Safe Memory Regions for Big Data Processing

Gowtham Kaki
Purdue University, USA
gkaki@purdue.edu

G Ramalingam
Microsoft Research, India
grama@microsoft.com

Kapil Vaswani
Microsoft Research, India
kapilv@microsoft.com

Dimitrios Vytiniotis
Microsoft Research, UK
dimitris@microsoft.com

Abstract

Recent work in high-performance systems written in managed languages (such as Java or C#) has shown that garbage-collection can be a significant performance bottleneck. A class of these systems, focused on big-data, create many and often large data structures with well-defined lifetimes. In this paper, we present a language and a memory management scheme based on user-managed memory regions (called *transferable regions*) that allow programmers to exploit knowledge of data structures' lifetimes to achieve significant performance improvements.

Manual memory management is susceptible to the usual perils of dangling pointers. A key contribution of this paper is a refinement-based region type system that ensures the memory safety of C# programs in the presence of transferable regions. We complement our type system with a type inference algorithm that infers principal region types for first-order programs, and practically useful types for higher-order programs. This eliminates the need for programmers to write region annotations on types, while facilitating the reuse of existing C# libraries with no modifications. Experiments demonstrate the practical utility of our approach.

1. Introduction

Consider the example, from [4], shown in Fig. 1. This code represents the logic for a streaming query operator. The operator receives a stream of input messages, each associated with a time (window) t, processed by method `onReceive`. Each input message contains a list of inputs, each of which is processed by applying a user-defined function to create a corresponding output. The operator may receive multiple messages with the same timestamp (and messages with different timestamps may be delivered out of order). A timing-message (an invocation of method `OnNotify`) indicates that no more input messages with a timestamp t will be subsequently delivered. At this point, the operator completes the processing for time window t and sends a corresponding output message to its successor.

```
class SelectVertex<TIn, TOut> {
 Func<TIn, TOut> selector;
 Dictionary<Time, List<TOut>> map;
 ...
  void onReceive(Time t, List<TIn> inList) {
    if (!map.ContainsKey(t))
      map[t] = new List<TOut>();
    foreach (TIn input in inList) {
      TOut output = selector(input);
      map[t].add(output);
    }
  }
  void onNotify(Time t) {
     List<TOut> outList = map[t];
     map.Remove(t);
     transfer(successorId, t, outList);
  }
}
```

Figure 1: `SELECT` dataflow operator

This example is an instance of a general pattern, where a producer creates a data structure and passes it to a consumer. In a system where most of the computation takes this form, and these data structures are very large, as is the case with many streaming big-data analysis systems, garbage collection overhead becomes significant [4]. Furthermore, in a distributed dataflow system, the GC pause at one node can have a cascading adverse effect on the performance of other nodes, particularly when real-time streaming performance is required [4, 10]. In particular, a GC pause at an upstream actor can block downstream actors that are waiting for messages. However, often much of the GC overhead results from the collector performing avoidable or unproductive work. For example, in the process executing the code from Fig. 1, GC might repeatedly traverse the `map` data structure, although its objects cannot be collected until a suitable timing message arrives.

An important observation, in the context of processes of the kind described above, is that the data-structures exchanged between them can be partitioned into sets of fate-sharing objects with common lifetimes, which makes them good candidates for a region-based memory management discipline. A region is a block of memory that is allocated and freed in one shot, consuming constant time. A region may contain one or more contiguous range of memory locations, and individual objects may be dynamically allocated within the region over time, while they are deallocated en masse when the region is freed. Thus, a region is a good fit for a set of fate-sharing objects. In Fig. 1, the output to be constructed for each time window t (i.e., `map[t]`) can be a separate region that is allocated when the first message with timestamp t arrives, and deallocated after `map[t]` is transferred in `onNotify`.

Region-based memory management, both manual as well as automatic, has been known for a long time. Manual region-based memory management suffers from the usual drawbacks, namely the potential for invalid references and the consequent lack of memory safety. Automatic region-based memory management systems guarantee memory safety, but impose various restrictions. MLKit, which implements the approach pioneered by Tofte and Talpin [15, 16], for example, uses lexically scoped regions. At runtime, the set of all regions (existing at a point in time) forms a stack. Thus, the lifetimes of all regions must be well-nested: it is not possible to have two regions whose lifetimes overlap, with neither one's lifetime contained within the other. Unfortunately, the data structures in the above example do not satisfy this restriction (as the output messages for multiple time windows may be simultaneously live, without any containment relation between their lifetimes). We refer to regions with lexically scoped lifetimes as *stack regions* and to regions that do not have such a lexically scoped region as *dynamic* regions.

The goal of this work is a *memory-safe* region-based memory management technique that supports dynamic regions as first-class objects. Our focus, in this paper, is on dynamic regions, which can be safely transferred across address spaces. We refer to such dynamic regions as *transferable* regions. As with allocation and deallocation, transferring a memory region is fast, and therefore, transferring a data structure contained in a region is more efficient that traversing its objects in heap, and transferring them independently[1]. In the `SelectVertex` example from Fig. 1, the proposed region to contain the output for each time window t must be transferable due to the `transfer` operation on Line 16. As it is the case with `SelectVertex`, the transferred data is no longer accessed by the producer, so the transfer operation in our system deallocates the region once the transfer is complete.

With respect to memory safety, the key property we wish to ensure is that there are no invalid references: i.e., references to objects that were deallocated, or simply never existed. Transfer operation, with no additional checks, may cause memory safety violations, both at the producer of the data structure, and its (possibly remote) consumer. At the producer, any existing references into the data structure become invalid post transfer. If the data structure contains references to objects outside its (transferable) region, then such references become invalid in the context of the consumer. Safety violations of this kind are particularly unwelcome as the program with GC did not have them to begin with. Note that, while the references of later kind (i.e., references that escape a region), defeat the very purpose of a transferable region, hence need to be prohibited, references of the former kind (i.e., references into a region from outside) are not at odds with the concept of a transferable region, hence need to be permitted. In fact, allowing such references is crucial to performance, as any non-trivial program creates temporary objects, and it is undesirable to allocate them in a transferable region; such regions are meant for the data being transferred. Since transferable regions are first class objects in our setting, controlling references to and from such regions while ensuring their safety without significantly diluting the performance advantage of regions over GC is a challenging exercise.

In this paper, we describe an approach that restores memory safety in presence of transferable regions through a combination of a static typing discipline and lightweight runtime checks. The cornerstone of our approach is an `open` lexical block for transferable regions, that "opens" a transferable region and guarantees that the region won't be transferred/freed while it is open. Our observation is that by nesting a [15]-style `letregion` lexical block, that delimits the lifetime of a stack region, inside an `open` lexical block for a transferable region, we can guarantee that the transferable region will remain live as long as the stack region is live. We say that the former "outlives" the later[2], and any references from the stack region to the transferable region are therefore safe. Next, we note that by controlling the "outlives" relationships between various regions, we only allow safe cross-region references, while prohibiting unsafe ones. In the above example, an outlives relationship *from* the stack region *to* the transferable region means that the references in that direction are allowed, but not the references in the opposite direction. In contrast, if an `open` block of a transferable region R_0 is nested inside an `open` block of another transferable region R_1, we do not establish any outlives relationships, thus declaring our intention to not allow any cross-region references between R_0 and R_1. Finally, we observe that outlives relationships are established based on the lexical structure of the program, hence a static type system can enforce them effectively. By assigning region types to objects, which capture the regions such objects are allocated in, and by maintaining outlives relationships between

[1] Empirical studies in [4] support this claim

[2] We borrow the *outlives* relation from [2]. A comparison of our approach with [2] can be found in § 6.

various regions, we can statically decide the safety of all references in the program.

Of course, the utility of our approach described above is predicated on the assumption that we can enforce certain invariants on transferable regions. Firstly, a transferable region cannot be transferred/freed inside an open block of that region (i.e, while it is still open). Secondly, a transferred/freed region cannot be opened. These are typestate invariants on the transferable region objects, which are hard to enforce statically due to the presence of unrestricted aliasing. Techniques like linear types and unique pointers can be used to restrict aliasing, but the constraints they impose are often hard to program around. We therefore enforce typestate invariants at runtime via lightweight runt-time checks. In particular, we define an acceptable state transition discipline for transferable regions (Fig. 3), and check, at runtime, whether a given transition of a transferable region (*e.g.*, from *open* state to *freed* state) is valid or not. The check is lightweight since it only involves checking a single tag that captures the current state. We believe that this is a reasonable choice since regions are coarse-grained objects manipulated infrequently, when compared to the fine-grained objects that are present inside these regions, for which safety is enforced statically. An added advantage of delegating the enforcement of typestate invariants to runtime is that our region type system is simple[3], which made it possible to formulate a type inference that completely eliminates the need to write region type annotations. This, we believe, significantly reduces the impediment to adopt our approach in practical setting.

Contributions

The paper makes the following contributions:

- We present BROOM, a C# -like typed object-oriented language that eschews garbage collection in favour of programmer-managed memory regions . BROOM extends its core language, which includes *lambdas* (higher-order functions) and *generics* (parametric polymorphism), with constructs to create, manage and destroy static and transferable memory regions. Transferable regions are first-class values in BROOM.
- BROOM is equipped with a region type system that statically guarantees safety of all memory accesses in a well-typed program, provided that certain typestate invariants on regions hold. The later invariants are enforced via simple runtime checks.
- We define an operational semantics for BROOM, and a type safety result that clearly defines and proves safety guarantees described above.
- We describe a region type inference algorithm for BROOM that (a). completely eliminates the need to annotate BROOM programs with region types, and (b). enables seamless interoperability between region-aware BROOM programs and legacy standard library code that is region-oblivious. The cornerstone of our inference algorithm is a novel constraint solver that performs abduction in a partial-order constraint domain to infer weakest solutions to recursive constraints.
- We describe an implementation of BROOM frontend in OCaml, along with case studies where the region type system was able to identify unsafe memory accesses statically.

[3] § 3 contains all the type rules of our language, which occupy less than a page.

2. An Informal Overview of BROOM

BROOM enriches a simple object-oriented language (supporting parametric polymorphism and lambdas) with a set of region-specific constructs. In this section, we present an informal overview of these region-specific constructs.

2.1 Using Regions in BROOM

Stack Regions The "`letregion R { S }`" construct creates a new stack region, with a static identifier `R`, whose scope is restricted to the statement `S`. The semantics of `letregion` is similar to Tofte and Talpin [15]'s `letregion` expression: objects can be allocated by `S` in the newly created region while `R` is in scope, but the region and all objects allocated within it are freed at the end of `S`.

Object Allocation The "`new@R T()`" construct creates a new object of type `T` in the region `R`. The specification of the allocation region `R` in this construct is optional. At runtime, BROOM maintains a stack of *active* regions, and we refer to the region at the top of the stack as the *allocation context*. The statement `new T()` allocates the newly created object in the current allocation context. This is important as it enables BROOM applications to use existing region-oblivious C# libraries. In particular, given a C# library function `f` (that makes no use of BROOM's region constructs), the statement "`letregion R { f(); }`" invokes `f`, but has the effect that all objects allocated by this invocation are allocated in the new region `R`.

Region Identifiers Every live region in BROOM is associated with a static identifier that uniquely identifies the region within its scope. At runtime, if a `letregion` expression is evaluated multiple times in a loop or a recursive method, the corresponding identifier is bound to a new stack region each time. Any proposition involving static region identifiers is considered true at a program location if and only if the proposition is true under all possible evaluation contexts of that program location. For instance, consider the following example:

```
for (int i=0; i<=10; i++) {
  letregion R0 {
    letregion R1 {
      A a1 = new@R1 A();
      ...  } } }
```

The identifiers `R0` and `R1` are bound to new stack regions each time the loop is evaulated. Nonetheless, the propositions (a). `R0` $\succeq$ `R1`, and (b). `a1 : A@R1` (read as `a1` *refers to an object of type* `A` *contained in region* `R1`) are true at line 5, as they are true under all possible bindings of `R0` and `R1` at line 5.

Transferable Regions BROOM's *transferable regions* are an encapsulation of a data-structure that can be transferred between autonomous entities (*e.g.*, between two concurrently executing threads or actors). Hence, unlike stack regions, transferable regions are not constrained to have a lexically scoped lifetime. (Hence, we also refer to them as *dynamic* regions.)

Furthermore, transferable regions, unlike stack regions, are first class values of BROOM: they are objects of the class `Region`, they are created using the `new` keyword, and can be passed as arguments, stored in data structures, and returned from methods. A transferable region is intended to encapsulate a single data-structure, consisting of a collection of objects with a distinguished root object of some type `T`, which we refer to as the region's *root* object. The class `Region` is parametric over the type `T` of this root object.

The `Region` constructor takes as a parameter a function that constructs the root object: it creates a new region and invokes this function, with the new region as the allocation context, to create the root object of the region. The following code illustrates the creation of a transferable region, whose root is an object of type `A`.

```
Region<A> rgn = new Region<T>(() => new A())
```

In the above code, `rgn` is called the *handler* to the newly created region, and is required to read the contents of the region, or change its state. The class `Region` offers two methods: a `free` method that deallocates the region (and all the objects allocated within it), and `transfer` method that transfers the region to a consumer process. It is an abstraction of two possible forms of transfer: a transfer between two processes in a shared memory setting or a transfer between two processes in a distributed, message-passing, setting. The precise semantics of `transfer` are unimportant in the context of the region type system and we will not discuss them further.

Open and Closed Regions A transferable region must be explicitly *opened* using BROOM's `open` construct in order to either read or update or allocate objects in the region. Specifically, the construct "`open rgn as v@R { S }`" does the following: (a). It opens the transferable region handled by `rgn` for allocation (i.e., makes it the current allocation context), (b). binds the identifier `R` to this open region, and (c). initializes the newly introduced local variable `v` to refer to the root object of the region. The `@R` part of the statement is optional and may be omitted. The `open` construct is intended to simplify the problem of ensuring memory safety, as will be explained soon. We refer to a transferable region that has not been opened as a *closed* region.

```
1  class SelectVertex<TIn, TOut> {
2    Func<TIn, TOut> selector;
3    Dictionary<Time, Region<List<TOut>>> map;
4    ...
5    void onReceive(Time t,Region<List<TIn>> inRgn){
6      if (!map.ContainsKey(t))
7         map[t] = new Region<List<TOut>>
8                        (() => new List<TOut>());
9      open inRgn as inList {
10       letregion R0 {
11         foreach (TIn input in inList) {
12           open map[t] as outList {
13             TOut output = selector(input);
14             outList.add(output); } } } }
15     inRgn.free();
16   }
17   void onNotify(Time t) {
18      Region<List<TOut>> outRgn = map[t];
19      map.Remove(t);
20      outRgn.transfer();
21   }
22 }
```

Figure 2: `SELECT` dataflow operator in BROOM

Motivating Example Fig. 2 shows how the motivating example of Fig. 1 can be written in BROOM. The `onReceive` method receives its input message in a *transferred* region (*i.e.*, a *closed* region whose ownership is transferred to the recipient). Line 7 creates a new region to store the output for time `t`, initializing it to contain an empty list. Line 9 opens the input region to process it[4]. Line 10 creates a stack region `R0`. Thus, the temporary objects created by the iteration in line 11, for example, will be allocated in this stack region that lives just long enough. We open the desired output region in line 12, so that the new output objects created by the invocation of `selector` in line 13 are allocated in the output region. Finally, the input region is freed in line 15. The output region at `map[t]` stays as along as input messages with timestamp `t` keep arriving. When the timing message for `t` arrives, the `onNotify` method transfers the `outRgn` at `map[t]` to a downstream actor.

2.2 Memory Safety

Our goal is a type system that can ensure the memory safety of programs that use the region constructs described above. The key to memory safety in BROOM is the following restriction: an object o_1 in a region R_1 is allowed to store a pointer to an object o_2 in a region R_2 only if R_2 is guaranteed to outlive R_1. (A similar restriction applies in the case where o_1 is a stack-allocated variable.)

