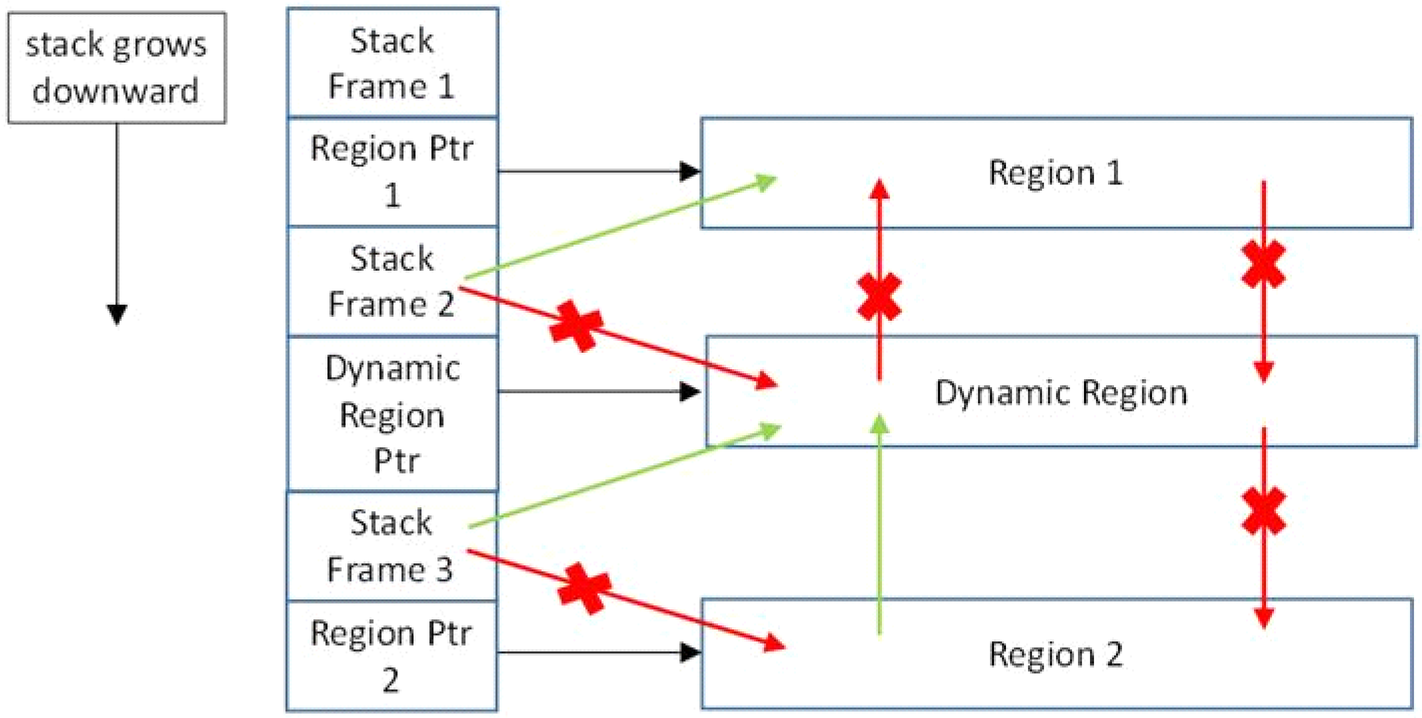
Adding Dynamically-Scoped Regions

Statically-scoped regions are inadequate to realize multiple-action-scope regions (e.g., transferable regions). We add dynamically-scoped regions to the above, permitting both types of regions to co-exist. We extend the constraints previously described (for statically-scoped regions) as follows to ensure memory safety in the presence of dynamically-scoped regions.

- Stack variable restrictions Note that a statically-scoped region has a very simple life-cycle (identical to a stack-frame): the region is created (allocated), and a pointer to the region is conceptually pushed onto the stack. Eventually we pop the pointer off the stack, freeing the region at the same time. This ensures that no stack-variable points to objects inside a freed region. For dynamic regions, we extend the protocol as follows:

  (1) A dynamic region can be allocated, and a pointer to it pushed onto the stack.
  (2) When the stack shrinks back and the dynamic region pointer goes off the stack, we can choose to free the region (as in the static case), or we can choose to temporarily "seal" (or "close") the region. A sealed dynamic region is temporarily inaccessible to the program.
  (3) We can "unseal" or "open" a sealed region, pushing a pointer to the region on to the stack. At this point, the dynamic region becomes accessible again. E.g., at this point we can start creating stack variables that point to objects in the dynamic region.
  (4) Thus, a dynamic region may be unsealed and sealed any number of times as above, before it is finally freed.

- Inter-Region pointer restrictions In addition, we impose a further restriction on pointers: a pointer from an object in a dynamic region cannot point to any object in another region. Thus, the picture now looks as follows:



Conceptually, sealing/unsealing a dynamic region can be thought of as decrementing/incrementing a reference-count (or, more simply, clearing/setting a boolean flag) that is used to ensure that an unsealed region is not freed. This allows us to work safely with pointers to objects inside an unsealed region (e.g., such pointers may be stored in stack-variables or in other static regions that are nested below).

Dynamic Regions With a Root Object

When a region has been sealed, we are guaranteed that there are no pointers to objects inside the region. However, we obviously need pointers to the region itself (so that we can use it in the future). We can ensure safety of such pointers by using either a linear type system (requiring that an actor store a unique pointer to a dynamic region) or by reference-counting.

Because of the above constraints, it is convenient to think of a dynamic-region as an object itself, with fields and methods. Or, equivalently, we can think of a dynamic region as having a root-object (of some given type) that can be accessed using the pointer to the region. This is necessary to start working with objects already in a dynamic region after we unseal the region, as illustrated below.

Example: In the sequel, the type Transferable<Foo> denotes a transferable region with a root object of type Foo, which can be retrieved as r.Value (where r is of type Transferable<Foo>, provided the region had been opened).

```
var region = AllocateRegion <Transferable<Foo>> ();
…
Foo foo = region.Value; // Invalid when region is sealed.
open region in {
    Foo foo = region.Value; // valid here
    free(region); // invalid here.
}
Foo foo = region.Value; // Invalid here
free(region); // Valid here
```

Mapping Private/Transferable/Shared Regions to Static/Dynamic Regions

Transferable and shared regions need to be realized using dynamic regions. However, private regions can sometimes be realized using static regions. Specifically, we can realize single-action scoped region using static regions and we can realize multiple-action scoped region using dynamic regions.

Note: An *action* refers to a single method/operation executed by an actor to process a single input message. In our earlier formulation, the execution of an action could contain *yields*, leading to a potentially interleaved execution of multiple actions. In the recent call, it was suggested that actions should terminate when a yield occurs, with subsequent processing done by other actions. We have not fully thought through the interactions between this proposal and the use of statically-scoped regions. Hopefully it is not a major issue.

Type system

Monday, February 3, 2014 10:34 PM

Domains

x,y,z variables

f field names

m method names

c class names

a actor names

r region variables

Syntax

$t \coloneqq int \mid bool$ // primitive types

$h \coloneqq string \mid c$ // heap allocated types

$r \coloneqq r \mid act$ // regions

$q \coloneqq t \mid r\ h$ // qualified types

$s \coloneqq x = y \mid x.f = y \mid x = y.f \mid x = y.m(z_1, z_2, \ldots, z_k) \mid return\ x \mid r\ \{\ s\ \} \mid x = new\ t \mid x = new\ h\ @\ r \mid send(m, x)$ // statements

$field \coloneqq q\ f$ // field declarations

$meth \coloneqq q\ m(q\ x_1, \ldots, q\ x_n)\ \{\ s; return\ x\ \}$ // method declarations

$T \coloneqq actor\ a\ \{\ field*\ \ meth*\ \} \mid class\ c\ \{\ field*meth*\ \}$ // type declarations

Type rules

$\Gamma \coloneqq [\ \Gamma, x : r]$

$act \in R$

$$\frac{\square}{\Gamma, R\ \vdash x\ \dashv \Gamma, R}$$

$$\frac{r\ \in R}{\Gamma, R\ \vdash x = new\ h\ @\ r\ \dashv [\ \Gamma, x{:}\, r\ ]}$$

$$\frac{r_2 \leq_l\ r_1}{[\ \Gamma, x{:}\, r_1, y{:}\, r_2\ ] \vdash x.f\ = y \dashv [\ \Gamma, x{:}\, r_1, y{:}\, r_2\ ]}$$

Examples

Wednesday, November 27, 2013 1:40 PM

The following are some examples to illustrate simple scenarios and how regions can be used in these scenarios. They are written in a made-up language (C# with few extensions/modifications) … hopefully the meaning is clear enough. Note that these are intended to be fully-typed (with respect to regions). In practice, if we have a type-inference algorithm, much of region-related code could be auto-generated.

Almost all calls that eventually translate to the BROOM API calls are written as "await BroomCallAsync(…)".

MapActor

Monday, December 2, 2013   1:36 PM

```
// A actor that applies a map function f to every input (of type TSource) and generates an output of type Tresult
actor MapActor<TSource,TResult> : IActor<TSource,TResult>
{
        // A reference to the next actor in the pipeline
        private IActor<TResult> _target;

        MapActor(IActor<TResult> target)
        {
                _target = target;
        }

        // We use the notation RegionType<T> as a shorthand for a region of type RegionType with a single object of type T
        action async onRecv(Transferable<TSource> messageRegion)
        {
                var targetRegion = await AllocateRegionAsync<Transferable<TResult>>();
                open messageRegion in
                open targetRegion in
                {
                        targetRegion.Value = Mapper(messageRegion.Value, targetRegion);
                }
                await SendLocalMessageAsync(_target, targetRegion);
                await FreeRegionAsync(messageRegion);
        }

        private TResult Mapper(TSource value, Transferable<TResult> targetRegion)
        {
                // Apply some function on value
                ...
                // Allocate result in the targetRegion
                return  new[targetRegion] TResult(args);
        }
}
```

BufferActor

Monday, December 2, 2013 1:36 PM

```
// An actor which buffers objects before sending them to the next stage in the pipeline
actor BufferActor<TSource> : IActor<TSource,List<TSource>>
{
        private Transferable<List<TSource>> _bufferRegion;
        private IActor<List<TSource>> _target;

        BufferActor(IActor<List<TSource>> target)
        {
                _bufferRegion = await AllocateRegionAsync<Transferable<List<TSource>>>();
                _target = target;
        }

        action async onRecv(Transferable<TSource> messageRegion)
        {
                var size;
                open _bufferRegion in
                {
                        var list = _bufferRegion.Value;
                        // Add a copy of the object to the buffer since pointers between transferable regions are not permitted
                        list.Add(messageRegion.Value.Copy(_bufferRegion));
                        // Storing a reference to an object in the region in a stack variable declared outside the scope is not permitted
                        // However the following is permitted since a value is being copied
                        size = list.Count();
                }

                if (size > 10)
                {
                        await SendLocalMessageAsync(_bufferRegion, _target);
                        _bufferRegion = await AllocateRegionAsync<Transferable<<List<Tsource>>>();
                }
                await FreeRegionAsync(messageRegion);
        }
}
```

GroupMinActor

Monday, December 2, 2013 1:36 PM

```
// An actor which groups inputs and computes an aggregation (min) for each group
actor GroupMinActor<TSource,TKey> : IActor<TSource, Tuple<TKey, int>>
{
    // All fields of the actor must be private
    // Fields are allocated in an (implicit) actor-scoped region
    private Dictionary<TKey, int> _mins;
    private IActor<Tuple<TKey, int>> _target;
    private Func<TSource, TKey> _keySelector;
    private Func<TSource, int> _valSelector;

    GroupMinActor(IActor<Tuple<TKey, int>> target, Func<TSource, TKey> keySelector, Func<TSource, int> valSelector)
    {
        // The keyword thisRegion is a reference to an actor scoped region for the current actor
        _mins = new[thisRegion] Dictionary<TKey, int>();
        _target = target;
        _keySelector = keySelector;
        _valSelector = valSelector;
    }

    action async OnRecv(Transferable<TSource> messageRegion)
    {
        open messageRegion in
        {
            // Should allocate the key in a new action-scoped region
            var key = _keySelector(messageRegion.Value);
            var val = _valSelector(messageRegion.Value);
            bool change = true;
            if (_mins.ContainsKey(key))
            {
                if (val < _mins[key])
                    _mins[key] = val;
                else
                    change = false;
            }
            else
                _mins[key.Copy(thisRegion)] = val;

            if (change)
            {
                var targetRegion = await AllocateRegionAsync<Transferable<Tuple<TKey, int>>();
                targetRegion.Value = new Tuple<TKey, int>(targetRegion, key, val);
                await SendLocalMessageAsync(targetRegion, _target);
            }
        }
        await FreeRegionAsync(messageRegion);
    }
}
```

DistinctCount

Monday, December 2, 2013 1:37 PM

```
class DistinctCountActor<S,T> : Actor<T>
{
        private Dictionary<T, Dictionary<S,int>> _counts;

        action async OnRecv(S msg, T time)
        {
                if (!_counts.ContainsKey(time)) {
                        _counts[time] = new Dictionary<S, int>();
                        this.NotifyAt(time);
                }
                if (!_counts[time].ContainsKey(msg)) {
                        _counts[time][msg] = 0;
                        this.SendBy(output1, msg, time);
                }
                _counts[time][msg]++;
        }

        action async void OnNotify(T time)
        {
                foreach (var pair in _counts[time])
                        this.SendBy(output2, pair, time);
                _counts.Remove(time);
        }
}
```

```
// This implementation assumes we want to the entire dictionary of message counts in one message
class DistinctCountActor<S,T> : Actor<T>
{
        // All fields of the actor must be private
        // Fields are allocated in an (implicit) actor-scoped immortal region
        private Dictionary<T, exclusive Transferable<Dictionary<S,int>>> _counts;

        action async OnRecv(Transferable<Pair<S, T>> messageRegion)
        {
                open messageRegion in
                {
                        bool send = false;
                        var time = messageRegion.Value.First;
                        var msg = messageRegion.Value.Second;
                        if (!_counts.ContainsKey(time)) {
                                var timeRegion = await AllocateRegionAsync<Transferable<Dictionary<S,int>>();
                                open timeRegion
                                {
                                        timeRegion.Value = new[timeRegion] Dictionary<S, int>();
                                }
                                _counts[time.Copy(timeRegion)] = timeRegion;
                                this.NotifyAt(time);
                }

                var timeRegion = _counts[time];
                open timeRegion in
                {
                    if (!timeRegion.Value.ContainsKey(msg))
                    {
                        timeRegion.Value[msg.Copy()] = 0;
                        send = true;
                    }
                    timeRegion.Value[msg]++;
                }
            }

            if (send == true)
            {
                await SendLocalMessageAsync(messageRegion);
            }
        }

        action async void OnNotify(Transferable<T> messageRegion)
        {
            open messageRegion in
            {
                var timeRegion = _counts[messageRegion.Value];
                Consume(_counts[messageRegion.Value]);
                await SendLocalMessageAsync(timeRegion);
                _counts.Remove(messageRegion.Value);
            }
            await FreeRegionAsync(messageRegion);
        }
    }
```

Join Example

Thursday, July 3, 2014 6:50 PM

```
/*
General comments is that so far the API does not have support for the following:
 - specifying whether a certain region (that is created) can or should be
   garbage collected, or not.
 - hinting the memory management on the location where this region should be
   allocated (local heap, global heap, RDMA buffer, RPC buffer, etc.
*/

enum RegionTypes : int {Transferable, ActorScoped, ActionScoped};

class RegionContext<T>
{
    public static void Open(Region<T> region) { }
    public static void OpenAlloc(Region<T> region) { }
    public static void Open() { }
}

class Region<T>
{
    RegionTypes regionType;
    T rootObject;

    public Region(RegionTypes t)
    {
        regionType = t;
    }
    public T    getData() { return this.rootObject; }
    public void setData(T rootObject) { this.rootObject = rootObject; }
    public void GiveUp() { }
    public void Transfer() { }
    public void suckObject() { }
}

class RegionExamples
{
    public class MyJoinVertex<TInput1, TInput2, TKey, TOutput, TTime>
    {
        // declare an actorScoped region handle that is accessible
        // during the lifetime of the vertex
        Region<Dictionary<TKey, Pair<List<TInput1>,
                                     Dictionary<TTime, List<TInput2>>>>> actorScoped;

        private readonly Dictionary<TKey, Pair<List<TInput1>,
                                          Dictionary<TTime, List<TInput2>>>> values;
        private readonly Func<TInput1, TKey> keySelector1;
        private readonly Func<TInput2, TKey> keySelector2;
        private readonly Func<TInput1, TInput2, TOutput> resultSelector;

        public void onReceive1(Region<Message<TTime, TInput1>> messageRegion)
        {
            int length;
            TTime time;
            Message<TTime, TInput1> msg;

            using (RegionContext.Open(messageRegion))
            {
                msg = messageRegion.getData();
                time = msg.time;
                length = msg.length;
            }
            Region<Buffer<TTime, TOutput>> outputRegion =
                new Region<Buffer<TTime, TOutput>>(RegionTypes.Transferable);

            Buffer<TTime, TOutput> output;
            int outputOffset = 0;

            using (RegionContext.OpenAlloc(outputRegion))
            {
                output = new Buffer<TTime, TOutput>(time);
                outputRegion.setData(output);
            }

            /* most of the references until now are from the stack to a transferable
               region, either the input message region or output region */

            using (RegionContext.Open(messageRegion))
            {
                Message<TTime, TInput1> message = messageRegion.getData();
                using (RegionContext.OpenAlloc(actorScoped))
                {
                    var tmpValues = actorScoped.getData();
                    // It will be the case that you have to write to two different
                    // regions within a for loop (see below) in which case it will
                    // be inevitable to open and close regions. This maybe can be
                    // simplified if we keep the old form of new [regionR]Object();
                    for (int i = 0; i < length; i++)
                    {
                        // keySelector1 allocates key in actorScoped region
                        // however, any temporary objects used by the key selector are
                        // also allocated in this region, which may end up creating garbage
                        var key = keySelector1(message.payload[i]);
                        Pair<List<TInput1>, Dictionary<TTime, List<TInput2>>> keyEntry;

                        if (!tmpValues.TryGetValue(key, out keyEntry))
                        {
                            // keyEntry is allocated in the actorScoped region
                            keyEntry = new Pair<List<TInput1>,
                                                Dictionary<TTime, List<TInput2>>>
                                (new List<TInput1>(), new Dictionary<TTime,List<TInput2>>());
                            tmpValues[key] = keyEntry;
                        }

                        // at this point we are copying values from the message
                        // region to the actor-scoped region
                        var tmpObject = actorScoped.suckObject(message.payload[i]);
                        keyEntry.First.Add(tmpObject);

                        var matches = keyEntry.Second.Values;

                        using (RegionContext.OpenAlloc(outputRegion))
                        {
                            var output = outputRegion.getData();
                            foreach (var match in matches)
                            {
                                foreach (var m in match)
                                {
                                    /* The inContextAllocator for the resultSelector is
                                    the inner-most region opened for allocation: outputRegion
                                 */
                                    output.set(outputOffset++, resultSelector(tmpObject, m));
                                }
                             }
                        }
                    }
                }
            }
            messageRegion.GiveUp();

            // prepare an InAllocationContext for the send function.
            // Since there are no input arguments we assume that the other
            // regions are not necessary. If the actor-scoped region is
            // required it should be (re-)opened internally?
            // Kapil: The region should not be open when it is sent
            using (RegionContext.OpenAlloc(outputRegion))
            {
                output.send();
            }
            // outputRegion.Transfer(this.getDownStreamActor());
        }

        public void onReceive2(Region<Message<TTime, TInput2>> messageRegion)
        {
            // These fields are placed on the stack, with actionScope lifetime
            int length;
            TTime time;
            Message<TTime, TInput2> msg;

            // We open the message region, so that we setup the
            // references from the fields to the messageRegion elements.
            // The tricky bit here is that if we give up on the message
            // region we must not! use the time and msg objects...
            using (RegionContext.Open(messageRegion))
            {
                msg = messageRegion.getData();
                time = msg.time;
                length = msg.length;
            }

            Region<Buffer<TTime, TOutput>> outputRegion =
                new Region<Buffer<TTime, TOutput>>(RegionTypes.Transferable);
            Buffer<TTime, TOutput> output;
            int outputOffset = 0;

            using (RegionContext.OpenAlloc(outputRegion))
            {
                output = new Buffer<TTime, TOutput>(time);
                outputRegion.setData(output);
            }



            using (RegionContext.Open(messageRegion))
            {
                using (RegionContext.OpenAlloc(actorScoped))
                {
                    for (int i = 0; i < length; i++)
                    {
                        // the keySelector is not allocating new objects. the key variable is
                        // allocated on the stack and will eventually have a reference to
                        // an element in the message region. Because the variable is allocated
                        // within in the using message using region, its lifetime scope should
                        // be the same as the object to which it points to.
                        var key = keySelector2(message.payload[i]);
                        Pair<List<TInput1>, Dictionary<TTime, List<TInput2>>> keyEntry;

                        if (!this.values.TryGetValue(key, out keyEntry))
                        {
                            // this new Pair will be allocated from the actorScoped region
                            keyEntry = new Pair<List<TInput1>, Dictionary<TTime,
List<TInput2>>>
                                (new List<TInput1>(), new Dictionary<TTime, List<TInput2>>());
                            // and is added to the actorScoped collection of values
                            this.values[key] = keyEntry;
                        }

                        if (!keyEntry.Second.ContainsKey(time))
                        {
                            // the new List is allocated also on the actorScoped region,
                            // and is added to the actorScoped keyEntry in the collection
                            // of values.
                            keyEntry.Second.Add(time, new List<TInput2>());
                        }

                        var currentEntry = keyEntry.Second[time];

                        // we have to suck the object from messageRegion to actorScoped region
                        // so that we can safely place it in the actorScoped collection
values.
                        currentEntry.Add(actorScoped.suckObject(message.payload[i]));

                        // We have to open the outputRegion now for allocating new objects
                        // from within the UDF (resultSelector).
                        using (RegionContext.OpenAlloc(outputRegion))
                        {
                            foreach (var match in currentEntry)
                                foreach (var m in keyEntry.First)
                                    // as before both m and message.payload are accessible for
reading.
                                    // should the UDF want to actually reuse the objects it
will need
                                    // to suck them internally
                                    output.set(outputOffset++, resultSelector(m,
message.payload[i]));
                        }
                    }
                }
            }
            messageRegion.GiveUp();
            using (RegionContext.OpenAlloc(outputRegion))
            {
                output.send();
            }
            // outputRegion.Transfer(this.GetDownStreamActor());
        }

        public void onNotify(Region<TTime> timeRegion)
        {
            using (RegionContext.Open(timeRegion))
            {
                TTime time = timeRegion.getData();
                // Since we are not allocating any new Objects, but removing,
                // it is enough to just open the actorScoped region
                using (RegionContext.Open(actorScoped))
                {
                    var allKeys = this.values.Values;
                    foreach (var k in allKeys)
                        k.Second.Remove(time);
                }
            }
            timeRegion.GiveUp();
        }



        // We could create and actor-scoped region in the Constructor.
        public MyJoinVertex(Func<TInput1, TKey> key1,
                            Func<TInput2, TKey> key2,
                            Func<TInput1, TInput2, TOutput> result)
        {
            actorScoped = new Region<Dictionary<TKey, Pair<List<TInput1>,
                                                           Dictionary<TTime,
                                      List<TInput2>>>>>(RegionTypes.ActorScoped);
            using (RegionContext.OpenAlloc(actorScoped))
            {
                this.values = new Dictionary<TKey, Pair<List<TInput1>, Dictionary<TTime,
List<TInput2>>>>();
                actorScoped.setData(this.values);
            }
            this.keySelector1 = key1;
            this.keySelector2 = key2;
            this.resultSelector = result;
        }
    }
}
```

Writing code with regions

Wednesday, July 9, 2014 5:55 PM

The aim of this document is to list the assumptions and comments regarding the scope covered with the new region API as specified in the email sent by Dimitrios on the 23rd of June, and updated in page Region Constraints.

Actor- and action-scoped regions with the new Region API

With the new API there is no longer specific support for: (1) actor- and (2) action-scoped regions. The exact use-cases for them were: (1) for (one or more) collections and/or objects whose life-span is the life-span of the actor (Naiad vertex) itself, (2) for any object allocation happening in the implementation of the standard actor functions (onNotify, onReceive, etc).

One idea of implementing an actor-scoped region is to specify the region handle as one more vertex field. This way we can achieve that it is accessible (visible) from anywhere else in the vertex code logic. The region can be allocated from within the vertex constructor, and the corresponding (root) objects will be properly allocated and linked to the region.
Gotham, however, suggested that 'this' object will be allocated in a 'static' region that is serving as the *inAllocationContext* when the vertex constructor is executed. So there is no need to explicitly create an actor-scoped region.

Once we have decided that we have an actor-scoped region available by default, what is the default *inAllocationContext* for the standard actor functions such as onNotify, onReceive, etc? Is it the actor-scoped region? Is there a hidden action-scoped region opened in the background, or should it be explicitly created once we start the function.

The questions here are the following:

- By 'static' here I assume you refer to the region as private to the vertex and non-transferable.
- If we do not explicitly create it, then what is the type of objects that can be allocated there?
- Will it also be a rooted-region, in which case what is the type of the expected root?
- Do we have control over whether it is going to be GC-ed or not?
- Should we specifically open- (or openAlloc-) the region or do we assume that its contents are available at any time as long as the vertex is alive?

Regions and *InAllocationContext* for UDFs

The assumptions made here are the following:
- no requirement for supporting higher-order UDFs, i.e, at code-generation time the whole vertex code is flattened out and available for analysis.
- the required regions for accessing the UDF's input arguments are opened before the UDF is called.
- the caller is responsible for preparing the *inAllocationContext* for actions and allocations within the UDF. In particular should the UDF return an object it must be allocated in the specified *inAllocationContext*.
- should the UDF want to use the objects given as parameters internally (e.g. include them in a collection) it is responsible for sucking the objects in its own region prior to doing that. (I assume this is a safe-assumption. It is attractive because it is the optimistic approach, i.e., we don't have to suck objects prior to the call if there is no real need for it).
- The UDF internally is allowed to overwrite the *inAllocationContext* by creating a separate private scratchpad region. As mentioned above, the only constraint is that the return value must be visible/accessible from outside the UDF, i.e., be allocated in the default *inAllocationContext*.

General assumptions and rules for opening and closing regions

Depending on the cost (of implementation) for opening and closing a region, a general rule would be to have the regions opened for reading before entering a tight loop.

The only mandatory exception is when one would have to write/allocate to two (or more) different regions within a loop in which case we would have to allow opening and closing these separate regions accordingly.

Questions about the examples

Saturday, June 21, 2014 6:23 PM

/*
Assumptions made when reading the examples: (based on our understanding of the Broom design document)

- We distinguish two types of private regions based on their scope: actor- and action-based.
- The private regions are automatically opened, within the actions performed by an actor.
- All shared/transferrable regions must be explicitly opened for performing any action on the objects stored in these regions (allocating, removing, reading, updating, etc).
- Whenever one needs to allocate a new object, one needs to explicitly specify in the allocator the region where this object will reside.

*/

Questions and comments on the examples listed in the Broom notebook.

- comments on MapActor example:
  - _target is allocated in the actor-scoped private region;
  - nothing is allocated in the action-scoped (in action onRecv) region.
  - var targetRegion resides on the stack.
  - we have to open both the messageRegion and the targetRegion because they are both transferrable regions and we want to read from (messageRegion) and allocate in (targetRegion).
  - there is no need for opening the transferrable regions also in the Mapper function/method because that one is not a separate action, and it has been invoked in a scope within an action where the two regions have been previously opened.

- comments on BufferActor example:
  - _target is allocated in the actor-scoped private region;
  - _bufferRegion is allocated in the actor-scoped private region, but points to a newly allocated transferable region.
  - ??? in the onRecv action, why don't we open also the messageRegion (which is also transferable) when we need to copy a value from that region to the bufferRegion).

- comments on GroupMinActor example:
  - in the action onRecv, there is a comment stating that we should allocate the key in a new action-scoped region. Why? Can't the key just be allocated on the stack? Is it by default that when you have multiple private regions opened (in this case both the actor and the action-scoped regions) one would always (implicitly) allocate the new objects in the inner-most region. Aren't key and val just pointers to objects in the transferable messageRegion region?
  - We do not have to open the actor-scoped region in order to access elements/objects in it.
  - By assigning _mins[key] = val -- does it mean that there is now a reference from thisRegion to msgRegion. (references between transferable regions are not allowed)
  - In contrast, in _mins[key.Copy(thisRegion)] = val; does it mean that we copy the value for key from messageRegion to thisRegion explicitly?
  - In the if(change) branch, we are allocating a new Transferable region called targetRegion, and then allocate a new object in the region without opening its scope.
  - In the constructor used for a new Tuple, do we copy the values for key and val or are they just references to the messageRegion?