Enforcing this restriction is simple in the case of stack regions since the outlives relation between stack regions can be inferred from their lexical nesting. Unfortunately, inferring outlives relations in presence of transferable regions is

[4] We omit `@R` annotation in `open` when we don't need `R`.

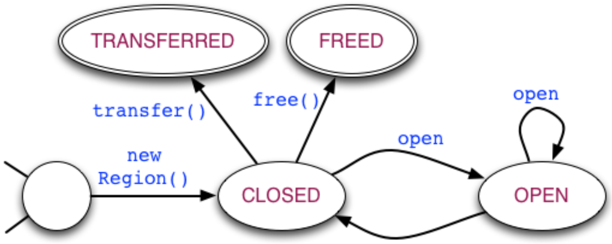

Figure 3: The lifetime of a dynamic (transferable) region in BROOM

not easy. BROOM imposes the following protocol on the use of transferable regions to help simplify this check.

A transferable region (that has not been freed or transferred) can be in one of two possible states, *open* or *closed*. A newly created region is in the closed state. A region must be opened, using the open construct (as explained previously), in order to read or update or allocate an object within that region. An open region cannot be freed or transferred. In particular, an open region is guaranteed to be live for the entire duration of the open construct. This allows the type system to infer a valid outlives relation between the opened region and any stack region that is nested within the open construct.

The protocol for transferable regions is presented as a finite state machine in Fig. 3. The safety of memory accesses in BROOM is now subject to the condition that every transferable region correctly follows the state transition discipline of Fig. 3. Under this condition, BROOM's region type system statically guarantees the safety of all memory accesses.

In BROOM, this enforcement is done at runtime by explicitly keeping track of the *current state* for `Region` objects, and checking the validity of every open, transfer, or free operation and throwing an exception if it is invalid. As explained previously, this is a reasonable trade-off in the context of BROOM, as regions are coarse-grained objects, which are manipulated infrequently, when compared to fine-grained objects that reside inside these regions. Therefore, runtime overhead of checking the region's state transition discipline is acceptable.

Cloning Note that in the example from Fig. 2 the object returned by the `selector` (on Line 13) should not contain any references to the input object, since the input region, where the object resides, will be freed at the end of the method. If there is a need for the output object to point to subobjects of the input object, such subobjects must be cloned (to copy them from the input region to the output region). Fortunately, BROOM's region type system (§ 3) is capable of capturing such nuances in the type of `selector` and the type checker will ensure correctness. Furthermore, the type can be automatically inferred by BROOM's region type inference (§ 4), which can perform the above reasoning on behalf of the programmer.

3. FEATHERWEIGHT BROOM

The purpose of BROOM's region type system is to enforce the key invariant required for memory safety, namely that an object o_1 in a region R_1 contains a reference to an object o_2 in R_2, only if R_2 is guaranteed to outlive R_1. Intuitively, the invariant can be enforced by (a). tracking *outlives* relationships between various regions in the program (b). tagging the C# type of every object in the program with its allocation region, and (c). ensuring that, when a reference is created, its target object (o_2) is allocated in a region that is known to be in outlives relationship with the object containing the reference. We now formally develop this intuition via FEATHERWEIGHT BROOM (FB), our explicitly typed core language (with region types) that incorporates the features introduced in the previous section. FEATHERWEIGHT BROOM builds on the Featherweight Generic Java (FGJ) [9] formalism, and reuses notations and various definitions from [9], such as the definition of type well-formedness for the core (region-free) language.

3.1 Syntax

Fig 4 describes the syntax of FB. We refer to the class types of FGJ as *core types*. The following definition of `Pair` class in FB illustrates some of the key elements of the formal language[5]:

```
class Pair<a ◁ Object, b ◁ Object>
          <ρ^a, ρ1, ρ2 | ρ1 ⪰ ρ^a ∧ ρ2 ⪰ ρ^a> ◁ Object<ρ^a> {
  a@ρ1 fst;
  b@ρ2 snd;
  Pair(a@ρ1 fst, b@ρ2 snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  a@ρ1 getFst() {
    return this.fst;
  }
}
```

A class in FB is parametric over zero or more type variables (as in FGJ) as well as one or more region variables ρ. We refer to the first region parameter, usually denoted ρ^a, as the *allocation region* of the class: it serves to identify the region where an instance of the class is allocated. An object in FB can contain fields referring to objects allocated in regions ($\overline{\rho}$) other than its own allocation region (ρ^a), provided that the former outlive the later (i.e., $\overline{\rho} \succeq \rho$). In such case, the definition of object's class needs to be parametric over allocation regions of its fields (i.e., their classes). Furthermore, the constraint that such regions must outlive the allocation region of the class needs to be made explicit in the definition, as the `Pair` class does in the above definition. We say that the `Pair` class exhibits *constrained region polymorphism*.

To construct objects of the `Pair` class, its type and region parameters need to be instantiated with core types

[5]The symbol $\lhd$ should be read *extends*, and the symbol $\succeq$ stands for *outlives*

$$\pi \in \mathtt{Static\ region\ ids} \qquad \rho \in \mathtt{Region\ variables} \qquad a, b \in \mathtt{Type\ variables} \qquad m \in \mathtt{Method\ names} \qquad x, y, f \in \mathtt{Variables\ and\ fields}$$

$$\begin{array}{lcllll}
cn & \in & \mathtt{Class\ names} & ::= & \mathtt{Object} \mid \mathtt{Region} \mid A \mid B \\
K & \in & \mathtt{FGJ\ class\ types} & ::= & cn\langle\overline{T}\rangle \\
T & \in & \mathtt{FGJ\ types} & ::= & a \mid K \mid \mathtt{unit} \mid \overline{T} \to T \\
N & \in & \mathtt{Region-annotated\ class\ types} & ::= & cn\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle \\
\tau & \in & \mathtt{types} & ::= & T@\pi \mid N \mid \mathtt{unit} \mid \langle\rho^a\overline{\rho} \mid \phi\rangle\overline{\tau} \xrightarrow{\pi} \tau \\
C & \in & \mathtt{Class\ definitions} & ::= & \mathtt{class}\ cn\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a\overline{\rho} \mid \phi\rangle \triangleleft N\{\overline{\tau}\ \overline{f};\ k\ \overline{d}\} \\
k & \in & \mathtt{Constructors} & ::= & cn(\overline{\tau}\ \overline{x})\{\ \mathtt{super}\ (\overline{x});\ \mathtt{this}.\overline{f} = \overline{x};\} \\
d & \in & \mathtt{Methods} & ::= & \tau\ m\langle\rho^a\overline{\rho} \mid \phi\rangle(\overline{\tau}\ \overline{x})\{\mathtt{return}\ e;\} \\
\phi, \Phi & \in & \mathtt{Region\ constraints} & ::= & \mathit{true} \mid \rho \succeq \rho \mid \rho = \rho \mid \phi \wedge \phi \\
e & \in & \mathtt{Expressions} & ::= & () \mid x \mid e.f \mid e.m\langle\pi^a\overline{\pi}\rangle(\overline{e}) \mid \mathtt{new}\ N(\overline{e}) \mid \lambda@\pi^a\langle\rho^a\overline{\rho} \mid \phi\rangle(\overline{x} : \overline{\tau}).e \mid e\langle\pi^a\overline{\pi}\rangle(\overline{e}) \\
 & & & & \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{letregion}\ \rho\ \mathtt{in}\ e \mid \mathtt{open}\ x\ \mathtt{as}\ y@\pi\ \mathtt{in}\ e
\end{array}$$

Figure 4: FEATHERWEIGHT BROOM: Syntax

$$\begin{array}{lcl}
\mathsf{allocRgn}(A\langle\pi^a\overline{\pi}\rangle\langle\overline{T}\rangle) & = & \pi^a \\
\mathsf{allocRgn}(\langle\rho^a\overline{\rho} \mid \phi\rangle\overline{\tau^1} \xrightarrow{\pi^a} \tau^2) & = & \pi^a \\
\mathsf{shape}(A\langle\rho^a\overline{\rho}\rangle\langle\overline{T}\rangle) & = & A\langle\overline{T}\rangle
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{bound}_\Theta(a@\pi) & = & \Theta(a)@\pi \\
\mathsf{bound}_\Theta(N) & = & N \\
\mathsf{fields}(\mathtt{Object}\langle\pi\rangle) & = & \bullet
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{ctype}(\mathtt{Object}\langle\pi\rangle) & = & \bullet \\
\mathsf{ctype}(B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle) & = & \mathsf{fields}(B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle)
\end{array}$$

$$\frac{\begin{array}{c} CT(B) = \mathtt{class}\ B\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a, \overline{\rho} \mid \phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ ...\} \\ \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a, \overline{T}/\overline{a}] \qquad \mathsf{fields}(\mathcal{S}(N)) = \overline{g} : \overline{\tau^g} \end{array}}{\mathsf{fields}(B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle) = \overline{g} : \overline{\tau^g},\ \overline{f} : \mathcal{S}(\overline{\tau^f})}
\qquad
\frac{\begin{array}{c} CT(B) = \mathtt{class}\ B\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a, \overline{\rho} \mid \phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ k\ \overline{d}\} \\ m \notin \overline{d} \qquad \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a, \overline{T}/\overline{a}] \end{array}}{\mathsf{mtype}(m, B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle) = \mathsf{mtype}(m, \mathcal{S}(N))}$$

$$\frac{\begin{array}{c} CT(B) = \mathtt{class}\ B\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a, \overline{\rho} \mid \phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ k\ \overline{d}\} \\ \tau^2\ m\langle\rho^a_m, \overline{\rho_m} \mid \phi_m\rangle(\overline{\tau^1}\ \overline{x})\{...\} \in \overline{d} \qquad \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a, \overline{T}/\overline{a}] \end{array}}{\mathsf{mtype}(m, B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle) = \mathcal{S}(\langle\rho^a_m, \overline{\rho_m} \mid \phi_m\rangle\overline{\tau^1} \to \tau^2)}
\qquad
\frac{\begin{array}{c} \mathsf{mtype}(m, N) = \langle\rho^a_1\overline{\rho_1}, \mid \phi_1\rangle\overline{\tau^{11}} \to \tau^{12} \\ \mathtt{implies}\quad \mathcal{A}.\Phi \vdash \phi_2 \Leftrightarrow \mathcal{S}(\phi_1) \quad \mathtt{and} \quad \overline{\tau^{21}} = \mathcal{S}(\overline{\tau^{11}}) \\ \mathtt{and}\quad \mathcal{A} \vdash \tau^{22} <: \mathcal{S}(\tau^{12}) \quad \mathcal{S} = [\overline{\rho_2}/\overline{\rho_1}][\rho^a_2/\rho^a_1] \end{array}}{\mathsf{override}(\mathcal{A}, N, \langle\rho^a_2\overline{\rho_2}, \mid \phi_1\rangle\overline{\tau^{21}} \to \tau^{22})}$$

Figure 5: FEATHERWEIGHT BROOM: Auxiliary Definitions

(T) and concrete region identifiers[6] (π), respectively. For example:

```
letregion π0 in
  let snd = new Object<π0>() in
  letregion π1 in
    let fst = new Object<π1>() in
    let p = new Pair<Object,Object><π1,π0,π1>
                (fst,snd);
```

In the above code, the instantiation of ρ^a and ρ_1 with π_0, and ρ_2 with π_1 is allowed because (a) π_0 and π_1 are live during the instantiation, and (b). $\pi_0 \succeq \pi_1$ and $\pi_1 \succeq \pi_1$ (since outlives is reflexive). Observe that the region type of `p` conveys the fact that (a). it is allocated in region π_1, and (b). it holds references to objects allocated in region π_0 and π_1. In contrast, if we choose to allocate the `snd` object also in π_1, then `p` would be contained in π_1, and its region type would be `Pair<Object,Object>`$\langle\pi_1, \pi_1, \pi_1\rangle$, which we abbreviate as `Pair<Object,Object>`$@\pi_1$. In general, we treat $B\langle\overline{T}\rangle@\pi$ as being equivalent to $B\langle\overline{T}\rangle\langle\overline{\pi}\rangle$. Region annotation on type a, where a is a type variable, assumes the form $a@\pi$. If a is instantiated with `Pair<Object,Object>`, the result is the type of a `Pair` object contained in π. The type `unit` is unboxed in FB, hence it has no region annotations.

Like classes, methods can also exhibit constrained region polymorphism. A method definition in FB is necessarily polymorphic over its allocation context (§ 2.1), and optionally polymorphic with respect to the regions containing its arguments. Region parameters, like those on classes, are qualified with constraints (ϕ). Note that if a method is not intended to be polymorphic with respect to its allocation context (for example, if its allocation context needs to be same as the allocation region of its *this* argument), then the required monomorphism can be captured as an equality constraint in ϕ.

FB extends FGJ's expression language with a lambda expression and an application expression ($e\langle\pi^a\overline{\pi}\rangle(\overline{e})$) to define

[6] Region variables (ρ) and region identifiers (π) belong to the same syntactic class. Region identifiers are to region variables, as types (T) are to type variables (a)

Type, and Type Constraint Well-formedness $\boxed{\mathcal{A} \vdash \tau\ \texttt{ok}, \quad \Delta \vdash \phi\ \texttt{ok}}$

$$\frac{\pi \in \mathcal{A}.\Delta}{\mathcal{A} \vdash \texttt{Object}\langle\pi\rangle\ \texttt{ok}} \qquad \frac{\begin{array}{c}\{\rho^a,\overline{\rho}\} \notin \mathcal{A}.\Delta \quad \mathcal{A}' = (\mathcal{A}.\Sigma, \mathcal{A}.\Delta \cup \{\rho^a,\overline{\rho}\}, \mathcal{A}.\Theta, \mathcal{A}.\Phi \wedge \phi)\\ \pi \in \mathcal{A}.\Delta \quad \mathcal{A}'.\Delta \vdash \phi\ \texttt{ok} \quad \mathcal{A}' \vdash \overline{\tau^1}\ \texttt{ok} \quad \mathcal{A}' \vdash \tau^2\ \texttt{ok}\end{array}}{\mathcal{A} \vdash \langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{\pi} \tau^2\ \texttt{ok}} \qquad \frac{\mathcal{A}.\Theta \Vdash T\ \texttt{ok}}{\texttt{Region}\,\langle T\rangle\langle\pi\rangle} \qquad \frac{\rho_0,\rho_1 \in \Delta}{\Delta \vdash \rho_0 \succeq \rho_1\ \texttt{ok}}$$

$$\frac{\begin{array}{c}CT(B) = \texttt{class}\ B\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a,\overline{\rho}\,|\,\phi\rangle \triangleleft N\{...\} \quad \pi^a,\overline{\pi} \in \mathcal{A}.\Delta\\ \mathcal{A}.\Theta \Vdash B\langle\overline{T}\rangle\ \texttt{ok} \quad \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a, \overline{T}/\overline{a}] \quad \mathcal{A}.\Phi \vdash \mathcal{S}(\phi)\end{array}}{\mathcal{A} \vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{T}\rangle\ \texttt{ok}} \qquad \frac{\begin{array}{c}\mathcal{A}.\Theta \Vdash T\ \texttt{ok} \quad \pi^a \in \mathcal{A}.\Delta\\ \mathcal{A}.\Theta \Vdash T <: \texttt{Object}\end{array}}{\mathcal{A} \vdash T@\pi^a\ \texttt{ok}} \qquad \frac{\begin{array}{c}\Delta \vdash \phi_0\ \texttt{ok}\\ \Delta \vdash \phi_1\ \texttt{ok}\end{array}}{\Delta \vdash \phi_0 \wedge \phi_1\ \texttt{ok}}$$