- comments on DistinctCount example:
  - _counts is a Dictionary of transferable regions. We open one such timeRegion in order to allocate a new value object within that region and immediately close it.
  - but in the next line we use the timeRegion to write in it, but it is closed.
  - This is contrast to the usage in the next couple of lines where we only close the timeRegion after we are done with allocating new objects and updating values.
  - Lastly in the onNotify action we transfer the transferrable region to another actor, and only afterwards do we remove the reference to it from the _counts variable. Isn't this illegal?

Bibliography

Tuesday, December 17, 2013 9:23 AM

Heaps:

The Knowledge Base article Q10758, "Managing Memory with calloc() and malloc()" (search on article ID number), contains more background on these topics. Also, a detailed discussion on the heap implementations and designs can be found in "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995 (http://www.cs.utexas.edu/users/oops/papers.html).

From <http://msdn.microsoft.com/en-us/library/ms810466.aspx>

Following the development of ML Kit, regions began to be generalized to other language environments:

- Various extensions to the C programming language:
  - The safe C dialect Cyclone, which among many other features adds support for explicit regions, and evaluates the impact of migrating existing C applications to use them.[13][14][15]
  - An extension to C called RC[16] was implemented that uses explicitly-managed regions, but also uses reference counting on regions to guarantee memory safety by ensuring that no region is freed prematurely.[17][18] Regions decrease the overhead of reference counting, since references internal to regions don't require counts to be updated when they're modified. RC includes an explicit static type system for regions that allows some reference count updates to be eliminated.[19]
  - A restriction of C called Control-C limits programs to use regions (and only a single region at a time), as part of its design to statically ensure memory safety.[20]
- Regions were implemented for a subset of Java,[21] and became a critical component of memory management in Real time Java, which combines them with ownership types to demonstrate object encapsulation and eliminate runtime checks on region deallocation.[22][23][24] More recently, a semi-automatic system was proposed for inferring regions in embedded real-time Java applications, combining a compile-time static analysis, a runtime region allocation policy, and programmer hints.[25][26] Regions are a good fit for real-time computing because their time overhead is statically predictable, without the complexity of incremental garbage collection.
- They were implemented for the logic programming languages Prolog[27][28] and Mercury[29][30] by extending Tofte and Talpin's region inference model to support backtracking and cuts.
- Region-based storage management is used throughout the parallel programming language ParaSail. Due to the lack of explicit pointers in ParaSail,[31] there is no need for reference counting.

From http://en.wikipedia.org/wiki/Region-based_memory_management

Midori paper:

midori-p21-gordon

Broom Type System Prelims

Monday, June 2, 2014 11:04 AM

Introduction and Motivation

Thursday, June 5, 2014 5:31 PM

Introduction

In this document, we motivate the need for a region type system for Broom programs, clearly identify the problems that the type system needs to solve, and discuss some related work along with the applicability of proposed solutions to our problem setting.

It may be useful to read the examples provided in the preceding pages before reading this writeup.

System Description

For the sake of the discussion, we assume that system comprises of a set of actors, each with an Actor Id (*actId*). An actor is an instantiation of a class that belongs to an application. All actors run on the same machine sharing same physical memory, and communicate by message passing.

Memory Management

In Broom, memory is allocated to actors in terms of blocks called regions. Three kinds of regions can be identified in Broom:

- *Private Regions* : A private region is a block of memory that is private to the actor to which is was allocated. Memory for local variables of methods within the actor, and data structures that are private to the actor are allocated in its private regions. Actor's private regions can be made to abide by the LIFO discipline, with following characteristics:
    1. A *private stack region* is allocated to an actor whenever it executes a method call. Such region essentially serves as an activation record for the method being called. As with activation records, private stack regions form a stack of regions, where regions at higher addresses in the stack (stack is assumed to grow downwards) have longer lifetime than those at lower addresses. A private stack region can be deallocated as soon as its corresponding method call returns.
    2. A *private heap region* is allocated to an actor at the beginning of actor's life time, and outlives all its stack regions. Data stored in actor's heap region is assumed either to persist throughout the lifetime of the actor, or to be garbage collected appropriately. In other words, there is no explicit way to deallocate memory in private heap region.
- *Transferable Regions* : A transferable region is a block of memory that is allocated/deallocated to an actor on demand. Subsequent to the allocation, the actor to which the transferable region is allocated is its unique *owner*. Owner of a transferable region can allocated/deallocate objects into the region. An actor can chose to transfer the ownership to another actor (whence *transferable* region). Subsequent to the transfer, the former is no longer allowed to refer into the region. It can be observed that a transferable region and the act of ownership transfer effectively model a message, and the act of passing the message between different actors, respectively. Ownership transfer effectively ends the lifetime of transferable region as far as its owner is concerned. Alternatively, the owner of a transferable region can chose to deallocate it, marking the real end of region's life time. Transferable regions mark the departure from LIFO discipline among lifetimes of regions. Some important observations regarding this point:
    1. Depending on when it is allocated and deallocated, a transferable region can outlive multiple private (stack) regions of its owner, or it may outlive none. Example for first case is when a transferable region is allocated at the beginning of actor's lifetime, updated throughout its lifetime via multiple method calls, and is transferred to another actor just

before the end of owner's life time. An example for second case is when the ownership of the transferable region is transferred to a different actor almost immediately after it has been allocated (i.e., allocation and ownership transfer are not separated by a method call). Nevertheless, LIFO discipline is not violated in both the examples, as the lifetime of regions can still be represented as in form of a stack involving private regions and transferable regions. However, LIFO discipline is violated when a transferable region allocated in a method A, is deallocated in a method B called subsequently (i.e., activation record for B occurs below that of A). One consequence of this is that a local variable of A that points into the transferable region suddenly becomes a dangling pointer after the call to B. Evidently, lifetimes of regions cannot be represented as a stack in such case.
    2. An actor can own multiple transferable regions. Lifetimes of such regions can overlap, but need not necessarily be ordered by the *outlives* relation. An example of such case is when two transferable regions are deallocated in different order than they were allocated. Neither outlives the other in such case. Pointers from one region to other become dangling when the later is deallocated before the former. A more complex case is when ownerships of both the allocated regions are transferred. Pointers between transferable region become dangling depending on whether ownerships of both are transferred to the same actor, or different actors.

  (I presume that) In order to avoid aforementioned complexities, we impose certain restrictions on how Broom programs should deal with transferable regions:

    1. We require that allocations and deallocations within a transferable region happen within a lexical block, during which the region is *open*. An open region cannot be deallocated; neither can it be transferred. Therefore, a call to a method made within the lexical block that opens the region cannot deallocate or transfer the region. There can be multiple non-overlapping lexical blocks which open a given transferable region. Consequently, the entire lifetime of a transferable region is split into multiple incarnations, where each incarnation is in *outlives* relation with lifetimes of actor's private regions. Effectively, we have reinstated the LIFO discipline among the lifetimes of private regions and lifetimes of transferable regions. With this restriction, a dangling pointer manifests straightforwardly in the syntax of the program - a pointer into a transferable region is dangling if its scope exceeds the scope of lexical block during which the region is open.
    2. We require that a transferable region be closed with respect to pointer references. That is, a transferable region should not contain pointers into other transferable regions, or even the private regions of an actor. With this restriction in place, the question of formulating an outlives relation among transferable regions is now moot.

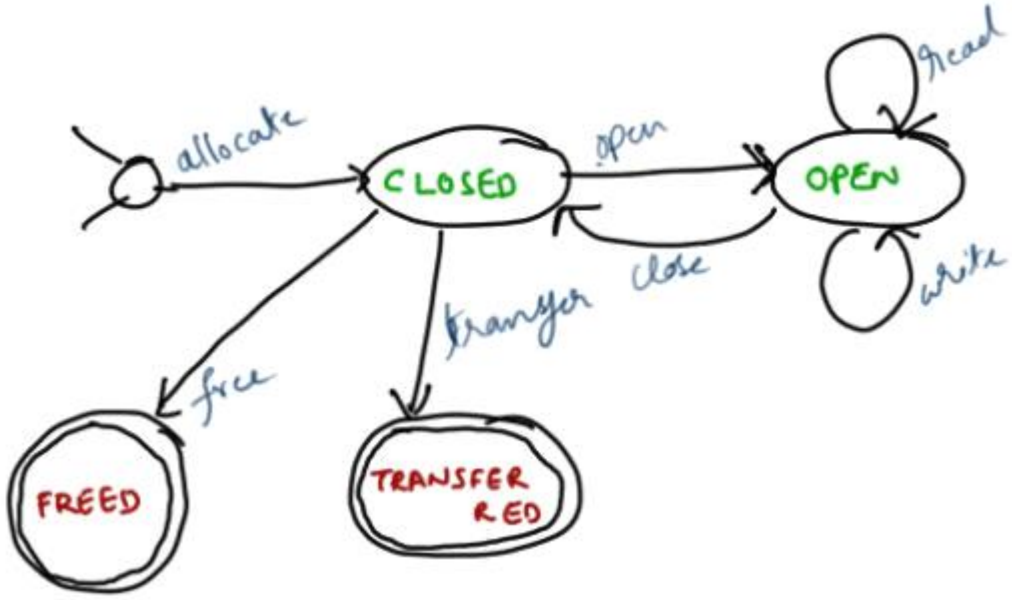
- *Shared Regions* : Shared regions are regions that are always live, and shared among multiple actors. We ignore share regions for the time being.

Region Types

Ensuring memory safety is equivalent to ensuring that

- There are no space leaks (or region leaks, in presence of explicitly allocated regions), and
- All memory accesses are safe. A dangling pointer (i.e., a pointer into a deallocated region) is never followed.

In the absence of explicitly allocated transferable regions, ensuring memory safety of Broom programs is only as hard as ensuring memory safety of C programs without `malloc` and `free`(recall that private heap region of an actor is either immortal, or automatically garbage collected). Since there are existing solutions that work well for this subset, we concentrate on containing unsafe interactions with memory that are possible due to the introduction of transferable regions. Such interactions can be understood by examining the following state machine representation of the lifetime of a transferable region:



The above figure succinctly captures all legal interactions with transferable regions in a Broom program. A transition not represented in above state machine is illegal. For example, following are some illegal actions that violate memory safety:

1. Any action on a region that is not yet allocated.
2. Closing a closed region, or opening an open region.
3. Reading from, or writing to, a closed, freed, or transferred region. To state more precisely, following a pointer into closed, freed, or transferred region is a violation of memory safety.
4. Freeing, or transferring an open region.
5. Any action on the region that is transferred, or freed.

Our goal is to formulate a region type system primarily to prevent aforementioned bad behaviours. To be more specific, our plan is to:

1. Define a model (a calculus) that can capture all the aforementioned unsafe operations, yet sufficiently constrained such that reasoning with the model is easy.
2. Define operational semantics, such that the evaluation gets *stuck* if a memory unsafe operation is attempted.
3. Define static semantics for the type-annotated programs of the calculus, such that type checking is a) decidable, and b) practical. Prove type safety (Well typed programs do not get *stuck*).
4. Construct an algorithm for type inference. Type inference may not be complete; our aim is to minimize manual annotations as much as possible.

Type system explained

Friday, July 4, 2014 6:33 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);
              // Where does this come from? Is it just a typo?
    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<$\rho_0^a$ $\rho_1$,$\rho_2$>(Region< $\rho_1$><Batch<Record>>@$\rho_2$ rin,
                     Func<$\rho_1^a$>(<Record@$\rho_1$,OutRec@$\rho_1^a$> udf)
```

Observe that this is (technically) higher-rank polymorphism.

Related Work

Thursday, June 5, 2014 7:50 PM

Related Work

We now closely examine some of the related work in the area of region type systems. Specifically, we will describe salient aspects of the following papers, while commenting on the applicability of the solution proposed to our problem setting:

- Tofte & Talpin, Implementation of the typed call-by-value λ-calculus using a stack of regions, POPL’94.
- Grossman et. al., Region-Based Memory Management in Cyclone, PLDI’02.
- Henglein et. al., A direct approach to control-flow sensitive region-based memory management, PPDP’01.
- Boyapati et. al., Ownership Types for Safe Region-Based Memory Management in Real-Time Java, PLDI'03.
- Walker D, and Watkins K, On Regions and Linear Types, ICFP'01.
- Wadler P, Linear Types Can Change The World!, PCC'90.

Due to the unconventional semantics of transferable regions, our problem setting is somewhat unique; so none of the aforementioned works apply directly in our case. Nevertheless, our motivation is to identify differences, and use their models as starting point to construct right model for our setting.

Tofte and Talpin

Summary

Introduces statically nested memory regions for ML programs. Elaborates well-typed ML programs, introducing following expressions (ρ denotes regions, e denotes expressions):

letregion ρ in e
e at ρ

We denote ML extended with above expressions as ML+ρ. letregion expression of ML+ρ introduces (allocates) a region, which is bound to ρ in e. The region handler (ρ) can be used in expressions form e' at ρ within e to store the result of evaluating e' (a runtime value - an integer, a tuple, or a closure) in ρ. The lifetime of a region introduced via letregion expression is apparent from its lexical structure; when ρ goes out of scope, the lifetime of corresponding region has ended and it can be deallocated. Since all runtime values are stored in regions, lexical scoping of regions essentially means that lifetimes of runtime values are statically determined (or conservatively approximated). A most conservative elaboration algorithm from ML to ML+ρ would simply enclose entire program inside one letregion expression. This approach entails:

- Making a conservative assumption that all runtime values are always needed, and
- Using a single region to store all such values, which is never deallocated.

Such an approach, though sound, does not help in memory reuse, defeating the primary purpose of region-based memory management. A good elaboration algorithm should localize lifetimes of runtime values as much as possible. Therefore, it should introduce several letregion annotations, making regions as distinct and local as possible. Such an elaboration scheme is the primary contribution of the paper.

In the absence of closures, determining the lifetime of values is relatively straightforward - one only has to take the scope of value in to account. For eg, consider the ML code:

```
let x = 2 in let y = let z = 1 in x-z in x+y
```

Its elaborated version is given below:

```
letregion r1
in let x = 2 at r1
   in letregion r2
      in let y = letregion r3
                 in let z = 1 at r3
                    in (op -) (r1,r3,r2) (x,z)
         in (op +) (r1,r2,r4) (x,y)
```

Regions *r*1, *r*2, and *r*3 hold the values bound to variables *x*, *y*, and *z* (resp.), and have same scope as the variables themselves. For example, the scope of region *r*3 is the (elaborated) expression *x*—*z*, after which it can be deallocated. The deallocation is safe, as *z* is not needed outside its lexical scope.

In the above example, Functions + and — are assumed to be region polymorphic, meaning that they are parameterized over regions of their arguments and also the region where the result has to be stored. For instance, (op +) is (+ in the below definition denotes machine integer addition):

```
Λ(ρ0,ρ1,ρ2). λ(a: int@ρ0, b : int@ρ1). a+b at ρ2.
```

In the paper, the *decorated type int@ρ* is written (*int*,*ρ*), and denotes an integer at region *ρ*. Instantiating the parameterized region variables of `(op -)` with (*r*1,*r*3,*r*2) yields a subtraction function that expects its arguments at *r*1 and *r*3, and stores the result in *r*2. Since this result is bound to variable *y*, whose scope is the (elaborated) expression *x*+*y*, region *r*2 has to be live at *x*+*y*, which is ensured by the corresponding letregion annotation.

It can be observed that region *r*4 occurs free in the elaborated version. This is because *r*4 is the region that holds the result of evaluating the above expression; so, it has to be live even after the expression is finished evaluating.

This simple scheme of using the lexical scope of the variable to determine the lifetime of the bound value fails in presence of closures. Consider, for example, the following simple ML expression:

```
let x = (2,3) in λy.(fst x,y)
```

The lexical scope of *x* is only the lambda expression under let, but the runtime value bound to *x* escapes the scope of *x* through the closure. In other words, the lifetime of runtime value bound to *x* exceeds the lexical scope of *x*. In order for the elaboration to ML+*ρ* to be memory safe, it is imperative to keep track of values that *hide* under closures and escape the lexical scope of their names.

The paper introduces *arrow effects* and *effect variables* for this purpose. An effect in the context of the paper is either a *read* or a *write* to/from a region. The intuition is to label the arrow type (i.e., type of a closure) with *read(ρ)* and *write(ρ)* tags, where ρ is a region, such that they indicate regions that could be read/written if the closure is executed. For example, the `fst` function, which projects the first component of an integer pair, could have the following type:

```
fst : ∀(ρ0,ρ1,ρ2). ((int@ρ0, int@ρ1)@ρ2) -ϕ-> (int@ρ0)
```

The label (ϕ) over the arrow denotes the following effect set: {*read*(*ρ*2),*read*(*ρ*0)}. Now, assuming that the expression `(2,3)` is elaborated to `(2 at r1, 3 at r2) at r3`, for some regions *r*1, *r*2, and *r*3, the lambda expression:

```
λy. (fst x,y)
```

will surely have the arrow in its type labelled with the effect set: {*read*(*r*3),*read*(*r*1)}, which tells us that if the corresponding closure is evaluated, regions *r*1 and *r*3 are read. Since the closure escapes the let expression, *r*1 and *r*3 should not be generalized (with letregion) in this expression. On the other hand, *r*2 can be generalized:

```
letregion r2 in let x = (2 at r1, 3 at r2) at r3 in ....
```

Region *r*2 can be safely deallocated once the let is evaluated, as the values stored in *r*2 do not escape through the closure.

The paper formalizes intuitions stated thus far and proposes an elaboration scheme together with a region type discipline for ML programs. The formalization accommodates parameterization over regions and effects in function types, which invariably leads to multiple categories of type schemes. It is not clear at this point of time how type safety was phrased and proved for the region type system.

Comparision with Broom

After discussion with Rama, I realized that private regions are intended to be statically nested regions similar to Tofte and Talpin (T&T) regions. They need not necessarily be activation records. Further, it is (currently) assumed that Broom programs have annotations introducing static regions; so, at first glance, it appears that the target language of T&T could be sufficient to model Broom programs with only private regions. In such case, Region type system of T&T could be reused for this subset. However, some questions:

1. In the target language of T&T, letregion syntax ensures LIFO discipline among static regions. Is there any such lexical block to introduce a static region in Broom programs? Since static nesting essentially means lexical nesting, I expect that we should be able to define a syntactic block to introduce static regions.
   Kapil: We don't have a construct yet. But we can definitely extend the RegionContext API to achive this.

2. Consider a pair of statically nested regions, *r*1 and *r*2, such that *r*1 outlives *r*2, and an expression *e*, which can see (i.e., is in scope of) both *r*1 and *r*2. In T&T target language, the example of such an expression is:
   ```
   letregion r1
   in ....
      letregion r2
      in e
   ```
   e can store values in *r*1 and *r*2; That is, *e* can contain sub-expressions of form *e*'*atr*1 and *e*'*atr*2. However, it was mentioned during the discussion today that under semantics of regions in Broom, sub-expression of later form is allowed, but the former sub-expression is disallowed. Futher, it was also mentioned that *e* can *allocate* memory in *r*2, but cannot *store* a value in *r*2. This difference is not clear.
   Kapil: We should be able to support both forms of expressions. At the moment, expressions of the former kind requires can be written by creating a special constructor that takes a region as the first parameter. With compiler support, we should be able to implement a construct similar to e at r

Nevertheless, T&T language has no non-statically nested regions, which are required to model transferable regions of Broom. Further, the language has no references needed to capture the possibility of dangling pointers in Broom programs that result from target region being freed/transferred.

Cyclone

Cyclone is a type-safe subset of C. The primary purpose of the type system is to ensure memory safety while allowing programmers to have control over memory management. Here are the main ideas:

Region Type System

consider a subset of C with only stack allocated memory (i.e., only local variables are allowed). This subset is still memory unsafe, as it is possible for a method to return the address of a local variable to its caller, which then becomes a dangling pointer. A simplified version of such example can be presented using lexical blocks:

```
int *p;        //1
L : {          //2
  int x = 0;   //3
  p = &x;      //4
}              //5
*p = 42;       //6
```

Since memory for variable x is deallocated as soon as lexical block L ends, the statement `p = &x` effectively makes p a dangling pointer at line 6. However, this unsafe operation can be detected by observing that :

1. Variables p and x belong to different lexical blocks,
2. Lexical block of p outlives that of x, and
3. p refers to x, which means that p points to deallocated memory after lifetime of block L.

T&T's region type system allows us to perform such reasoning. To see how, consider the following program in T&T's target language (extended with references), which is equivalent to above C code:

```
letregion r1
in let p = letregion rL
           in let x = 0 at rL
              in (ref x) at r1
   in p := 42
```

The assignment on last line is an unsafe operation as p is a reference into region rL, which is now deallocated. Fortunately, the program is ill typed under T&T's region type system, as decorated type of expression `ref x` is `(ref(int,rL),r1), which must be the type of p as per the type rule of let expression; but, such a type for p is ill-formed as region rL is not in scope.`

Cyclone makes use of the above insight to judge ill-typedness of the C program shown previously. A lexical block is considered a region in Cyclone. Static nesting among lexical blocks gives rise to the a stack of regions at run-time, which is effectively the C call stack. The region type system of T&T can now be applied to ensure memory safety.

Dynamic Regions or LIFO Arenas

Cyclone introduces dynamic regions with lexically scoped lifetime. The name "dynamic region" is somewhat a misnomer as dynamic regions of Cyclone are delimited by a lexical block, just like stack regions. However, Cyclone allows unlimited allocations into dynamic regions via `rmalloc` and `rnew` primitives. The allocated memory gets deallocated in tandem with the dynamic region, when the control exits the lexical block corresponding to the dynamic region. From the perspective of region type system, it is not interesting to distinguish between stack regions and dynamic regions.

Garbage Collected Heap

Memory that needs to survive limits of lexical blocks is dynamically allocated on heap through the usual `malloc` call. There is no `free`, hence no dangling pointers. Garbage collection is used to reclaim heap memory.

Outlives Relation & Subtyping

The LIFO discipline on region lifetimes gives rise to *outlives* relationship among region lifetimes, which allows Cyclone to define subtyping relation among pointer types. This subtyping discipline is quite useful in the context of C. For instance, consider the following example:

```
int p = 0; //1
L : { //2
  int q = 1; //3
  int *x = (p<q)? //4
     &p : &q; //5
  return *x; //6
}
```

Here is a similar example in T&T target language:

```
letregion r1
in let p = 0 at r1
   in letregion rL
      in let q = 1 at rL
         in let x = if p<q then (ref p) at rL
                           else (ref q) at rL
            in !x
```

The types of then and else branches of the if expression are (ref(int,r1),rL), and (ref(int,rL),rL), respectively. However, since r1 outlives rL, we can define following subtype relation among the types of both branches:

(ref(int,r1),rL) <: (ref(int,rL),rL)

The subtype relation effectively makes it safe to use a reference into r1, wherever a reference into rL is expected. Safety follows from the observation that r1 outlives rL, hence if its safe to dereference a pointer into rL, it should be safe to dereference a pointer into r1. Further, the subtyping relation lets us unify the types on both the branches to (ref(int,rL),rL). Hence x must be considered a reference into region rL, or any other region that lives atleast as long as rL.

Formalization

Full static and dynamic semantics for Cyclone language are given in a tech report. The formalization is of special interest to us, as the language includes full imperative features, such as pointers and aliasing. Small step operational semantics are presented as rewriting relation from machine states to machine states, and gets stuck if a memory unsafe operation is attempted. They prove the standard type safety theorem - well-typed Cyclone programs don't get stuck.

Comments

Outlives relation, and subtyping among reference types is the key takeaway from this paper. Cyclone's regions are all statically nested and stick to LIFO discipline. Therefore, transferable regions cannot be modeled in Cyclone. It is certainly conceivable to extend the formal language of Cyclone with constructs to allocate, open, close and transfer ownership of transferable regions. However, Broom is based on C#, which, unlike C, has no explcit pointers, and is inherently memory safe. Therefore, Cyclone's formal model extended with transferable regions could be an overkill, in the sense that it allows such unsafe memory interactions that are not possible in Broom programs. More work is needed to prove or refute this intuition. Further, there are no higher-order functions in Cyclone's formalization. I am not sure how often higher-order features are used in Broom (and C#) programs, but if they are indispensable then Cyclone's formal model may not be a good place to start.

Henglein, Makholm, & Niss (HMN01)

HMN01 is a variant of T&T memory management and type system. Following are salient points:

- First-Order Programs. No higher-order functions. However, C style function pointers can be supported. Basically, closures are not allowed, thereby preventing *hidden* effects that we discussed in T&T. The way this is manifest in the language is:
    - Absence of lambda expressions.
    - Presence of *declaration* syntactic class (d), which declares a function. At the top level, program is: *let \bar{d} in e.*
- No hierarchic letregion construct. Instead, embed an imperative region sublanguage in base language. The region sublanguage has constructs to allocate, release and alias regions. Allocation and release of of regions are completely independent. In other words, the lifetime of a region need not be delineated by a nice lexical block. Anything can be deallocated at any time. Therefore, the type system must be capable of identifying references that suddenly become dangling after a function call.

To compare with our problem setting:

+ We need higher-order state. Although explicit use of higher-order features in Broom/C# programs is minimal, there are classes, which encapsulate state that is implicitly shared among all methods of the class.

+ We need the letregion construct to model statically nested regions in Broom.

+ The imperative region sublanguage supports transferable regions of Broom. The lifetime of a transferable region is split into multiple incarnations, with each incarnation nicely delineated by a lexical block. This feature should make our reasoning with dynamic regions easier than in HMN01.

Base Types

Formal language includes boxed integer types, which are T&T style decorated types: (int,ρ). The type represents an address in region ρ which contains an integer. Since ρ is a value in the language, the decorated types are dependent types. Now, consider the following example:

```
[[new r1]]
let x = 1 at r1
in let y = [[new r2]] x + (1 at r2) at r2 [[release r1]]
   in e
```

where e is some expression. The type of x on line 2 is (int,r1). Now, what is its type in e? Since r1 has been released, the type of x should now be *updated* to (*int,* ⊤*). The* ⊤ *in the type denotes that x is an address into a released region. This ability to track state changes in types makes the type system a typestate system. Also, the type system can possibly ascribe different types to a variable at different program locations; which means that it is flow sensitive.*

Function Types

Functions in HMN01 have three kinds of region parameters - input regions, constant regions, and output regions. The set of constant regions is disjoint with the set of input and output regions. A function is expected to read its input from input regions, and write its output to output regions. Input and output regions need not be disjoint, so same formal parameter can occur as an input region and output region. Caller provides input and constant region arguments. As soon as the call is made, the name to value bindings of actual region arguments are lost in calling context. I understand this as a pessimistic approach - Callee releases all its input regions, unless it explicitly states otherwise.

Questions to Answer

1. Since variables are already addresses into regions, do we need to include references in our formalization? If we consider the dynamic semantics, we definitely need to as we have to capture updatation of a memory location. But, for static semantics does it matter?
2. HMN01 explicitly updates the types in Γ, when a ρ is released. My intuition was to use a separate environment ΓR for region handlers, and extend ΓR with new type of ρ. It would be interesting to understand how these methods compare, especially given that region handlers can be aliased and, unlike HMN01, releasing an alias deallocates the region in Broom.

On Regions and Linear Types (WW01)

This paper proposes a type system based on region types and linear types for a calculus with first class regions. At a high level, this is precisely what we too are interested in; so, it is natural to expect some overlap between our ideas, and the solution proposed in the paper. Here are some overlapping concepts:

1. There is a unique region name ρ, which is different from the region handler (r).
2. WW01 has $linear$ regions - regions whose handlers are linear; i.e., they can be *used* only once, and cannot be duplicated/aliased. Linear regions are allocated and deallocated via $alloc$ and $free$. These $alloc$ and $free$ need not be related by any lexical construct. In this respect, they are similar to transferable regions.
3. Allocations into linear regions can only happen within a lexical block delineated by a special let syntax. There can be multiple such let blocks between $alloc$ and $free$ of a linear region. These let blocks are intended to serve the same purpose as our $open$ blocks. Within these blocks, the linear regions behave as an $intuitionistic$ region, which means that the corresponding region handler can be used multiple times to allocate multiple objects into the linear region. However, region cannot be deallocated within this lexical block. Ensuring this invariant is straightforward as $free$ expects a value of linear type, whereas the region handler has intuitionistic type within the lexical block.
4. With their type system, it is possible to have lists of existentially typed linear objects, an example of which is the list l_r of region handlers in example 5 on the page "Calculus V2". Such lists are themselves linear.

There are some differences between their language and our current calculus (V2 and V3):

1. There is no ref In WW01. However, it is not clear if adding references to their language is safe.
2. There are no static regions.
3. A function closure is not expected to retain a handler to an intuitionistic region. Otherwise, the function is ill-typed. It is not clear if we need this requirement.
4. Their calculus seem to permit inter-region pointers. It is not clear how the safety is ensured. Nevertheless, we have stricter requirements on transferable regions - they should not contain pointers to other regions at any cost.

Linear Types Can Change the World (Wadler90)

In this seminal paper, Wadler describes a calculus with a combination of linear types and non-linear (intuitionistic) types. This work forms the bases for WW01 work.

Values belonging to linear types can be used exactly once: they can neither be duplicated, nor be discarded. "No duplication" essentially prevents aliasing, making reasoning about imperative style programs easier. For instance, when a linear reference to an object is $free$'d, we can safely free the memory allocated to the object, as we are sure that there exists no other reference to the memory region. "No discarding" means that a program cannot implicitly discard a reference to a linear object, for eg, by letting it go out of scope. Linear objects have to be explicitly deallocated.

The key idea in a linear type system is that each typing assumption in Γ must be used exactly once. If T is a linear type, then an assumption of form $x{:}T$ is a permission, and a requirement to use the variable x exactly once.

Type grammar of fully linear langauge is as following:
$T ::= !\,k \mid U\ -0\ V$
Read $U\ -0\ V$ as a function consuming an assertion U and producing an assertion V.
Characteristics of fully linear langauge:

1. Assertion consumption in variable typing rule : $x{:}\,U \vdash x{:}\,U$.
2. Weakening and contraction of Γ not allowed
3. "Splitting of resources" in type rule for application : $\frac{\Gamma_1 \vdash t{:}U\ -o\ V \quad \Gamma_2 \vdash u{:}U}{\Gamma_1,\Gamma_2 \vdash (!t\ u) : V}$
4. Linear lists: $!\,List$. Linear constructors : $!\,Cons$, which means that a linear value is "consumed" when it is put in the list.
5. Scrutinee gets "consumed" in the *case* expression. Pattern variables indicate new linear values that were "produced".

For practicality, we need a language with a combination of linear and intuitionistic values. In such language, we have:

1. Both linear and non-linear lists - a non-linear list cannot contain a linear value.
2. Both linear and non-linear functions - A non-linear function closure cannot trap a reference to a linear value.

In order to ensure the second condition, the Abs rule for non-linear function is stated thus:

$$\frac{\Gamma,x{:}U \vdash v{:}V}{\Gamma \vdash \lambda x{:}U.v : U \to V}\ (nonlinear\ \Gamma)$$

By ensuring that Γ is non-linear, we ensure that no linear value gets trapped inside the non-linear function closure. However, notice that U and V can still be linear.

Alternatively, a linear function closure can trap a linear assumption, by essentially consuming it from the typing context: $\frac{\Gamma,x{:}U \vdash v{:}V}{\Gamma \vdash\ !\lambda x{:}U.v : U \multimap V}$

Some observations:

1. There cannot be a curried intuitionistic function with multiple arguments, whose non-final argument is linearly typed.
2. In the combination type system, the rule to type check the application of an intuitionistic function is same as that of a linear function: $\frac{\Gamma_1 \vdash t : U \to V \quad \Gamma_2 \vdash u{:}U}{\Gamma_1,\Gamma_2 \vdash (t\ u) : V}$ $\frac{\Gamma_1 \vdash t : U \multimap V \quad \Gamma_2 \vdash u{:}U}{\Gamma_1,\Gamma_2 \vdash (!t\ u) : V}$
3. Then, what is exactly the difference between intuitionistic arrow ($\to$) and a linear arrow ($\multimap$)? An intuitionistic function can be duplicated, whereas a linear function is … well, linear.
4. Type of list functions:
   a. $hd : \forall a.\, a\ !\,list \to a$
   b. $tl : \forall a.\, a\ !\,list\ \to a\ !\,list$
   c. $rev : \forall a.\, a\ !\,list\ \to a\ !\,list$
   d. $concat : \forall a\ .\, a\ !\,list \to a\ !\,list \multimap a\ !\,list$
5. Observe the type of $concat$ above. The arrows indicate that $concat$ is an intuitionistic value, whereas partially applied $concat$ (closure) is a linear value. This is in tune with Wadler's following statement: "Each type is deemed linear or nonlinear depending on its topmost constructor. Hence, $T\ \multimap (U\ \to V)$ is linear, while $T\ \to (U\ \multimap V)$ is nonlinear (intuitionistic)"
6. Perhaps, linear typing in context of OOP naturally relaxes to ownership typing. This hypothesis needs to be tested.

Calculus V2 - Current

Monday, June 9, 2014 1:58 PM

Rama's suggestions:

- Static regions in Broom are more like dynamic regions in cyclone - region handlers are first class, and unlimited allocation is allowed. Moreover, we do not plan to extend the syntax of C#, so adding a new syntactic class to expression language is discouraged. Therefore, we should not separate ρ from *x*.
- We need references eventually to model pointers in C#. If adding references does not complicate type checking, then we should include ref in formal language. Comments : Adding references need not necessarily complicate type checking; but, experience of Standard ML designers suggests that it complicates type inference. Since we are not yet concerned with inference, we can add references to the language.

$n \in integers$
$x \in variables$
$\rho \in region\ identifiers$
$a \in addresses$
$\kappa\ ::=\ \mathcal{R} \mid \Theta$ //kinds
$T\ ::= int \mid unit \mid \alpha \mid \alpha\ list \mid T \to T$
$\sigma\ ::= \forall\alpha.T \mid T$
$e\ ::= x \mid e\ e \mid \lambda(x{:}T).e\ at\ e \mid n\ at\ e \mid e \otimes e\ at\ e \mid Nil\ T\ at\ e \mid Cons\ T\ e\ e\ \ at\ e \mid$
$\quad match\ e\ with\ Nil \Rightarrow e \mid Cons\ x\ y \Rightarrow e$
$\quad \mid new\langle\rho\rangle\langle T\rangle\ at\ e \mid transfer\ e \mid open\ e\ in\ e \mid letregion\langle\rho\rangle\ x\ in\ e \mid ref\ \ e\ at\ e$
$\quad \mid e\ T \mid \Lambda\alpha.e \mid let\ x = e\ in\ e \mid valueOf\ e\ \ \mid !\,e \mid e \coloneqq e$
$sv ::= n \mid Nil\ T \mid Cons\ T\ v\ v \mid region\langle\rho, o\rangle \mid ref\ v \mid \langle x, e, VE\rangle$ // Storable Values
$v\ \ ::= a$ //values

Remarks about calculus:

- We dropped pairs, but added lists to model potentially unbounded data structures
- Lists are made polymorphic so that it is possible to store region handlers in a list
- Unit type is to assign type to return value of *transfer*
- *valueOf* operator returns the reference to root of a region, given a region handler.

$$valueOf\ : \forall\alpha.\forall\rho.\, Region\langle\rho\rangle\langle\alpha\rangle \to ref\ \alpha$$

  Where are $Region\langle\rho\rangle\langle\alpha\rangle$ and $ref\ \alpha$ stored? Type of $valueOf$ should be a dependent type:

$$valueOf\ : \forall\tau_1 :: \mathcal{R}.\forall\alpha.\forall\rho.\ r_1{:}\tau_1 \to r_2{:}(Region\langle\rho\rangle\langle T\rangle, \mathrm{r}_1) \to (ref\ T, r_2)$$

  $\mathcal{R}$ is the kind of region handler types.
- When a region is newly allocated, $valueOf$ returns an invalid reference. Dereferencing such reference, or transferring a region whose root is not initialized must be a type error.
- It is not yet clear which universe ρ belongs to (value/type/kind ?).
- A region polymorphic function manifests as a function expecting static region handlers as arguments. We retain prenex quantification. Syntax of arrow types should reflect this.
- We use ρ_H as heap region identifier, and r_H as heap region handler.
- What are values in the calculus? Similar to T&T, all values are boxed; so, all our values are addresses. We define a syntactic class of storable values, which is the co-domain of the store (s) in dynamic semantics.

Dynamic Semantics:

$$\boxed{s, VE \vdash e\ \to v, s}$$

$$\frac{VE(x)=a}{s,VE\vdash x\rightarrow a,s}$$ [E-Var]

$$\frac{s,VE\vdash e_2\rightarrow a_1,s_1 \quad s_1(a_1)=region\langle\rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o)}{s,VE\vdash \lambda(x{:}T).e_1\ at\ e_2\rightarrow a,\ s_1+\{a\mapsto\langle x,e_1,VE\rangle\}}$$ [E-Abs]

$$\frac{s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1,VE\vdash e_2\rightarrow a_2,s_2 \quad s_1(a_2)=region\langle\rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o)}{s,VE\vdash ref\ e_1\ at\ e_2\rightarrow a,\ s_2+\{a\mapsto ref\ a_1\}}$$ [E-Ref]

$$\frac{s,VE\vdash e\rightarrow a_1,s_1 \quad s_1(a_1)=ref\ a}{s,VE\vdash\ !e\rightarrow a,\ s_1}$$ [E-DeRef]

$$\frac{s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1(a_1)=ref\ a \quad s_1,VE\vdash e_2\rightarrow a_2,s_2}{s,VE\vdash e_1{:=}e_2\rightarrow\bot,\ s_2+\{a_1\mapsto ref\ a_2\}}$$ [E-Assn]

$$\frac{s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1,VE\vdash e_2\rightarrow a_2,s_2 \quad s_2,VE\vdash e_3\rightarrow a_3,s_3 \quad s_3(a_3)=region\langle\rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o)}{s,VE\vdash Cons\ T\ e_1\ e_2\ at\ e_3\rightarrow a,\ s_3+\{a\mapsto Cons\ a_1\ a_2\}}$$ [E-Cons]

$$\frac{s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1(a_1)=Nil \quad s_1,VE\vdash e_2\rightarrow a_2,s_2}{s,VE\vdash match\ e_1\ with\ Nil\Rightarrow e_2\ |\ Cons\ x\ y\Rightarrow e_3\rightarrow a_2,\ s_2}$$ [E-Match1]

$$\frac{s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1(a_1)=Cons\ a_{11}a_{12} \quad s_1,VE+\{x\mapsto a_{11},y\mapsto a_{12}\}\vdash e_3\rightarrow a_3,s_3}{s,VE\vdash match\ e_1\ with\ Nil\Rightarrow e_2\ |\ Cons\ x\ y\Rightarrow e_3\rightarrow a_3,\ s_3}$$ [E-Match2]

$$\frac{s,VE\vdash e_2\rightarrow a_1,s_1 \quad c\in\{n,Nil\} \quad s_1(a_1)=region\langle\rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o)}{s,VE\vdash c\ at\ e_2\rightarrow a,\ s_1+\{a\mapsto c\}}$$ [E-Const]

$$\frac{s,VE\vdash e\rightarrow a_1,s_1 \quad s_1(a_1)=region\langle\rho',\_\rangle \quad o'\notin Dom\Big(s\big(\rho'\big)\Big) \quad a=\big(\rho',o'\big) \quad \rho\notin Dom(s_1)}{s,VE\vdash new\langle\rho\rangle\langle T\rangle\ at\ e\rightarrow a,s+\big\{\rho\mapsto\{o\mapsto ref\bot\}\big\}+\{a\mapsto region\langle\rho,o\rangle\}}$$ [E-NewReg]

$$\frac{s,VE\vdash e\rightarrow a_1,s_1 \quad s_1(a_1)=region\langle\rho,o\rangle \quad o\neq\bot \quad a=(\rho,o)}{s,VE\vdash valueOf\ e\rightarrow a,s_1}$$ [E-ValueOf]

$$\frac{\rho\notin Dom(s) \quad s+\big\{\rho\mapsto\{o\mapsto region\langle\rho,\bot\rangle\}\big\},VE+\{x\mapsto(\rho,o)\}\vdash e\rightarrow a,s_1}{s,VE\vdash letregion\langle\rho\rangle\ x\ in\ e\rightarrow a,s_1-\{\rho\}}$$ [E-LetReg]

$$\frac{s,VE \vdash e \to a_1,s_1 \quad s_1(a_1)=region\langle\rho,o\rangle \quad o\neq\bot}{s,VE \vdash transfer\ e \to \bot, s_1-\{\rho\}} \qquad \text{[E-Transfer]}$$

Notes on dynamic semantics:

- Side condition on all store accesses - $a \neq \bot$ and $a \neq (\rho, \bot)$
- Semantics does not capture safety condition on *transfer* completely. The pre-condition on *transfer* should be that region being transferred should not contain pointers to/into other regions. Currently, we pretend that *transfer* has same semantics as *free*.
- We elided rules for *let* and application expressions, as they are standard. They are similar to corresponding rules in Tofte &Talpin .
- *open .. in ..* expression is not relevant for dynamic semantics. It should be treated as a static construct required to ensure safety of memory accesses into transferable regions.
- One way to understand the semantics of a region handler is to treat it as an immutable reference to a record containing a single field named $value$, which is again a reference. If r is an immutable reference, then the expression $r := e$ is not allowed.

Examples:

In our examples, we use SML-style $let\ val\ x = e \ldots in\ e\ end$ expression to group multiple declarations together.

Example 1: Create a transferable region, Store a list in transferable region, and transfer it.

```
let
  val  $r_1 = new\langle\rho_1\rangle\langle int\ list\rangle\ at\ r_H$  (* region handler stored in heap region *)
  val  x =  open $r_1$ in
              let
                val l = $Cons\ int\ (1\ at\ r_1)\ (Cons\ int\ (2\ at\ r_1)\ (Nil\ int\ at\ r_1)\ at\ r_1)\ at\ r_1$
                val rval = valueOf $r_1$
              in
                rval := l
              end
           end
in
  transfer r1
end
```

Example 2:Receive a transferable region of type $int\ ref\ list$, access the list, perform computation, update the list, and transfer the region.

```
λ($r_1$: $Region\langle\rho\rangle\langle int\ ref\ list\rangle$).
    let
      val  x =  open $r_1$ in
                  let
                    val rval = valueOf $r_1$
                    val x = hd !rval
                    val y = hd (tl !rval)
                    val z = ref (!x + !y at $r_1$) at $r_1$
                    val l = Cons (int ref) z !rval at $r_1$
                  in
                    rval := l
                  end
               end
    in
      transfer r1
    end
```

Example 3: Receive a transferable region of type $_{i}int\ ref\ list$, access list, create a static region, create a list (l_2) with its spine in the static region, but refererring to elements of the list (l_1) in the transferable region, reverse the list l_2, perform computation on its head, update the list l_1, and transfer the region.

```
λ(r: Region⟨ρ⟩⟨int ref list⟩).
    let
      val x =
          open r in
            letregion⟨ρ_s⟩ r_s in
                let
                  val rval = valueOf r
                  val l_1 = !rval
                  val l_2 = rev (int ref) r_s l_1 (* Since calculus has no fixpoint, assume that rev is predefined. *)
                      (*  Also, assume that rev stores output list in the  static region, whose handler (r_s) is
                          provided as an argument to rev *)
                  val y = ref (2 * !(hd l_2) at r) at r
                  val l_1'=Cons (int ref) y  l_1 at r
                in
                  rval:= l_1'
                end
            end
          end
    in
      transfer r
    end
```

- Consider a variant of above example, where r_s is substituted for r in the expression: *ref (2 * !(hd* l_2) *at r) at r* . Type checking the subsequent expression (*Cons (int ref) y* l_1) should fail, as it is trying to store a pointer into static region (r_S) in transferable region *r*. Transferable regions must be closed under referencing.
- Inverting the order of *open r in … end* and *letregion .. in .. end* in the above example should result in type error. The reason being that after such inversion, static region (r_S) outlives the current incarnation of transferable region (*r*) resulting in the possbility of dangling references from the static region.
- Why do we need *open … in … end* ? *open* block serves the same purpose as *let*! block in Wadler's original formulation of linear types - to identify lexical blocks where the linear resource is accessed in a read-only fashion. In the context of the calculus, a dynamic region handler (eg: r above) is the linear resource. Using the handler to allocate objects into the region (eg: *at r*) is considered its read-only use, whereas using it to *free* or *transfer* the region is considered its destructive use. Inside an *open* block, the region handler is treated as any other intuitionistic value, thereby 1. allowing aliasing, and 2. preventing destructive updates (i.e., *transfer* or *free*). Consequently, when a dynamic region is open, its region handler can be freely copied and passed to functions, which allocate objects into the region. Without the open block, passing a region handler to a function (eg: *rev*) essentially "consumes" the region handler resource. If the caller has to gain the resource back, the called function (eg: *rev*) has to explicitly return the region handler along with its output (eg: reversed list). Since most functions only allocate objects into the region (hence, use it in a read-only way), explicitly returning the handler is an overkill. Opening a region prevents this by allowing callee to retain the region handler even after passing it to a function that uses it in a read-only way. Further, an open dynamic region cannot be deallocated; so, any region with lesser lifetime than the open block (For eg, r_s in the above example) can safely contain references to objects inside the dynamic region. Thus, *open* block helps increase expressitivity and productivity without losing safety by temporarily relaxing the linearity requirement.

Example 4: Consider a variant of Example 3, where we replace the static region with another transferable region (r'):

```
λ(r: Region⟨ρ⟩⟨int ref list⟩).
    let
      val r' = new⟨ρ'⟩⟨int ref list⟩ at r_H
      val x =
          open r in
            open r' in
                let
                  val rval = valueOf r
                  val l_1 = !rval
```

```
                    $val\ l_2 = rev\ (int\ ref)\ r'\ l_1$
                    $val\ y = ref\ (2 * !(hd\ l_2)\ at\ r)\ at\ r$
                    $val\ l_1' = Cons\ (int\ ref)\ y\ \ l_1\ at\ r$
                  $in$
                    $rval := l_1'$
                  $end$
              $end$
            $end$
      $in$
        $transfer\ r$
      $end$
```

- The example should fail the type check as we have inter-region pointers between transferable regions.

Example 5: Create some regions, each storing an integer or an integer list. Store their region handlers as a list (l_r) in the heap:

```
  $let$
    $val\ l_r = Cons\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ (new\langle\rho_1\rangle\langle int\rangle\ at\ r_H)$
              $(Cons\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ (new\langle\rho_2\rangle\langle int\ list\rangle\ at\ r_H)$
                 $(Nil\ \ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ at\ r_H)\ \ at\ r_H)\ at\ r_H$
    (* Note: 1. All region handlers stored in the heap region,
             2. The list itself is stored in the heap region,
             3. We introduced existential typing (not yet in calculus) to make the list well-typed
             4. Since region handlers are linear, the list itself must be linear. Recall that an intuitionistic
                value cannot contain linear values. *)
    $val\ x = case\ l_r\ of\ Nil \Rightarrow ()$
      $|\ Cons\ r_0\ Nil\ \ \Rightarrow\ \ ()$   (* Merged successive pattern matches into one top-level match *)
      $|\ Cons\ r_0\ (Cons\ r_1\ Nil) \Rightarrow\ \ open\ r_0\ in$
                    $valueOf\ r_0\ \ := 2\ at\ r_0$
                  $end;$
                  $open\ r_1\ in$
                    $let$
                      $val\ l = Cons\ int\ (1\ at\ r_1)\ (Cons\ int\ (2\ at\ r_1)\ (Nil\ int\ at\ r_1)\ at\ \ r_1)\ at\ r_1$
                      $val\ rval = valueOf\ r_1$
                    $in$
                      $rval := l$
                    $end$
                  $end;$
                  $transfer\ r_0;$
                  $transfer\ r_1;$
  $in$
    $x$
  $end$
```

Example 6: A modification of previous example. We create aliases for regions before adding them to the list:

```
  $let$
    $val\ r_1 =\ new\langle\rho_1\rangle\langle int\rangle\ at\ r_H$
    $val\ r_2 =\ new\langle\rho_2\rangle\langle int\ list\rangle\ at\ r_H$
    $val\ l_r = Cons\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ r_1$
              $(Cons\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ r_2$
                 $(Nil\ \ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\ at\ r_H)\ \ at\ r_H)\ at\ r_H$
    (* Linear values $r_1$ and $r_2$ have been "used" in the above line. They no longer serve as
    evidences for liveness of regions ρ_1 and ρ_2, respectively.*)
  val l'_r = rev (∃ρ. ∃T. Region⟨ρ⟩⟨T⟩) r_H l_r (* As noted previously, rev stores the result list in heap region,
                  whose handler (r_H) is passed to rev. *)
  (* Continutation is same as previous example, except that l'_r is used instead of l_r *)
  ...
  ...

in
  ...
end
```

- The example is well-typed
- Using r_1 or r_2 in the continuation makes the example ill-typed.
- rev consumes its input list (l_r), if l_r is a linear value. A possible type is:
  $$rev : \forall a. a\,!\,list \ \rightarrow a\,!\,list.$$
  However, rev need not necessarily take linear values. We can parameterize its type over linearity requirement:
  $$rev : \forall \phi. \forall a\,.\,a\ list^{\phi} \rightarrow \ a\ list^{\phi}$$
  where, $\phi \in \{\cdot, 1\}$, where $\cdot$ denotes intuitionistic type, and 1 denotes linear type. Similarly, for concat:
  $$concat : \forall \phi. \forall a\,.\,a\ list^{\phi} \rightarrow \ a\ list^{\phi} \ \ \rightarrow^{\phi} a\ list^{\phi}$$
  where, $\rightarrow^{\phi}$ denotes 1) $\multimap$ when $\phi = 1$, and 2) $\rightarrow$ when $\phi = \cdot$
- Using l'_r instead of l_r in the continuation makes the example ill-typed.

Calculus V3

Monday, June 9, 2014 1:58 PM

- Calculus V2 has no mutable linked-lists. This causes problems, as enforcing linearity with immutable lists is difficult. With mutable lists, we can simply insist on using *pop* to access an element in the list.

  $n \in integers$
  $x \in variables$
  $\rho \in region\ identifiers$
  $a \in addresses$
  $\kappa\ ::=\ \mathcal{R} \mid \Theta$ //kinds
  $T\ ::= int \mid unit \mid \alpha \mid \alpha\ list \mid T \to T$
  $\sigma\ ::= \forall \alpha. T \mid T$
  $e\ ::= x \mid e\ e \mid \lambda(x{:}T).e\ at\ e \mid n\ at\ e \mid e \otimes e\ at\ e \mid \emptyset\ \langle T \rangle\ at\ e \mid push\ (e,e)\ \mid pop\ e$
  $\mid new\langle\rho\rangle\langle T\rangle\ at\ e \mid transfer\ e \mid open\ e\ in\ e \mid letregion\langle\rho\rangle\ x\ in\ e \mid ref\ e\ at\ e$
  $\mid e\ T \mid \Lambda\alpha.e \mid let\ x = e\ in\ e \mid valueOf\ e\ \mid !\,e \mid e \coloneqq e$
  $sv ::= n \mid Nil \mid Cons\ (v,v) \mid region\langle\rho,o\rangle \mid ref\ v \mid \langle x,e,VE\rangle$ // Storable Values
  $v\ ::= a$ //values

  Remarks about calculus:

- Language has references.
- All region handlers are first class. They can be stored, passed around, can be returned from functions.
- We have higher-order functions (hence, closures).
- We dropped pairs (from calculus v1), but added mutable lists to model potentially unbounded data structures. Expression $\emptyset\ \langle T\rangle$ creates a reference to an empty list (of type T). Operators $push$ and pop can be used to destructively update the list.
- Lists are made polymorphic so that it is possible to store region handlers in a list.
- The type/kind language is not yet complete. In this article, we only concentrate on dynamic semantics. We will develop the type/kind language when we start working on static semantics.
- Unit type is to assign type to return value of *transfer*
- $valueOf$ operator returns the reference to root of a region, given a region handler.
  $$valueOf : \forall\alpha.\forall\rho. Region\langle\rho\rangle\langle\alpha\rangle \to ref\ \alpha$$
  Where are $Region\langle\rho\rangle\langle\alpha\rangle$ and $ref\ \alpha$ stored? Type of $valueOf$ should be a dependent type:
  $$valueOf : \forall \tau_1 :: \mathcal{R}.\forall\alpha.\forall\rho.\ r_1{:}\tau_1 \to r_2{:}(Region\langle\rho\rangle\langle T\rangle, r_1) \to (ref\ T, r_2)$$
  $\mathcal{R}$ is the kind of region handler types.
- When a region is newly allocated, $valueOf$ returns an invalid reference. Dereferencing such reference, or transferring a region whose root is not initialized must be a type error.
- It is not yet clear which universe ρ belongs to (value/type/kind ?).
- A region polymorphic function manifests as a function expecting static region handlers as arguments. We retain prenex quantification. When type language is developed, syntax of arrow types should be modified reflect this.
- We use ρ_H as heap region identifier, and r_H as heap region handler.
- What are values in the calculus? Similar to Tofte and Talpin (T&T), all our values are boxed; so, all our values are addresses. We define a syntactic class of storable values, which is the co-domain of the store (s) in dynamic semantics.
- Observe that there are some storable values that are not part of the expression language. An example is the runtime value of a region handler ($region\langle\rho,o\rangle$).

Dynamic Semantics:

- Our dynamic semantics are store semantics . Reduction rules are shown below:

$$\boxed{s, VE \vdash e \rightarrow v, s}$$

$$\frac{VE(x)=a}{s,VE\vdash x \rightarrow a,s} \qquad \text{[E-Var]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_2\rightarrow a_1,s_1 \\ s_1(a_1)=region\langle \rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o) \end{array}}{s,VE \vdash \lambda(x{:}T).e_1\ at\ e_2\rightarrow a,\ s_1+\{a\mapsto\langle x,e_1,VE\rangle\}} \qquad \text{[E-Abs]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1,VE\vdash e_2\rightarrow a_2,s_2 \\ s_1(a_2)=region\langle \rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o) \end{array}}{s,VE \vdash ref\ e_1\ at\ e_2\rightarrow a,\ s_2+\{a\mapsto ref\ a_1\}} \qquad \text{[E-Ref]}$$

$$\frac{s,VE\vdash e\rightarrow a_1,s_1 \quad s_1(a_1)=ref\ a}{s,VE \vdash\ !e \rightarrow a,\ s_1} \qquad \text{[E-DeRef]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1(a_1)=ref\ a \\ s_1,VE\vdash e_2\rightarrow a_2,s_2 \end{array}}{s,VE \vdash e_1{:=}e_2 \rightarrow \bot,\ s_2+\{a_1\mapsto ref\ a_2\}} \qquad \text{[E-Assn]}$$

$$\frac{\begin{array}{c} s,VE\vdash e\rightarrow a_1,s_1 \\ s_1(a_1)=region\langle \rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o) \end{array}}{s,VE \vdash \emptyset\ \langle T\rangle\ \ at\ e\rightarrow a,\ s_1+\{a\mapsto ref\ Nil\}} \qquad \text{[E-Nil]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_1\rightarrow a_1,s_1 \quad s_1,VE\vdash e_2\rightarrow a_2,s_2 \quad a_2=(\rho_2,o_2) \\ s_2(a_2)=ref\ a \quad o\notin Dom\big(s(\rho_2)\big) \quad a'=(\rho_2,o) \end{array}}{s,VE \vdash push\ (e_1,e_2) \rightarrow\ \bot,\ s_2+\{a'\mapsto Cons\ (a_1,a)\}+\{a_2\mapsto ref\ a'\}} \qquad \text{[E-Push]}$$

$$\frac{s,VE\vdash e \rightarrow a,s' \quad s'(a)=ref\ a' \ \ s'\big(a'\big)=Cons\ (a_1,a_2)}{s,VE \vdash pop\ e \rightarrow a_1,\ s_2+\{a\mapsto ref\ a_2\}} \qquad \text{[E-Pop]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_2\rightarrow a_1,s_1 \\ s_1(a_1)=region\langle \rho,\_\rangle \quad o\notin Dom\big(s(\rho)\big) \quad a=(\rho,o) \end{array}}{s,VE \vdash n\ at\ e_2\rightarrow a,\ s_1+\{a\mapsto n\}} \qquad \text{[E-Const]}$$

$$\frac{\begin{array}{c} s,VE\vdash e_2\rightarrow a_1,s_1 \quad s_1(a_1)=region\big\langle \rho',\_\big\rangle \\ o'\notin Dom\Big(s\big(\rho'\big)\Big) \quad a=\big(\rho',o'\big) \quad \rho\notin Dom(s_1) \quad fresh(o) \end{array}}{s,VE \vdash new\langle\rho\rangle\langle T\rangle\ at\ e \rightarrow a,s+\big\{\rho\mapsto\{o\mapsto ref\bot\}\big\}+\{a\mapsto region\langle\rho,o\rangle\}} \qquad \text{[E-NewReg]}$$

$$\frac{s,VE \vdash e \rightarrow a_1,s_1 \quad s_1(a_1)=region\langle\rho,o\rangle \quad o \neq \perp \quad a=(\rho,o)}{s,VE \vdash valueOf\ e \rightarrow a,s_1} \qquad \text{[E-ValueOf]}$$

$$\frac{\rho \notin Dom(s) \quad s+\{\rho \mapsto \{o \mapsto region\langle\rho,\perp\rangle\}\},VE+\{x \mapsto (\rho,o)\} \vdash e \rightarrow a,s_1}{s,VE \vdash letregion\langle\rho\rangle\ x\ in\ e \rightarrow a,s_1-\{\rho\}} \qquad \text{[E-LetReg]}$$

$$\frac{s,VE \vdash e \rightarrow a_1,s_1 \quad s_1(a_1)=region\langle\rho,o\rangle \quad o \neq \perp}{s,VE \vdash transfer\ e \rightarrow \perp,s_1-\{\rho\}} \qquad \text{[E-Transfer]}$$

Notes on dynamic semantics:

- Side condition on all store accesses - $a \neq \perp$ and $a \neq (\rho, \perp)$
- Semantics does not capture safety condition on *transfer* completely. The pre-condition on *transfer* should be that region being transferred should not contain pointers to/into other regions. Currently, we pretend that *transfer* has same semantics as *free*.
- We elided rules for *let* and application expressions, as they are standard. They are similar to corresponding rules in Tofte &Talpin .
- *open .. in ..* expression is not relevant for dynamic semantics. It should be treated as a static construct required to ensure safety of memory accesses into transferable regions.
- One way to understand the semantics of a region handler is to think of it as an immutable reference to a record containing a single field named $value$, which is again a reference. If r is an immutable reference, then the expression $r \coloneqq e$ is not allowed.

Examples:

In our examples, we use SML-style $let\ val\ x = e\ \ldots in\ e\ end$ expression to group multiple declarations together.

Example 1: Create a transferable region, Store a list in transferable region, and transfer it.