Expression Typing $\boxed{\mathcal{A}, \pi^a, \Gamma \vdash e : \tau}$

$$\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash ()\ :\ \texttt{unit}\\ \mathcal{A},\pi^a,\Gamma \vdash x : \Gamma(\tau)\end{array} \qquad \frac{\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash e : \tau'\\ \overline{f} : \overline{\tau}\ =\ \mathsf{fields}(\mathsf{bound}_{\mathcal{A}.\Theta}(\tau'))\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash e.f_i : \tau_i} \qquad \frac{\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash \overline{e} : \overline{\tau}\\ \mathcal{A} \vdash N\ \texttt{ok} \quad \mathsf{fields}(N) = \overline{f} : \overline{\tau}\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{new}\ N(\overline{e}) : N} \qquad \frac{\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash e_1 : \tau_1\\ \mathcal{A},\pi^a,\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\begin{array}{c}\mathcal{A}.\Theta \Vdash T\ \texttt{ok} \quad K = \texttt{Region}\,\langle T\rangle\\ (\emptyset, \{\rho^a\}, \mathcal{A}.\Theta, \mathit{true}), \rho^a, \cdot \vdash e : T@\rho^a\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{new}\ K\langle\pi_\top\rangle(\lambda@\pi^a\langle\rho^a\rangle().e) : K\langle\pi_\top\rangle} \qquad \frac{\begin{array}{c}\pi \in \mathcal{A}.\Sigma \quad \mathcal{A}' = (\{\}, \{\pi\}, \mathcal{A}.\Theta, \mathit{true})\\ \pi \notin \mathcal{A}.\Delta \cup \{\pi_\top\} \quad \mathcal{A}' \vdash T@\pi\ \texttt{ok} \quad \mathcal{A}',\pi,\Gamma \vdash e : T@\pi\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{new Region}\,\langle T\rangle\langle\pi\rangle(e) : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle}$$

$$\frac{\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash e_0 : \tau \quad \overline{\pi} \in \mathcal{A}.\Delta\\ \mathsf{mtype}(m, \mathsf{bound}_{\mathcal{A}.\Theta}(\tau)) = \langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \to \tau^2 \quad \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a]\\ \mathcal{A} \vdash \langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \to \tau^2\ \texttt{ok} \quad \mathcal{A},\pi^a,\Gamma \vdash \overline{e} : \mathcal{S}(\overline{\tau^1}) \quad \mathcal{A}.\Phi \vdash \mathcal{S}(\phi)\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash e_0.m\langle\pi^a\overline{\pi}\rangle(\overline{e}) : \mathcal{S}(\tau^2)} \qquad \frac{\begin{array}{c}\mathcal{A} = (\Sigma,\Delta,\Theta,\Phi) \quad \pi \notin \Delta \quad \phi = \Delta \succeq \pi \quad \mathcal{A} \vdash \tau\ \texttt{ok}\\ \mathcal{A}' = (\Sigma, \Delta \cup \{\pi\}, \Theta, \Phi \wedge \phi) \quad \mathcal{A}',\pi,\Gamma \vdash e : \tau\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{letregion}\ \pi\ \texttt{in}\ e : \tau}$$

$$\frac{\begin{array}{c}\pi \in \mathcal{A}.\Delta \quad \rho^a,\overline{\rho} \notin \mathcal{A}.\Delta\\ \mathcal{A}' = (\mathcal{A}.\Sigma, \mathcal{A}.\Delta \cup \{\rho^a,\overline{\rho}\}, \mathcal{A}.\Theta, \mathcal{A}.\Phi \wedge \phi) \quad \mathcal{A}'.\Delta \vdash \phi\ \texttt{ok}\\ \mathcal{A}' \vdash \overline{\tau^1}\ \texttt{ok} \quad \mathcal{A}' \vdash \tau^2\ \texttt{ok} \quad \mathcal{A}',\rho^a,\Gamma[\overline{x} \mapsto \overline{\tau^1}] \vdash e : \tau^2\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \lambda@\pi\langle\rho^a\overline{\rho}\,|\,\phi\rangle(\overline{x} : \overline{\tau^1}).e : \langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{\pi} \tau^2} \qquad \frac{\begin{array}{c}\mathcal{A},\pi^a,\Gamma \vdash e_a : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle\\ \mathcal{A} = (\Sigma,\Delta,\Theta,\Phi) \quad \pi \notin \Delta \quad \mathcal{A}' = (\Sigma, \Delta \cup \{\pi\}, \Theta, \Phi)\\ \Gamma' = \Gamma[y \mapsto T@\pi] \quad \mathcal{A}',\pi,\Gamma' \vdash e_b : \tau \quad \mathcal{A} \vdash \tau\ \texttt{ok}\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash \texttt{open}\ e_a\ \texttt{as}\ y@\pi\ \texttt{in}\ e_b : \tau}$$

$$\frac{\begin{array}{c}\overline{\pi} \in \mathcal{A}.\Delta \quad \mathcal{A},\pi^a,\Gamma \vdash e : \langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{\pi} \tau^2\\ \mathcal{S} = [\overline{\pi}/\overline{\rho}][\pi^a/\rho^a] \quad \mathcal{A}.\Phi \vdash \mathcal{S}(\phi) \quad \mathcal{A},\pi^a,\Gamma \vdash \overline{e} : \mathcal{S}(\overline{\tau^1})\end{array}}{\mathcal{A},\pi^a,\Gamma \vdash e\langle\pi^a\overline{\pi}\rangle(\overline{e}) : \mathcal{S}(\tau^2)} \qquad \frac{\mathcal{A},\pi^a,\Gamma \vdash e : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle}{\mathcal{A},\pi^a,\Gamma \vdash e.\mathsf{transfer}\langle\pi^a\rangle() : \texttt{unit}}$$

Figure 6: FEATHERWEIGHT BROOM: Static Semantics

and apply functions[7]. Functions, like methods, exhibit constrained region polymorphism, as evident in their arrow region type ($\langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau} \xrightarrow{\pi} \tau$). Note that any of the τ's in the arrow type can themselves be region-parametric arrow types. In this respect, our region type system is comparable to System F's type system, which admits higher-rank parametric polymorphism. Note that a lambda expression creates a closure, which can escape the context in which it is created. It is therefore important to keep track of the region in which a closure is allocated in order to avoid unsafe dereferences. The π annotation above the arrow in the arrow type denotes the allocation region of the corresponding closure. Note that it is important to distinguish between the allocation context argument (ρ^a) of a function and the allocation region (π) of its closure. In BROOM, the later corresponds to the region where a `Func` object is allocated, while the former corresponds to the region where it is applied. For instance, in the following example:

```
letregion π {
  let f = λ⟨ρ^a⟩().new Object⟨ρ^a⟩()
  in f
}
```

The type of `f` is $\langle\rho^a\rangle\,\texttt{unit} \xrightarrow{\pi} \texttt{Object}\,\langle\rho^a\rangle$, coveying that (a). `f`'s closure is allocated in π, and (b). when executed under an allocation context ρ^a, the closure returns an object allocated in ρ^a.

[7] We distinguish between functions and methods. The former result from lambda expressions, whereas the latter come from class definitions

Method Well-formedness $\boxed{d \texttt{ ok in } B}$

$$\frac{\begin{array}{c}\Delta = \{\rho^a, \overline{\rho}, \rho^a_m, \overline{\rho_m}\} \quad \Theta = [\overline{a} \mapsto \overline{K}] \quad \Phi = \phi \wedge \phi_m \quad \mathcal{A} = (\Sigma, \Delta, \Theta, \Phi) \quad \Delta \vdash \phi_m \texttt{ ok} \\ \mathcal{A} \vdash \overline{\tau^1}, \tau^2 \texttt{ ok} \quad \texttt{class } B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \rho^a, \overline{\rho} \,|\, \phi\rangle \triangleleft N\{...\} \quad \Gamma = \cdot[\overline{x} \mapsto \overline{\tau^1}][\texttt{this} \mapsto B\langle\overline{a}\rangle\langle\rho^a\overline{\rho}\rangle] \\ \mathsf{override}(m, N, \langle \rho^a_m \overline{\rho_m} \,|\, \phi_m\rangle \overline{\tau^1} \to \tau^2) \quad \mathcal{A}, \rho^a_m, \Gamma \vdash e : \tau \quad \mathcal{A} \vdash \tau <: \tau^2\end{array}}{\tau^2\ m\langle \rho^a_m \overline{\rho_m} \,|\, \phi_m\rangle(\overline{\tau^1\ x})\{\ \texttt{return } e;\} \texttt{ ok in } B}$$

Class Well-formedness $\boxed{B \texttt{ ok}}$

$$\frac{\begin{array}{c}\Delta = \{\rho^a, \overline{\rho}\} \quad \Theta = [\overline{a} \mapsto \overline{K}] \quad \Phi = \phi \quad \mathcal{A} = (\Sigma, \Delta, \Theta, \Phi) \quad \Delta \vdash \phi \texttt{ ok} \\ \Theta \Vdash \overline{K} \texttt{ ok} \quad \mathcal{A} \vdash N, \overline{\tau^f} \texttt{ ok} \quad \mathsf{shape}(N) \neq \texttt{Region}\langle T\rangle \quad \Phi \vdash \mathit{allocRgn}(\overline{\tau^f}) \succeq \rho^a \\ \mathsf{ctype}(N) = \overline{\tau^N} \quad k = B(\overline{\tau^f\ x}, \overline{\tau^N\ y})\{\ \texttt{super}\,(\overline{x});\ \texttt{this}.\overline{f} = \overline{y};\} \quad \overline{d} \texttt{ ok in } B\end{array}}{\texttt{class } B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \rho^a, \overline{\rho} \,|\, \phi\rangle \triangleleft N\{\overline{\tau^f\ f};\ k\ \overline{d}\} \texttt{ ok}}$$

Figure 7: FEATHERWEIGHT BROOM: Method and Class Well-formedness

3.2 Types and Well-formedness

Well-formedness and typing rules of FEATHERWEIGHT BROOM establish the conditions under which a region type is considered well-formed, and an expression is considered to have a certain region type, respectively. Fig. 6 contains the entire set of the rules. The rules refer to a context ($\mathcal{A}$), which is a tuple of:

- A set ($\Delta \in 2^{\pi}$) of static identifiers of regions that are estimated to be live,
- A finite map ($\Theta \in a \mapsto K$) of type variables to their bounds[8], and
- A constraint formula (Φ) that captures the outlives constraints on regions in Δ.

$\mathcal{A}$ also contains Σ, which is primarily an artifact to facilitate the type safety proof, and can be ignored while type checking user-written programs in FB. The context for the expression typing judgment also includes:

- A type environment ($\Gamma \in x \mapsto \tau$) that contains the type bindings for variables in scope, and
- The static identifier (π^a) of the allocation context for the expression is being typechecked.

Like the judgments in FGJ [9], all the judgments defined by the rules in Fig. 6 are implicitly parameterized on a class table ($CT \in cn \mapsto D$) that maps class names to their definitions in FB.
The well-formedness judgment on region types ($\mathcal{A} \vdash \tau \texttt{ ok}$) makes use of the well-formedness and subtyping judgments on core types [9]. The class table ($[\![CT]\!]$) for such judgments is derived from FB's class table (CT) by erasing all region annotations on types, and region arguments in expressions ($[\![\cdot]\!]$ denotes the region erasure operation). The well-formedness rule for class types ($B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle$) is responsible for enforcing the safety property that prevents objects from containing unsafe references. It does so by insisting that regions $\pi^a\overline{\pi}$ satisfy the constraints (ϕ) imposed by the class on its region parameters. The later is enforced by checking the validity of ϕ, with actual region arguments substituted[10] for formal region parameters, under the conditions ($\mathcal{A}.\Phi$) guaranteed by the context. The semantics of this sequent is straightforward, and follows directly from the properties of outlives and equality relations. For any well-formed core type T, $T@\pi$ is a well-formed region type if π is a valid region. The type $\texttt{Region}\langle T\rangle\langle\pi\rangle$ is well-formed only if $\pi = \pi_\top$, where $\pi_\top$ is a special immortal region that outlives every other live region. This arrangement allows `Region` handlers to be aliased and referenced freely from objects in various regions, regardless of their lifetimes. On the flip side, this also opens up the possibility of references between transferable regions, which become unsafe in context of the recipient's address space. Fortunately, such references are explicitly prohibited by the type rule of `Region` objects, as described below.

The type rules distinguish between the `new` expressions that create objects of the `Region` class, and `new` expressions that create objects of other classes. The rule for the later relies on an auxiliary definition called fields that returns the sequence of type bindings for fields (instance variables) of a given class type. Like in FGJ, the names and types of a constructor's arguments in FB are same as the names and types of its class's fields, and the type rule relies on this fact to typecheck constructor applications. Note that this rule does not apply to `new` expressions involving `Region` class, as we do not define fields for `Region`.

The type rule for `new Region` expressions expects the `Region` class's constructor to be called with a nullary function that returns a value in its allocation context. It enforces this by typechecking the body (e) of the function un-

[8] A bound of a type variable (a) in FGJ [9] is the class (K) that type variable was declared to extend.

[9] We use a double-piped turnstile ($\Vdash$) for judgments in FGJ [9], and a simple turnstile ($\vdash$) for those in FB.

[10] The notation $[a/b](e)$ stands for "a is substituted for b in e"

$\boxed{\Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma')}$

$$\frac{\mathsf{allocRgn}(N) \in \Delta \quad \mathsf{fields}(N) = \overline{f} : \overline{\tau}}{\Delta \vdash ((\texttt{new}\ N(\overline{v})).f_i, \Sigma) \longrightarrow (v_i, \Sigma)} \qquad \frac{\mathsf{fresh}(\pi') \quad \Delta \cup \{\pi'\} \vdash ([\pi'/\pi]e, \Sigma) \longrightarrow (e', \Sigma')}{\Delta \vdash (\texttt{letregion}\ \pi\ \texttt{in}\ e, \Sigma) \longrightarrow (\texttt{letregion}\ \pi'\ \texttt{in}\ e', \Sigma')}$$

$$\frac{}{\Delta \vdash (\texttt{letregion}\ \pi\ \texttt{in}\ v, \Sigma) \longrightarrow (v, \Sigma)} \qquad \frac{K = \texttt{Region}\langle T\rangle \quad \pi^a \in \Delta \quad \mathsf{fresh}(\pi) \quad \Sigma' = \Sigma[\rho \mapsto \texttt{C}]}{\Delta \vdash (\texttt{new}\ K\langle \pi_\top\rangle(\lambda@\pi^a\langle\rho^a\rangle().e), \Sigma) \longrightarrow (\texttt{new}\ K\langle\pi\rangle([\pi/\rho^a]e), \Sigma')}$$

$$\frac{\begin{array}{c}\{\pi\} \vdash (e, \Sigma) \longrightarrow (e', \Sigma') \\ K = \texttt{Region}\langle T\rangle \quad \pi \in dom(\Sigma)\end{array}}{\Delta \vdash (\texttt{new}\ K\langle\pi\rangle(e), \Sigma) \longrightarrow (\texttt{new}\ K\langle\pi\rangle(e'), \Sigma')} \qquad \frac{v_a = \texttt{new Region}\langle T\rangle\langle\pi\rangle(v_r) \quad \Sigma(\pi) \neq \texttt{X}}{\Delta \vdash (\texttt{open}\ v_a\ \texttt{as}\ x@\pi_0\ \texttt{in}\ v_b, \Sigma) \longrightarrow (v_b, \Sigma)} \qquad \frac{\Delta \vdash (e, \Sigma) \longrightarrow \bot}{\Delta \vdash (E[e], \Sigma) \longrightarrow \bot}$$

$$\frac{\begin{array}{c}v_a = \texttt{new Region}\langle T\rangle\langle\pi\rangle(v_r) \quad \Sigma(\pi) \neq \texttt{X} \quad \mathsf{fresh}(\pi_1) \\ \{\pi_1\} \vdash ([\pi_1/\{\pi, \pi_0\}][v_r/x]e_b, \Sigma[\pi \mapsto \texttt{O}]) \longrightarrow (e'_b, \Sigma') \quad \Sigma'' = \Sigma'[\pi \mapsto \Sigma(\pi)]\end{array}}{\Delta \vdash (\texttt{open}\ v_a\ \texttt{as}\ x@\pi_0\ \texttt{in}\ e_b, \Sigma) \longrightarrow (\texttt{open}\ v_a\ \texttt{as}\ x@\pi_1\ \texttt{in}\ e'_b, \Sigma'')} \qquad \frac{v_a = \texttt{new Region}\langle T\rangle\langle\pi\rangle(v_r) \quad \Sigma(\pi) = \texttt{X}}{\Delta \vdash (\texttt{open}\ v_a\ \texttt{as}\ x@\pi_0\ \texttt{in}\ e_b, \Sigma) \longrightarrow \bot}$$

$$\frac{\mathsf{allocRgn}(N), \pi^a \in \Delta \quad \mathsf{mbody}(m\langle\pi^a\overline{\pi}\rangle, N) = \overline{x}.e}{\Delta \vdash ((\texttt{new}\ N(\overline{v})).m\langle\pi^a\overline{\pi}\rangle(\overline{v'}), \Sigma) \longrightarrow ([\overline{v'}/\overline{x}][\texttt{new}\ N(\overline{v})/\texttt{this}]\,e, \Sigma)} \qquad \frac{N = \texttt{Region}\langle T\rangle\langle\pi\rangle \quad \Sigma(\pi) \neq \texttt{O} \quad \Sigma' = \Sigma[\pi \mapsto \texttt{X}]}{\Delta \vdash ((\texttt{new}\ N(v)).\mathsf{transfer}\langle\pi^a\rangle(), \Sigma) \longrightarrow ((), \Sigma')}$$

$$\frac{N = \texttt{Region}\langle T\rangle\langle\pi\rangle \quad \Sigma(\pi) = \texttt{O}}{\Delta \vdash ((\texttt{new}\ N(v)).\mathsf{transfer}\langle\pi^a\rangle(), \Sigma) \longrightarrow \bot} \qquad \frac{v_a = \lambda@\pi_a\langle\rho^a\overline{\rho}\rangle(\overline{\tau}\ \overline{x}).e \quad \pi_a, \pi^a \in \Delta}{\Delta \vdash (v_a\langle\pi^a\overline{\pi}\rangle(\overline{v}), \Sigma) \longrightarrow ([\overline{v}/\overline{x}][\overline{\pi}/\overline{\rho}][\pi^a/\rho^a]\,e, \Sigma)} \qquad \frac{\Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma')}{\Delta \vdash (E[e], \Sigma) \longrightarrow (E[e'], \Sigma')}$$

$$\frac{\begin{array}{c}\mathsf{fresh}(\pi') \\ \Delta \cup \{\pi'\} \vdash ([\pi'/\pi]e, \Sigma) \longrightarrow \bot\end{array}}{\Delta \vdash (\texttt{letregion}\ \pi\ \texttt{in}\ e, \Sigma) \longrightarrow \bot} \quad \frac{\begin{array}{c}K = \texttt{Region}\langle T\rangle \quad \pi \in dom(\Sigma) \\ \{\pi\} \vdash (e, \Sigma) \longrightarrow \bot\end{array}}{\Delta \vdash (\texttt{new}\ K\langle\pi\rangle(e), \Sigma) \longrightarrow \bot} \qquad \frac{\begin{array}{c}v_a = \texttt{new Region}\langle T\rangle\langle\pi\rangle(v_r) \quad \Sigma(\pi) \neq \texttt{X} \quad \mathsf{fresh}(\pi_1) \\ \{\pi_1\} \vdash ([\pi_1/\{\pi, \pi_0\}][v_r/x]e_b, \Sigma[\pi \mapsto \texttt{O}]) \longrightarrow \bot\end{array}}{\Delta \vdash (\texttt{open}\ v_a\ \texttt{as}\ x@\pi_0\ \texttt{in}\ e_b, \Sigma) \longrightarrow \bot}$$

Evaluation Context $\boxed{E}$

$$\begin{array}{rcl} E & ::= & \bullet \mid (\bullet).f \mid \bullet.m\langle\pi^a\overline{\pi}\rangle(\overline{e}) \mid v.m\langle\pi^a\overline{\pi}\rangle(...,\bullet,...) \mid \texttt{new}\ N(...,\bullet,...) \mid \texttt{new Region}\langle T\rangle\langle\pi_\top\rangle(\bullet) \mid \bullet\langle\pi^a\overline{\pi}\rangle(\overline{e}) \\ & & \mid v\langle\pi^a\overline{\pi}\rangle(...,\bullet,...) \mid \texttt{open}\ \bullet\ \texttt{as}\ y@\pi\ \texttt{in}\ e \end{array}$$

Figure 8: FEATHERWEIGHT BROOM: Operational Semantics

der an empty context containing nothing but the allocation context of the function. This step ensures that the value returned by the function stores no references to objects allocated elsewhere, including the top region ($\pi_\top$), thus preventing cross-region references originating from transferable regions[11].

The type rule for `letregion` expression requires that the static identifier introduced by the expression be unique under the current context (i.e., $\pi \notin \Delta$). This condition is needed in order to prevent the new region from incorrectly assuming any existing outlives relationships on an eponymous region. Provided this is satisfied, the expression (e) under `letregion` is then typechecked assuming that the new region is live ($\pi \in \Delta$) and that it is outlived by all existing live regions ($\Delta \succeq \pi$). The result of a `letregion` expression must have a type that is well-formed under a context not containing the new region. This ensures that the value obtained by evaluating a `letregion` expression contains no references to the temporary objects inside the region[12].

The rule for `open` expression, unlike the rule for `letregion`, does not introduce any outlives relationship between the newly opened region and any pre-existing region while checking the type of the expression (e) under `open`. This prevents new objects allocated inside the transferable region from storing references to those outside. Environment (Γ) is extended with binding for the type of root object while typechecking e.

The type rule for lambda expression typechecks the lambda-bound expression (e) under an extended type environment containing bindings for function's arguments, assuming that region parameters are live, and that declared constraints over region parameters hold. The constraints (ϕ) are required to be well-formed under Δ', which means that

[11] The body of the function (e) might, however, create new (transferable) regions while execution, but that is fine as long as such regions, and objects allocated in them, don't find their way into the result of evaluating e

[12] The appendix contains a formal proof of this claim

ϕ must only refer to the region variables in the set Δ'. Note that the closure is always allocated in the current allocation context (π^a). This prevents the closure from escaping the context in which it is created, thus trivially ensuring the safety of any dereferences inside the closure.

3.3 Operational Semantics and Type Safety

Fig. 8 defines a small-step operational semantics for FEATHERWEIGHT BROOM via a five-place reduction relation:

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

The reduction judgment should be read as following: given a set (Δ) of regions that are currently live, and a map (Σ) from unique identifiers of transferable regions to their current states (Fig. 3), the expression e reduces to e', while updating Σ to Σ'. The semantics gets "stuck" if e attempts to access an object whose allocation region is not present in Δ, or if e tries to `open` a transferable region, whose identifier is not mapped to a state by Σ. On the other hand, if e attempts to commit an operation on a `Region` object that is not sanctioned by the transition discipline in Fig. 3, then it raises an exception value ($\perp$).

To help state the type safety theorem, we define the syntactic class of values:

$$v \;\in\; \texttt{values} \;::=\; \texttt{new}\ N(\overline{v}) \mid \lambda @\pi^a \langle \rho^a \overline{\rho} \mid \phi \rangle (\overline{\tau}\ \overline{x}).e \quad \texttt{new Region}\,\langle T\rangle\langle\pi\rangle(v)$$

Note that for $\texttt{new Region}\,\langle T\rangle\langle\pi\rangle(v)$ to be considered a value, $\pi \neq \pi_\top$. The semantics reduces a $\texttt{new Region}\,\langle T\rangle\langle\pi_\top\rangle(e)$ expression in the user program to a runtime $\texttt{new Region}\,\langle T\rangle\langle\pi\rangle(v)$ value that is tagged with a unique identifier (π) for this region. A binding is also added to Σ, mapping π to the "closed" state. Fig. 6 defines a type rule for such values, allowing them to be typed. Type safety theorem is now stated thus[13]:

THEOREM 3.1. (**Type Safety**) *$\forall e, \tau, \Delta, \Sigma, \pi$, such that $\pi \in \Delta$ and $\Delta \vdash \Phi$ ok, if $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$, then either e is a value, or e raises an exception ($\Delta \vdash (e, \Sigma) \longrightarrow \perp$), or there exists an e' and a Σ' such that $\Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma')$ and $(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$.*

Furthermore, we prove the following theorem about FB, which, in conjunction with the type safety theorem, implies the safety of region transfers across address spaces:

THEOREM 3.2. (**Transfer Safety**) *$\forall\, v, \Delta, \Delta', \Sigma, \Sigma', \Phi, \Phi', \pi, \pi'$, such that $\pi \in \Delta$, $\pi' \in \Delta'$, and $\pi_i \notin dom(\Sigma') \cup \Delta \cup \{\pi_\top\}$, if $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(v) : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle$, then $(\Sigma'[\pi_i \mapsto \texttt{C}], \Delta', \cdot, \Phi') \vdash \texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(v) : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle$*

The above theorem states that if a $\texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(v)$ value is well-typed under one context, then it is also well-typed under every other context, whose Σ maps π_i to closed (`C`) state. Thus, a recipient of a transferable region only needs to add a binding for the region to its Σ in order to preserve its type safety. Notably, this result could not have been established if it were possible for a transferable region to contain references to objects outside the region.

[13] Theorems 3.1 and A.7 are restated and proved in the appendix

4. Type Inference

BROOM's region type system imposes a heavy annotation burden, and the cost of manually annotating C# standard libraries with region types is prohibitive. We now present our region type inference algorithm that completely eliminates the need to write region type annotations, except on some higher-order functions. Formally, the type inference algorithm is an elaboration function from programs in $[\![FB]\!]$ (i.e., FB without region types, but with `letregion` and `open` expressions, similar to the language introduced in § 2) to programs in FB. The elaboration proceeds in four steps. In the first step we make use of the observation that region types are *refinements* of FGJ types with region annotations and constraints over such region annotations, and compute polymorphic region type templates for methods and classes from their FGJ types. The templates contain region variables (ρ) to denote unknown region annotations, and predicate variables (φ) to denote unknown constraints over such region annotations. Free region variables are generalized in types (hence, polymorphic). Second, we make use of the computed region type templates to elaborate expressions by introducing region variables to denote unknown region arguments in `new` expressions, method calls and function applications. While elaborating expressions, we also build a system of constraints that capture well-formedness requirements and subtyping relationships between type templates that must hold (as per the static semantics in Fig. **??**) for the elaboration to be valid. Third, we lift expression elaboration and constraint generation to methods and classes. Finally, we solve the constraints by making use of our fixpoint constraint solving algorithm CSOLVE*, which reduces the constraint solving problem to an abduction problem in a Herbrand constraint system, and then relies on CSOLVE, our abduction solver for that domain.

4.1 Region Type Templates

Region type templates are FGJ types extended with fresh region variables (ρ) and predicate variables (φ) to denote unknown region annotations and region constraints, respectively. For instance, if a variable x has type `Object` in FGJ, its region type template is of form $\texttt{Object}\,\langle\rho_0\rangle$, where ρ_0 is a fresh region variable. Likewise, given the region-annotated definition of `Pair` class from §3.1 a region type template for a method with FGJ type $\texttt{Pair}\,\langle \texttt{A}, \texttt{B}\rangle \rightarrow \texttt{A}$ is

[14,15] $\langle \rho_0^a, \rho_1, \rho_2, \rho_3 \,|\, \varphi_0 \rangle$ `Pair` $\langle$ `A , B` $\rangle \langle \rho_1, \rho_2, \rho_3 \rangle \rightarrow$ `unit`, where ρ_0^a and ρ_{1-3} are fresh region variables, and φ_0 is a fresh predicate variable denoting unknown constraints over ρ_0^a and ρ_{1-3}. Region type template of a type variable `a` is `a` $@\rho$, where ρ is fresh. For a class, a template is computed in two steps. In the first step, we templatize the types of all its fields, constructor arguments, and arguments and return values of all its methods, along with the type of its superclass. In the second step, we generalize all the free region variables occuring in the templates computed in the first step as region parameters of the class. Finally, we add a fresh allocation region parameter (ρ^a) to the list of parameters, and introduce a new predicate variable (φ) to denote unknown constraints on region parameters. For example, consider the standard FGJ definition of `Pair` $\langle$ `a , b` $\rangle$ class, where `a` and `b` are the types of `fst` and `snd` fields, respectively. It can be templatized as following:

```
class Pair⟨a ◁ Object , b ◁ Object ⟩
          ⟨ρ0^a, ρ0−4 | φ0⟩ ◁ Object ⟨ρ4⟩ {
  a@ρ0 fst;
  b@ρ1 snd;
  Pair(a@ρ2 fst, b@ρ3 snd) { ... }
  ...
}
```

The template elides `Pair`'s methods, whose region type templates contain no free variables. Among the region parameters of the class template, ρ_{0-4} are obtained by generalizing free region variables in the types of its class's fields, constructor arguments, and its superclass type. The remaining parameter (ρ_0^a) is a fresh region variable denoting the allocation region argument. Fresh predicate variable φ_0 denotes unknown constraints over ρ_0^a and ρ_{0-4} that need to hold for template to be a well-formed region-annotated class definition in FB.

The type template for a recursively defined class is computed in two steps. First, all recursive occurances of the class among the types of its fields are ignored and the class is templatized as if it is a non-recursive class. Next, all the recursive occurances are templatized with respect to the class template computed in the first step, such that their region annotations are exactly same as the region parameters of the class. For example, consider a generic `ListNode` $\langle$ `a` $\rangle$ class containing two fields: `data` of type `a` and `next` of type `ListNode` $\langle$ `a` $\rangle$. where `a` is the type of the data stored in the linked list node. To templatize the `ListNode` $\langle a \rangle$ class, we first ignore its recursive occurance in the type of `next` field, and templatize the type `a` of `data` field as `a` $@\rho_0$. Based on this type template of `data` field, we compute the class's template as following:

[14]In our exposition, we assume that classes `A` and `B` are trivial subclasses of `Object` with no fields/methods. Like `Object`, they accept one region parameter - the allocation region of their objects.

[15]We abuse arrow notation to also represent types of methods, but unlike function types, there is no allocation region annotation atop the arrow in a method type.