```
let
  val r_1 = new⟨ρ_1⟩⟨int list⟩ at r_H  (* region handler stored in heap region *)
  val x = open r_1 in
              let
                val l = push (1at r_1, push (2 at r_1, ∅ ⟨int⟩ at r_1)
                val rval = valueOf r_1
              in
                rval ≔ ! l
              end
          end
in
  transfer r1
end
```

Example 2:Receive a transferable region of type $int\ ref\ list$, access the list, perform computation, update the list, and transfer the region.

```
λ(r_1: Region⟨ρ⟩⟨int ref list⟩).
    let
      val x = open r_1 in
                  let
```

```
                        $val\ rval = valueOf\ r_1$
                        $val\ x = pop\ rval$
                        $val\ y = pop\ rval$
                        $val\ z = ref\ (!x + !y\ at\ r_1)\ at\ r_1$
                      $in$
                        $push\ (z, push\ (y, push\ (x, rval)))$
                      $end$
                  $end$
        $in$
          $transfer\ r1$
        $end$
```

Example 3: Receive a transferable region of type $int\ ref\ list$, access list, create a static region, create a list (l_2) with its spine in the static region, but refererring to elements of the list (l_1) in the transferable region, reverse the list l_2, perform computation on its head, update the list l_1, and transfer the region.

```
    $\lambda(r{:}\ Region\langle\rho\rangle\langle int\ ref\ list\rangle)$.
        let
          val x =
              open r in
                $letregion\langle\rho_s\rangle\ r_s$ in
                    let
                      val rval = valueOf r
                      val $l_2$ = rev (int ref) $r_s$ rval (* Since calculus has no fixpoint, assume that rev is predefined. *)
                            (*  Also, note that rev creates a new list in the  static region, whose handler ($r_s$) is
                                provided as an argument to rev *)
                      val y = ref (2 * !(pop $l_2$) at r) at r
                    in
                      push (y, rval)
                    end
                end
              end
        in
          transfer r
        end
```

- Consider a variant of above example, where r_s is substituted for r in the expression: *ref (2 * !(hd l_2) at r) at r* . Type checking the subsequent expression ($push\ (y, rval)$) should fail, as it is trying to store a pointer into static region (r_s) in transferable region *r*. Transferable regions must be closed under referencing.
- Inverting the order of *open r in ... end* and *letregion .. in .. end* in the above example should result in type error. The reason being that after such inversion, static region (r_S) outlives the current incarnation of transferable region (r) resulting in the possbility of dangling references from the static region.

Example 4: Consider a variant of Example 3, where we replace the static region with another transferable region (r'):

```
    $\lambda(r{:}\ Region\langle\rho\rangle\langle int\ ref\ list\rangle)$.
        let
          val r' = $new\langle\rho'\rangle\langle int\ ref\ list\rangle$ at $r_H$
          val x =
              open r in
                open r' in
                    let
                      val rval = valueOf r
                      val $l_2$ = rev (int ref) $r'$ rval
                      val y = ref (2 * !(hd $l_2$) at r) at r
                    in
                    $push\ (y, rval)$
                  $end$
              $end$
            $end$
      $in$
        $transfer\ r$
      $end$
```

- The example should fail the type check as we have inter-region pointers between transferable regions.

Example 5: Create some regions, each storing an integer or an integer list. Store their region handlers as a list (l_r) in the heap:

```
let
  $val\ l_r = \emptyset\ \langle\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\rangle\ at\ r_H)$
  $val\ x = push\ (new\langle\rho_1\rangle\langle int\rangle\ at\ r_H,$
               $push\ (new\langle\rho_2\rangle\langle int\ list\rangle\ at\ r_H,\ \ l_r))$
  (* Note: 1. All region handlers stored in the heap region,
           2. The list itself is stored in the heap region,
           3. We introduced existential typing (not yet in calculus) to make the list well-typed *)
  $val\ r_0 = pop\ l_r$
  $val\ x = open\ r_0\ in$
             $valueOf\ r_0\ \coloneqq 2\ at\ r_0$
           $end$
  $val\ r_1 = pop\ l_r$
  $val\ \ x = \ open\ (pop\ l_r)\ as\ r_1\ in$
             $valueOf\ r_1 \coloneqq !\ (push\ (1\ at\ r_1, push\ (2\ at\ r_1,\ \ \emptyset\ \langle int\rangle\ at\ r_1)))$
           $end$
in
  $transfer\ (r_0)$; (* Sequencing expression. Should be treated as syntactic sugar. *)
  $transfer\ (r_1)$
end
```

Example 6: A modification of previous example. We create aliases for regions before adding them to the list:

```
let
  $val\ r_1 = \ new\langle\rho_1\rangle\langle int\rangle\ at\ r_H$
  $val\ r_2 = \ new\langle\rho_2\rangle\langle int\ list\rangle\ at\ r_H$
  $val\ l_r = \emptyset\ \langle\ (\exists\rho.\exists T.Region\langle\rho\rangle\langle T\rangle)\rangle\ at\ r_H)$
  $val\ x = push\ (r_1, push\ r_2, l_r))$
  …
  …
in
  $transfer\ (r_0)$;
  $transfer\ (r_1)$
end
```

- Linearity implies that once r_1 and r_2 are pushed into the list, the names r_1 and r_2 lose their bindings.
- Observe that r_1 rebound later. This is perfectly fine. The example should pass the type check.

Example 7: Consider the case where one might want to access region handlers in the reverse order. We store another list of region handlers in the heap (l_r') which is equal to $rev\ l_r$.

```
let
  val $l_r = \emptyset \langle (\exists \rho. \exists T. Region\langle \rho \rangle \langle T \rangle) \rangle$ at $r_H$)
  val $x$ = push (new$\langle \rho_1 \rangle \langle int \rangle$ at $r_H$,
                 push (new$\langle \rho_2 \rangle \langle int\ list \rangle$ at $r_H$,   $l_r$))
  val $l'_r$ = rev $(\exists \rho. \exists T. Region\langle \rho \rangle \langle T \rangle)$ $r_H$ $l_r$ (* As noted previously, rev creates a new list  in the heap region,
                 whose handler ($r_H$) is passed to rev. *)
in
  transfer (pop $l_r$);
  transfer (pop $l'_r$) (* Notice the use of reversed list here. *)
end
```

- The example should fail the type check, as rev duplicates region handlers of l_r in l'_r, violating the linearity requirement on region handlers.
- But, precisely what type check should fail? The application of *rev* should fail the type check, as *rev* is being applied to a list of non-duplicatable values, whereas *rev* requires that the elements in its input list be duplicatable.
- What should be the type of *rev*, such that it specifies the aforementioned pre-condition?
    - Let us introduce kind $\kappa_{.}$ of linear types, and kind $\kappa_{\circ}$ of regular types. The type of region handler should be of kind $\kappa_{.}$ (i.e., $Region\langle \rho \rangle \langle T \rangle :: \kappa_{.}$).Then, the type of rev should be:
    $$rev : \forall (a :: \kappa_{\circ}).\ a\ list \rightarrow a\ list$$
    Now, the type application ($rev\ (\exists \rho. \exists T. Region\langle \rho \rangle \langle T \rangle)$) fails the typecheck (kindcheck, to be precise).
    - In (A Relational Framework for Higher-Order Shape Analysis, ICFP'14), authors show how to assign precise type to *rev*, such that the type declares that resulting list contains same elements as the input list. Such a type clearly declares that input values are duplicated/aliased in output list, and can also be used to prevent the *rev* application in this example.
- This example can be used to demonstrate problems due to uncontrolled aliasing, and the motivation for the linearity requirement: Let us assume that there is no linearity requirement, and the above example is well-typed. Consider a case where we iterate through l_r and transferred some region. Now, we iterate through l'_r and allocate objects in some of the regions. For this code to be safe, it must be the case that the set of regions where allocations happened should not contain the region transferred in first iteration. Since it is very difficult (if not impossible) to statically locate the freed region in l'_r, we cannot be sure that the code is safe.

Region Constraints

Monday, June 9, 2014 1:37 PM

Kapil's summary of constraints on region system, with some added comments:

1. "Regions are typed. Each region has a root object that can be accessed using a special property of the region object."
   - My guess is that primary motivation for this is that ease of garbage collection. A collector can collect all such objects in the region that are unreachable from the root.
   - This constraint is also sensible at the high-level, as a transferable region is semantically a message, therefore expected to have a well-defined structure.
2. "Objects in (immortal region)/(GC heap) can store references to other regions, but cannot reference objects inside other regions. This simplifies the type system, and will hold us in a good stead when we GC heap memory."
   - Let us consider a region type system with Tofte & Talpin style *decorated types* (eg: (int, ρ)), such that type of an object identifies the region ($\boldsymbol{\rho}$) to which it belongs. In case of reference types, the type identifies the region to which referred object belongs, along with the region where the reference itself is stored (eg: $(ref\ (int, \rho_1), \rho_2)$). Assume there are no recursive datatypes (and fixpoint operator) in the language. Under this setting, if heap region contains references to objects within other regions, the type of such reference clearly identifies the region (through its lexically manifest identifier $\boldsymbol{\rho}$) to which it is referring. If this region is freed, the type system can update the type(state) of references to objects in this region to convey that the pointers are now dangling. Clearly, there is no need to restrict heap from storing pointers to objects in other regions under this language setting. Now, let us add polymorphic *list* recursive datatype to the language. Since heap can now store lists, assume that heap stores a list of references to objects in other regions. However, since type system requires that all references in the list have same (decorated) type, all references should refer to objects of same type in same region. For example, such a list would have following type: $((ref\ (int, \rho_1), \rho_2)\ list, \rho_H)$, which indicates that spine of the list is stored in heap region (ρ_H), and list contains references to integers in region ρ_1, where references themselves are stored in region ρ_2. Now, if either of ρ_2 or ρ_1 are freed, the type system can update the type(state) of the list stored in heap, and can subsequently flag either accessing a reference in the list, or dereferencing such reference (resp.) as a type error. Therefore, it seems that even with a language containing recursive datatypes (or a fixpoint operator), there is no need to constrain heap from having pointers to objects in other regions. However, this conclusion is wrong as, with the extended language, region identifiers (ρ) need not be lexically manifest. For instance, consider a program that (1). creates several transferable regions, (2). stores region handlers in a list l_1, (3). iterates through the list l_1 while flipping a coin, and when the coin lands heads for the first time, (4). stores several objects in the region at current position of the list, and stores references to such objects as a list (l_2) in the heap region, and (5). exits. Now, what is the type of the list l_2? List l_2 contains references to objects stored in *some* region, whose region handler is contained in l_1, but not lexically manifest in the program. We can assign an existentially quantified type to l_2, but it is very imprecise - if list l_1 is unfolded again, and some region (handler) in the list is freed/transferred, we cannot be sure if references in list l_2 refer to objects in freed region or not.
     A straightforward way to circumvent this problem is to avoid inter-region pointers, which is achieved through restrictions in (2) and (4). However, region handlers (or pointers to the roots of regions) have to be maintained somewhere, lest it leads to space leak. Therefore, let us relax the restriction, and allow region handlers to be stored in other regions. But, it leads to the same problem as before. To see how, consider a case when we have two lists containing region handlers to same set of regions, but in different order. We unfold one list and free one of the regions. Now, which region handler in 2nd list is dangling? Observing that root of this problem is aliasing, we forbid aliasing of region handlers, which is achieved through restriction in (3).
3. "References to regions are linear, i.e., at any point, there can only be one reference to a region."
   - This solves the aliasing problem.
4. "Objects in transferable regions cannot store references to other regions. This is to ensure that we can collect a region independently without having to worry about regions that may be referenced by objects in freed regions".
5. "There may be references to objects within a region from stack variables. We shall provide a mechanism to identify lexical blocks where such references exist. The type system will ensure that regions cannot be freed from within a code region where a reference to an object in that region may exist".

Source Language Example

Modified and linear/region type-annotated version of Dimitrios's example.

- Some constraints/features, which are not required/supported by our current type system were striken off.
- The implicit "this" parameter to methods of a class is made explicit. Some methods of $Region$ class expect the "$this$" object to be linear, whereas some expect it to be non-linear.
- Linear types have 1 as superscript (eg:$Rgn^1\langle\rho\rangle\langle T\rangle$).
- Typing assertions are shown in blue coloured background.

- `class Region<T> {`

    `public Region<T>(`~~`RegionType t`~~`);  // Creates a new region of type 'Transferrable'` ~~`(could also be 'Static' and 'LongLived')`~~ `- 1. This class offers methods which only make sense for transferable regions. So, it is only natural that this class creates transferable regions. Furthermore, 2. We want to restrict the other type of regions to be only used via OpenAlloc.`

    `public T GetData();              // Gets the root of this region`
    `//` $GetData :\ this\!: Rgn\langle\rho\rangle\langle T\rangle \rightarrow unit$

    `public GiveUp();                 // Gives up references to this region (i.e. can never open it again)`
    `//` $GiveUp :\ this\!: Rgn^1\langle\rho\rangle\langle T\rangle \rightarrow\ unit$

    `public Transfer(Actor a);        // Gives up references to this region (i.e. can never open it again) and transfers`
    `//` $Transfer : (this\!: Rgn^1\langle\rho\rangle\langle T\rangle\ , a\!: Actor) \rightarrow unit$

    `public SuckObject(Object o);     // Deeply clones an object from a region` ~~`that outlives the current one to this one`~~
    `// ***` ~~`Precondition: transitive closure of object 'o' must be allocated in regions that outlive 'this'`~~ `- When we are deeply cloning an object, it is not clear why we require the cloned object to be in`

```
        a region which outlives transferable region.
            // SuckObject : (this: Rgn⟨ρ⟩⟨T⟩, o: Object) → unit

        public SetData(T t);              // Sets the root pointer of a region.
                                           // *** Precondition: ‘t’ must be allocated in ‘this’
            // SetData : (this: Rgn⟨ρ⟩⟨T⟩, t: T@ρ) → unit
    }
```

The following methods are supposed to be used only with ‘using’ (see example)

- `RegionContext.Open(Region<T> r)      // opens a region for accessing contents`
- `RegionContext.OpenAlloc(Region<T> r) // opens a region for access and also guarantees that all allocations happen in that region (inAllocCtxt)`
- `RegionContext.OpenAlloc()            // opens a temporary (statically scoped) region (just a shorthand for create new region and OpenAlloc it))`

The above methods temporarily update the typestate of the region handler they accept without *consuming* the region handler. We cannot assign a sensible type to these methods in our linear type system, for they effectively simulate the "*let*!"construct (Wadler'90) for linear resources, which is a meta-level entity that cannot be expressed in the language itself.

```
// Accepts a region that hosts a batch of records, and a UDF to process them with
void select(Region<Batch<Record>> rin, Func<Record,OutRec> udf) {

   // The method select is region-polymorphic with one region parameter (identified as ρ_iac), such that its
   // arguments and any temporary objects that it creates are allocated in ρ_iac. That is, ρ_iac identifies
   // in-allocation-context of select.
   // rin : Rgn^1⟨ρ_in⟩⟨Batch⟨Record⟩⟩
   // Observe that the type of rin is not decorated. Where is rin stored? We assume that rin is stored in the
   // in-allocation-context of the current method call. We adopt the notation that if ρ identifies the
   // region corresponding to in-allocation-context in T@ρ, we simply write T instead of T@ρ.

   // Let us create a new transferrable region
   var rtrans = new Region<Batch<OutRecord>>(); // rtrans : Rgn^1⟨ρ_tr⟩⟨Batch⟨OutRecord⟩⟩

   // Let us open the received region for access
   using (RegionContext.Open(rin)) {
        // rin : Rgn⟨ρ_in⟩⟨Batch⟨Record⟩⟩
       // in here we are allowed to refer to objects in rin

       var batch = rin.GetData();
      // batch : Batch⟨Record@ρ_in⟩@ρ_in

      // let us initialize the batch in the transferrable region
      // notice the ‘using’ which ensures that the new batch object
      // is indeed allocated in rtrans.

      using (RegionContext.OpenAlloc(rtrans)) {
            // rtrans : Rgn⟨ρ_tr⟩⟨Batch⟨OutRecord⟩⟩
            rtrans.SetData(new Batch<OutRecord>());
            // Since, inAllocCtxt is a region with Id ρ_tr:
            // new Batch<OutRecord>() : Batch⟨OutRecord@ρ_tr⟩@ρ_tr
       }
      // rtrans : Rgn^1⟨ρ_tr⟩⟨Batch⟨OutRecord⟩⟩

       // Creates and opens a new un-named region to be used statically
       // and performs all allocations in it
       using (RegionContext.OpenAlloc()) {
            // Assume that newly allocated static region has id ρ_0, which is the
            // the new inAllocCtxt.
            // in here all allocations are done in an un-named temporary region (ρ_0)
            Iterator ix = new Iterator(batch); // ok to store pointer to outer ‘batch’
            //  ix : Iterator⟨Record@ρ_in⟩@ρ_0
            foreach Record x in ix {
                   // x : Record@ρ_in
                   // y will be in the temp region, as well as all allocations of the UDF
                   var y = udf(x);
                   // y : OutRecord@ρ_0
                   // but now open the ‘rtrans’ region to use for allocation
                   using (RegionContext.OpenAlloc(rtrans)) {
                     // rtrans : Rgn⟨ρ_tr⟩⟨Batch⟨OutRecord⟩⟩
                     var z = rtrans.SuckObject(y); // z is now on rtrans
                     // z : OutRecord@ρ_tr
                     rtrans.GetData().Append(z); // ... and hence can be appended safely
```

```
                    // The call typechecks because:
                    // $rtrans.GetData() : Batch\langle OutRecord@\rho_{tr}\rangle@\rho_{tr}$
                  }
           }
     }
  }
  // $rin :\ Rgn^1\langle\rho_{in}\rangle\langle Batch\langle Record\rangle\rangle$
  rin.GiveUp(); // no longer need to have a handle on rin
  // $rin$ is now consumed.
  rtrans.Transfer(this.getDownStreamActor()); // same with rtrans but also transfer

}
```

T&T with refs and subtyping

Tuesday, June 17, 2014 10:12 PM

Before we attempt full static semantics of our calculus, let us first consider Tofte and Talpin language extended with references and subtyping based on region lifetimes, and give type rules for the language. This is interesting for two reasons:

1. The original T&T paper presents elaboration rules from ML to ML+ρ as static semantics. No explicit type rules were given. It would be useful to formulate type rules for the elaborated language (ML+ρ) independent of the elaboration rules.
2. Adding references to the language introduces new non-trivial ways of creating dangling references. Absence of dynamic regions (and linear typing) in T&T lets us concentrate on this problem.

Monomorphic Language

We first consider a simply typed region-monomorphic language.

The Language:

$\rho \in \ \ region\ names$
$\tau \ ::= int@\rho \mid unit \ \mid \tau\ ref@\rho \mid \tau_1 \to \tau_2\ @\ \rho$
$e \ ::= x \mid e\ e \mid n\ at\ \rho \mid \lambda(x{:}\tau).e\ \ at\ \rho \mid letregion\ \rho\ in\ e \mid e \otimes e\ at\ \rho \mid let\ x = e\ in\ e$
$\quad \mid ref\ \ e\ at\ \rho \mid\ !\,e \mid e \coloneqq e$

- Note that we do not distinguish between run-time value of a region, and its name in our language. Therefore, the calculus should be considered stable under substitution.

Type Rules (Incorrect):

$$\frac{\Gamma(x)=\tau}{\Delta;\Gamma\vdash x{:}\tau}\ [\text{T-Var}] \qquad \frac{\Delta;\Gamma\vdash e_2{:}\tau \quad \Delta;\Gamma\vdash e_1{:}\tau\to\tau_2\ @\ \rho}{\Delta;\Gamma\vdash e_1\ e_2{:}\ \tau_2}\ [\text{T-App}] \qquad \frac{\Delta;\Gamma,x{:}\tau_1\vdash e:\tau_2 \quad \Delta\vdash\tau_1 \quad \rho\in\Delta}{\Delta;\Gamma\vdash\lambda(x{:}\tau_1).e\ \ \text{at}\ \rho{:}\ \tau_1\to\tau_2@\rho}\ [\text{T-Abs}]$$

$$\frac{\rho\in\Delta}{\Delta;\Gamma\vdash n\ \text{at}\ \rho : int@\rho}\ [\text{T-Int}] \qquad \frac{\Delta;\Gamma\vdash e_1{:}int@\rho_1\ \ \Delta;\ \ \Gamma\vdash e_2{:}int@\rho_2 \quad \rho\in\Delta}{\Delta;\Gamma\vdash e\otimes e\ \text{at}\ \rho : int@\rho}\ [\text{T-IntOp}]$$

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

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

$$\frac{\Delta;\Gamma\vdash e_1{:}\tau_1 \qquad \Delta;\Gamma,x{:}\tau_1\vdash e_2 : \tau_2}{\Delta;\Gamma\vdash \text{let}\ x = e_1\ \text{in}\ e_2 : \tau_2}\ [\text{T-Let}]$$

Well-Formedness:

$$\frac{regOf(\tau)\subseteq\Delta}{\Delta;\Gamma\vdash\tau}\ [\text{WF-1}]$$

We define the function $regOf$ as following:

$regOf\ (int@\rho)$	=	$\{\rho\}$
$regOf\ (unit)$	=	$\emptyset$
$regOf\ (\tau\ ref@\rho)$	=	$regOf(\tau) \cup \{\rho\}$
$regOf\ (\tau_1 \rightarrow \tau_2\ @\ \rho)$	=	$regOf(\tau_1) \cup regOf(\tau_2) \cup \{\rho\}$

Counterexamples:

The language is unsafe in the sense that it allows dangling pointer dereferences. Following are counterexamples.

Example 1:
```
letregion r1 in
let f =
    letregion r2 in
    let y = 2 at r2 in
        λ(x:int@r1). (x + y) at r1 in
let z = f (1 at r1) in
…….…
```

When f is executed, y is read from $r2$, which is already deallocated. The type of f is $(int@r1 \rightarrow int@r1)@r1$, which is well formed when $\Delta = \{r1\}$.

Example 2: Previous example, with references:

```
letregion r1 in
let f =
    letregion r2 in
    let y = ref (2 at r2) at r1 in
        λ(x:int@r1). (x + !y) at r1 in
let z = f (1 at r1) in
…….…
```

Effect System:

Counterexamples demonstrate the need to track latent effects in a closure. Accordingly, we update our type language as following:
$v ::= \ get(\rho) |\ set(\rho)$
$\varphi \ \overset{\text{def}}{=}\ Finite\ set\ of\ v$
$\tau\ ::= int@\rho\ |\ unit\ \ |\ \tau\ ref@\rho\ |\ \tau_1 \rightarrow^{\varphi} \tau_2\ @\ \rho$

Similarly, we extend our typing judgment to track effects. The new rules are given below:

$\boxed{\Delta, \Gamma \vdash e :^{\varphi} \tau}$

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

$$\frac{\begin{array}{c}\Delta\vdash\tau_1 \quad \rho\in\Delta \\ \Delta;\Gamma,x{:}\tau_1\vdash e :^{\varphi_{12}}\tau_2 \quad \varphi=\{set(\rho)\}\end{array}}{\Delta;\Gamma\vdash\lambda(x{:}\tau_1).e \text{ at } \rho :^{\varphi} \tau_1 \rightarrow^{\varphi_{12}}\tau_2@\rho}\ \text{[T-Abs]} \qquad \frac{\rho\in\Delta \quad \varphi=\{set(\rho)\}}{\Delta;\Gamma\vdash n \text{ at } \rho : int@\rho}\ \text{[T-Int]}$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash e_1{:}^{\varphi_1} int@\rho_1 \ \Delta; \quad \Delta;\Gamma\vdash e_2{:}^{\varphi_2} int@\rho_2 \\ \rho\in\Delta \qquad \varphi=\varphi_1\cup\varphi_2\cup\{set(\rho)\}\end{array}}{\Delta;\Gamma\vdash e\otimes e \text{ at } \rho : int@\rho}\ \text{[T-IntOp]} \qquad \frac{\rho\notin\Delta \quad \Delta\cup\{\rho\};\Gamma\vdash e :^{\varphi}\tau \quad \Delta\vdash\tau}{\Delta;\Gamma\vdash \text{letregion } \rho \text{ in } e :^{\varphi}\tau}\ \text{[T-Reg]}$$

$$\frac{\Delta;\Gamma\vdash e{:}^{\varphi_1}\tau \quad \rho\in\Delta \quad \varphi=\{set(\rho)\}\cup\varphi_1}{\Delta;\Gamma\vdash(ref\ e)\ at\ \rho :^{\varphi} \tau\ ref@\rho}\ \text{[T-Ref]} \qquad \frac{\begin{array}{c}\Delta;\Gamma\vdash e :^{\varphi_1} \tau\ ref\ @\ \rho \\ \varphi=\{get(\rho)\}\cup\varphi_1\end{array}}{\Delta;\Gamma\vdash !e :^{\varphi}\tau}\ \text{[T-DeRef]}$$

$$\frac{\begin{array}{c}\varphi=\varphi_1\cup\varphi_2\cup\{set(\rho)\} \\ \Delta;\Gamma\vdash e_1{:}^{\varphi_1}\ \tau\ ref\ @\ \rho \quad \Delta;\Gamma\vdash e_2{:}^{\varphi_2}\tau\end{array}}{\Delta;\Gamma\vdash e_1{:=}e_2{:}^{\varphi}\ unit}\ \text{[T-Assn]} \qquad \frac{\begin{array}{c}\varphi=\varphi_1\cup\varphi_2 \\ \Delta;\Gamma\vdash e_1{:}^{\varphi_1}\tau_1 \quad \Delta;\Gamma,x{:}\tau_1\vdash e :^{\varphi_2}\tau_2\end{array}}{\Delta;\Gamma\vdash \text{let } x=e_1 \text{ in } e_2 :^{\varphi}\tau_2}\ \text{[T-Let]}$$

Well-formedness remains the same:

$\boxed{\Delta\vdash\tau}$

$$\frac{regOf(\tau)\subseteq\Delta}{\Delta;\Gamma\vdash\tau}\ \text{[WF-1]}$$

But, we update the definition of $regOf$ for function types to account for latent effects. The new definition of $regOf$ is as following:

$regOf\ (int@\rho)$	=	$\{\rho\}$
$regOf\ (unit)$	=	$\emptyset$
$regOf\ (\tau\ ref@\rho)$	=	$regOf(\tau)\cup\{\rho\}$
$regOf\ (\tau_1 \rightarrow^{\varphi} \tau_2\ @\ \rho)$	=	$regOf(\tau_1)\cup regOf(\tau_2)\cup frv(\varphi)\cup\{\rho\}$

Where, $frv(\varphi)$ denotes the set of free region variables in φ.

Now, we convince ourselves that previous two counterexamples are rejected by our type system:
Example 1: The type of f now is $(int@r1 \rightarrow^{\{get(r2)\}} int@r1)@r1$. Under $\Delta=\{r1\}$, the type of f is ill-formed, as

$$regsOf\big((int@r1 \rightarrow^{\{get(r2)\}} int@r1)@r1\big)=\{r1,r2\}$$

which is not a subset of $\{r1\}$. Type checking fails while applying T-Reg rule.
Example 2: The type of f now is $(int@r1 \rightarrow^{\{get(r2),get(r1)\}} int@r1)@r1$. The program is rejected for the same reason as above.

Subtyping and References

We have previously considered the extension of T&T with references to model some C examples from Cyclone (see related work). We reprise those examples, and provide some new examples below. We ignore effect typing for some time.

Example 1: Consider the following program in T&T's target language (extended with references):

```
letregion r1
in let p = letregion rL
           in let x = 0 at rL
              in (ref x) at r1
   in p := 42
```

The assignment on last line is an unsafe operation as p is a reference into region rL, which is now deallocated. Fortunately, the program is ill typed under T&T's region type system, as decorated type of expression $ref\ x$ is $((int@rL)\ ref@r1)$, which must be the type of p as per the type rule of let expression; but, such a type for p is ill-formed as region rL is not in scope.

Example 2 : This example demonstrates the need for subtyping based on region lifetimes:

```
letregion r1 in
let x =  0 at r1 in
letregion r2 in
let y =  1 at r2 in
let z = if x<y then x else y in
.…
```

What is the type of z above? The types of both branches of if expression are $int@r1$ and $int@r2$, respectively. To unify both the branches, we need subtyping relationship induced by LIFO order on region lifetimes. Since r1 outlives r2, (denoted $r1 \succcurlyeq r2$), we can provide an integer value stored at r1, when an integer value stored at r2 is expected. Consequently, we have that: $int@r1 <: int@r2$. Using this subtype judgement, we can unify the tyoes of both the branches of if expression as $int@r2$.

Example 3: Let us consider a similar example inthe context of a language extended with references:

```
letregion r1
in let p = 0 at r1
   in letregion rL
      in let q = 1 at rL
         in let x = if p<q then (ref p) at rL
                            else (ref q) at rL
            in !x at rH
```

The types of then and else branches of the if expression are $((int@r1)\ \ ref@\ rL)$, and $((int@rL)\ \ ref@rL)$, respectively. Since r1 outlives rL, we have that $int@r1 <: int@rL$. By covariant subtyping on references, we have that:

$$((int@r1)\ ref@rL)\ \ <:\ ((int@rL)\ ref@rL)$$

The intuitive reasoning behind the above subtype relation is as follows:
The subtype relation effectively makes it safe to use a reference into r1, wherever a reference into rL is expected, provided that reference into r1 lives atleast as long as the reference into rL. In this example, both references are stored in same region. Safety follows from the observation that r1 outlives rL, hence if it is safe to dereference a pointer into rL, it should be safe to dereference a pointer into r1. Further, the subtyping relation lets us unify the types on both the branches to $((int@rL)\ ref@rL)$. Hence x must be considered a reference into region rL, or any other region that lives atleast as long as rL.

The above paragraph argues for covariant subtyping among references (Assume both refs are stored in same regions. We omit region annotations for brevity):

$$\frac{T_1 <: T_2}{T_1 ref <: T_2\ ref}$$

However, covariant subtyping (or rather, typecast allowed by such subtyping) is, in general, unsound for references.

In this example, using the covariant downcast (from $(int, r1)\ ref@rL)$ to $((int, rL)\ ref@rL$) did not result in any manifest safety violation, as there are no references outside the lifetime of region r_L. The next example demonstrates the safety violation.

To summarize, subtyping rules that we used to type examples 1 and 2:

$$\frac{r_1 \geqslant r_2}{int@r_1 <: int@r_2}\ \text{[ST-1]} \qquad \frac{\tau_1 <: \tau_2 \quad r_1 \geqslant r_2}{\tau_1\ ref@r_1 <: \tau_2\ ref@r_2}\ \text{[ST-2]}$$

And a rule for downcast:

$$\frac{\Gamma \vdash e: \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e: \tau_2}\ \text{[T-Sub]}$$

Example 4: Consider the following code:

```
letregion r1 in
let p = 0 at r1 in
let x = ref p at r1 in
let _ =
    letregion rL in
    let f = (λ(x: (int@rL) ref@rL). x := 2 at rL) at rL in
        f x in
.….…
```

The above example is well-typed owing to the subtype rules shown above. Variable x has the type $int@r1\ ref@r1$.

1. Using rule ST-1, $int@r1 <: int@rL$
2. Using ST-2, $int@r1\ ref@r1 <: int@rL\ ref@rL$
3. Using T-Sub, and previous conclusion, we derive that x has type $int@rL\ ref@rL$

Owing to the cast allowed by the last conclusion, the call $f\ x$ is well-typed. But, this example creates a dangling pointer (through x), when region rL's lifetime ends.

Example 5: Another example that involves updating of a reference:

```
 $letregion\ r1\ in$
$let\ p = ref\ (0\ at\ r1)\ at\ r1\ in$
$let\ \_ =$
  $letregion\ r2\ in$
  $let\ x = 2\ at\ r2\ in$
     $p := x\ \ in$
$let\ y = 1\ in$
  $!p + y$
```

The type of p is $((int@r1)\ ref@\ r1)$, whereas type of x is $(int@r2)$. Subtyping rules allow p to have the type of $((int@r2)\ ref@\ r2)$, resulting in the above unsafe code to type check.

The above examples demonstrate that a general covariant subtyping among references is unsound. Similar counter examples can be given to rule out contravariant subtyping. It follows that general subtyping scheme for references is not possible.
Therefore, we eliminate the covariant subtyping for references (ST-2), but retain the usual subtyping based on region lifetimes:

$$\frac{r_1 \geqslant r_2 \quad \mathrm{T} \in \{int, \tau\ ref\}}{\mathrm{T}@r_1 <: \mathrm{T}@r_2}\ \text{[ST-1]}$$

The Subtyping Judgment

We now concretize the intuitions about subtyping discussed so far. We also extend it to the function types with latent effects.
Formally, $\succcurlyeq$ is the transitive closure of the *outlives* relation. In the typing judgment formulated previously, the set Δ tells us which regions are live while type checking an expression, but does not currently capture the LIFO order among their lifetimes. Since this order is needed for $r_1 \succcurlyeq r_2$ judgment, we redefine Δ as an ordered set, and add it as a context to the subtype judgment. Accordingly, $\Delta \cup \{\rho\}$ in the premise of T-Reg rule should be interpreted as Δ, ρ. Further, we need a subtyping rule for functions. The full description of subtype judgment is given below:

$$\boxed{\Delta \vdash \tau_1 <: \tau_2}$$

$$\frac{\rho_1 \succcurlyeq \rho_2 \quad T \in \{int, \tau\ ref\}}{\Delta \vdash T@\rho_1 <: T@\rho_2}\ \text{[ST-Base]} \qquad \frac{\begin{array}{c}\varphi_1 \subseteq \varphi_2 \quad \rho_1 \succcurlyeq \rho_2 \\ \Delta \vdash \tau_{21} <: \tau_{11} \quad \Delta \vdash \tau_{12} <: \tau_{22}\end{array}}{\Delta \vdash \tau_{11} \to^{\varphi_1} \tau_{12}\ @\ \rho_1 <:\ \tau_{21} \to^{\varphi_2} \tau_{22}\ @\ \rho_2}\ \text{[ST-Fun]}$$

The ST-Fun rule defines subtyping among functions to be contravariant among arguments and covariant among result types. This is the usual notion of subtyping among functions. However, we also require that the subtype function produce no more effects than the original function itself.

Finally, we need one more type rule to relate subtyping and typing judgments:

$$\frac{\Delta, \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_1 <: \tau_2}{\Delta, \Gamma \vdash e : \tau_2}\ \text{[T-Sub]}$$

The above rules are sufficient to correctly let the following example type check:

Example 6:

```
letregion r1 in
let p = 0 at r1 in
let x = ref p at r1 in
letregion r2 in
let y = ref (1 at r2) at r2 in
let _ = if (!x < !y) then y := !x else y := !y at r2 in
.…..…
```

In the above code:
1. $y : int@r2\ ref@r2$,
2. $x : int@r1\ ref@r1$,
3. $!x : int@r1$. Since $r1 \succcurlyeq r2$, we have that $int@r1 <: int@r1$. Therefore, $x : int@r2$
4. Due to (3) above, $y := !x$ typechecks.
5. The example is typesafe

However, our rules are too weak to type the following example, which is a minor modification of above example:

Example 7 :

```
letregion r1 in
let p = 0 at r1 in
let x = ref p at r1 in
letregion r2 in
let y = ref (1 at r2) at r2 in
```

```
        let _ = y := !(if (!x < !y) then x else y) in
        .….…
```

The reason for the failure of typechecking is the inability to unify types $int@r1\ ref@r1$ and $int@r2\ ref@r2$ on "then" and "else" branches, respectively. Subtyping may allow us to cast $int@r1\ ref@r1$ to $int@r1\ ref@r2$, but lack of covariant subtyping over references means that we cannot cast $int@r1\ ref$ to $int@r2\ ref$. Although covariant subtyping is unsound for references in general, it is sound if references are used in a read-only way, which is the case here. A possible solution is to allow covariance for read-only references:

$$\frac{\Gamma\vdash e{:}const\ ref\ \tau}{\Gamma\vdash !e{:}\tau} \qquad \frac{\Gamma\vdash e{:}ref\ \tau}{\Gamma\vdash e{:}const\ ref\ \tau} \qquad \frac{\tau_1 <: \tau_2}{const\ ref\ \tau_1 <: const\ ref\ \tau_2}$$

Similar example can be conceived to demonstrate the usefulness of contravariant subtyping for write-only references. Nevertheless, it is not known how often we encounter such examples in practice (in the context of Broom and C#). To keep things simple, we do not admit any subtyping among references for now. We will revisit this decision when we find enough practical motivation for admitting subtyping.

Polymorphic Language

We now extend the language with type and region polymorphism. Due to the presence of higher-order functions and effect typing, we need effect polymorphism to quantify over effects of higher-order functions.

The Language:

$$\begin{array}{l}
\rho \in \ region\ names \\
\alpha \in \ type\ variables \\
\epsilon \in \ effect\ variables \\
\upsilon ::= get(\rho) \mid set(\rho) \\
\varphi ::= \{\} \mid \{\upsilon\} \cup \varphi \mid \epsilon \cup \varphi \\
\tau ::= \alpha@\rho \mid int@\rho \mid unit \mid \tau\ ref@\rho \mid \tau_1 \rightarrow^{\varphi} \tau_2\ @\rho \\
\sigma ::= \forall\alpha.\sigma \mid \forall\epsilon.\sigma \mid \forall\rho.^{\varphi}\ \sigma \mid \tau \\
e ::= x \mid e\ e \mid n\ at\ \rho \mid \lambda(x{:}\tau).e\ \ at\ \rho \mid letregion\ \rho\ in\ e \mid e \otimes e\ at\ \rho \mid let\ x = e\ in\ e \\
\qquad \mid ref\ e\ at\ \rho \mid !\,e \mid e \coloneqq e \mid \lambda\rho.e \mid \Lambda\epsilon.\mathrm{e} \mid \Lambda\alpha.e \mid e\ \rho \mid e\ \varphi \mid e\ \tau
\end{array}$$

Type Rules

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

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

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

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

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

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

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

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

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

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

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

- 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 $\Lambda\rho.e$.

Linear Types, Refs & Usage Polymorphism

Thursday, June 26, 2014 9:10 PM

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 both linearly and non-linearly. 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.

This simple language is extended with usage polymorphism, and then with type polymorphism towards the end of this document.

The Monomorphic Language

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

Syntactic Conventions

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

Type Rules for Monomorphic Language

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

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

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

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

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

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

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

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. 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.
4. Since $Fix\ f\ =\ f\ (f\ (f\ \ldots\ ))$, we require that f has non-linear type.
5. 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. This is a design choice for the sake of simplicity.

Examples

Example 1

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

- 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 discareded, 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):

$$\begin{aligned}
&Fix\ (\lambda(f : int^1\ list^1 \rightarrow\ int^1\ list^1\ \rightarrow^1\ int^1\ list^1).\\
&\quad \lambda(xs : int^1\ list^1).\lambda(ys : int^1\ list^1).\\
&\quad\quad match\ xs\ with\ Nil \Rightarrow ys \mid Cons\ \ x\ xs' \Rightarrow Cons\ x\ (f\ xs'\ ys)
\end{aligned}$$

- 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:

$¡\,T^0$	=	T^0
$¡\,int^1$	=	int^0
$¡\,unit^1$	=	$unit^0$
$¡\,(\tau\ list^1)$	=	$(¡\,\tau)\ list^{\,0}$
$¡\,(\tau\ ref^1)$	=	$(¡\,\tau)\ ref^{\,0}$

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:
  $let\ l = ref\ (Nil\ int\ @\rho_H)\ @\rho_H\ in$
  $let!\,(x)\ y = match\ x\ with\ Nil \Rightarrow\ () \mid Cons\ h\ t \Rightarrow l \coloneqq t\ in$
  …

- 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 strategy of its arguments.

The Usage Polymorphic Language

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

Type Rules for Usage Polymorphic Language

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : \tau \to^0 \tau}{\Gamma \vdash Fix\ e : \tau} \qquad \frac{\upsilon \notin fuv(\Gamma) \quad \Gamma, \upsilon \vdash e{:}\sigma}{\Gamma \vdash \Lambda\upsilon.e : \forall\upsilon.\sigma} \qquad \frac{\Gamma \vdash e{:}\forall\upsilon.\sigma \quad \Gamma \vdash u}{\Gamma \vdash e\ u : [u/\upsilon].\sigma}$$

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 $int^{\upsilon_0} \rightarrow\ int^{\upsilon_1} \rightarrow^{\upsilon_0} int^{0}$.
- 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)

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

Example 6 (Usage Polymorphic Linked List Add)

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

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

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

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:

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

  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.

In Allocation Context

Wednesday, July 2, 2014   7:15 PM

In this wiki, our aim is to adapt the region type system that we have previously developed to the concept of *inAllocationContext*.

<u>The source language</u>

- 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.

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

- $\vec{A}$ 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 $((\forall \rho\vec{\rho}.\tau_1 \to \tau_2) \to \tau_3$ 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

$$\boxed{\Delta, \gamma, \Gamma \vdash \tau_1 <: \tau_2}$$

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

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

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

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

$$\frac{\begin{array}{c}\varphi_1=\text{frv}(\vec{\rho})\cup\{\rho\} \quad \varphi_1\not\subseteq\Delta\\ \varphi_2=\text{frv}(\vec{\rho})\cup\text{frv}(\tau_1\to\tau_2) \quad \Delta\cup\varphi_1;\gamma\cup\{\varphi_2\succcurlyeq\rho\};\rho;\Gamma\vdash e{:}\,\tau_1\to\tau_2\end{array}}{\Delta;\gamma;\rho^a;\Gamma\vdash\lambda\rho\vec{\rho}.e{:}\ \forall\rho\vec{\rho}.\tau_1\to\tau_2}\ \text{[T-RegGen]}$$

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

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

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

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

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

- 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:
    - $closed_\rho(rgn\langle\rho'\rangle\langle T\rangle) \Leftrightarrow \rho = \rho'$
    - $closed_\rho(rgn^1\langle\rho'\rangle\langle T\rangle) \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\tau \rightarrow \tau) = false$
    - $closed_\rho\left(\forall\rho'\overrightarrow{\rho'}.\tau \rightarrow \tau\right) = false$
    - $closed_\rho(unit) = true$
    - $closed_\rho(int@\rho') \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\alpha@\rho') \Leftrightarrow \rho = \rho'$
    - $closed_\rho(\tau\ ref@\rho') \Leftrightarrow closed_\rho(\tau) \wedge \rho = \rho'$
    - $closed_\rho(\tau\ list@\rho') \Leftrightarrow closed_\rho(\tau) \wedge \rho = \rho'$
  - If T denotes unannotated type of an object, then $\|T@\rho\|$ denotes its region-annotated type when the objected is owned by (i.e., completely allocated within) region ρ. It is defined recursively:
    - $\|T@\rho\| = T@\rho$, when $T \in \{\alpha, int\}$
    - $\|T\ ref@\rho\| = \|T@\rho\|\ ref@\rho$
    - $\|T\ list@\rho\| = \|T@\rho\|\ list@\rho$
- 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.

Objects, regions & LinkedList Example

Thursday, July 10, 2014 3:52 PM

FJ with Regions

The Language

$\rho, p \in \quad region\ names$
$cn \ \in Class\ Names\ (A, B, C \ldots)$
$mn \in Method\ Names\ (m, n, \ldots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle p^a \bar{p}\rangle\langle\bar{\tau}\rangle$ //Instantiated class type
$C \ ::= \ \text{class } cn\langle \rho^a \bar{\rho}\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N \left\{\overline{\tau f}; k\ ; \bar{d}\right\}$ //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
$\phi \ \ ::= true \mid \rho \succcurlyeq \rho \mid \rho = \rho \mid \phi \wedge \phi$ //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⩾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 $\bar{\rho} \succcurlyeq \rho^a$. 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 $\pi^a$and $\bar{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\bar{\pi} \succcurlyeq \pi^a$ 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.

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

ctype: Types of constructor arguments of a class.

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

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.

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

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

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

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

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

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.
-

$\boxed{\Delta;\gamma\vdash\tau_1<:\tau_2}$

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

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

$\boxed{\Delta;\ \gamma\vdash\tau\ OK\quad\Delta\vdash\phi\ OK}$

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

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

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

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

$$\frac{x{:}\tau\in\Gamma}{\Delta;\gamma;p^a;\Gamma\vdash x{:}\tau}\qquad\frac{\begin{array}{c}\Delta;\ \gamma;\ p^a;\ \Gamma\vdash e{:}\tau'\\ f{:}\tau\in fields\big(bound_\Delta(\tau')\big)\end{array}}{\Delta;\ \gamma;\ p^a;\ \Gamma\vdash e.f{:}\tau}\qquad\frac{\begin{array}{c}\Delta;\gamma\vdash N\ OK\quad allocRgn(N)=p\quad\gamma\vdash p^a\succcurlyeq p\\ ctype(N)=\bar{\tau}\quad\Delta;\ \gamma;\ p^a;\ \Gamma\vdash\overline{e{:}\tau}\end{array}}{\Delta;\ \gamma;\ p^a;\ \Gamma\vdash new\ N(\bar{e}){:}\ N}$$

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

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

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

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

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

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

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

$\boxed{d\ OK\ in\ B}$

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

$\boxed{B\ OK}$

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

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:

$class\ A\langle R^a\rangle\ extends\ Object$ {
  $A()$ { $super()$; }
}
$class\ B\langle R^a\rangle\ extends\ Object$ {
  $B()$ { $super()$; }
}
$class\ Pair\langle R^a,R0,R1\rangle\langle X\ extends\ Object\langle R0\rangle,Y\ extends\ Object\langle R1\rangle\rangle\ extends\ Object\langle R^a\rangle$ {

```
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  $X\ getfst\langle R^a_m \rangle() \{$
    return this.fst;
  }
  $unit\ setfst\langle R^a_m \rangle(X\ newfst) \{$
    this.fst = newfst;
    return ();
  }
}
```

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.

Should there be a constraint between the lifetimes of R_a and R0?

The constraint that R0 >= Ra is implicit. Implicit constraints are detailed in point 4 of notes as the beginning of this wiki.

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

Is there a way of specifying the region in which primitive returned objects are allocated?
I assume primitive objects are ints, bools and units. We treat them as unboxed values. Nevertheless, it is possible to use their boxed counterparts (eg: Integer for int). Boxed values can be annotated to specify their allocation region (eg: Integer<R0>).

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);
                     rout.set(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.

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

- 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.

Region Type Inference

Thursday, July 24, 2014 7:24 PM

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

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

The Target Language

$\rho, p \in \ \ region\ names$
$cn \ \in Class\ Names\ (A, B, C \dots)$
$mn \in Method\ Names\ (m, n, \dots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle p^a \bar{p} \mid \phi \rangle \langle \bar{\tau} \rangle$ //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
$\phi \ \ ::= true \mid \rho \succcurlyeq \rho \mid \rho = \rho \mid \phi \wedge \phi$ //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
$\tau \ ::= \ \tau_{\lhd} \mid T$
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid \ letregion\langle \rho \rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$

$$
\begin{array}{rl}
 & \mid open^{a}\ e\ \{\ s\ \} \mid s;s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \\
e\ ::= & c \mid x \mid e.f \mid e.mn\langle p^{a}\overline{p}\rangle(\bar{e}) \mid \text{new } N(\bar{e}) \mid (N)\ e \mid e.get()\ \ \text{//Expressions}
\end{array}
$$

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.
  $bound_{\Delta}(\alpha) = \Delta(\alpha)$
  $bound_{\Delta}(N) = N$
  $bound_{\Delta}(T) = T$

$\boxed{B\ OK, B\ header\_OK, B\ cons\_OK}$

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

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

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

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

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

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

$$\frac{\begin{array}{c}
CT(B)=\mathit{class}\ B\langle\overline{\alpha\lhd N_S}\rangle\lhd N_S\ \{\overline{T\ f};k_S\ ;\overline{d_S}\} \\
\mathit{class}\ B\langle\overline{\alpha\lhd N_S}\rangle\lhd N_S\ \{\overline{T\ f};k_S\ ;\overline{d_S}\}\hookrightarrow\mathit{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;\mathit{return}\ e;\}\quad \tau_p=\mathit{templateTy}(T_r)\quad \overline{\tau_a}=\mathit{templateTy}(\overline{T_x})\quad \mathit{new}(\pi^a) \\
\overline{\pi}=\mathit{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'[B\mapsto\mathit{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\}]\quad \Gamma=\cdot,\mathit{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{\alpha\lhd N_B} \\
\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}\succcurlyeq\rho_B^a)\wedge\phi_B\quad (D,\psi_i)=\mathit{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}=\mathit{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}=\mathit{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}\succcurlyeq\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'';\mathit{return}\ e'';\}
\end{array}}{\mathit{class}\ B\langle\overline{\alpha\lhd N_S}\rangle\lhd N_S\ \{\overline{T\ f};k_S\ ;\overline{d_S}d_S\}\hookrightarrow\mathit{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
  $\alpha$|int|bool|unit => T
| Object => Object<ρ> where new(ρ)
| A($\overline{T}$) => if A ∈dom(CT') ∧CT'(A) = $\mathit{class}\ A\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N$ then
              then A<$\pi^a\overline{\pi}$><$\overline{\tau}$> where new($\pi^a\overline{\pi}$)∧ $|\overline{\pi}|=|\overline{\rho}|$ ∧ $\overline{\tau}$ = templateTy($\overline{T}$)
              else T
| Region<$T_{root}$> =>let $\tau'$= templateTy($T_{root}$) in
                  let $\tau_{root}$= $[\rho/\mathit{frv}(\tau')]\,\tau'$ where new($\rho$) in
                      Region[$\rho$]<$\pi$><$\tau_{root}$> where new($\pi$)
```

```
fun superClasses(B<$\pi^a \overline{\pi}$><$\overline{\tau}$>) = case B of
  Object => {}
| Region[$\rho$] => Object<$\pi^a$>
| _ =>
    let $class\ B\langle \rho^a \overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N$ = CT'(B)in
    let N' = $[\overline{\pi}/\overline{\rho}][\pi^a/\rho^a]$ N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<$\pi^a \overline{\pi}$><$\overline{\tau}$>)= $\pi^a$
  | _ => 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.
- $\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash s \hookrightarrow s' : \tau \dashv \Gamma'$ 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.

$\boxed{C \vdash \tau\ OK}$

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

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

$\boxed{\Delta;\ C \vdash \tau_1 <: \tau_2}$

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

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

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

$$\frac{x{:}\tau \in \Gamma}{\Sigma;\Delta;\Gamma;\ \rho^{a};\top \vdash x \hookrightarrow x{:}\tau} \qquad \frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C \vdash e \hookrightarrow e'{:}\tau' \\ f{:}\tau \in fields(bound_{\Delta}(\tau'))\end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C \vdash e.f \hookrightarrow e'.f{:}\tau} \qquad \frac{\begin{array}{c}\rho \in \Sigma \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^{a};C_{R} \vdash e \hookrightarrow e'{:}Region[\rho]\langle \pi^{a} \rangle \langle \tau \rangle\end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};C_{R} \vdash e.get() \hookrightarrow e'.get(){:}Region[\rho]\langle \pi^{a} \rangle \langle \tau \rangle}$$

$$\frac{\begin{array}{c}N{=}templateTy(N_{s}) \quad C_{N} \vdash N\ OK \\ \pi^{a}{=}allocRgn(N) \quad \overline{\tau_{B}}{=}ctype(N) \quad \Sigma;\ \Delta;\ \Gamma;\rho^{a};C_{e} \vdash \bar{e} \hookrightarrow \overline{e_{1}}{:}\overline{\tau_{e}} \\ \Delta;\ C_{\lhd} \vdash \overline{\tau_{e}} <:\ \overline{\tau_{B}} \quad C{=}C_{N} \wedge C_{e} \wedge C_{\lhd} \wedge (\rho^{a} \succcurlyeq \pi^{a})\end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C \vdash new\ N(\bar{e}) \hookrightarrow new\ N(\overline{e_{1}}){:}N} \qquad \frac{eraseRgn\ (\tau) \lhd Object}{\Sigma;\Delta;\Gamma;\ \rho^{a};\top \vdash Null{:}\tau}$$

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C_{0} \vdash e_{0} \hookrightarrow e_{0}'{:}\tau_{0} \quad mtype\big(m, bound_{\Delta}(\tau_{0})\big) = \langle \rho_{m}^{a}\overline{\rho_{m}}\ |\phi_{m} \rangle \overline{\tau_{x}} \to \tau \\ new(\overline{\pi}) \quad |\overline{\pi}|{=}|\overline{\rho_{m}}| \quad \psi{=}[\overline{\pi}/\overline{\rho_{m}}][\rho^{a}/\rho_{m}^{a}] \quad C_{x} \vdash \overline{\psi(\tau_{x})}\ OK \quad C_{r} \vdash \psi(\tau)\ OK \\ \Sigma;\ \Delta;\ \Gamma;\rho^{a};C_{e} \vdash \bar{e} \hookrightarrow \overline{e'}{:}\overline{\tau_{e}} \quad \Delta;\ C_{\lhd} \vdash \overline{\tau_{e}} <:\ \overline{\psi(\tau_{x})} \quad C{=}C_{x} \wedge C_{r} \wedge C_{e} \wedge C_{\lhd} \wedge \psi(\phi_{m})\end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C \vdash e_{0}.m(\bar{e}) \hookrightarrow e_{0}'.m\langle \rho^{a}\overline{\pi} \rangle \big(\overline{e'}\big){:}\psi(\tau)}$$

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

$$\frac{\begin{array}{c}\tau{=}templateTy(T) \quad C_{T} \vdash \tau\ OK \quad \Gamma'{=}\Gamma,x{:}\tau \\ \Sigma;\Delta;\ \Gamma;\rho^{a};C \vdash e \hookrightarrow e'{:}\tau_{e} \quad \Delta;\ C_{\lhd} \vdash \tau_{e} <: \tau \quad C{=}C_{T} \wedge C_{e} \wedge C_{\lhd}\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash let\ T\ x{=}e \hookrightarrow let\ \tau\ x{=}e'\ {:}\dashv \Gamma'} \qquad \frac{\begin{array}{c}e_{1} \in \{x, e.f\} \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C_{1} \vdash e_{1} \hookrightarrow e_{1}'{:}\tau_{1} \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C_{2} \vdash e_{2} \hookrightarrow e_{2}'{:}\tau_{2} \quad \Delta;\ C_{\lhd} \vdash \tau_{e} <: \tau \\ C{=}C_{1} \wedge C_{2} \wedge C_{\lhd}\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash e_{1}{=}e_{2} \hookrightarrow e_{1}'{=}e_{2}' \dashv \Gamma}$$

$$\frac{\begin{array}{c}new(\rho) \quad \Sigma \cup \{\rho\};\ \Delta;\ \Gamma;\ \rho;\ C_{s} \vdash s \hookrightarrow s' \dashv \Gamma' \\ C{=}\rho \notin \Sigma \wedge \Sigma \succcurlyeq \rho \wedge C_{s}\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash\ letregion\ \{\ s\ \} \hookrightarrow letregion\langle \rho \rangle\ \{\ s'\ \} \dashv \Gamma} \qquad \frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C_{1} \vdash s_{1} \hookrightarrow s_{1}' \dashv \Gamma_{1} \\ \Sigma;\ \Delta;\ \Gamma_{1};\ \rho^{a};\ C_{2} \vdash s_{2} \hookrightarrow s_{2}' \dashv \Gamma' \quad C{=}C_{1} \wedge C_{2}\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash s_{1};s_{2} \hookrightarrow s_{1}';s_{2}' \dashv \Gamma'}$$

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

$$\frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};C_{R} \vdash e \hookrightarrow e'{:}Region[\rho]\langle \pi^{a} \rangle \langle \tau \rangle \quad \rho \in \Sigma \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^{a};\ C_{2} \vdash e_{2} \hookrightarrow e_{2}'{:}\tau' \quad C{=}C_{R} \wedge C_{2} \wedge (\tau'{=}\tau)\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash e.set(e_{2}) \hookrightarrow e'.set\big(e_{2}'\big) \dashv \Gamma} \qquad \frac{\begin{array}{c}\Sigma;\ \Delta;\ \Gamma;\ \rho^{a};C_{R} \vdash e \hookrightarrow e'{:}Region[\rho]\langle \pi^{a} \rangle \langle \tau \rangle \quad \rho \notin \Sigma \\ a \in \{transfer, giveUp\}\end{array}}{\Sigma;\Delta;\Gamma;\ \rho^{a};C \vdash e.a() \hookrightarrow e'.a() \dashv \Gamma}$$

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: $\rho \notin \{\rho1,\rho2\}$ can be elaborated to $\rho \neq \rho1 \wedge \rho \neq \rho2$). 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,R3,R4 | R2>=R0,R4>=R3,R3=R0,R4=R1 || R0->R1> (LinkedListNode<R3,R4>
<T> node) {
    node.next = this.head;
    this.head.prev = node;
    this.head = node;
    this.count=this.count+1;
    return ();
  }

  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 ();
  }

  reverse: forall(R1,R2).R1>=R2. unit -> LinkedList<R2,R1><T>
  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;
  }
}

Object<R3> Foo<R3,R1,R2 | R1=R2, R1,R2>=R3>(Object<R1> a, Object<R2> b) {
    Object o = new Object();
```

```
     If (a > 0)
       o.x = a;
     Else
       o.x = b;
     this.head = a; // R0->R1
     Return o;
}
Object<R3,R1>;

{
   if (a>0)
       y=a
       x=…
   else
       y=b
       x=new Object();
   return x;
}

(R3)->(R1,R2)
(R3)->…
```