$\mathsf{elabExpr}(CT, \mathcal{A}, \pi^a, \Gamma, e)$ =
  match e **with**
  | `new` $K(\overline{e}) \longrightarrow$
    let N = $\mathsf{templateTy}(K)$ **in**
    let C_1 = $\mathsf{typeOk}(\mathcal{A}, N)$ **in**
    let C_2 = **match** K **with** `Region` $\langle T \rangle \longrightarrow \top$
        | $\_ \longrightarrow \{\mathcal{A}.\Phi \vdash \mathsf{allocRgn}(N) = \pi^a\}$ **in**
    let $(\_ : \overline{\tau})$ = $\mathsf{fields}(N)$
    let $(\overline{e'} : \overline{\tau'}, C_3)$ = $\mathsf{elabExpr}(\mathcal{A}, \pi^a, \Gamma, \overline{e})$ **in**
    let C_4 = $\mathsf{subtypeOk}(\mathcal{A}, \overline{\tau'}, \overline{\tau})$ **in**
    (`new` $N(\overline{e'}) : N, \bigcup_{i=1}^{4} C_i$)
  | $e_a(\overline{e}) \longrightarrow$
    let $(e'_a : \langle \rho^a \overline{\rho} \,|\, \phi \rangle \overline{\tau} \xrightarrow{\pi} \tau, C_1)$ =
        $\mathsf{elabExpr}(\mathcal{A}, \pi^a, \Gamma, e_a)$ **in**
    let $\overline{\rho'}$ = $\mathsf{fresh}_\rho()$ **in**
    let C_2 = $\{\overline{\rho'} \in \mathcal{A}.\Delta\}$ **in**
    let $\mathcal{S}$ = $[\overline{\rho'}/\overline{\rho}][\pi^a/\rho^a]$ **in**
    let C_3 = $\{\mathcal{A}.\Phi \vdash \mathcal{S}(\phi)\}$ **in**
    let $(\overline{e'} : \overline{\tau'}, C_4)$ = $\mathsf{elabExpr}(\mathcal{A}, \pi^a, \Gamma, \overline{e})$ **in**
    let C_5 = $\mathsf{subtypeOk}(\mathcal{A}, \overline{\tau'}, \overline{\mathcal{S}(\tau)})$ **in**
    $(e'_a \langle \pi^a \overline{\rho'} \rangle (\overline{e'}) : \mathcal{S}(\tau), \bigcup_{i=1}^{5} C_i)$
  | `letregion` π `in` $e_a \longrightarrow$
    let π' = $\mathsf{fresh}_\pi()$ **in**
    let (Δ, Θ, Φ) = $\mathcal{A}$ **in**
    let $\mathcal{A}'$ = $(\Delta \cup \{\pi'\}, \Theta, \Phi \wedge (\Delta \succeq \pi'))$ **in**
    let $(e'_a : \tau, C_1)$ = $\mathsf{elabExpr}(\mathcal{A}', \pi', \Gamma, [\pi'/\pi]e_a)$ **in**
    (`letregion` π' `in` $e'_a : \tau, C_1$)
  | `open` e_a `as` $y@\pi$ `in` $e_b \longrightarrow$
    let $(e'_a :$ `Region` $\langle T \rangle \langle \rho \rangle, C_1)$ =
        $\mathsf{elabExpr}(\mathcal{A}, \pi^a, \Gamma, e_a)$ **in**
    let π' = $\mathsf{fresh}_\pi()$ **in**
    let (Δ, Θ, Φ) = $\mathcal{A}$ **in**
    let $(\mathcal{A}', \Gamma')$ = $((\Delta \cup \{\pi'\}, \Theta, \Phi), \Gamma[y \mapsto T@\pi'])$ **in**
    let $(e'_b : \tau, C_2)$ = $\mathsf{elabExpr}(\mathcal{A}', \pi', \Gamma', [\pi'/\pi]e_a)$ **in**
    (`open` e'_a `as` $y@\pi'$ `in` $e'_b : \tau,\ C_1 \cup C_2$)
  | $\_ \longrightarrow$ ...

Figure 9: Constraint generation for expressions in $[\![FB]\!]$

`ListNode` $\langle$ `a` $\triangleleft$ `Object` $\rangle \langle \rho_0^a, \rho_0 \,|\, \varphi_0 \rangle \triangleleft$ `Object` $\{\ldots\}$

Next, we trivially templatize the type of `next` field in the body of the class as `ListNode` $\langle$ `a` $\rangle \langle \rho_0^a, \rho_0 \rangle$. The resulting class represents a linked list with spine in the region ρ_0^a and data objects in the region ρ_0.

The templatization technique we described above for recursive class definitions can be extended to mutually recursive definitions in a straightforward manner, by simultaneously templatizing them. Using the techniques outlined above, we compute region type templates for all classes bound in the class table of the $[\![FB]\!]$ program before we proceed to elaborate expressions.

4.2 Expression Elaboration

Elaborating $[\![FB]\!]$ expressions to FB expressions involves (a). replacing core types in variable declarations and `new` expressions with fresh region type templates, and (b). explicitly instantiating region parameters of methods with fresh region variables in method calls and function applications. This elaboration is performed with respect to the polymor-

$$
\begin{array}{l}
\mathsf{elabMeth}(CT, B, \tau\ m\langle \rho^a_m \overline{\rho_m} \mid \varphi_m\rangle(\overline{\tau}\ \overline{x})\{\,\mathtt{return}\,e;\}) \ = \\
\quad \mathbf{let}\ \ \mathtt{class}\ B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \rho^a \overline{\rho} \mid \varphi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{x};\ k\ \overline{d}\} \ = \ CT(B)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (\Delta,\Theta,\Phi)\ \ \mathbf{as}\ \ \mathcal{A} \ = \\
\qquad\qquad (\{\rho^a, \overline{\rho}, \rho^a_m, \overline{\rho_m}\}, \overline{a} \triangleleft \overline{K}, \varphi \wedge \varphi_m)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ C_1 \ = \ \{\Delta \vdash \varphi_m\ \mathtt{ok}\}\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ \Gamma \ = \ \cdot[\mathtt{this} \mapsto B\langle \overline{a}\rangle\langle \rho^a \overline{\rho}\rangle][\overline{x} \mapsto \overline{\tau}]\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (e' : \tau', C_2) \ = \ \mathsf{elabExpr}(\mathcal{A}, \rho^a_m, \Gamma, e)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ C_3 \ = \ \mathsf{subtypeOk}(\mathcal{A}, \tau', \tau)\ \ \mathbf{in} \\
\qquad (\tau\ m\langle \rho^a_m \overline{\rho_m} \mid \varphi_m\rangle(\overline{\tau}\ \overline{x})\{\,\mathtt{return}\,e';\},\ \ C_1 \cup C_2 \cup C_3)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{elabClass}(CT, B) \ = \\
\quad \mathbf{let}\ \ \mathtt{class}\ B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \rho^a \overline{\rho} \mid \varphi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{x};\ k\ \overline{d}\} \ = \ CT(B)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (\Delta,\Theta,\Phi)\ \ \mathbf{as}\ \ \mathcal{A} \ = \ (\{\rho^a, \overline{\rho}\}, \overline{a} \triangleleft \overline{K}, \varphi)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ C_1 \ = \ \Delta \vdash \varphi\ \mathtt{ok}\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (C_2, C_3) \ = \ (\mathsf{typeOk}(\mathcal{A}, N), \mathsf{typeOk}(\mathcal{A}, \overline{\tau^f}))\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ C_4 \ = \ \{\Phi \vdash \mathsf{allocRgn}(\overline{\tau^f}) \succeq \rho^a \wedge \mathsf{allocRgn}(N) = \rho^a\}\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (k', C_5) \ = \ \mathsf{elabCons}(B, k)\ \ \mathbf{in} \\
\quad \mathbf{let}\ \ (\overline{d'}, C_6) \ = \ \mathsf{elabMeth}(B, \overline{d})\ \ \mathbf{in} \\
\qquad (\mathtt{class}\ B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \rho^a \overline{\rho} \mid \varphi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{x};\ k'\ \overline{d'}\},\ \ \bigcup_{i=1}^{6} C_i)
\end{array}
$$

Figure 10: Method, class and class table elaboration

phic type templates of classes and methods computed as per §4.1.

Function elabExpr, shown in Fig. 9, performs this elaboration for a subset of expressions in $[\![FB]\!]$, whose corresponding FB expressions have been ascribed static semantics in Fig. 6. elabExpr is defined under the same context as the expression typing judgment in Fig. 6 with symbols $\mathcal{A}$,π^a, and Γ retaining their meaning. The function traverses expressions in a syntax-directed manner of a type checker, introducing fresh region type templates for unknown region types, while generating constraints over region and predicate variables. The precise nature of generated constraints is explained in §4.4, but in summary, they capture the relationships between the type templates of various subexpressions and their well-formedness. Note that elabExpr returns the region type template of the subexpression, which is used to generate constraints for the expression. Functions typeOk and subtypeOk (definitions not shown) used by elabExpr implement type well-formedness and subtype judgments from Fig. 6, respectively.

4.3 Method and Class Elaboration

Functions elabMeth and elabClass shown in Fig. 10 lift expression elaboration to method and class definitions, respectively. Both functions first build a context ($\mathcal{A}$) containing a set (Δ) of region variables denoting regions that are currently live, a map (Θ) mapping type variables to their bounds, and a constraint formula (Φ) capturing constraints over live region variables. We use predicate variables (φ and φ_m) to capture constraints over variables in Δ denoting the fact that such constraints are yet to be inferred.

Function elabMeth elaborates a method definition of class B. It calls elabExpr with the context $\mathcal{A}$, its allocation context parameter (ρ^a_m), and a type environment (Γ) that contains region type bindings for all the arguments of the method, including the implicit `this` argument. The region type template returned by elabExpr for the method body is checked against its expected type (derived from the type template of the method) generating more constraints. The function then returns the elaborated method definition and the set of constraints.

elabClass elaborates the definition of a class B. It relies on elabCons[16] and elabMeth functions to elaborate B's constructor (k) and method definitions ($\overline{d}$), respectively. To the set of constraints returned by these functions, elabClass adds constraints generated by checking the well-formedness of the type templates of its superclass and fields, and also a new constraint capturing a couple of safety conditions: first, the allocation regions of objects referred by the instance variables should outlive the allocation region of the instance itself, and second, the allocation regions of a class type and its superclass type must be the same.

Function elabClassTable (Fig. 10) elaborates every definition in the class table CT, while accumulating constraints. The constraints are finally solved by solve (§4.5), which returns substitution functions $\mathcal{S}_\rho$ and $\mathcal{S}_\varphi$ for free region and predicate variables, respectively, introduced during templatization and elaboration stages. The substitutions are applied to the class table (and to the artifacts that make up the class table, recursively) to compute a class table that maps classes to their fully region-annotated definitions in FB.

Note that if the original program in $[\![FB]\!]$ contains unsafe references, for example, a reference from a transferable region to a stack regions, then the constraints generated during the elaboration are not satisfiable. In such case, solve fails to solve constraints, causing the program to be rejected.

4.4 Constraints

Our constraint generation algorithm generates three kinds of constraints:

- Well-formedness constraints of form $\rho \in \Delta$, restricting the domain of unification for a region variable (ρ) to the set (Δ) of regions in scope,
- Well-formedness constraints of form $\Delta \vdash \varphi\ \mathtt{ok}$, restricting the domain of a predicate variable (φ) to the set of all possible constraint formulas over region variables (Δ) in scope, and
- Validity constraints of form $\Phi \vdash \phi$, where Φ and ϕ are region constraints (Fig. 4) extended with predicate variables and *pending substitutions*[17]:

$$
\begin{array}{lcl}
\Phi, \phi & ::= & \mathit{true} \ \mid\ \rho \succeq \rho \ \mid\ \rho = \rho \ \mid\ F(\varphi) \ \mid\ \phi \wedge \phi \\
F & ::= & \cdot \ \mid\ [\rho/\rho]F
\end{array}
$$

A pending substitution (F) is a substitution function over region variables/identifiers. They represent the substitu-

[16] The definition of elabCons is straightforward, hence not shown.

[17] we borrowed this terminology from [12]

tions that need to be carried out when a predicate variable (φ) is replaced by a concrete formula in a validity constraint. For instance, in the validity constraint $\pi_1 \succeq \pi_2 \vdash [\pi_1/\rho_1][\pi_2/\rho_2]\varphi$, pending substitution is $[\pi_1/\rho_1][\pi_2/\rho_2]$. Any concrete formula (call it ϕ_{sol}) over variables ρ_1 and ρ_2 is a solution to φ if and only the formula obtained by substituting π_1 and π_2 for ρ_1 and ρ_2 (resp.) in ϕ_{sol} is deducible from $\pi_1 \succeq \pi_2$.

In general, validity constraints generated by our algorithm assume one of the following two forms:

$$\phi_{cx} \wedge \bigwedge_i \varphi_i \vdash \phi_{cs} \qquad \phi_{cx} \wedge \bigwedge_i \varphi_i \vdash F_j(\varphi_j)$$

Where φ_i's denote the unknown preconditions of the class and the method under elaboration. If the constraint is generated while checking the well-formedness of a type or elaborating an expression, then φ_j's denote the unknown preconditions of classes and methods that were used in that type or expression. Each use of a (region-polymorphic) class or a method may instantiate region parameters differently, resulting in a different pending substitution (F_j). Formulas ϕ_{cx} and ϕ_{cs} are concrete, i.e., free of predicate variables and pending substitutions. While ϕ_{cx} captures relationships that are *known* to hold between concrete region identifiers (i.e., π's) when the constraint was generated, ϕ_{cs} captures relationships that are *required* to hold among region varibles (i.e., ρ's), or relationships between region variables and identifiers. Each region variable occuring in ϕ_{cs} has an associated well-formedness constraint, which specifies its unification domain. The unification domain of a constraint is the union of unification domains of all region variables occuring in the constraint.

Constraints Example 1 Consider the `Pair` class template from §4.1. Following constraints are generated during its elaboration (Constrains are identified with $\mathbf{c_i}$'s. Some trivial constraints, such as $\rho_4 \in \Delta_0$ and $\rho_5 \in \Delta_1$, where $\Delta_0 = \{\rho_0^a, \rho_{0-4}\}$ and $\Delta_1 = \Delta_0 \cup \{\rho_5\}$, have been elided):

$$[\mathbf{c_1}] : \Delta_0 \vdash \varphi_0 \ \mathtt{ok} \qquad [\mathbf{c_2}] : \varphi_0 \vdash \rho_0 \succeq \rho_0^a \wedge \rho_1 \succeq \rho_0^a \wedge \rho_4 = \rho_0^a$$
$$[\mathbf{c_3}] : \varphi_0 \vdash \rho_2 = \rho_0 \qquad [\mathbf{c_4}] : \varphi_0 \vdash \rho_3 = \rho_1$$
$$[\mathbf{c_5}] : \varphi_0 \wedge \varphi_1 \vdash \rho_5 = \rho_0 \qquad [\mathbf{c_6}] : \Delta_1 \vdash \varphi_1 \ \mathtt{ok}$$

Constraints Example 2 Let us add to the `Pair` class a contrived method `alt` that accepts a `Region` object `r`, a `Pair<A,A>` object `q`, and an `A` object `y`. It assigns `y` to `fst` and `snd` fields of `q`, and calls itself recursively with the same region, a new `Pair` object allocated in a local region, and an `A` object referred by the `snd` field of the pair inside the region. `alt` never terminates. Elaboration phase elaborates the method to the following region-annotated definition[18](The original definition of `alt` can be obtained by erasing all the region annotations from the elaborated version):

[18]In reality, elaboration uses new region variables as parameters to the constructor and method calls, and then generates constraints that unify them with actuals. In our examples, to avoid clutter due to trivial constraints,we coalesced both steps and show the actuals instead.

```
unit alt<ρ₂ᵃ,ρ6−9 | φ2>(Region<Pair<A,A>><π⊤> r,
                  Pair<A,A><ρ6−8> q, A<ρ9> y) {
  q.fst := y; q.snd := y;
  open r as π0 withroot p in
    letregion π1 in
      let x = new Pair<A,A><π1,π0,π0>
                   (p.fst,p.fst) in
        alt<π1,π1,π0,π0,π0>(r,x,p.snd)
}
```

Constraints generated during the elaboration are shown below (let $\Delta_2 = \{\rho_0^a, \rho_{0-4}, \rho_2^a, \rho_{6-9}\}$):

$$[\mathbf{c_7}] : \Delta_2 \vdash \varphi_2 \ \mathtt{ok} \qquad [\mathbf{c_8}] : \varphi_1 \wedge \varphi_2 \vdash \rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6$$
$$[\mathbf{c_9}] : \varphi_1 \wedge \varphi_2 \vdash \rho_7 = \rho_9 \qquad [\mathbf{c_{10}}] : \varphi_1 \wedge \varphi_2 \vdash \rho_8 = \rho_9$$
$$[\mathbf{c_{11}}] : \varphi_1 \wedge \varphi_2 \wedge \pi_0 \succeq \pi_1 \vdash [\pi_1/\rho_2^a][\pi_1/\rho_6][\pi_0/\rho_{7-9}]\varphi_2$$

4.5 Constraint Solving

Our constraint generation algorithm traverses the entire program, performing elaboration and collecting constraints, which are subsequently solved en masse. The motivation behind the whole-program approach to constraint generation is twofold: it simplifies elaboration functions and makes presentation easier, and second, it naturally generalizes to mutual recursion. Nonetheless, we do not intend our type inference to be a whole-program analysis for (a). it preempts opportunities for separate compilation and dynamic linking, and (b). it is expensive and an overkill in most practical cases. We therefore reclaim the compositionality of type inference by solving the constraints in a compositional fashion. In more practical terms this means that our constraint solving algorithm visits and solves every constraint (or, every set of mutually dependent constraints) only once. It composes computed solutions to solve other constraints that depend on the solved constraints. Importantly, the failure to solve a dependent constraint does not result in backtracking. We now describe our compositional constraint solving algorithm in detail. To simplify the presentation of our algorithm, we assume that are no mutually recursive definitions in the source program. Recursive definitions are nonetheless allowed.