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);
```

Simplified Type Inference

Tuesday, August 12, 2014 8:50 PM

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
$N ::= cn\langle\bar{T}\rangle$ //Instantiated class type
$C ::= \text{class } cn\langle\overline{\alpha \lhd N}\rangle \lhd N \{\overline{T\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k ::= cn\ (\overline{T\ x})\{\text{ super }(\bar{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d ::= T\ mn\ (\overline{T\ x})\ \{s;\ \text{return } e;\}$ //Methods
$T ::= \alpha \mid N \mid Object \mid Region\langle T\rangle \mid int \mid bool \mid unit$ // Types
$v ::= c \mid x \mid new\ N(\bar{v})$
$s ::= \cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid letregion\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e ::= c \mid x \mid e.f \mid e.mn(\bar{e}) \mid \text{new } N(\bar{e}) \mid (N)\ e \mid e.get()$ //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
$N ::= cn\langle p^a\bar{p} \mid \phi\rangle\langle\bar{\tau}\rangle$ //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
$\phi ::= true \mid \rho \succcurlyeq \rho \mid \rho = \rho \mid \phi \wedge \phi$ //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
$v ::= c \mid x \mid new\ N(\bar{v})$
$s ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid letregion\langle\rho\rangle\ \{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\quad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp()$
$e ::= c \mid x \mid e.f \mid e.mn\langle p^a\bar{p}\rangle(\bar{e}) \mid \text{new } N(\bar{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle\rho\rangle\langle\tau\rangle()$ //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 $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\,f}; k_s\,; \overline{d_s}\}$ = CT(B)
    class $B\langle\rho^a\bar{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\,f}\}$ = header-template(B)
    C1 = type-ok($\overline{N}$)
    C2 = type-ok($N$)
    C3 = type-ok($\bar{\tau}$)
    C = C1 ∧ C2 ∧ C3 ∧ $\bar{\rho} \succcurlyeq \rho^a$
    (D,$\psi_i$) = normalize(C)
    $\overline{N_T} = \psi_i(\overline{N})$
    $N_T = \psi_i(N)$
    $\overline{\tau_T} = \psi_i(\bar{\tau})$
    $\rho^a_T = \psi_i(\rho^a)$
    $\overline{\rho_T} =$ (frv($\overline{N_T}, N_T, \overline{\tau_T}$)) $- \{\rho^a_T\}$
    $\phi = D - \{\overline{\rho_T} \succcurlyeq \rho^a_T\}$ (* We need not record implicit constraints*)
  in
    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
    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($\overline{T}$)
    $\rho^{a}$ = allocRgn($^{X}N$)
    $\overline{\rho}$= (frv($\overline{^{X}N}, {}^{X}N, \overline{^{X}\tau}$))−{$\rho^{a}$}
    $\psi_i$ = $[B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle/B\langle\overline{\alpha}\rangle]$ (* 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. *)
    $\overline{N} = \psi_i(\overline{^{X}N})$
    $N = \psi_i(^{X}N)$
    $\overline{\tau} = \psi_i(\overline{^{X}\tau})$
  in
    class $B\langle\rho^a\overline{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}\}$
  end

fun elaborate-cons(B, hdB) =
  let
    class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s\ ; \overline{d_s}\}$  = CT(B)
    class $B\langle\rho^a_B\overline{\rho_B} \mid \phi_B\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f}\}$ = hdB
    $\overline{\tau_A}$= ctype($N_B$)   (* Types of super class constructor args *)
    $B(\overline{T_x\ x})\{super(\overline{\upsilon_g});\ \overline{this.f = \upsilon_f;}\}$  =  $k_s$
    $\overline{\tau_a}$ = templateTy($\overline{T_x}$)
    Ca = type-ok($\overline{\tau_a}$)
    _ = CT'[B $\mapsto$  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  =  ·, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle$, x: $\overline{\tau_x}$
    Σ = $\rho^a_B \cup \overline{\rho_B}$
    Δ = $\overline{\alpha \lhd N_B}$
    ($\overline{\upsilon'_g : \tau_g}$, Cg)  = elab-expr(Σ; Δ; Γ; $\rho^a_B \vdash \overline{\upsilon_g}$)
    Csub = subtype-ok(Δ ⊢ $\overline{\tau_g} <: \overline{\tau_a}$) (* Actual types of args to super should be subtype of
                 expected types. *)
    ($\overline{this.f = \upsilon'_f}$, _, Cf)  = elab-stmt(Σ; Δ; Γ; $\rho^a_B \vdash \overline{this.f = \upsilon_f}$)
    C  = Ca  ∧  Cg  ∧  Csub  ∧  Cf  ∧  ($\overline{\rho_B} \succcurlyeq \rho^a_B$) ∧ $\phi_B$
    (D, $\psi_i$)  = normalize(C)
    $(\overline{N}, N, \overline{\tau}, \overline{\tau_x}, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\rho^a_B))$
    $\overline{\rho}$  =  (frv($\overline{N}$, N, $\overline{\tau}$)) − {$\rho^a$}
    $\phi$  = project-constraints(D, {$\rho^a, \overline{\rho}$})  -  ($\overline{\rho} \succcurlyeq \rho^a$)   (* Collect residual constraints
                 over region params of class B that need to be recorded explicitly as refinement *)
    ($\overline{\upsilon''_g}$,  $\overline{\upsilon''_f}$)  =  ($\psi_i(\overline{\upsilon'_g})$,  $\psi_i(\overline{\upsilon'_f})$)
    k  =  $B(\overline{\tau_x\ x})\{super(\overline{\upsilon''_g});\ \overline{this.f = \upsilon''_f;}\}$
  in
    class $B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\}$
  end

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

fun elaborate-methods-rec(Bdef, consB) = case Bdef of
  class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s\ ;\}$ => consB  (* If there are no methods, we are done *)
| class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s\ ; \overline{d_s} d_s\}$  =>
  let
    fullB' = elaborate-methods-rec (class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s\ \{\overline{T\ f}; k_s\ ; \overline{d_s}\}$, consB)
    class $B\langle \rho^a_B \overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\ f};\ k;\ \overline{d_B}\}$ = fullB'
    (* Our task is to elaborate method $d_s$*)
    $T_r\ m(\overline{T_x\ x})\{s; \text{return } e;\}$ = $d_s$
    $\tau_p$ = templateTy($T_r$)
    $\overline{\tau_a}$ = templateTy($T_x$)
    $(\pi^a, \rho^a_m, \overline{\pi})$ = (new(), new(), frv($\tau_p, \overline{\tau_a}$)) (* $\pi^a$ denotes allocation context param of "m".
                $\rho^a_m$ is to be used as a dummy variable to facilitate the unification of allocation
                contexts for recursive calls of "m" with $\pi^a$. In other words, no region polymorphic
                recursion *)
    $d_t = \tau_p\ m\langle\rho^a_m \mid \rho^a_m = \pi^a\rangle(\overline{\tau_a\ x})\{\cdot\}$ (* 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 $B\langle \rho^a_B \overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\ f};\ k;\ \overline{d_B} d_t\}$]   (* temporarily
                update CT' so that "this.m" gives correct type*)
    $\Gamma = \cdot, \text{this}: B\langle\rho^a_B\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle, x: \overline{\tau_a}$
    $\Sigma = \rho^a_B \cup \overline{\rho_B} \cup \pi^a \cup \overline{\pi}$
    $\Delta = \overline{\alpha \lhd N_B}$
    (s', $\Gamma'$, Cs) = elab-stmt($\Sigma;\ \Delta;\ \Gamma; \pi^a \vdash s$)
    (e' : $\tau_q$, Ce) = elab-expr($\Sigma;\ \Delta;\ \Gamma'; \pi^a \vdash e$)
    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 $\wedge$ $(\overline{\rho_B} \succcurlyeq \rho^a_B) \wedge \phi_B$   (* Set of all constraints *)
    (D, $\psi_i$) = normalize(C)
    $(\overline{N}, N, \overline{\tau}, \overline{\tau_x}, \tau_r, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\tau_p), \psi_i(\rho^a_B))$
    $\overline{\rho}$ = (frv $(\overline{N}, N, \overline{\tau})) - \{\rho^a\}$
    $\Sigma_\rho$ = $\rho^a \cup \overline{\rho}$  (* $\rho^a$ and $\overline{\rho}$ are new region vars that replace $\rho^a_B$ and $\overline{\rho_B}$ as region params of class
    B *)
    $(\rho^a_m, \phi^a_m)$ = 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$(\overline{\tau_x}, \tau_r)) - \{\rho^a_m\}$
    $\Sigma_\pi$ = $\rho^a_m \cup \overline{\rho_m}$
    $\phi$ = project-constraints(D, $\Sigma_\rho$) - $(\overline{\rho} \succcurlyeq \rho^a)$ (* Explicit constraints over region
    params of B *)
    $\phi_m$ = project-constraints(D, $\Sigma_\rho \cup \Sigma_\pi$) (* Constraints over region params of method
    m *)
    $(s'', e'')$ = ($\psi_i$(s'), $\psi_i$(e'))
    d = $\tau_x\ m\langle\rho^a_m\overline{\rho_m} \mid \phi^a_m \wedge \phi_m\rangle(\overline{\tau_x\ x})\{s''; return\ e''\}$
  in
    class $B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f};\ k;\ \overline{d} d\}$
  end

(* AUXILIARY FUNCTIONS *)

fun templateTy(T) = case T of
  $\alpha$|int|bool|unit => T
| Object => Object<$\rho$> where new($\rho$)
| A($\overline{T}$) => if A $\in$ dom(CT') $\wedge$ CT'(A) = $class\ A\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N$ then
```

```
             then A<$\pi^a\bar{\pi}$><$\bar{\tau}$> where new($\pi^a\bar{\pi}$)$\wedge$ $|\bar{\pi}| = |\bar{\rho}|$ $\wedge$ $\bar{\tau}$= templateTy($\bar{T}$)
             else T
| Region<$T_{root}$> =>let $\tau'$= templateTy($T_{root}$) in
                   let $\tau_{root}$= $[\rho/frv(\tau')]\,\tau'$ where new($\rho$) in
                      Region[$\rho$]<$\pi$><$\tau_{root}$> where new($\pi$)

fun superClasses(B<$\pi^a\bar{\pi}$><$\bar{\tau}$>) = case B of
  Object => {}
| Region[$\rho$] => Object<$\pi^a$>
| _ =>
    let $class\ B\langle\rho^a\bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N$ = CT'(B)in
    let N' = $[\bar{\pi}/\bar{\rho}][\pi^a/\rho^a]$ N in
        {N'} $\cup$ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<$\pi^a\bar{\pi}$><$\bar{\tau}$>)= $\pi^a$
  | _ => error()

fun project-constraints (D,S) = case D of
  true => D
| $\varphi \wedge$ D'=>
    let $\phi$ = project-constraints (D',S) in
      if frv($\varphi$)$\subseteq$ S then $\varphi \wedge \phi$ else $\phi$
```

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)`.

$\boxed{C \vdash \tau\ OK}$

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

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

$\boxed{\Delta;\ C \vdash \tau_1 <: \tau_2}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Type inference (Informal outline)

Wednesday, July 30, 2014 9:10 PM

This note presents a simplified, informal, overview of the type inference algorithm. More formal and specific details can be found in the other page Region Type Inference.

Overview

The inference algorithm processes a set of class-definitions. The class-definitions are processed in an order that respects the dependences between classes:

1. If class X is a derived class of Y, then X depends on Y.
2. If class X has a field of type Y, then X depends on Y.
3. If a parametric class X<T> has constraints of the form "T extends S", then X depends on S.
4. If method of class X makes use of a variable of type Y, then X depends on Y.

In the presence of mutually recursive dependences (e.g., X depends on Y and Y depends on X), the inference algorithm needs to process these classes together. To begin with, let us ignore this complication and assume that there is no mutually recursive dependence. If X depends on Y, then the type inference algorithm is first applied to Y and then applied to X. Let us now address the problem of type inference for a single class X, assuming that all classes that X depends on have already been processed.

Challenges: Interfaces and abstract methods [and, more generally, method overriding] pose a challenge to the inference. Specifically, we end up with a mutual dependence between an interface and its implementation, or a overridden method and the overriding method [since the region-constraints of the overriding method must be incorporated in the region-constraints of the overridden method]. This can be handled if we do inference for a whole program, but is difficult if we do inference for a library or incomplete program. We may have to exploit some heuristics to infer a type that is not the most-general type: e.g., restricting an interface or an abstract method (signature) to have a single region-variable. E.g., in the general case, a *List<T>* can be parameterized over two region-variables, one indicating where the list itself is allocated, and one indicating where the elements of the list are allocated. For the interface *IList<T>*, we may have to heuristically force these two regions to be the same, or analyze the interface together with an implementation to produce the more general typing.

The inference consists of two aspects:

1. Identify the set of region-variables to be introduced (and express the types of various entities in terms of these region-variables).
2. Identify constraints among these region-variables, which may cause some of these variables to be unified, and to incorporate these constraints in the inferred types.

Part 1: Introducing region-variables and elaborated-types

Let us consider problem (1).

a) The set RV(X) of region-variables for the class X is the union of the
    i) The set RV(Y) of region-variables associated with the base-class Y of X (if any), and
    ii) The set RV(f) of region-variables for every field f in class X, with the exception of any recursive field (that is, any field of type X), and
    iii) The set RV(tp) of region-variables for every type-parameter of class X (if X is a parametric class), and
    iv) One distinguished region-variable (representing the allocation-region).
b) The set RV(f) of region-variables of a (non-recursive) field f of X is obtained by introducing a new

region-variable for every region-variable in the type of the field f. Thus, if f is of type Z, then RV(f) consists of a new region-variable for every region-variable in RV(Z). Note that if the type of f is of the form Z<W>, where Z is a parametric class, then, we will need to introduce new region-variables corresponding to those in RV(W) union RV(Z). (Note that this applies when the classes Z and W are real classes, not when Z or W is a type parameter of the parametric class X. For type parameters, we don't need to introduce any new region-variables.)

c) The set RV(tp) of a bounded type-parameter tp of class X with a constraint "tp extends Z" is obtained by introducing new region-variables for every region-variable in the type Z.
d) The set RV(f) for a recursive field f of X is the same as RV(X).
e) For any method m, RV(m) is defined as the union of (fresh copies) of the
   i) region-variables RV(type(p)) of every parameter p, and
   ii) region-variables RV(type(v)) of the return value v
f) For any variable v of type T, RV(v) is obtained from the type T (just like in the case of fields).

This can be extended to deal with mutually-recursive classes. If classes X and Y are mutually recursive, then the set RV(X) and RV(Y) are defined to be the same and this set is computed by analyzing the definitions of class X and Y together as above.

We order the set of region-variables associated with any entity (class, field, variable, etc.) into a sequence, and this produces the region-parameterized types such as C<R1,R2>. As usual, the distinguished allocation-region is specified as the first element in this sequence (where appropriate). This is mostly straightforward, with the exception of (higher-order) functions, as detailed below.

Challenge: Consider a higher-order function $f(\,X \rightarrow void\ g)$. Assume that the class *X* has a single region-variable, namely its allocation-region. Thus, the type *X* is elaborated into $X\langle R\rangle$. Now, we have two choices for elaborating the type of f: (a) As type $\forall R.\,((X\langle R\rangle \rightarrow void) \rightarrow void)$ or as (b) $(\forall R\ (X\langle R\rangle \rightarrow void)) \rightarrow void$. This choice introduces complicates the type inference algorithm. In principle, we could try each choice and see if the program type-checks under this choice, but this can introduce a potential explonential blowup in the number of choices we must consider.

Challenge: In C#, functions and closures are typically represented as objects (of various types such as Action<>, Func<>, etc.). To get the equivalent of polymorphic functions, we will need similar polymorphic classes/objects (e.g. an object of type $\forall R.\,Action\ \langle X\langle R\rangle\rangle$ can be used to represent a procedure that expects an argument of type X, allocated in any region whatsoever. The current type-system does not include such types.

Note: The above process tries to infer as general a type as possible. Hence, it introduces a new region variable wherever possible, forcing different region variables to be the same (via unification) only when necessary. However, this could potentially lead to an explosion in the number of region variables. Some ad-hoc heuristics to limit this may be useful. E.g., consider the following class: "class X { Y y; Z z; }". The general process will utilize different region variables for the container X and the objects that y and z point to. Thus, we could use a "X<R1,R2,R3>" to denote a X allocated in region R1, whose y-field points to an object allocated in region R2, and z-field points to an object allocated in region R3. If class X has hundreds of fields, this could lead to a potential explosion in the number of region-variables.

Example.

Part 2: Constraint generation and simplification

Work in progress: This section is currently incomplete.

In this phase we generate "outlives" constraints on the region variables of the form $(\rho_1 \succcurlyeq \rho_2)$. Simplifying these constraints and projecting these constraints onto the relevant variables produces the types of classes as well as their methods. As an example of the final outcome, we may determine that a region-parametric class "X<R1,R2,R3>" requires "R2 ≽ R3". Similarly, we may determine that a procedure "f(Y<R1> y, Z<R2> z)" requires "R1 ≽ R2". Such constraints are incorporated as part of these respective types.

The key sources of constraints are described below:

a) Any field f of any object o1 must point to an object o2 that outlives o1. This is captured as:
    i) For a class C and a field f in the class we have "AllocationRegion(f) ≽ AllocationRegion(C)".
b) Any stack-variable v points to an object which outlives the variable itself:
    i) "AllocationRegion(v) ≽ VariableScope(v)"
c) For any assignment statement "lhs = rhs", we have a sub-typing constraint "type(rhs) ≽ type(lhs)". This sub-typing constraint can be transformed or simplified to a set of outlives constraints.
    i) In the simplest instance, where lhs and rhs are just Objects, for example, this reduces to "AllocationRegion(rhs) ≽ AllocationRegion(lhs)".
d) For any method call, we have similar typing-constraints between the types of the actual parameters and the formal parameters (and, similarly, for return-values).
    i) To map this to region-variable constraints, we first need to produce a new "instance" of the method's type-signature (by creating new region-variables for every (outer-quantified) region-variable in the method's type). The sub-typing constraints will then generate constraints on these new region-variables.

In addition, there are a number of straightforward ways in which these constraints are propagated or inherited, as follows:

a) For a class C that derives from a class B, we have “AllocationRegion(C) = AllocationRegion(B)”. [The entire object is allocated in one region.]
b) A class C that derives from a class B also inherits "Constraints(B)".
c) A parametric class with a bounded type parameter T (of the form "T extends B") inherits "Constraints(B)".
d) A variable x of type C inherits "Constraints(C)".

Slides - Broom Type System

Saturday, August 9, 2014 5:13 PM

Here is the PDF:

BroomSlides

If you are seeing some blank slides in the PDF, it could be because of some problem with OneNote attachments. Please use this link to get full PDF: https://msft-my.spoppe.com/personal/t-gokaki_microsoft_com/_layouts/15/guestaccess.aspx?guestaccesstoken=Kj5cAgISehXVQ2nOIXBtVsQVE5Yv%2frJZrvIXB1IqijY%3d&docid=0a25db93938554527a5eb71a23d8c3fca

Here is the Keynote:

BroomType System

Region Type System + Exists

Tuesday, August 12, 2014 11:32 AM

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

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

FJ with Regions

The Language

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

Notes

1. A note on notation: We write $\bar{a}$ to denote sequence of a's (i.e., $a_0, a_1, \dots, a_i, \dots, a_n$). 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 $\{b \odot a_0, \dots, b \odot a_i, \dots\}$, depending on the context. Similarly, $\bar{b} \odot \bar{a}$ or $\overline{(b \odot a)}$ denotes point-wise $\odot$ relation between b's and a's.
2. We write $\alpha \lhd N$ to denote that α is subclass of N. We call the instantiated class N as the bound of the type variable α.
3. A region name (ρ, p or π) is a static identifier for a region created by either a $letregion$ expression or a $new\ Region$ expression. If such an expression is inside a loop or a recursive function, it creates multiple regions at run-time, all of which have same static identifier. Any outlives relation that holds on two static identifiers also holds between corresponding regions at run-time. For eg, consider the following code:

```
let var x = …;
while (.…){
  letregion<R0> {
    letregion<R1> {
      …
    }
  }
}
```

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

4. We have region-polymorphic classes and region-polymorphic methods. Examples are given at the end of this wiki (LinkedListNode and LinkedList). The syntax of a region-polymorphic class is given below:
   $\text{class } B\langle \rho^a \bar{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \left\{ \overline{\tau\, f}; k\, ; \bar{d} \right\}$
   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 $\bar{\rho} \succcurlyeq \rho^a$. 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 $\pi^a$and $\bar{\pi}$ are concrete region identifiers, the expression is judged OK only if the relation $\bar{\pi} \succcurlyeq \pi^a$ 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.

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

ctype: Types of constructor arguments of a class.

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

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.

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

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

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

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

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

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.
-

$$\boxed{\Sigma;\Delta;\gamma\vdash\tau_1<:\tau_2}$$

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

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

$$\boxed{\Sigma;\Delta;\ \gamma\vdash\tau\ OK\quad\Delta\vdash\phi\ OK}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\boxed{d\ OK\ in\ B}$

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

$\boxed{B\ OK}$

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

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:

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

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.

```
$class\ LinkedListNode\langle R^a, R0\rangle\langle T\ extends\ Object\langle R0\rangle\rangle\ extends\ Object\langle R^a\rangle$ {
  $T\ val$;
  $LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ prev$;
  $LinkedListNode\langle R^a, R0\rangle\langle T\rangle\ next$;
  $LinkedListNode(T\ val)$ {
    $super()$;
    $this.val\ =\ val$;
    $this.prev\ =\ Null$;
    $this.head\ =\ Null$;
  }
}
$class\ LinkedList\langle R^a, R1\rangle\langle T\ extends\ Object\langle R1\rangle\rangle\ extends\ Object\langle R^a\rangle$ {
  $LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ head$;
  $int\ count$;
  $LinkedList(T\ v)$ {
    $super()$;
    $this.head\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v)$;
   $this.count = 1$;
  }
  $unit\ add\langle R^a_m \mid R^a_m \succcurlyeq R^a\rangle(T\ v)$ {
    $let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ n\ =\ new\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle(v)$;
    $n.next\ =\ this.head$;
    $this.head.prev\ =\ n$;
    $this.head\ =\ n$;
    $this.count = this.count + 1$;
    $return\ ()$;
  }
  $T\ head\langle R^a_m\rangle(unit\ u)$ {
    $return\ this.head.val$;
  }
  $LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ reverse\langle R^a_m \mid\ R1 \succcurlyeq R^a_m\rangle(unit\ u)$ {
    $let\ LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ xs\ =\ new\ LinkedList\langle R^a_m, R1\rangle\langle T\rangle\ (this.head.val)$;
    $let\ LinkedListNode\langle R^a, R1\rangle\langle T\rangle\ cur\ =\ this.head.next$;
```

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

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<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.

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

Region Type Inference + Exists

Wednesday, August 13, 2014 11:09 PM

In this wiki, we extend type inference to existential types of transferable regions.

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
$N \ ::= \ cn\langle \bar{T} \rangle$ //Instantiated class type
$C \ ::= \ \text{class } cn\langle \overline{\alpha \lhd N} \rangle \lhd N \ \{\overline{T\ f}; k\ ; \bar{d}\}$ //Class Definitions
$k \ ::= \ cn\ (\overline{T\ x})\{\text{ super } (\bar{v});\ \overline{\text{this}.f = v;}\ \}$ //Constructors
$d \ ::= \ T\ mn\ (\overline{T\ x})\ \{s;\ \text{return } e;\}$ //Methods
$T ::= \alpha \mid N \mid Object \mid Region\langle T \rangle \mid int \mid bool \mid unit$ // Types
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid \ letregion\ \{\ s\ \} \mid open\ x\ \{\ s\ \}$
$\quad \mid open^a\ x\ \{\ s\ \} \mid s; s \mid x.set(e) \mid x.transfer() \mid x.giveUp()$
$e \ ::= \ c \mid x \mid e.f \mid e.mn(\bar{e}) \mid \text{new } N(\bar{e}) \mid (N)\ e \mid x.get()$ //Expressions

The Target Language

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

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 ∧ C2 ∧ C3 ∧ $\bar{\rho} \succcurlyeq \rho^a$
    (D,$\psi_i$) = normalize(C)
    $\overline{N_T} = \psi_i(\bar{N})$
    $N_T = \psi_i(N)$
    $\overline{\tau_T} = \psi_i(\bar{\tau})$
    $\rho^a_T = \psi_i(\rho^a)$
    $\overline{\rho_T} =$ (frv($\overline{N_T}, N_T, \overline{\tau_T}$)) $- \{\rho^a_T\}$
    $\phi = D - \{\overline{\rho_T} \succcurlyeq \rho^a_T\}$ (* 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)
    $\overline{^XN}$ = templateTy($\overline{N_s}$) (* templateTy is an auxiliary fn defined at the end *)
    $^XN$ = 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. *)
    $\bar N = \psi_i(\overline{^X N})$
    $N = \psi_i(^X N)$
    $\bar\tau = \psi_i(\overline{^X \tau})$
  in
    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
    class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s}\}$  = CT(B)
    class $B\langle\rho^a_B\overline{\rho_B} \mid \phi_B\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\, f}\}$ = hdB
    $\overline{\tau_A}$= ctype($N_B$)  (* Types of super class constructor args *)
    $B(\overline{T_x\, x})\{\mathrm{super}(\overline{\upsilon_g});\ \overline{\mathrm{this}.f = \upsilon_f;}\}$  =  $k_s$
    $\overline{\tau_a}$ = templateTy($\overline{T_x}$)
    Ca = type-ok($\overline{\tau_a}$)
    _ = CT'[B $\mapsto$ 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*)
    $\Gamma$  =  $\cdot$, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\bar\alpha\rangle$, x: $\overline{\tau_x}$
    $\Sigma = \rho^a_B \cup \overline{\rho_B}$
    $\Delta = \overline{\alpha \lhd N_B}$
    ($\overline{\upsilon'_g : \tau_g}$, Cg)  = elab-expr($\Sigma$; $\Delta$; $\Gamma$; $\rho^a_B \vdash \overline{\upsilon_g}$)
    Csub = subtype-ok($\Delta \vdash \overline{\tau_g} <: \overline{\tau_a}$) (* Actual types of args to super should be subtype of
                  expected types. *)
    ($\overline{\mathrm{this}.f = \upsilon'_f}$, _, Cf)  = elab-stmt($\Sigma$; $\Delta$; $\Gamma$; $\rho^a_B \vdash \overline{\mathrm{this}.f = \upsilon_f}$)
    C  = Ca $\wedge$ Cg $\wedge$ Csub $\wedge$ Cf $\wedge$ ($\overline{\rho_B} \succcurlyeq \rho^a_B$) $\wedge\ \phi_B$
    (D, $\psi_i$)  = normalize(C)
    $(\bar N, N, \bar\tau, \bar\tau_x, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\rho^a_B))$
    $\bar\rho$  =  (frv($\bar N$, N, $\bar\tau$)) − {$\rho^a$}
    $\phi$  = project-constraints(D, {$\rho^a$, $\bar\rho$})  -  ($\bar\rho \succcurlyeq \rho^a$)  (* Collect residual constraints
                  over region params of class B that need to be recorded explicitly as refinement *)
    ($\overline{\upsilon''_g}$,  $\overline{\upsilon''_f}$)  =  ($\psi_i(\overline{\upsilon'_g})$,  $\psi_i(\overline{\upsilon'_f})$)
    k  =  $B(\overline{\tau_x\, x})\{\mathrm{super}(\overline{\upsilon''_g});\ \overline{\mathrm{this}.f = \upsilon''_f;}\}$
  in
    class $B\langle\rho^a\bar\rho \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\}$
  end