Terminology In a validity constraint, a predicate variable occuring on the left side of the turnstile is said to occur negatively, or with *negative polarity*. In contrast, a predicate variable occuring on the right side is said to occur positively, or with *positive polarity*. A validity constraint *constrains* the set of predicate variables that occur negatively in the constraint, while it *uses* the set of predicate variables that occur positively. A constraint is said to be *recursive* if it constrains and uses a predicate variable.

Given a set of validity constraints, we first build a dependency graph (G_c) with constraints as nodes, and dependencies between them captured as edges. There exists an edge from a constraint c_2 to a constraint c_1 in the graph (i.e., c_2 *depends on* c_1) if any of the following conditions hold:

- c_1 constrains a predicate variable that c_2 uses.

- c_1 constrains a (non-strict) subset of predicate variables that c_2 constraint.

The first condition intuitively corresponds to a case, where expression or type, whose region elaboration is constrained by c_2 refers to a method or a class, whose unknown precondition is constrained by c_1. The common predicate variable (φ) represents the unknown precondition in this case. The dependency from c_2 to c_1 means that c_1 must be solved to compute φ before c_2 is solved, thus enforcing the rule that the precondition of a method must not depend on its calling context. The second condition captures two kinds of dependencies. First, the dependency from the constraints over a method precondition to the constraints over the precondition of the class containing the method. This captures our preference that the constraints over a class's region parameters should not depend on the idiosyncracies of its methods. Any additional constraints required by any of its methods must be captured in the precondition of the method itself (well-formedness rules allow this possibility). The second condition adds bidirectional dependencies between validity constraints that constraint the same set of predicate variables.

Next, we convert the dependency graph over constraints into a dependency DAG (G_C) over sets of constraints, where each set represents a strongly connected component in the dependency graph.

Example The dependency DAG (G_C) over validity constraints from the `Pair` example (§ **??**) is shown below:

$$\mathtt{C_2}:\{\mathtt{c_5}\} \longrightarrow \mathtt{C_1}:\{\mathtt{c_{2-4}}\} \longleftarrow \mathtt{C_3}:\{\mathtt{c_{8-11}}\}$$

Each node (labeled C_i) is a set of constraints that belong to a strongly connected component in the dependency graph (G_c), hence are mutually dependent. All dependencies, except the self-dependency on $\mathbf{c_{10}}$, are type-2 dependencies.

A dependency DAG makes the dependencies between constraints explicit. Constraints in each set are mutually dependent, and need to be solved simultaneously, whereas constraints in different sets can be solved as per any valid topological ordering of the graph's transpose. Accordingly, we obtain a topological ordering of nodes in the graph G_C^T (G_C's transpose), and solve the sets of constraints in that order. The solutions obtained after solving a constraint set are applied to the constraints in subsequent sets before attempting to solve them. Consequently, when the turn of a constraint set (C) arrives during the constraint solving process, it satisfies certain properties:

- There exists only one predicate variable (φ) that is either constrained or used by the constraints in the set (C). The variable is called set's *subject*. This property follows from (a). the fact that all the dependency constraints have already been solved (and solutions applied), and (b). the assumption that there are no mutually recursive definitions.
- All the constraints that constrain the set's subject are present in the set. This follows from our definition of the dependency relation.

For the DAG in figure above, we consider the topological order $[C_1, C_2, C_3]$ of its transpose, and solve the sets of constraints in that order. Solving the set (C) of constraints entails finding an assignment for C's subject (φ), and also all the region variables ($\overline{\rho}$) that occur free in C, such that the solution satisfies well-formedness constraints on φ and $\overline{\rho}$. To simplify presentation, we think of C as being parameterized on φ and $\overline{\rho}$, and write it as $C[\varphi, \overline{\rho}]$. We now formalize the constraint satisfaction problem, and its solution.

Definition ***(Constraint Satisfaction Problem (CSP))*** A constraint satisfaction problem is a tuple $(C[\varphi, \overline{\rho}], \Delta_\varphi, \overline{\Delta})$, where $C[\varphi, \overline{\rho}]$ is a set of validity constraints, where i'th validity constraint assumes one of the following forms:

$$\phi^i_{cx} \wedge \varphi \vdash \phi^i_{cs} \qquad \phi^i_{cx} \wedge \varphi \vdash F_i(\varphi)$$

$\overline{\Delta}$ are the unification domains for $\overline{\rho}$. We call the union of all unification domains ($\bigcup \overline{\Delta}$) as the unification domain (Δ) of the CSP. The solution to the constraint satisfaction problem is a pair $(\mathcal{S}, \phi_{sol})$, where $\mathcal{S}$ is a map from $\overline{\rho}$ to Δ such that $\mathcal{S}(\rho_j) \in \Delta_j$, for every j, and ϕ_{sol} is a constraint formula such that:

- ϕ_{sol} is well-formed under Δ_φ (i.e., $\Delta_\varphi \vdash \phi_{sol}\ \mathtt{ok}$).
- Every sequent in $C[(\phi_{sol}, \mathcal{S}(\overline{\rho}))]$ is valid.
- ϕ_{sol} is maximally weak. That is, $\nexists \phi'_{sol}$ such that ϕ'_{sol} is well-formed, $C[(\phi'_{sol}, \mathcal{S}(\overline{\rho}))]$ is valid, and ϕ'_{sol} is strictly weaker than ϕ_{sol} (i.e., $\cdot \vdash \phi_{sol} \Rightarrow \phi'_{sol}$ but $\cdot \nvdash \phi'_{sol} \Rightarrow \phi_{sol}$).

4.6 Solving the CSP

The first step of solving the CSP $(C[\varphi, \overline{\rho}], \Delta_\varphi, \overline{\Delta})$ is to cast it as an equivalent problem $(c[\varphi, \overline{\rho}], \Delta_\varphi, \overline{\Delta})$ involving a single validity constraint. The single constraint ($[\varphi, \overline{\rho}]$) is:

$$\Phi_{cx} \wedge \varphi \vdash \Phi_{cs} \wedge \bigwedge_i F_i(\varphi)$$

Where $\Phi_{cx} = \bigwedge_i \phi^i_{cx}$ and $\Phi_{cs} = \bigwedge_i \phi^i_{cs}$. To see why both CSPs are equivalent, consider two distinct constraints, $c_i[\varphi, \overline{\rho}_i]$ and $c_j[\varphi, \overline{\rho}_j]$. Since $i \neq j$, $\overline{\rho}_i \neq \overline{\rho}_j$. Without the loss of generality, assume that c_i was generated before c_j by the constraint generation algorithm. Observe that when our constraint generation algorithm generates a constraint, all relationships between concrete region identifiers referred by the consequent of constraint are already present in its antecedent (this includes the unification domains of region variables referred by the consequent). Since no new relationships between existing regions are added when a new constraints is generated, ϕ^j_{cx} either describes the same relationships that are already described by ϕ^i_{cx}, or describes relationships among region identifiers that are new, and not relevant to c_i. Therefore, strengthening the context of c_i with that of c_j (or vice versa) neither weakens nor strengthens c_i (resp. c_j).

For the constraint sets C_1, C_2, and C_3 in the running example, equivalent constraints are c_{12}, c_{13}, and c_{14}, respectively, shown below:

$$\begin{array}{l} [\mathbf{c_{12}}]:\ \varphi_0 \vdash \rho_0 \succeq \rho_0^a \wedge \rho_1 \succeq \rho_0^a \wedge \rho_4 = \rho_0^a \wedge \rho_2 = \rho_0 \wedge \rho_3 = \rho_1 \\ [\mathbf{c_{13}}]:\ \varphi_0 \wedge \varphi_1 \vdash \rho_5 = \rho_0 \\ [\mathbf{c_{14}}]:\ \varphi_1 \wedge \varphi_2 \wedge \pi_0 \succeq \pi_1 \vdash \rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6 \wedge \rho_7 = \rho_9 \wedge \\ \qquad \rho_8 = \rho_9 \wedge [\pi_1/\rho_2^a][\pi_1/\rho_6][\pi_0/\rho_{7-9}]\varphi_2 \end{array}$$

4.6.1 Non-Recursive Constraints

We first describe how we solve a non-recursive CSP $(c[\varphi, \overline{\rho}], \Delta_\varphi, \overline{\Delta})$, where $c[\varphi, \overline{\rho}]$ is of the form $\Phi_{cx} \wedge \varphi \vdash \Phi_{cs}$.

Our approach is based on the observation that FB does not admit null values or uninitialized variables, thus forcing every region variable to be unified with some concrete region. The constraint formula Φ_{cs} captures all such unification constraints on $\overline{\rho}$. Since the set of all concrete region identifiers is the unification domain (Δ) of CSP, this means for every ρ_i with a well-formedness constraint as $\rho_i \in \Delta_i$, there exists a $\pi \in \Delta$ such that and $\Phi_{cx} \wedge \Phi_{cs} \vdash \rho = \pi$. However, π may not belong to Δ_i, in which case the well-formedness constraint on ρ_i is not satisfied, and constraint solving must fail. Therefore, there exists a unique assignment $\mathcal{S}$ such that $c[\phi_{sol}, \mathcal{S}(\overline{\rho})$ is satisfied, regardless of ϕ_{sol}.

To obtain ϕ_{sol}, we make use of another observation. Let $c[\varphi, \mathcal{S}(\overline{\rho})]$ be the following constraint:

$$\Phi_{cx} \wedge \varphi \vdash \Phi'_{cs}$$

Consider a maximally weak formula ϕ such that $\Phi_{cx} \vdash \phi \Leftrightarrow \Phi'_{cs}$. Clearly, $\Delta \vdash \phi\ \texttt{ok}$. However, since $\Delta_\varphi \subseteq \Delta$, we have two cases:

- Case $\Delta_\varphi \vdash \phi\ \texttt{ok}$: This means that ϕ is a solution to φ.
- Case $\Delta_\varphi \nvdash \phi\ \texttt{ok}$: In this case, ϕ contains at least one equality or outlives constraint on two region identifiers, π_i and π_j, where (a). $\pi_i, \pi_j \in \Delta - \Delta_\varphi$, or (b). $\pi_i \in \Delta - \Delta_\varphi$ and $\pi_j \in \Delta_\varphi$, such that the constraint is not implied by Φ_{cx} (if it is implied, then ϕ is not maximally weak). Let us denote such constraint on π_i and π_j as ϕ_{ij}. Now, let us consider a solution ϕ_{sol} to φ, which means that $\Phi_{cx} \wedge \phi_{sol} \vdash \Phi'_{cs}$. Since $\Phi_{cx} \vdash \phi \Leftrightarrow \Phi'_{cs}$, we have $\Phi_{cx} \wedge \phi_{sol} \vdash \phi$. Since $\phi \vdash \phi_{ij}$, we have $\Phi_{cx} \wedge \phi_{sol} \vdash \phi_{ij}$, although $\Phi_{cx} \nvdash \phi_{ij}$. But this is impossible. To see why, recall that all the constraints on identifiers in Δ_φ occuring in Φ_{cx} are subsumed by $(\Delta - \Delta_\varphi) \succeq \Delta_\varphi$. Therefore, it is impossible to derive ϕ_{ij} from Φ_{cx} by only adding constraints on $\pi \in \Delta_\varphi$. Hence, such a solution ϕ_{sol} cannot exist.

The above discussion hints at an algorithm to compute a solution to φ: find a maximally weak ϕ_{sol} such that $\Phi_{cx} \vdash \phi_{sol} \Leftrightarrow \Phi'_{cs}$. If $\Delta_\varphi \vdash \phi_{sol}\ \texttt{ok}$, then ϕ_{sol} is the solution. Otherwise, there is no solution to φ. We now describe a graph-based algorithm to compute maximally weak ϕ_{sol}.

Definitions Given a constraint formula ϕ, we define its graph encoding $G(\phi) = (V(\phi), E(\phi))$ as a digraph whose vertices ($V(\phi)$) are free region variables and identifiers in ϕ, and whose edges ($E(\phi)$) denote outlives constraints in[19] ϕ. That is, if ϕ contains a constraint $\rho_1 \succeq \rho_2$, then $\rho_1, \rho_2 \in V(\phi)$ and $(\rho_1, \rho_2) \in E(\phi)$. Equality constraints are treated as a conjunction of symmetric outlives constraints for the purpose of graph encoding. Conversely, given a digraph G, we define its constraint encoding $\Phi(G)$ in a straightforward manner. We say that a graph $G_1 = (V_1, E_1)$ is as connected as graph $G_2 = (V_2, E_2)$ if $V_2 \subseteq V_1$, and for every $\rho_1, \rho_2 \in V_2$, if there exists a path between ρ_1 and ρ_2 in G_2, then there must exist a path between same pair of vertices in G_1.

A maximally weak ϕ_{sol} that satisfies $\phi_{cx} \vdash \phi_{sol} \Leftrightarrow \phi_{cs}$ is a constraint encoding of the smallest graph G (i.e., $\Phi(G)$) such that $G(\phi_{cx}) \cup G$ is as connected as $G(\phi_{cs})$. The problem of computing such a G is equivalent to the problem finding of finding minimum number of edges to add to $G(\phi_{cx})$ such that it is as connected as $G(\phi_{cs})$. Algorithms to solve the later problem are known to exist [3].

Solving the constraints c_{12} and c_{13} by reducing them to graph augmentation problems, as described above, results in the following solutions for φ_0 and φ_1 (symmetric outlives constraints are replaced with equalities):

$$\begin{array}{c} \varphi_1 \Leftrightarrow \rho_0 \succeq \rho_0^a \wedge \rho_1 \succeq \rho_0^a \wedge \rho_4 = \rho_0^a \wedge \rho_2 = \rho_0 \wedge \rho_3 = \rho_1 \\ \varphi_2 \Leftrightarrow \rho_5 = \rho_0 \end{array}$$

4.6.2 Solving Recursive Constraints

Substituting the above solutions for φ_0 and φ_1 in c_{14} gives us a recursive constraint of the following form:

$$\phi_{cx} \wedge \varphi \vdash \phi_{cs} \wedge F(\varphi)$$

Where ϕ_{cx} and ϕ_{cs} are concrete constraint formulas, and F is a substitution function, not necessarily idempotent. We now extend constraint solving to recursive CSP $(c[\varphi, \overline{\rho}], \Delta_\varphi, \overline{\Delta})$, where $c[\varphi, \overline{\rho}]$ assumes the form $\Phi_{cx} \wedge \varphi \vdash \Phi_{cs} \wedge \bigwedge_i F_i(\varphi)$. For the sake of brevity, we define $G(\varphi) = \Phi_{cx} \wedge \bigwedge_i F_i(\varphi)$, and use F in $c[\varphi, \overline{\rho}]$:

$$\Phi_{cx} \wedge \varphi \vdash G(\varphi)$$

To solve the above recursive constraint, we start with the observation that the set of all constaints (ϕ) over $\Delta \cup \{\overline{\rho}\}$ is a lattice, where $\phi_1 \leq \phi_2 \triangleq \cdot \vdash \phi_1 \Rightarrow \phi_2$. Note that G is a monotone over the lattice:

$$\forall \phi_1, \phi_2. \cdot \vdash (\phi_1 \Rightarrow \phi_2) \Rightarrow (G(\phi_1) \Rightarrow G(\phi_2))$$

From Knaster-Tarski's theorem, we know that G has a greatest fixed point (ϕ_f), with following properties:

- **Property 1** $\phi_f = G(\phi_f)$, which means $\cdot \vdash \phi_f \Leftrightarrow G(\phi_f)$
- **Property 2** $\forall \phi'_f$ such that $\phi'_f = G(\phi'_f)$, we have $\phi'_f \leq \phi_f$, which means $\cdot \vdash \phi'_f \Rightarrow \phi_f$.