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

fun elaborate-methods-rec(Bdef, consB) = case Bdef of
  class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s;\}$ => consB  (* If there are no methods, we are done *)
| class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s} d_s\}$  =>
  let
    fullB' = elaborate-methods-rec  (class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s}\}$, consB)
    class $B\langle\rho^a_B\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\, f};\ k;\ \overline{d_B}\}$  = fullB'
    (* Our task is to elaborate method $d_s$*)
    $T_r\ m(\overline{T_x\, x})\{s; \mathrm{return}\ e;\}$  =  $d_s$
    $\tau_p$ = templateTy($T_r$)
    $\overline{\tau_a}$ = templateTy($\overline{T_x}$)
    $(\pi^a, \rho^a_m, \bar{\pi})$ = (new(), new(), frv($\tau_p, \overline{\tau_a}$)) (* $\pi^a$ denotes allocation context param of "m".
                $\rho^a_m$ is to be used as a dummy variable to facilitate the unification of allocation
                contexts for recursive calls of "m" with $\pi^a$. In other words, no region polymorphic
                recursion *)
    $d_t = \tau_p\ m\langle\rho^a_m \mid \rho^a_m = \pi^a\rangle(\overline{\tau_a\ x})\{\cdot\}$ (* 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 $B\langle\rho^a_B\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\ f};\ k;\ \overline{d_B}d_t\}$]   (* temporarily
                update CT' so that "this.m" gives correct type*)
    $\Gamma$ = $\cdot$, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\bar{\alpha}\rangle$, x: $\overline{\tau_a}$
    $\Sigma = \rho^a_B \cup \overline{\rho_B} \cup \pi^a \cup \bar{\pi}$
    $\Delta = \overline{\alpha \lhd N_B}$
    s0 = redec-rgn-handler($\overline{T_x\ x}$) (* Re-declare any arguments that are transferable
    region
                handlers. Elaboration ensures that region handlers are unpacked *)
    (s', $\Gamma'$, Cs) = elab-stmt($\Sigma;\ \Delta;\ \Gamma; \pi^a \vdash$s0;s)
    (e' : $\tau_q$, Ce) = elab-expr($\Sigma;\ \Delta;\ \Gamma'; \pi^a \vdash$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 $\wedge$ $(\overline{\rho_B} \succcurlyeq \rho^a_B) \wedge \phi_B$   (* Set of all constraints *)
    (D, $\psi_i$) = normalize(C)
    $(\bar{N}, N, \bar{\tau}, \overline{\tau_x}, \tau_r, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\tau_p), \psi_i(\rho^a_B))$
    $\bar{\rho}$ = (frv ($\bar{N}$, N , $\bar{\tau}$)) $- \{\rho^a\}$
    $\Sigma_\rho$ = $\rho^a \cup \bar{\rho}$ (* $\rho^a$ and $\bar{\rho}$ are new region vars that replace $\rho^a_B$ and $\overline{\rho_B}$ as region params of class
    B *)
    $(\rho^a_m, \phi^a_m)$ = 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($\overline{\tau_x}, \tau_r$)) $- \{\rho^a_m\}$
    $\Sigma_\pi$ = $\rho^a_m \cup \overline{\rho_m}$
    $\phi$ = project-constraints(D,$\Sigma_\rho$) - ($\bar{\rho} \succcurlyeq \rho^a$) (* Explicit constraints over region
    params of B *)
    $\phi_m$ = project-constraints(D,$\Sigma_\rho \cup \Sigma_\pi$) (* Constraints over region params of method
    m *)
    ($s''$, $e''$) = ($\psi_i$(s'), $\psi_i$(e'))
    d = $\tau_x\ m\langle\rho^a_m\overline{\rho_m} \mid \phi^a_m \wedge \phi_m\rangle(\overline{\tau_x\ x})$ {$s''$; $return\ e''$}
  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 redec-rgn-handler($\overline{T\ x}$) =
  foldr ($\overline{T\ x}$,$nop$, fn ($T\ x$,s) => case $T$ of
    $Region\langle T'\rangle$ => $(let\ T\ x = x); s$
  | _ => $s$)
```

```
fun templateTy(T) = case T of
  $\alpha$|int|bool|unit => T
| Object => Object<ρ> where new(ρ)
| A($\bar{T}$) => if A ∈ dom(CT') ∧ CT'(A) = $class\ A\langle\rho^a\bar{\rho} \mid \phi\rangle\overline{\langle\alpha \lhd N\rangle} \lhd N$ then
             then A<$\pi^a\bar{\pi}$><$\bar{\tau}$> where new($\pi^a\bar{\pi}$)∧ $|\bar{\pi}| = |\bar{\rho}|$ ∧ $\bar{\tau}$= templateTy($\bar{T}$)
             else T
| Region<$T_{root}$> =>let $\tau'$= templateTy($T_{root}$) in
                 let $\tau_{root}$= $[\rho/frv(\tau')]\,\tau'$ where new($\rho$) in
                   $\exists\rho.$ Region[$\rho$]<$\pi$><$\tau_{root}$> where new($\pi$)

fun superClasses(B<$\pi^a\bar{\pi}$><$\bar{\tau}$>) = case B of
  Object => {}
| Region[$\rho$] => Object<$\pi^a$>
| _ =>
    let $class\ B\langle\rho^a\bar{\rho} \mid \phi\rangle\overline{\langle\alpha \lhd N\rangle} \lhd N$ = CT'(B)in
    let N' = $[\bar{\pi}/\bar{\rho}][\pi^a/\rho^a]$ N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<$\pi^a\bar{\pi}$><$\bar{\tau}$>)= $\pi^a$
  | _ => error()

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

fun elab-stmt($\Sigma$; $\Delta$; $\Gamma$ ; $\rho^a \vdash$s) = case s of
  (let Region⟨T⟩ x = e) =>
  let
    $\exists\rho_0.$ Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩ = templateTy(Region⟨T⟩)
    Ct = type-ok(Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩)
    (e':$\tau_e$,Ce) = elab-expr($\Sigma$; $\Delta$; $\Gamma$ ; $\rho^a \vdash$ e)
    ($\tau_2$,s') = case $\tau_e$ of
       $\exists\rho_1.\tau_1$ => ([$\rho_0/\rho_1$] $\tau_1$, let ($\rho_0$, Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩ x) = unpack e′)
      | _ => ($\tau_e$, let Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩ x = e′)
    Csub = subtype-ok($\Delta \vdash$ $\tau_2$ <: Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩ )
    C = Ct ∧ Ce ∧ Csub
    $\Gamma'$ = $\Gamma$, x: Region[$\rho_0$]⟨$\pi_0^a$⟩⟨$\tau_0$⟩
  in
    (s',$\Gamma'$,C)
  end
| (let T x = e) => …
| ($e_1$ = $e_2$) where $e_1 \in \{x, e.f\}$ =>
  let
    ($e_1'$:$\tau_1$,C1) = elab-expr($\Sigma$; $\Delta$; $\Gamma$ ; $\rho^a \vdash e_1$)
    ($e_2'$:$\tau_2$,C2) = elab-expr($\Sigma$; $\Delta$; $\Gamma$ ; $\rho^a \vdash e_2$)
    (Csub, s') = case ($\tau_1$,$\tau_2$) of
        ($\exists\rho_1.$ Region[$\rho_1$]⟨$\pi_1^a$⟩⟨$\tau_1$⟩, Region[$\rho_2$]⟨$\pi_2^a$⟩⟨$\tau_2$⟩) =>
          (subtype-ok($\Delta \vdash \tau_2$ <: [$\rho_1/\rho_0$]$\tau_1$),
           $e_1$ = pack[$\rho_2$, $e_2$] as $\exists\rho_1.$ Region[$\rho_1$]⟨$\pi_1^a$⟩⟨$\tau_1$⟩)
       | _ => (subtype-ok($\Delta \vdash \tau_2 <: \tau_1$), $e'_1$ = $e'_2$)
     C = C1 ∧ C2 ∧ Csub
   in
     (s',Γ,C)
   end
| (open x { s }) =>
| (open$^a$ x { s }) =>

fun elab-subtype-ok ($\Delta \vdash e : \tau_1 <: \tau_2$) = case ($\tau_1$,$\tau_2$) of
   ($\mathrm{Region}[\rho_1]\langle\pi_1^a\rangle\langle\tau_1\rangle, \exists\rho_2.\tau'_2$) =>
   let
     Csub = subtype-ok($\Delta \vdash \tau_1 <: [\rho_1/\rho_2]\tau_2$)
     e' = $\mathrm{pack}[\rho_1, e]$ as $\exists\rho_2.\tau'_2$
   in
     (e',Csub)
   end
| _ => (e, subtype-ok($\Delta \vdash \tau_1 <: \tau_2$))

fun elab-expr ($\Sigma; \Delta; \Gamma; \rho^a \vdash e$) = case e of
   (x.get()) =>
   let
     ($e' : \mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\tau_0\rangle$,Cr) = elab-expr ($\Sigma; \Delta; \Gamma; \rho^a \vdash x$)
     C = Cr ∧ ($\rho_0 \in \Sigma$)
   in
     ($e'.\mathrm{get}() : \tau_0$, C)
   end
| (new N($\overline{e}$)) =>
   let
     N = templateTy ($N_s$)
     Cn = type-ok (N)
     $\pi^a$ = allocRgn(N)
     $\overline{\tau_B}$ = ctype(N)
     ($\overline{e_1} : \overline{\tau_e}$,Ce) = elab-expr ($\Sigma; \Delta; \Gamma; \rho^a \vdash \overline{e}$)
     ($\overline{e_2}$,Csub) = elab-subtype-ok($\Delta \vdash \overline{e_1} : \overline{\tau_e} <: \overline{\tau_B}$)
     C = Cn ∧ Ce ∧ Csub ∧ ($\rho^a \succcurlyeq \pi^a$)
   in
     (new N($\overline{e_2}$) : N, C)
   end
| ($e_0.m(\overline{e})$) =>
   let
     ($e'_0 : \tau_0, C_0$) = elab-expr($\Sigma; \Delta; \Gamma; \rho^a; C_0 \vdash e_0$)
     $\langle\rho_m^a\overline{\rho_m} \mid \phi_m\rangle\overline{\tau_x} \to \tau$ = mtype(m, $\mathrm{bound}_\Delta(\tau_0)$)
     $\overline{\pi}$ = new(length($\overline{\rho_m}$))
     $\psi$ = $[\overline{\pi}/\overline{\rho_m}][\rho^a/\rho_m^a]$
     Cx = type-ok($\psi(\overline{\tau_x})$)
     Cr = type-ok ($\psi(\tau)$)
     ($\overline{e_1} : \overline{\tau_e}$,Ce) = elab-expr ($\Sigma; \Delta; \Gamma; \rho^a \vdash \overline{e}$)
     ($\overline{e'}$,Csub) = elab-subtype-ok($\Delta \vdash \overline{e_1} : \overline{\tau_e} <: \psi(\overline{\tau_x})$)
     C = Cx ∧ Cr ∧ Ce ∧ Csub ∧ $\psi(\phi_m)$
   in
     ($e'_0.m(\overline{e'}) : \psi(\tau)$, C)
   end
|
```

Design notes (SVC)

Friday, June 20, 2014 9:48 AM

1. Issues with current memory management solutions

Currently, programming languages with automatic memory management rely heavily on garbage collection (GC). A lot of research effort has gone into developing GC algorithms that work well in most of the cases. While the currently available solutions work well, there are still some applications (e.g. real-time data processing systems) that underperform due to GC. However, most of the data processing systems follow a simply pattern in which various operators (e.g. join, select) are applied over partitioned data. Moreover, these so called operators do not share state among them. This is certainly true for Naiad and Dryad which use as their core computational model, a collection of single-threaded actors. The actors are instantiations of classes that can have private state. Similarly, to Erlang's model actors can only communicate via message passing. Currently, these actors run in a single CLR process and end up sharing the GC. In the following examples we will highlight few problems with the current general memory management systems:

1.1. Examples

TODO: Examples
TODO: Run examples and get preliminary characterizations

1.1.1. Time based aggregation

1.1.2. Lists of objects

https://github.com/MicrosoftResearch/Naiad/blob/6a9f73e4adae3412c6fcbda030f9d4614bfe958a/Naiad/DataStructures/SpinedList.cs

1.1.3. Dictionaries of dictionaries

https://github.com/MicrosoftResearch/Naiad/blob/6a9f73e4adae3412c6fcbda030f9d4614bfe958a/Frameworks/DifferentialDataflow/CollectionTrace/CollectionTrace.cs

1.1.4. Sliding window aggregation

1.1.5. Private vertex dictionary

https://github.com/MicrosoftResearch/Naiad/blob/6a9f73e4adae3412c6fcbda030f9d4614bfe958a/Examples/Naiad/WordCount.cs

In the following sections of the document we will present a preliminary design of Broom, bespoke memory management for actor based systems. Broom improves performance by using simpler sequential memory management for the private state of the actors. Instead of using a single general GC, Broom provides a region based memory management system combined with a general GC.

2. Introduction

In this section we briefly discuss the currently available memory management solutions together with the advantages and disadvantages they have in an actor based world.

TODO: Come up with better dimensions, and distinguish between stop the world and incremental collectors.

	Mark-sweep GC	Mark-compact GC	Copying GC	RC GC	Generational GC	Per thread heap	Region based
Stops program	🟧	🟧	🟧	🟩	🟧	🟧	🟩
Traverses the entire object graph	🟥	🟥	🟥	🟥	🟧	🟩	🟩
Requires program changes	🟩	🟩	🟩	🟩	🟩	🟩	🟥 maybe

There are multiple available algorithms for GC. In this section we will briefly talk about the five main categories:

2.1. Mark-sweep garbage collection

2.2. Mark-compact garbage collection

2.3. Copying garbage collection

2.4. Reference counting

The main issue with garbage collectors based on reference counting is that tracking all the updates to references is very expensive. Most RC collectors try to apply various optimizations to reduce the amount of references tracked. Among the optimizations are:

- Deferred reference counting: defer counting for variables that are frequently modified. Buffer the

increments/decrements and only process them from time to time
- Coalescing: only the initial and the final states of a reference are necessary to calculate correct reference counts.
- Limited bit counts: most of objects have only a few references. Limit the number of bits allocated for storing the count. If it happens that the limit of count has been reached then move the counter into a stuck state and treat it differently
- Young objects: most objects die young. Most collectors exploit this information with the help of copying generational nursery

Reference counting garbage collectors in theory could be suited for the actor based world because they do not stop the execution of the program. However, in practice counting adds a lot of overhead and only recently RC GCs have managed to be competitive with other GC algorithms.

2.5. Generational garbage collection

2.5.1 The generational CLR GC

The current CLR GC is a generational garbage collector. The collector explicitly treats objects differently depending on their size. Objects that are < 85,000 bytes are labelled as small objects whereas the ones bigger or equal are labelled as large objects. Small objects are stored in 3 generations called: gen0, gen1, gen2. Gen0, also known as nursery, stores newly allocated objects. Objects from genX that survive a garbage collection are promoted to genX+1.

Details:
- when we collect genX we also collect all genY where Y < X
- when we collect the last generation storing small objects we also collect the generation storing large objects
- objects in genX can have references to objects in genY where Y > X
- in order to avoid traversing the entire object graph the GC maintains a card table (i.e. mark array)
- memory is always zeroed before allocating objects

Issues with the CLR GC

The CLR GC is a general collector that was not designed for the actor based world. Currently, it uses a per process heap (or 4, depending on how one thinks about it) for allocating objects. This means that whenever a GC is triggered, the collector will traverse the entire object graph. Depending on the type of the collector that is used this can give rise to long pauses due to collection. This increases the runtime of Naiad jobs which is many cases require all the actors to "synchronously" make progress.

TODO: Run the examples using the CLR GC

TODO: It may be interesting to run Naiad using 1 process with 4 threads vs Naiad using 4 processes with 1 thread. The purpose of this experiment is to check if there is any difference in have 4 heaps handled by 4 instances of the CLR GC vs having 1 heap handled by 1 instance of the CLR GC. Need to check how messages are passed among threads.

2.6. Region based memory allocation

Region based memory management is a type of memory management in which objects are assigned to a region. Similarly to stack allocation the goal of region-based memory allocation is to efficiently support allocation and de-allocation of objects. In contrast to stack allocation, region based memory management trade-offs ease of allocation/de-allocation and reasoning with a more expressive object lifetime management.

In comparison to heap based allocation region based memory management has several advantages:
- Splits the heap into smaller entities/blocks that are easier to manage (e.g. depending on the region semantics, garbage collection can be activate on a per region basis. Hence, it will have a smaller impact on the runtime of the programs)
- Region-based memory management combined with static object lifetime inference can avoid GC by simply freeing the regions not in use
- Fast object deletions for collections or object structures with identical lifetime

Before we proceed to explain the types of regions present in Broom we briefly define the terminology:
- actor = an instantiation of a class that runs in a single thread
- action = a method of an actor that receives a single parameter. The parameter represents the input (i.e. message) to the action. Similarly, an action can only return a message which in turn will become input for one or more actors.

QUESTION: Should we try to model everything as a region or just use the current design which has 3 regions?
TODO: Think/add other potential dimensions

Regions can have different properties depending on where they sit across a set of dimensions. In Figure 1 we can see several of the aforementioned dimensions. The most important ones are scope and lifetime. In the first stab at defining the different types of regions we only consider scope and lifetime. Hopefully, the other ones will require new types of regions:)

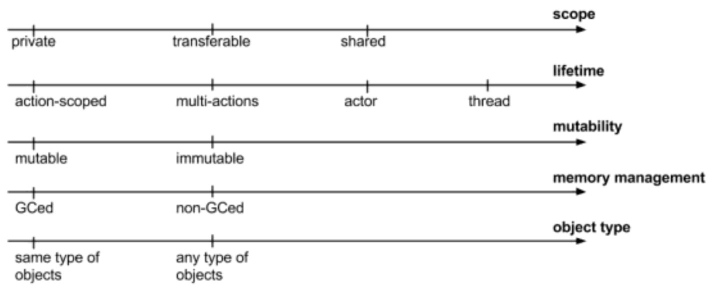


Figure 1

In Broom there currently are 5 different types of regions: **local**, **private**, **transferable**, **shared** and **public**. In the following subsections we describe each one of them.

2.6.1. Local regions
Local regions are the most restricted type of region. They can only be allocated within an action of an actor. Their scope is private and their lifetime is action scoped. They cannot be used across multiple actions, even if they are part of the same actor. One can think of local regions as a scratch pad. Their purpose is to store local data such as collections that currently go on the heap. Their main requirement is to provide very fast allocation and de-allocation of objects that are not shared with any other threads or even actions.

Local regions can move on all dimensions. They can be **GCed/non-GCed**, store objects of the **same /different type**, **mutable/immutable**. The only combination that does not make sense is local region, GCed and immutable.

2.6.1. Private regions
Private regions have a private scope and multi-action lifetime. They are declared per actor and mainly used to store data that is passed between the actions of an actor. Moreover, private regions can be used to store actor-level local data such as information about the where function in a Naiad Where vertex.

Private regions can vary on all the dimensions as well. Although, they will **most likely** be **GCed**, storing **different types of objects** and **mutable**.

2.6.2. Transferable regions
Transferable regions are less restricted that private regions. They are allocated by an actor. In contrast to private regions they can span multiple actors and implicitly multiple actions. However, only one actor can have a reference to a transferable region at a time. They are mainly used to store the messages that get passed between actors.

Transferable regions can be **GCed/non-GCed**, storing objects of the **same/different type** and **mutable**. Unless the same message is passed on between actors, it does not make sense to have immutable transferable regions.

2.6.3. Shared regions
Shared regions have multi-actor scope. Moreover, they can be accessed by more than an actor at a time. A shared region is reference counted and it is released when no actor holds a reference to it. The purpose of shared regions is to allow multiple actors to access shared data. It is not intended to be used as a mechanism to avoid communicating via message passing.

The shared regions can be **GCed/non-GCed**, storing objects of the **same/different type** and **mutable/immutable**. However, in practice they will mostly be **non-GCed**, storing objects of the **different type** and **immutable**.

We also considered a per-thread region. However, it does not make sense to have one because actors can be stopped and later on restarted on a different thread.

2.6.4. Public regions
There is only one public region. The public region is just a name of the shared standard heap space. This is just to unify everything under the region concept.

The public region is **GCed**, storing objects of **different type** and is **mutable**.

3. Design
In this section we discuss the properties of the regions with regards to the previously mentioned dimensions: scope, lifetime, mutability, memory management, types of stored objects.

(x,y) = green if x can have y scope, y lifetime etc.
        = orange if certain conditions apply
        = red if it cannot

In the following table we show the properties of the regions with regards to the **scope**:

	Private scope	Transferable scope	Shared scope
local	🟩	🟥	🟥
private	🟩	🟥	🟥
transferable	🟩	🟩	🟥

shared	🟩	🟩	🟩
public	🟩	🟩	🟩

In the following table we show the properties of the regions with regards to the **lifetime**:

	Single-action	Multi-action	Multi-Actors	Immortal
local	🟩	🟥	🟥	🟥
private	🟩	🟩	🟥	🟥
transferable	🟩	🟩	🟩	🟥
shared	🟩	🟩	🟩	🟥
public	🟩	🟩	🟩	🟩

In the following table we show the properties of the regions with regards to the **memory management**:

	non-GCed	GCed
local	🟩	🟩
private	🟩	🟩
transferable	🟩	🟩
shared	🟩	🟧
public	🟥	🟩

In the following table we show the properties of the regions with regards to the **mutability**:

	Mutable	Immutable
local	🟩	🟩
private	🟩	🟩
transferable	🟩	🟧
shared	🟧	🟩
public	🟩	🟥

In the following table we show the properties of the regions with regards to the **object type stored**:

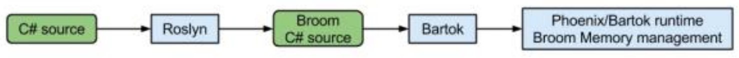
	Same object size	Different object size
local	🟩	🟩
private	🟩	🟩
transferable	🟩	🟩
shared	🟧	🟩
public	🟥	🟩

3.1. Design overview

In a region based memory management environment we are looking to answer the following 4 main questions:

1. **How** is the mechanism chosen? What is the input knowledge we have (Section 3.2)?
- statically analyze code
- dataflow graph
- statistics, cardinality
- SLAs
2. **Where** is the data stored (Section 3.3)?
- Region management
- Shared heap
- Stack
- I/O buffers
- RPC/RDMA buffers
3. **Who** manages the data (Section 3.5)?
- Reference counting + GC
- Standard GC
- Non-managed
4. **When** is the data freed (Section 3.4)?
- at the end of an action
- during the action
- garbage collected

We hope to answer the questions with the following system/design:



3.2. Roslyn transformations

TODO:

3.3. Roslyn to Bartok API

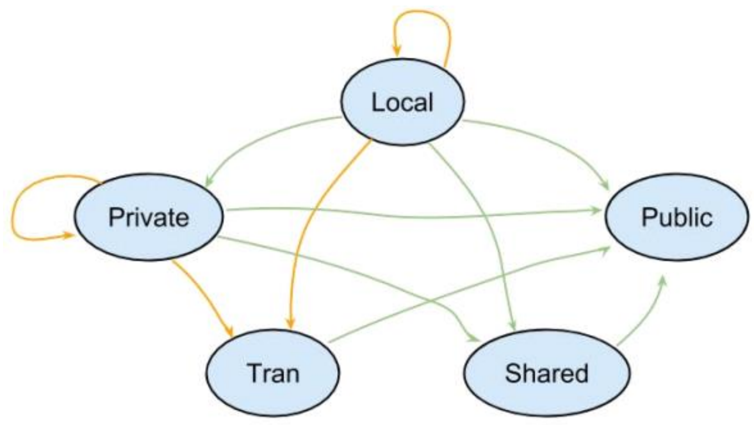
TODO:

- Arguments in favor of open: safety checks, avoids alias analysis
- No pointers to regions in the GC heap
- Private statically allocated and de-allocated actor regions: do not need to be collected, can contain refs to objects in regions
- UDF example

3.4. Memory safety

Green means that the region is always allowed to store a pointer to an object located in the other type of the region. Orange means that the region is allowed to store a pointer to an object located in the other type of region only if certain conditions are met:

- Local-local, Private-private: only if the lifetime contains the lifetime of the referenced region
- Local-transferable, Private-transferable: only if the actor has control over the transferable region (i.e. does not access it if the ownership has been passed)



TODO: things to consider: within a vertex vs between vertices on the same machine, different machines, same process

3.4.1. Statically scoped regions

Most original region type systems use statically-scoped regions. They guarantee memory safety by making sure that a program meets the following constraints:

- if an object in R1 references an object in R2, then region R2's lifetime must contain the lifetime of region R1
- if a stack-allocated variable V points to an object in region R, then R's lifetime must contain the lifetime of variable V

These properties can be achieved using a stack based solution. Whenever we create a region we store a pointer to the region on the stack. Eventually, we pop the pointer off the stack and freeing the region at the same time. This ensures that no stack-variable points to objects inside a free region.

However, statically scoped regions are limited in expressivity. A region's lifetime is identical to the call stack. This is inadequate for multi-action scoped regions such as: transferable, shared.

3.4.2. Dynamically scoped regions

Dynamically scoped regions work in a similar way as statically scoped regions. We also store a pointer to the region on the stack. However, when we pop the pointer we can choose either to free it or to temporarily "close" it. A closed region is temporarily inaccessible to the program. We can open again a region by pushing a pointer to the region back on the stack.

In order to guarantee memory safety we enforce the following restrictions:

- A pointer from an object in a dynamic region cannot point to any object in another region
- When a region has been sealed, we guarantee that there are no pointers to objects inside the region
- We still need to store pointers to the region. Memory safety can be achieve with a linear type (i.e. an actor can store only a unique pointer to a dynamic region) or by reference-counting

Moreover, transferable regions do not respect the LIFO constraint. In other words, we can allocate a region in method A and free it in a later method B. If during method A we created any pointers to the region, then we will end up with dangling pointers. This problem is solved with the help of the following constraints:

- All allocations and de-allocations within a transferable region happen within a lexical block
- An open region cannot be de-allocated or transferred
- The effect of this is that a dangling pointer can be caught in the syntax of the language
- A transferable region should not have pointers to other regions

Note: In order to be able to work with a region after we unseal it we need a root object or some sort of information about what's in the region.

3.5. Memory management

Questions that must be addressed:

1. Does every region have a root object?
2. How aggressive do we want to be with regards to de-allocation? If we store pointers to regions on the stack then we can only de-allocate the regions in a LIFO manner.
3. What page sizes are we using?
4. If we are going to build around the heap of pointers to regions abstraction then how are we going to ensure safety? (see later) Safety is easy to achieve in the stack based version.

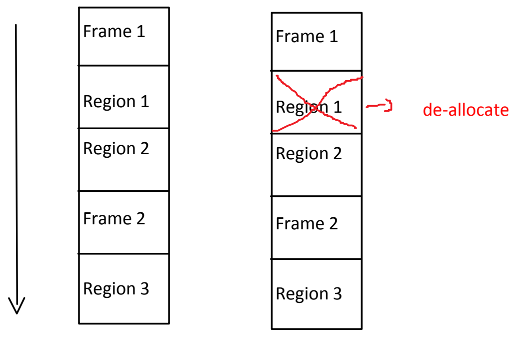
3.5.1. Local region

There are two options when it comes to allocating local regions:

Stack of local regions

Each entry in the stack contains a pointer to the root object of the local region. The problem with this approach is that one cannot free the regions just after their usage. Instead, the regions are occupying memory until the stack frame is popped. Let's look at an example:

```
void example(message) {
   LocalRegion<List<Object> localReg;
   populate localReg;
   for () {
      use localReg;
   }
   // localReg could be freed here
   moreProcessing();
}
```



In this example, we create a local region that is used to store some temporary data. However, the local region is not used in the moreProcessing method. Memory could be saved by freeing the region before moreProcessing is called. This is not possible under the model in which we store references to the root object of the region on the stack. See the example to the right.

One big advantage of the stack-based allocation is that the lifetime of the regions respects LIFO. Thus, it is very easy to reason about which regions can be referenced by another region.

Metadata block for local regions

A block of memory is allocated on the heap for storing local region metadata information (i.e. references to root objects). The area of the heap which stores the metadata block is unmanaged, but the block itself is managed using simple RC GC. Moreover, the metadata information for each region should be of constant size. With these constraints we can use a simple free list allocator together with a reference counting collector.

Note: Collection is not really required here because we have information about the lifetime of the regions. However, I think we should re-use the collector logic implemented for handling metadata blocks for other types of region in order to reduce the implementation workload. Using a RC collector should not add any significant overheads.

Using a metadata block for storing references to local regions has certain advantages:

- Regions can be released as soon as they get out of scope. Under a stack based solution a region can be freed only after all the subsequently allocated regions are freed.
- Metadata information per region is of constant size. This means that garbage collecting the metadata block should be fairly easy.

However, there are some disadvantages that ought to be considered:

- We are departing from the simple LIFO region lifetime. We can either use a similar solution as the one used for transferable/shared regions or do not provide memory safety and require the programmer to use his or hers regions appropriately.
- Stack variables can end up pointing to objects located in regions that have been freed. One could extend the RC for regions to include stack variables. Under such a scenario regions will only be freed after all the variables pointing to them will be popped. In the case when no stack variable is pointing to a region then the region can be de-allocated as soon as it goes out of scope.

Note: If we are going to use the stack based solution for local regions then we should probably get rid of them altogether. With the stack based solution everything a local region can do can also be done with a private region.

3.5.2. Private region

Similarly to local regions, private references to root objects of the private regions can be stored either on the stack or in a metadata block. One thing to note is that if references are stored on the stack then the differences between local and private become a bit blurry. In such cases we may want to completely drop local regions and just use private regions for storing local data as well.

3.5.3. Transferable region

They will be handled using a metadata block as well.

3.5.4. Shared region

Upon the start of a process the runtime will allocate an unmanaged block of memory. Let us call this block the shared metadata block. In this block we will store pointers to the head block of each shared region. Whenever, a new shared region is created we allocate space for a new pointer (plus some state such as reference counter) in the shared metadata block. The pointer will point to a freshly allocated segment of memory into which the objects will be stored. If the data the user wants to store in a shared region does not fit into a single segment then we will store it in a linked list of segments.

TODO: Investigate a better option that just representing a shared region as a list of segments

The metadata block is GC using a reference counting garbage collector. However, in this context a counter is increased whenever an actor is using the associated shared region and decremented whenever an actor stops using the region. Moreover, if the region is mutable then the region itself is garbage collected using either a generational GC or a RC GC.

Shared regions can only store pointers to the public region. They cannot store any references to objects located in any of the other types of regions. However, local and public regions can store references to objects store in shared regions. It is the duty of the application to make sure that these regions do no end up with dangling pointers.

3.5.5. Public region

The public region is essentially the current per process heap. We hope to be able to reuse the CLR GC code for managing the public region. Hence, it will GCed using a generational collector. We will have to adapt the traversal which is generating the object root set.

Every other type of region can store pointers to objects located in the public region. In order to provide memory safety the runtime has to do one of the following two things:

1. Traverse every region in order to find all the references to the objects stored in the public region. This is likely to be prohibitively expensive.
2. Use a reference counting garbage collector for the public region. However, it is difficult to implement a competitive RC GC in practice.

4. Examples/Evaluation

- run the same Naiad examples as in motivation
- change spine arrays in Naiad and show improvement over simple implementation. Compare with spine as well.
- maybe port something written using Orleans

5. Discussion

This is the best thing in the world maybe universe (aliens don't tend to publish too much)!!!

6. Related work

Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance
Stephen M. Blackburn, Kathryn S. McKinley

Taking Off the Gloves with Reference Counting Immix
Rifat Shahriyar, Stepher M. Blackburn, Xi Yang, Kathryn S. McKinley

Myths and Realities: The Performance Impact of Garbage Collection
Stephen M. Blackburn, Perry Cheng, Kathryn S. McKinley

The Case for Profile-Directed Selection of Garbage Collectors
Robert Fitzgerald, David Tarditi

Region-Based Memory Management in Cyclone
Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, James Cheney

Combining Region Inference and Garbage Collection
Niels Hallenberg, Martin Elsman, Mads Tofte

Experience With Safe Manual Memory-Management in Cyclone
Michael Hicks, Greg Morrisett, Dan Grossman, Trevor Jim

Memory Management with Explicit Regions
David Gay, Alex Aiken

Towards region-based memory management for Go
Matthew Davis, Peter Schatte, Zoltan Somogyi, Harald Sondergaard

Lifetime Management in System C# (describes Borrowed)
https://microsoft-my.sharepoint.com/personal/jaredpar_microsoft_com/_layouts/15/WopiFrame.aspx?sourcedoc=%7bffb845de-01ad-4930-9ec6-56f01347ab5e%7d

The garbage collection handbook: the art of automatic memory management
Richard Jones, Anthony Hosking, Eliot Moss

Orleans: Cloud computing for everyone
Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, Jorgen Thelin

Region annotations and code-inference

Thursday, July 31, 2014 9:19 AM

These notes are still work in progress. Will send an email when the first draft is ready.

Region-based cost-functions

- The cost functions are derived from the implementation and micro-benchmarks for the region based memory management implemented on top of Bartok.

- The cost for creating and removing a region
- The cost for growing (increasing the size) of a region
- The cost for garbage collecting a region
  - Is there a difference in the cost when we have single-root (transferable) vs. multi-root regions (actor-scoped, etc.).
  - How does it correlate to the size of the regions?
  - How does it correlate to the cardinality and size/type of a region elements?
- The cost for opening and closing a region for reading/writing? Or for inAllocationContext?

- Based on this information, we can decide:
  - Appropriate initial size of a region (based on program runtime statistics) .
  - Whether we have higher preference for one (small number) of large temporary regions, or many (large number of) smaller regions.
  - Whether a region should be grow-able?
    - Up until which point (limit)?
    - How should the limit be determined, is it the size of a region, object cardinality, or something else?
  - How often should we open/close a region. (for read/write or allocation).

- e.g. if resizing (growing) a region involves making a copy of all its elements to another block it maybe makes more sense the do garbage collection instead. Of course, it depends on the initial size of the region in the first place.

Flattening steps for C# Naiad (Lindi) vertices in Roslyn

- Nested method calls

```
var output = m1 (a1, m2( a2, m3()), a4);

var x1 = m3 ();
var x2 = m2 (a2, x1);
var output = m1 (a1, x2, a4);
```

- Chains of processing methods

```
List.ofType<>().Descendants().First().Count();

var x1 = list.ofType<>();
var x2 = x1.Descendants();
var x3 = x2.First();
```

var x4 = x3.Count();

- Simplify method arguments. Replace object field access with a new variable.

  var buf = new Buffer<TTime, TOutput>(message.time);

  var time = message.time;
  var bug = new Buffer<TTime, TOutput>(time);

- var matches = KeyEntry.Second.Values;

- What about cases like this:
  If (!this.Values.TryGetValue(key, out keyEntry) {}
- Consider that the statement has two different meanings for
- (a) control flow/liveness analysis
- (b) region grouping of objects, argument/return type manipulation as a result of method invocation.
- We can identify these methods by checking for the out keyword.

Brainstorming about code-inference and region-annotation

- Transferable regions should be self-sufficient.
  - once it is transferred all the objects referenced from its objects need to be accessible. Hence, the notion of cloning objects.
- Transferable regions can be garbage collected (on demand), if marked as such by an actor. The actor is responsible for doing the clean-up prior to sending it to another actor. TODO: Add example why this is needed.
  - Anything that is not linked to the root of a region can be garbage collected.
- Collections may not contain objects of different regions,
  but may also contain different regions (region handlers) as items. TODO: Add examples why this is useful.
- There can be only one actor-scoped region. TODO: Add why.
- If a collection is marked as actor-scoped then it may only hold items belonging to the actor-scoped region, or transferable regions as items.
  - There is an additional constraint that must be enforced making the transferable regions possible (we delete it from the collection once it has been sent out / transferred to another actor). Otherwise, we will have dangling pointers.
- If a collection is marked as transferable region, then all its elements must be also in the transferable region (self-sufficiency rule).

Preliminary action-plan for region-annotation

- Region-annotate all classes (except the Actor / Vertex class) prior to this phase. This can be done using Gowtham's region type inference rules explained in Region Type Inference.
- Build a reference-to graph for all objects in the actor/vertex class/methods.
- "=" in a statement means reference. -> add an edge in the reference-to graph.
- Every node in the graph corresponds to an object and its region
- Rules for set-up and processing the graph:
  - Every actor field is in the actor scoped region (AS)
    Potential exceptions are the items in a collection (?). They can also be transferable regions.
  - Every input message is marked as read only and is input transferable region (TRin).
  - Every outgoing message is an output transferable region (TRout).
  - If there is a reference between:
    - Tmp -> AS: do nothing, everything is OK. Because the AS region outlives the Tmp.

    - Tmp->TRin: the input region (TRin) needs to be open.
    - Tmp->tmp: can unify the regions, policy depends on cost function
    - AS->tmp: deep-copy to either AS, or TR (depending on the final destination/usage).*
    - AS->TRin: deep-copy
    - AS->TRout: deep-copy
    - TRin-> x: the input transferable region is read-only, and self-sufficient, so there are no outgoing references.
    - TRout->tmp: deep-copy
    - TRout->AS: deep-copy
    - TRout->TRin: deep-copy
- The allocation context of a function call is always a tmp. Even though the object returned may be annotated with more than one region (i.e. different fields, etc. belong to potentially different regions) at this point they are all in one same region. If we want to copy out only the content of one of the regions, we should be able to do that with the help of the runtime.
  As example imagine a function returning a List<Result> result_list object where every Result object will end up being transferred as a message, i.e., is a transferable region. The spine of the list can be potentially placed in a different region than its elements but in this case they are all allocated in the tmp region context. When copying the content of the corresponding Result element to the transferable region we do not want to copy the entire result_list object but just the necessary fragment of it.
- When invoking a function the tmp allocation context will be removed (go out of scope) as soon as the function returns. Therefore, at the closing brackets of the using block we will need to copy the rooted (return object, and function input arguments) to the corresponding regions. The order of copying is, however, important in order to preserve consistency among the different copies of the objects.
- The copy object implementation keeps track of whether an object has been copied and what its destination is.
  At the return of a function, i.e., at the closing bracket of a tmp using block, we order the regions that outlive the current tmp region context in decreasing order. We first copy thing that go to the AS (longest living region). When copying then objects to, for example, an action-scoped region we first check whether the object is already present in a region that outlives the current one and if so, add a reference to it, otherwise copy the object to the action-scoped tmp region.
  Additional optimizations, if an object was marked to be copied to tmp and then to output transferable region, but does not undergo modifications in between, then we copy it to the output region directly and avoid unnecessary overhead.
- Because of the constraints imposed on tmp regions, tmp->tmp has to be unified. By constraints here we refer to the fact that the tmp/scratchpad regions do not have region handles and unique names and therefore cannot be opened/closed upon demand. They are lexically scoped.

Liveness analysis

Live regions (definition taken from the Region allocation and transformation for Java programs)
- The region is reachable from at least one live variable.
- The execution of the program may access the region in the future.

A live region can be 'achieved' differently for different types of regions:
- If 'tmp'; a region can become 'live' after its creation, and 'dead' after its removal.
- If 'AS': a region is 'live' by definition; the same holds for action-scoped region.
- If 'TR': a region is 'live' in the scope between 'open' and 'close'

Regarding actual creation and removal of transferable regions:
- An input transferable region is not created; it is removed when getting out of scope (return of an actor-scoped method), unless it is referenced by the AS. Then we remove it, as soon as the reference from the AS is removed.

- An output transferable regions is not removed -> because it is either transferred or referenced by the AS. The creation can be when sending a message. There will be cases when instead of sending it (within the same method) where it is created, we assign a reference to it from the AS. We can create the output transferable region when setting up the reference from the AS.
  Whether a reference from within the AS is in the AS region or a transferable region is decided after doing a pass over all the methods in the Actor/Vertex class.

Merging TMP regions

- Unifying two temporary regions when there is a reference-to edge between them.
- How do we determine what should be action-scoped or in a separate scratch pad region?
  Maybe in the initial region-annotation, we can collect some statistics about the regions (examples include: number/size of objects currently in the region. Maybe if a region is too small, it will not justify its creation/removal so we'd be better off merging this region with another one. Also have a look at the cost-functions discussion above).
- I guess this fine-tuning can come at the very end. For the time being we can assume to either work with one big temporary (action-scoped) region or many small regions (one for each object, as generated after the first round of unification).

Enlisting and compiling Broom

Tuesday, March 18, 2014 9:34 AM

Enlisting into Broom sources

Sources are checked into the tools-broom branch in Midori
Set SDPORT=SDMIDORI:3268 and point client to //depot/branches/broom
You may have to request for CG-PARTNER access if you want to submit changes via ramweb

Building Broom

Follow directions in "MDN117 Bartok Developers Guide.docx" to build bartok and midori runtime.

Note, it will be convenient to slightly modify the Midori shortcut from the above document to:
Shortcut command line

```
%SystemRoot%\system32\cmd.exe /k setenv.cmd /x64 /codegenCLR /codegenTST /release
/wtkSystem /wtkRunDebug /bartokprebuilt=live /noselfhost /bartokFastAll
```

bhome -build -fast:RUNTIME-MIDORI-COMPILER-CROSSBUILDS-PHOENIX
bhome -build -fast:COMPILER-PHOENIX for incremental builds if you make changes to the Midori runtime

IDE support

Under the <ROOT>/Midori/Bin folder, you will find a folder MsharpIde.

1. Launch a Midori shell in administrator mode and run <ROOT>\Midori\setenv.cmd
2. In MSharpIde, run InstallIde.cmd

This will let you to create and build MSharp projects in Visual Studio.
Note: This has the side-effect of turning of squiggles even for C# projects. Run the corresponding UninstallXXX.cmd script to get back to the C# development environment.

Compiling Broom sources

Use bc -midori <FILE>.cs to compile sources with the broom runtime. If you have set the build configuration to debug, you will need to use the -build:debug option.

Object allocation in Bartok/Midori

Saturday, March 15, 2014 2:36 PM

MSIL to Bartok IR (compiler\ir\Msil2ir.cs, ConvertNewObject)
- Creates a NewObject ir effect

Bartok IR to Bartok Mir (compiler\convert\ToMir.cs, Run\ConvertNewObjectEffect
- Splits into allocation (AllocObj) and call to constructor

Bartok Mir to Phoenix Mir (compiler\convert\MirToMirPhoenix.cs, ConvertEffect\ConvertAllocObj)
- Generate a phx instruction GetVTable which gets the vtable reference for a type
- Generate an phx instruction AllocateObject with the vtable as an operand

Phoenix Mir (phx\mir-lower.cpp)

Bartok runtime (compiler\Backend\PhxBaseRuntime.cs)
- ExpandAllocateObject

Region allocator implementation

Tuesday, June 24, 2014 8:29 AM

Region metadata management

- **How do we access the objects stored in regions**.Regions are typed; each region has a single root object
  - If you want to allocate multiple objects (and access them in them in future :-)) in a transferrable region then you must do this through something like tuple types. Or an array, or some collection of some form.
  - For long-lived regions we can arrange so that the root object of the region **is the actor itself**, which means that it can contain references to whatever data the actor wants to keep for long periods of time.
  - Hence, no need for the runtime to keep extra roots. Ionel, do you agree?

- **Where do we store what is currently known as "region object" (i.e. an object containing pointer to the root object, reference counts, size of region, type of region, # of segments, etc).** Currently we allocate these "Region Objects" in a special rootRegion. Each of these objects contains an unmanaged reference to the storage for that region. Since we want to be able to store region pointers in object we have to be very careful to not move these Region Objects around in that rootRegion, so collecting that rootRegion is not straightforward.

  Instead we propose the alternative plan:
  PLAN
  - The region object is allocated at the beginning of the region storage and is never moved around.
  - We have unmanaged pointers from the GC heap to these region objects.
  - BUT (contrary to now) the region object can store a managed pointer to its root object.
  - GC heap contains TLS region-stacks that contain **unmanaged** pointers to regions.

  Notice that the invariant of the above plan is that it does not allow the GC to go into region storage. Just for completeness, note that there is another possibility:

  ALTERNATIVE BUT WORSE PLAN
  - The region object is still allocated at the beginning of the region storage.
  - We have **managed** pointers from the GC heap to these region objects
  - BUT everything inside the region object, including pointer to the root object, pointers to next free segments, metadata etc have to be **unmanaged**.

  It will be quite tedious to implement such untyped region objects and we believe that the first plan is much better. It also cleanly enforces our invariant "no managed pointers from GC to regions".

- **Growable regions**: A region is a (potentially non-contiguous) set of blocks. Each block is a fixed size, contiguous block of memory (multiple of page size)
  - Blocks belonging to a region are linked. Each blocks in the region stores a reference to the next segment in the region.
  - The region allocator (**who lives in the GC heap**) maintains an allocated and free list of blocks (that is, UIntPtrs).
  - A reference to the block storing an object can be obtained by masking the lower order bits of the object pointer.
  - The list of allocated blocks is used by the CLR GC to check if a reference belongs to region (in which case the GC does not traverse the reference nay further). Can someone think of a good way to design this check? One possibility (maybe stupid or quite tricky to implement?) is to allocate space for regions from the opposite end as the ordinary allocator so the check becomes a single range check (is this pointer larger than the smallest allocated segment?)

Region open/close

- Allocator maintains a per-thread stack of regions (using thread local storage to store a reference to the top of stack). The stack is itself allocated in a special root region (which has no GC yet).

Some notes from Kapil and DV about implementing ``region allocated region objects''

1. Current region allocation design

The current implementation of the region allocator stores all metadata about regions in a root region. When the allocator is initialized it allocates space for the root region on the unmanaged memory. There currently is only one root region per process. All the threads are using this space to store metadata objects (i.e. Region objects). The region objects contain information such as pointers to the beginning and end of the region, region size and a reference to a RegionRange object. Similarly to the region object, the RegionRange object is allocated on the root region. The RegionRange is just an object that stores the lower and the upper limit of the region.

References to region ranges are also stored in an array. This array is used to check if an address is located within a region. The check goes through all the region ranges and verifies if the address is within the limits of the region. The complexity of the operation is O(N) in the number of the regions. If we go ahead with this design I think we should store the region ranges in an enhanced segment tree that supports segment addition or in an interval tree. The insertion time for interval trees is O(logN) and the complexity of checking if an address is stored into a region is O(logN) as well.

Whenever a region is freed or goes out of context the region is marked as not allocated and it is appended to the list of free regions. The free list is intended to be used as a mechanism to avoid going to the operating system for every region allocation and deallocation. Whenever, a new region is requested, the allocator first checks if there's a free region large enough to meet the demands. Only if it cannot find such a region it does a VirtualAlloc. This essentially implements first fit.

However, the implemented first fit algorithm is basic. It does not return back onto the freeList the extra space within a region. For example, imagine a case when we want to allocate an object of size 8 bytes. The size of the first region in the free list is 8MB. In such a cases, the allocator will just use the whole region for just storing the 8 byte object. One could improve the algorithm by splitting the large region into a region of 8 bytes and a remainder region of 8MB-8bytes:). Moreover, the remainder region can be appended back onto the free list. Even with this improvement a free list allocator exhibit the following properties with time:

1. Small regions will accumulate towards the beginning of the free list
2. It will end up almost doing best-fit because the first match will be just a bit bigger than the requested region size

A better way to implement this is using segregated-fits allocation. The idea behind segregated-fits is to place regions into k bins. Bin j can store regions of size equal or greater than s(j-1) and smaller than s(j). Within each of these bin we can implement first-fit allocation or something better.

Currently, a region is declared using:
uint size = 100;
var region = RegionAllocator.AllocateRegion(size);

One currently has two options for allocating a new object within a region:

1. Pass the region as the first parameter when instantiating a class: TestObject test = new TestObject(region);
2. Open a RegionContext. Within a RegionContext all the objects are stored on the opened region:

```
var region = RegionAllocator.AllocateRegion(size);
using (RegionContext regContext = RegionContext.Create(region)) {
        var test = new TestObject();
}
```

The second option does not work yet. Need to investigate. Moreover, if the second option is used then the region is not appended to the free list upon exit from the using block.

Finally, if a region is used within a context then a reference to it is stored on a region stack that is stored on the unmanaged space as well. Each thread has its own region stack which is stored in the root region. Moreover, each thread stores a reference to the its region stack in its thread local storage.

Upon the creation of a RegionContext a reference to the region metadata object is pushed on the thread local stack. All the objects within a using block happen in the region at the top of the stack.

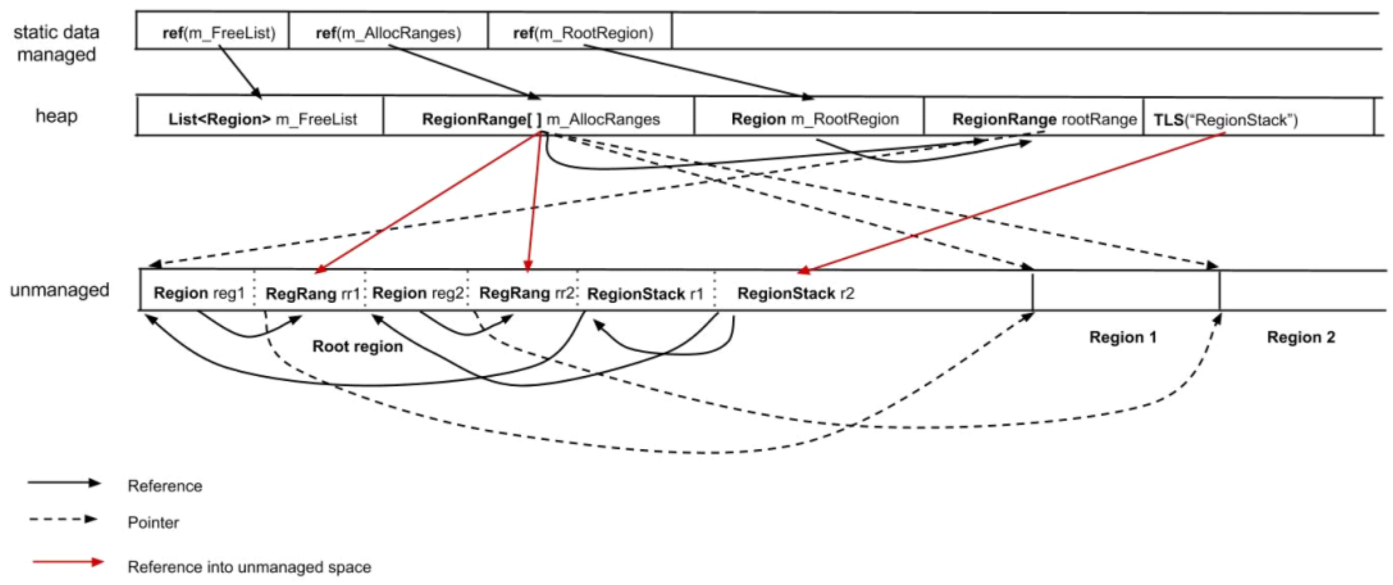


Figure 1. Memory pseudo-layout for the current implementation

NOTE: Regions are not allocated in contiguous blocks of memory. In other words, region 2 does not have to start where region 1 ends, there can be memory in-between that is used for other purposes. We represent them one after another just to simplify the figure.

Bugs or problems to be addressed in the next iteration of the implementation:

1. The initialization is currently done lazily (i.e. the allocator is initialized only when AllocateObject is called for the first time). From my understanding, in the current implementation, this can cause some leaks as multiple threads may end up allocating space for the root region before they update the isInitialized value.
2. Storing the RegionRanges in an array is not efficient with regards to querying if an address is part of a region. I think we should use a different data structure such as an interval or a segment tree.
3. The root region has a fixed size.
4. Garbage collection is required within the root region. However, this is not trivial to achieve because we have on the heap and the stack references to the region/regionRange objects stored in the root region. One could implement a non-compacting GC which is practice may work quite well because region and regionRange objects are of the same size. Only RegionStacks can create gaps. The downside of such a GC is that the Region and RegionRange objects of a region will potentially not be stored contiguously.
5. The regions cannot grow or shrink in size.
6. At the moment there's no way to GC within a region because we don't have information about

the root objects.

7. There is no implementation for transferable or shared regions.
8. The code fetching regions from the FreeList should be synchronized. In the current implementation two threads may end up re-using the same region.

2. Bugs fixed and issues to thinks about

1. References leftovers
2. Zero memory
3. Static in collections
4. Thread local storage
5. m_FreeList
6. Multi-threaded environment
7. Region Context
8. Fast Allocation Path

3. Proposed region allocation design

Issues we're trying to address:

1. How do we store objects within a region
2. How do we grow/resize regions
3. How is GC conducted within and among regions

```
Block {
 UIntPtr BasePtr;
 UIntPtr LimitPtr;
 UIntPtr AllocPtr;
 bool IsUsed;
 uint size;
 Block nextBlock;
 Block prevBlock;
}
```

```
Region {
 UIntPtr AllocPtr;   // These two pointers are only kept to avoid an extra indirection on the allocation
path.
 UIntPtr LimitPtr;  // However, block AlloctPtr needs to be updated lazily when a new object
allocation doesn't fit in the current block
 UIntPtr HeadBlock; // We may be able to store them as Block instead of UIntPtr. However, it
depends on where
 UIntPtr TailBlock;   // we store the region and block objects
 bool isGCed;
}
```

Mega-blocks: blocks can be merged into mega-blocks. Each mega-block will only have one block metadata object. Upon free the mega-blocks are again split into basic blocks.
TODO: Think about how we store the basic blocks such that they can efficiently be merged when we need to allocate large objects.

3.1. Where do we store the RegionStack<Region>?

1. On the heap
    a. Cannot keep pointer to the RegionStack because the stack may be moved around by the GC
    b. Need to make sure that all the objects created within the region stack end up on the heap as well. In other words, we need to change everything to new Object(null, args). This is to make sure that objects do not end up being stored in a region if a region context is open.

    c. Can keep references to regions if the region objects are stored on the heap.
2. In the root region
    a. The root region will need a GC. The GC may be relatively easy to implement because we will only store 2-3 types of objects. Hence, the GC will not have to compact the space and the allocator will only have to fill in the gaps left by the dead objects.
    b. Can keep pointer to the RegionStack
    c. Can keep references to regions if the region objects are stored within the blocks.
3. In the regions
    a. Each entry into the stack is stored in the corresponding region.
    b. RegionStack can hold Region objects because the GC will not go into the RegionStack.

3.2. Where do we store the blocks and regions?

1. On the heap
    a. Can add overhead to the GC if we have small regions
    b. Cannot hold pointers to metadata objects. In other words cannot store region/block references within regions.
    c. We need to use new Object(null, args) to make sure that metadata objects don't end up on a region if a region context is open.
    d. The RegionStack will have to be on the heap as well which means we won't be able to keep pointers to it. In other words, we won't be able to reference it from the BumpAllocContext. BumpAllocContext is used by the fast object allocator.
2. In the RootRegion
    a. The root region needs to be collected. However, there will only be around 3 types of objects stored on it.
3. Hybrid: Regions within blocks and blocks on the heap
    a. Easy to allocate new blocks
    b. Regions can be allocated by changing new such that new Object(block, args) allocates the object within the specified block
    c. Cannot easily get the block metadata object from an address. This operation may be required in the implementation of the region GC
    d. Cannot hold pointers to block objects because the objects may be moved around by the GC
4. At the beginning of blocks
    a. Doesn't answer the question of where we store the RegionStack
    b. Very easy to access the block metadata

    4.1. Store the block metadata directly at the beginning the block
    a. When we want to create a new Region we need to pass in a region. We could fake it by creating a stub region with region.AllocPtr = block.AllocPtr. Upon return we updated the block.AllocPtr to the value of reigon.AllocPtr

    4.2. Store the block metadata as an object at the beginning of the block
    a. Need to updated new Object(block..). I think we can do the same trick as in the previous case.

Compiler support for region allocator

Monday, March 17, 2014

Changes to Phoenix

- Add AllocateRegionObject as a new instruction to common.md. We specify the semantics of this instruction as one whose destination is a new object. The region is passed as the third parameter. This is to ensure that most of the logic in phoenix that assumes the vtable and type as the first and second parameters does not change.
- Dead code elimination
  - Checks if the destination is a new object. We don't need any changes here since the same logic for AllocateObject applies
- Stack allocation
  - Allocates objects on the stack whenever possible. Conditions include
    - If the object is scoped (does not escape?) OR
    - Size of the object is a constant and less than a specified max
    - Object does not escape
    - Alignment is compatible with the local stack alignment
    - Allocation does not occur inside a loop

  At the moment, this optimization does not apply to objects allocated in regions. Perhaps it should?
- AliasLattice
  - Checks if the new object has a finalizer side effect. Phoenix assumes that all types (except box types) may have a finalizer. We assume that's the case for region objects too.
- Changes to the runtime interface (runtime.pdl)
  - Add a method to expand AllocateRegionObject instruction

Open questions

- Semantics of finalizers for objects allocated in regions
- Phoenix has a GC type checker which checks if operands are the right type (GCed objects or not). Do we need to adapt this type checker in the presence of region-allocated objects?
- Generate GC safe points at actor boundaries

Todo

- Enable concurrent GC for a fair comparison, also test with different segment sizes to get the best performing GC

Notes on GC.cpp

Friday, April 11, 2014 10:48 AM

1. `init_gc_heap`:
    - Here we observe that every heap has a segment table. Each segment can contain stuff from multiple generations
    - It calls `create_gc_thread()`
2. `mark` class (specifies a range of some sort)
3. How to be able to make the GC work in one of our regions?
4. `garbage_collect()` does the work

RegionContext API

Tuesday, May 20, 2014 2:07 PM

The RegionContext API is designed to simplify the use of regions, espeically in existing library code.

```
using (RegionContext.Create(region))
{
        var dictionary = new Dictionary<int, int>();
        dictionary.Add(10, 10);
}
```

Design

Tuesday, May 27, 2014 1:48 PM

Implementation notes

Tuesday, May 27, 2014 1:48 PM

- Bartok uses a fast path for allocating certain kinds of objects. We disable fast memory allocation (PhxBaseRuntime.cs) through calls to native allocation routines