[19] We alternate between viewing a constraint formula (ϕ) as a set of constraints and a conjunction of constraints in this description

We therefore compute the greatest fixed point (ϕ_f) of G, and convert the recursive constraint to the following non-recursive constraint:

$$\Phi_{cx} \wedge \varphi \vdash \phi_f$$

The technique described in § 4.6.1 now suffices to solve the above constraint.

Let $H(\varphi_2)$ denote the consequent of the recursive constraint c_{14}. Its greatest fixed point (ϕ_f) is shown below:

$$\rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6 \wedge \rho_7 = \rho_9 \wedge \rho_8 = \rho_9 \wedge \pi_0 \succeq \pi_1 \wedge \pi_0 = \pi_0$$

Computing a maximally weak ϕ_{sol} such that $\varphi_1 \wedge \pi_0 \succeq \pi_1 \vdash \phi_{sol} \Leftrightarrow \phi_f$ results in the following solution for φ_2 that meets its well-formedness requirement ($\{\rho_0^a, \rho 0 - 4, \rho_2^a, \rho_{6-9}\} \vdash \varphi_2 \ \texttt{ok}$):

$$\rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6 \wedge \rho_7 = \rho_9 \wedge \rho_8 = \rho_9$$

5. Implementation and Evaluation

We have implemented the prototype of BROOM compiler frontend, including its region type system and type inference, in 3k+ lines of OCaml. The input to our system is a program in $[\![FB^+]\!]$, an extended version of $[\![FB]\!]$ that includes assignments, conditionals, loops, more primitive datatypes (*e.g.*strings), and a null value. Our implementation of region type inference closely follows the description given in Sec. 4. To solve the constraints that arise during type inference, we built a solver called CSOLVE that implements constraint solving approach based on fixpoint computation and graph augmentation described in § 4.6.1. If the input $[\![FB^+]\!]$ program does not create any unsafe references, our compiler annotates it with region types, which act as a witness to program's memory safety. On contrary, if the input program does create a reference that is potentially unsafe, then the type inference fails during the constraint solving phase. We currently do not implement error localization and feedback mechanisms.

To evaluate the practical utility of our region type system and type inference, we translated some of the Naiad streaming query operator benchmarks (Naiad vertices) used in [4] to $[\![FB^+]\!]$, and used our prototype compiler to assign region types to these programs, and thus prove their safety. During the process, we found multiple instances of potential memory safety violations in the $[\![FB^+]\!]$ translation of benchmarks, which we verified to be present in the original C# implementation as well. The cause of all safety violations is the creation of a reference from the outgoing message (a transferable region) to the payload of the incoming message. For example, the implementation of `RegionSelectVertex` contains the following:

```
if (this.selectFn(inMsg.payload[i])) {
  outMsg.set(outputOffset, inMsg.payload[i]);
  ...
}
```

The `outMsg` is later transferred to a downstream actor, where the reference to `inMsg`'s payload becomes unsafe[20]. We eliminated such unsafe references by creating a clone of `inMsg.payload[i]` in `outMsg`, and our compiler was subsequently able to certify the safety of all references.

6. Related Work

Tofte and Talpin in [11, 15, 16] introduce the concept of a region type system to statically ensure the safety of region-based memory management in ML. Following their seminal work, static type systems for safe region-based memory management have been extensively studied in the context of various languages and problem settings [1, 2, 5–8, 13, 18, 19]. Our work differs from the existing proposals in a number of ways. Firstly, our problem setting includes lexically scoped stack regions and dynamic transferable regions (both programmer-managed) in context of an object-oriented programming language equipped with higher-order functions. Second, we adopt a two-pronged approach to memory safety that relies on a combination of a simple static type discipline and lightweight runtime checks. In particular, our approach requires neither restrictive static mechanisms (e.g., linear types and unique pointers) nor expensive runtime mechanisms (e.g., garbage collection and reference counting) in order to guarantee safety. Lastly, our region type system comes equipped with full type inference that completely eliminates the need to write region annotations on types to convince the type checker that the program is safe.

[16] proposes to extend Standard ML, a higher-order functional language, with lexically-scoped stack regions, and defines an elaboration from Standard ML to the region-annotated version of Standard ML. The aim of the elaboration is to introduce stack regions and do away with GC in a transparent fashion without jeopardizing memory safety. We too define an elaboration, but our focus is on introducing region annotations necessary to prove the safety of an object-oriented program that already uses (stack and dynamic) regions. Similar to their region inference algorithm, our region type inference algorithm can deal with region-polymorphic recursion by computing fixed points for recursive constraints. While their inference algorithm only ever generates equality constraints, which can be solved via unification, our type inference algorithm also generates partial order outlives constraints, which are required to capture subtle relationships between lifetimes of transferable regions and stack regions. Consequently, our constraint solving algorithm is more sophisticated, and is capable of inferring unknown outlives constraints over region arguments of polymorphic recursive functions.

Cyclone [5] equips C with programmer-managed stack regions, and a typing discipline that statically guarantees the safety of all pointer dereferences. Later proposals [7, 14] ex-

[20] This unsafe reference could have gone unnoticed during experiments in [4] because their experimental setup included only one actor.

tends Cyclone with dynamic regions. BROOM differs from Cyclone fundamentally because of its non-intrusiveness design principle, which requires its safety mechanisms to not intrude on the the programming practices of C#. BROOM programmers, for example, shouldnt be forced to abandon iterators in favor of for-loops, annotate region types, or rewrite C#'s standard libraries to use in BROOM. Non-intrusiveness is not a design consideration for Cyclone, which requires C programmers to use new language constructs and abandon some standard programming idioms in the interest of preserving safety. For instance, Cyclone programmers are required to write region types for functions; the type inference is only intraprocedural. Ensuring safety in presence of dynamic regions requires using either unique pointers or reference-counted objects. Both approaches are intrusive. For example, unique pointers constrain, or in some cases forbid, the use of the familiar iterator pattern, which requires creation of aliases to objects in a collection. Some standard library functions, for example, those that use caching, may need to be rewritten. Moreover, even with unique pointers, safety cannot be guaranteed statically; checks against `NULL` are needed at run-time to enforce safety. For ref-counted objects, Cyclone requires programmers to use special functions (`alias_refptr` and `drop_refptr`) to create and destroy aliases. Reference count is affected only by these functions. An alias going out of scope, for instance, does not decrement the ref-count. The requirement to use additional constructs to manage aliases makes reference counting more-or-less as intrusive as unique pointers.

Our work differs from Cyclone also in terms of its technical contributions. While Cyclone equips C with a range of region constructs [14], the semantics of (a significant subset of) such constructs, and the safety guarantees of the language are not formalized. In contrast, the (static and dynamic) semantics of Broom has been rigorously defined with respect to a well-understood formal system (FGJ). The safety guarantees have been formalized and proved. The core of Broom is very simple; the rules that make up static and dynamic semantics occupy less than a page each. We believe that the rigor and simplicity of Broom makes it easy to understand the the underlying ideas, and apply them in various problem settings. Similar contrast can be made of region type inference in both the languages. Cyclones type inference was only ever described as being similar to Tofte and Talpins, and its effectiveness in presence of tracked pointers is not clear. In contrast, the complete Ocaml (pseudo) code of Broom's inference algorithm, was given in the supplement and the ideas underlying type inference have been described elaborately in the paper

An ownership type system for safe region-based memory management in real-time Java has been proposed by [2]. Like us, they too assume a source language with programmer-managed memory regions, and focus on proving safety of programs written in that language. Their source language admits various kinds of regions in order to support shared-memory concurrency. However, all their regions have lexically-scoped lifetimes. In contrast, we admit regions with dynamically determined lifetimes in order to support message-passing concurrency. We borrow outlives relation from their formal development, and our type system bears some similarities to their's. However, our language also admits parametric polymorphism (generics) and higher-order functions, whose interaction is non-trivial in context of a region type system. On the other hand, our type system eschews the notions of region kinds and ownership, leading to a more succinct formalization. Furthermore, we establish a type safety result that formalizes its guarantees with respect to a well-defined operational semantics. Like Cyclone, [2]'s language is explicitly typed. Although there is some support for local type inference, region types for methods and classes need to be written explicitly. In contrast, we support full type inference that eliminates any such need.

[6] proposes a flow-sensitive region-based memory management for first-order programs that proposes to overcome some of the drawbacks of [16] by generalizing [16]'s approach to regions with dynamic lifetimes. However, dynamic regions are still not first-class values of the language, and reference counting is nonetheless needed to ensure memory safety. [18] extends lambda calculus with first-class regions with dynamic lifetimes, and imposes linear typing to control accesses to regions. Our open/close lexical block for transferable regions traces its orgins to the `let!` expression in [18] and [17], which safely relaxes linear typing restrictions, allowing variables to be temporarily aliased. We don't have linear typing, thus admit unrestricted aliasing. Moreover, [18]'s linear type system is insufficient to enforce the invariants needed to ensure safety under region transfers, such as the absence of references that escape a transferable region.

The idea of using region-based memory management to facilitate the safe transfer of rich data structures between computational nodes has been previously explored by [8] in the context of Scheme language. However, their setting only includes lexically-scoped regions for which Tofte and Talpin-style analysis [16] suffices. In contrast, our language provides first-class support for transferable regions with dynamic lifetimes. We require this generality in order to support streaming query operators, such as the one shown in Fig. 1.

References

[1] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097. URL `http://doi.acm.org/10.1145/1640089.1640097`.

[2] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 324–337, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781168. URL `http://doi.acm.org/10.1145/781131.781168`.

[3] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM Journal on Discrete Math*, 5(1):25–53, 1992.

[4] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015. URL `https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog`.

[5] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

[6] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 175–186, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X. doi: 10.1145/773184.773203. URL `http://doi.acm.org/10.1145/773184.773203`.

[7] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029883. URL `http://doi.acm.org/10.1145/1029873.1029883`.

[8] E. Holk, R. Newton, J. Siek, and A. Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 141–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660244. URL `http://doi.acm.org/10.1145/2660193.2660244`.

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL `http://doi.acm.org/10.1145/503502.503505`.

[10] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 1–1, Berkeley, CA, USA, 2015. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2831090.2831091`.

[11] Mads Tofte and J.-P. Talpin. A Theory of Stack Allocation in Polymorphically Typed Languages. Technical Report DIKU-report 93/15, University of Copenhagen, 1993. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6564`.

[12] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602. URL `http://doi.acm.org/10.1145/1375581.1375602`.

[13] Rust. The Rust Programming Language, 2015. URL `https://doc.rust-lang.org/book`. Accessed: 2015-11-7 13:21:00.

[14] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Science of Computer Programming*, 62(2):122 – 144, 2006. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2006.02.003. URL `http://www.sciencedirect.com/science/article/pii/S0167642306000785`. Special Issue: Five perspectives on modern memory management - Systems, hardware and theory.

[15] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: 10.1145/174675.177855. URL `http://doi.acm.org/10.1145/174675.177855`.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997. ISSN 0890-5401. doi: 10.1006/inco.1996.2613. URL `http://dx.doi.org/10.1006/inco.1996.2613`.

[17] P. Wadler. Linear Types Can Change the World! In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, Apr. 1990. North-Holland.

[18] D. Walker and K. Watkins. On regions and linear types (extended abstract). In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 181–192, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507658. URL `http://doi.acm.org/10.1145/507635.507658`.

[19] B. N. Yates. A type-and-effect system for encapsulating memory in java. Master's thesis, Department of Computer Science and Information Science, University of Oregon, 1999.

A. Appendix

LEMMA A.1. (**Substitution Preserves Typing**) $\forall e, z, \tau_1, \tau_2, \Sigma, \Delta, \Gamma, \Phi$, *if* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma[z \mapsto \tau_1] \vdash e : \tau_2$ *and* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma \vdash v : \tau_1$, *then* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma \vdash [v/z]e : \tau_2$

Proof Intros e. Induction on e. For every subexpression e_0, inductive hypothesis says the following:

$$\begin{array}{l} \forall(z, \tau_1, \tau_2, \Sigma, \Delta, \Gamma, \Phi). \quad (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma[z \mapsto \tau_1] \vdash e_0 : \tau_2 \quad \wedge \quad (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma \vdash v : \tau_1 \quad IH1 \\ \qquad\qquad \Rightarrow (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma \vdash [v/z]e_0 : \tau_2 \end{array}$$

In all the inductive cases, we have the following hypotheses:

$$\begin{array}{cl} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma[z \mapsto \tau_1] \vdash e : \tau_2 & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \Gamma \vdash v : \tau_1 & H4 \end{array}$$

In each case, proof strategy is the same: invert on $H2$, apply $IH1$, and then construct the proof term for the goal by applying type rules. ∎

LEMMA A.2. (**Weakening**) $\forall v, \tau, \Delta, \Delta_0, \Sigma, \Phi, \pi, \pi_0$, *such that* $\mathsf{value}(v)$, $\pi \in \Delta$, $\pi_0 \notin \Delta$, *and* $\Delta_0 \subseteq \Delta$, *if* $(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0), \pi_0, \cdot \vdash v : \tau$ *and* $(dom(\Sigma), \Delta, \cdot, \Phi) \vdash \tau\ ok$, *then* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash v : \tau$.

Proof Intros. Hypotheses:

$$\begin{array}{cl} \pi \in \Delta & H1 \\ \pi_0 \notin \Delta & H2 \\ \Delta_0 \subseteq \Delta & H4 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0), \pi_0, \cdot \vdash v : \tau & H6 \\ (dom(\Sigma), \Delta, \cdot, \Phi) \vdash \tau\ \mathtt{ok} & H8 \end{array}$$

Proof by induction on $H6$. Cases:

- Case ($v = \mathtt{new}\ N_0(\overline{v})$ and $\tau = N_0$): Inversion on $H6$:

$$\begin{array}{cl} \mathsf{fields}(N_0) = \overline{f} : \overline{\tau} & H10 \\ (dom(\Sigma), \Delta, \cdot, \Phi) \vdash N_0\ \mathtt{ok} & H11 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0), \pi_0, \cdot \vdash \overline{v} : \overline{\tau} & H12 \end{array}$$

  Inductive hypothesis on $\overline{v}$:

$$(dom(\Sigma), \Delta, \cdot, \Phi) \vdash \overline{\tau}\ \mathtt{ok} \Rightarrow (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \overline{v} : \overline{\tau} \quad IH1$$

  Inversion on $H8$ tells us that N_0 is of the form $B\langle\overline{T}\rangle\langle\pi\overline{\pi}\rangle$, where, $\mathtt{class}\ B\langle\overline{a} \triangleleft \overline{K}\rangle\langle\rho^a, \overline{\rho} \mid \phi\rangle \triangleleft N\{\overline{\tau^h\ h}, ...\}$ is a well-formed class definition. Furthermore, we get:

$$\begin{array}{cl} \pi, \overline{\pi} \in \Delta & H13 \\ \cdot \Vdash \overline{T}\ \mathtt{ok} & H14 \\ \Phi \vdash [\pi/\rho^a][\overline{\pi}/\overline{\rho}]\phi & H16 \end{array}$$

  Let $\mathsf{fields}(N) = \overline{g} : \overline{\tau^g}$. Inverting the well-formedness of class B, we get the following:

$$(\emptyset, \{\rho^a, \overline{\rho}\}, [\overline{a} \mapsto \overline{K}], \phi) \vdash \overline{\tau^h}\overline{\tau^g}\ \mathtt{ok} \quad H18$$

  Since $\cdot \Vdash B\langle\overline{T}\rangle\ \mathtt{ok}$, $H18$ gives:

$$(\emptyset, \{\rho^a, \overline{\rho}\}, \cdot, \phi) \vdash [\overline{T}/\overline{a}](\overline{\tau^h}\overline{\tau^g})\ \mathtt{ok} \quad H20$$

  And, since $\pi \neq \rho^a$ and $\{\overline{\pi}\} \cap \{\overline{\rho}\} = \emptyset$:

$$(\emptyset, \{\pi, \overline{\pi}\}, \cdot, [\overline{\pi}/\overline{\rho}][\pi/\rho^a]\phi) \vdash [\overline{\pi}/\overline{\rho}][\pi/\rho^a][\overline{T}/\overline{a}](\overline{\tau^h}\overline{\tau^g})\ \mathtt{ok} \quad H22$$

  From the definition of fields, we have:

$$\mathsf{fields}(B\langle\overline{T}\rangle\langle\pi\overline{\pi}\rangle) = \overline{h} : [\overline{\pi}/\overline{\rho}][\pi/\rho^a][\overline{T}/\overline{a}]\overline{\tau^h},\ \overline{g} : [\overline{\pi}/\overline{\rho}][\pi/\rho^a][\overline{T}/\overline{a}]\overline{\tau^g} \quad H24$$

  From $H10$ and $H24$, we know that $\overline{\tau} = [\overline{\pi}/\overline{\rho}][\pi/\rho^a][\overline{T}/\overline{a}](\overline{\tau^h\tau^g})$. Substituting this in H22:

$$(\emptyset, \{\pi, \overline{\pi}\}, \cdot, [\overline{\pi}/\overline{\rho}][\pi/\rho^a]\phi) \vdash \overline{\tau}\ \mathtt{ok} \quad H26$$

Using $H13$ and $H16$, from $H26$, we derive:

$$(dom(\Sigma), \Delta, \cdot, \Phi) \vdash \overline{\tau} \; \texttt{ok} \quad H28$$

From $H28$ and $IH1$, we get:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \overline{v} : \overline{\tau} \quad H30$$

From $H10$, $H11$, and $H30$, we prove the required goal:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new } N_0(\overline{v}) : N_0$$

- Case ($v = \texttt{new Region}\langle T\rangle\langle\pi_i\rangle(v_0)$ and $\tau = \texttt{Region}\langle T\rangle\langle\pi_\top\rangle$): Inversion on $H6$:

$$\begin{array}{cl} \pi_i \in dom(\Sigma) & H10 \\ \pi_i \notin \Delta \cup \{\pi_0, \pi_\top\} & H12 \\ (dom(\Sigma), \Delta \cup \{\pi_i, \pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0) \vdash T@\pi_i \; \texttt{ok} & H14 \\ (dom(\Sigma), \Delta \cup \{\pi_i, \pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0), \pi, \cdot \vdash v_0 : T@\pi_i & H16 \end{array}$$

Inductive hypothesis on v_0:

$$(dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi) \vdash T@\pi_i \; \texttt{ok} \;\Rightarrow\; (dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi, \cdot \vdash v_0 : T@\pi_i \quad IH1$$

Inverting $H14$, we get:

$$\begin{array}{cl} \cdot \Vdash T <: \texttt{Object} & H18 \\ \cdot \Vdash T \; \texttt{ok} & H19 \\ \pi_i \in \Delta \cup \{\pi_i, \pi_0\} & H20 \end{array}$$

$H20$ implies $\pi_i \in \Delta \cup \{\pi_i\}$. Using this, $H18$, and $H19$, and applying the type well-formedness rule for $T@\pi_i$, we derive the following:

$$(dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi) \vdash T@\pi_i \; \texttt{ok} \quad H22$$

$H22$ and $IH1$ gives:

$$(dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi, \cdot \vdash v_0 : T@\pi_i \quad H24$$

$H12$ implies $\pi_i \notin \Delta \cup \{\pi_\top\}$. Using this, and $H10$, $H22$ and $H24$, we conclude:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{Region}\langle T\rangle\langle\pi_i\rangle : \texttt{Region}\langle T\rangle\langle\pi_\top\rangle$$

- Case ($v = \lambda@\pi^a\langle\rho^a\overline{\rho} \,|\, \phi\rangle(\overline{\tau^1 \; x}).e$ and $\tau = \langle\rho^a\overline{\rho} \,|\, \phi\rangle\overline{\tau} \xrightarrow{\pi^a} \tau^2$): By inversion on $H6$:

$$\begin{array}{cl} \pi^a = \pi_0 & H10 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta_0 \succeq \pi_0), \pi_0, \cdot \vdash \lambda@\pi_0\langle\rho^a\overline{\rho} \,|\, \phi\rangle(\overline{\tau^1 \; x}).e : \langle\rho^a\overline{\rho} \,|\, \phi\rangle\overline{\tau} \xrightarrow{\pi_0} \tau^2 & H12 \end{array}$$

From $H8$:

$$(dom(\Sigma), \Delta, \cdot, \Phi) \vdash \langle\rho^a\overline{\rho} \,|\, \phi\rangle\overline{\tau} \xrightarrow{\pi_0} \tau^2 \; \texttt{ok} \quad H14$$

Inverting $H14$ tells us that $\pi_0 \in \Delta$. But $H2$ tells us that $\pi_0 \notin \Delta$. This is a contradiction, telling us that it cannot be the case that the type of lambda is well-formed under $(dom(\Sigma), \Delta, \cdot, \Phi)$. Intuitively, this means that a `letregion` expression or an `open` expression can never return a closure, thereby ensure that a function application never dereferences an invalid reference.

LEMMA A.3. (**Safe Region Renaming**) *$\forall e, \Sigma, \tau, \Delta, \Phi, \rho, \pi$, such that $\Delta \vdash \Phi$ ok, $\rho \notin \Delta \cup \Sigma$ and* $\mathsf{fresh}(\pi)$ *(i.e., $\pi \notin \Delta \cup \Sigma \cup \mathsf{RVars}(e) \cup \mathsf{RVars}(CT)$), if $(\Sigma, \Delta \cup \{\rho\}, \cdot, \Phi), \rho, \cdot \vdash e : \tau$, then $(\Sigma, \Delta \cup \{\pi\}, \cdot, [\pi/\rho]\Phi), \pi, \cdot \vdash [\pi/\rho]e : [\pi/\rho]\tau$*

Proof By induction on e. Proof in each case follows directly from the following facts:

- The inductive hypothesis, which says that for every sub-expression e_0 of e, the following holds:

$$\begin{array}{cl} \forall\Sigma, \tau, \Delta, \Phi, \rho, \pi, \; \textit{such that } \Delta \vdash \Phi \; \texttt{ok}, \rho \notin \Delta \cup \Sigma \; \textit{and } \mathsf{fresh}(\pi), \; \textit{if } (\Sigma, \Delta \cup \{\rho\}, \cdot, \Phi), \rho, \cdot \vdash e : \tau, \; \textit{then} & IH1 \\ (\Sigma, \Delta \cup \{\pi\}, \cdot, [\pi/\rho]\Phi), \pi, \cdot \vdash [\pi/\rho]e : [\pi/\rho]\tau & \end{array}$$

- If $\rho \notin \mathsf{RVars}(e)$ and e_0 is a subexpression of e, then $\rho \notin \mathsf{RVars}(e_0)$
- If $\pi \notin \mathsf{frv}(\phi_1 \wedge \phi_2)$, then $\phi_1 \vdash \phi_2$ implies $[\pi/\rho]\phi_1 \vdash [\pi/\rho]\phi_2$.

THEOREM A.4. (**Progress**) *$\forall e, \tau, \Delta, \Sigma, \Phi, \pi$, if $\pi \in \Delta$ and $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$, then one of the following holds:*

$$\begin{array}{rl} (i) & \exists (e', \Sigma').\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') \\ (ii) & \mathsf{value}(e) \\ (iii) & \Delta \vdash (e, \Sigma) \longrightarrow \bot \end{array}$$

Proof Intros e. Induction on e. For every subexpressions e_0, inductive hypothesis gives us the following:

$$\begin{array}{ll} \forall(\tau_0, \Delta_0, \Sigma_0, \Phi_0, \pi_0).\pi_0 \in \Delta_0 \wedge (dom(\Sigma_0), \Delta_0, \cdot, \phi_0), \pi_0, \cdot \vdash e_0 : \tau_0 \Rightarrow & IH1 \\ \quad (\exists(e_0', \Sigma_0').\Delta \vdash (e_0, \Sigma_0) \longrightarrow (e_0', \Sigma_0')) \vee (\mathsf{value}(e_0)) \vee (\Delta \vdash (e_0, \Sigma_0) \longrightarrow \bot) & \end{array}$$

Cases from the induction:

- Cases ($e = ()$ and $e = x$): proof trivial.
- Case ($e = e_0.f_i$): Intros. Hypothesis:

$$\begin{array}{cc} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H5 \end{array}$$

  Inverting $H5$:

$$\begin{array}{cc} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0 : \tau' & H7 \\ \overline{f} : \overline{\tau} = \mathsf{fields}(\mathsf{bound}.(\tau')) & H9 \end{array}$$

  Applying $H7$ in $IH1$, we have three cases:
  - SCase (e_0 takes a step): Hypotheses:

$$\Delta \vdash (e_0, \Sigma) \longrightarrow (e_0', \Sigma_0') \quad H11$$

    Therefore $(e_0.f_i, \Sigma)$ takes a step to $(e_0'.f_i, \Sigma_0')$ under Δ.
  - SCase (e_0 is a value): Since e_0 has type τ' and $\mathsf{bound}.$ is defined for τ', it follows that e_0 is $\texttt{new}\, N(\overline{v})$. From $H7$:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new}\, N(\overline{v}) : \tau' \quad H14$$

    Inverting $H14$:

$$(dom(\Sigma), \Delta, \cdot, \Phi) \vdash N\ \texttt{ok} \quad H16$$

    Inverting $H16$:

$$\mathsf{allocRgn}(N) \in \Delta \quad H18$$

    From $H9$, $H18$, we know that $(e_0.f_i, \Sigma)$ takes a step to (v_i, Σ)
  - SCase (e_0 raises $\bot$): $e_0.f_i$ also raises $\bot$.
- Case ($e = \texttt{letregion}\ \pi_0\ \texttt{in}\ e_0$): Intros. Hypothesis:

$$\begin{array}{cc} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H5 \end{array}$$

  Inverting $H5$:

$$\begin{array}{cc} \pi_0 \notin \Delta & H7 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta \succeq \pi_0), \pi_0, \cdot \vdash e_0 : \tau & H9 \end{array}$$

  From $H9$ and $IH1$, we have three cases:
  - SCase (e_0 takes a step). Hypotheses:

$$\Delta \cup \{\pi_0\} \vdash (e_0, \Sigma) \longrightarrow (e_0', \Sigma_0') \quad H11$$

    From $H7$ and $H11$, $\Delta \vdash (e, \Sigma) \longrightarrow (\texttt{letregion}\ \pi_0\ \texttt{in}\ e_0', \Sigma_0')$.
  - SCase (e_0 is a value v_0): From $H7$, $\Delta \vdash (e, \Sigma) \longrightarrow (v_0, \Sigma)$
  - SCase (e_0 raises $\bot$): e raises $\bot$ too.

- Case ($e = \texttt{open}\ e_a\ \texttt{as}\ y@\pi_0\ \texttt{in}\ e_b$): Intros. Hypotheses:

$$\begin{array}{rl} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H5 \end{array}$$

Inverting $H5$:

$$\begin{array}{rl} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_a : \texttt{Region}\langle T\rangle\langle\pi_\top\rangle & H7 \\ \pi_0 \notin \Delta & H9 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, [y \mapsto T@\pi_0] \vdash e_b : \tau & H11 \end{array}$$

We have three cases. First case deals with (e_a, Σ) taking a step to (e'_a, Σ'_a) under Δ. Under this context, (e, Σ) takes a step to $(\texttt{open}\ e'_a\ \texttt{as}\ y@\pi_0\ \texttt{in}\ e_b, \Sigma'_a)$. So, there is progress. Second case deals with e_a raising $\bot$. In this case, execution of e also raises $\bot$. So, we again have progress. Third case deals with e_a being a value $\texttt{new}\ N(\overline{v})$. Inverting $H7$, we have to consider two possible derivations: one from the generic type rule for values of any type, and another from the type rule tailor-made for $\texttt{Region}$ values. The first rule does not apply because fields($\texttt{Region}\langle T\rangle\langle\pi_\top\rangle$) is undefined. The only rule that applies is the special type rule for $\texttt{Region}$ values. Hence, e_a is $\texttt{new Region}\langle T\rangle\langle\pi_i\rangle(v)$, where:

$$\begin{array}{rl} \pi_i \notin \Delta & H12 \\ \pi_i \in dom(\Sigma) & H13 \\ (dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi_i, \cdot \vdash v : T@\pi_i & H16 \end{array}$$

From $H9$, $H13$, $H16$, and Lemma A.3:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [\pi_0/\pi_i]v : T@\pi_0 \quad H18$$

From $H11$, $H18$ and Lemma A.1:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [[\pi_0/\rho^a]v/y]e_b : \tau \quad H19$$

From $H12$ and $H13$, we know that $\pi_i \in dom(\Sigma)$. We now have three cases:
  - SCase ($\Sigma(\pi_i) \neq \texttt{X}$ and e_b is not a value): By inductive hypothesis, $([[\pi_0/\rho^a]v/y]e_b, \Sigma[\rho \mapsto \texttt{0}]$ can either (a). take a step to (e'_b, Σ') under $\Delta \cup \{\pi_0\}$, or (b). e_b evaluates to $\bot$. In the first case, (e, Σ) itself evaluates to:

$$(\texttt{open}\ (\texttt{new Region}\langle T\rangle\langle\rho\rangle(\lambda@\pi\langle\rho^a\rangle().v))\ \texttt{as}\ e'_b@y\ \texttt{in}\ , \Sigma'[\rho \mapsto \Sigma(\rho)])$$

    In the second case, the evaluation of e also raises $\bot$.
  - SCase($\Sigma(\pi_i) \neq \texttt{X}$ and e_b is a value v_b): Trivially, $\Delta \vdash (e, \Sigma) \longrightarrow (v_b, \Sigma)$.
  - SCase($\Sigma(\pi_i) = \texttt{X}$): e raises $\bot$.
- (e = $e_a.m\langle\pi^a, \overline{\pi}\rangle(\overline{e})$): Intros. Hypotheses:

$$\begin{array}{rl} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H4 \end{array}$$

By inversion on $H4$:

$$\begin{array}{rl} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_a : \tau_a & H6 \\ \pi^a = \pi & H7 \\ \overline{\pi} \in \Delta & H8 \\ \mathsf{mtype}(m, \mathsf{bound}.(\tau_a)) = \langle\rho^a\overline{\rho} \mid \phi\rangle\overline{\tau^1} \to \tau^2 & H10 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \overline{e} : [\overline{\pi}/\overline{\rho}][\pi/\rho^a]\overline{\tau^1} & H11 \\ \Phi \vdash [\overline{\pi}/\overline{\rho}][\pi/\rho^a]\phi & \end{array}$$

Three cases:
  - SCase (e_a isn't a value): From $H16$ and IH, we know that either (a). e_a can take a step, or (b) e_a reduces to $\bot$. In first case, e can also take a step, and in second case, e also reduces to $\bot$.
  - SCase ($e_a = v_a$, but $\exists i$ such that e_i isn't a value): From $H11$, we know that (e_i, Σ) can either (a). take a step to (e'_i, Σ') under Δ, or (b). e_i reduces to $\bot$. In the first case, we have $\Delta \vdash (v_a.m\langle\pi^a, \overline{\pi}\rangle(..., e_i, ...), \Sigma) \longrightarrow (v_a.m\langle\pi^a, \overline{\pi}\rangle(..., e'_i, ...), \Sigma')$.

- SCase ($e_a = v_a$ and $\forall i.\, e_i = v_i$): $H10$ says that bound for τ_a is defined under empty Θ. This is possible only if $\tau_a = N$ and $v_a = \texttt{new}\ N(\overline{v'})$. Furthemore, N cannot be of form $\texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle$ because, mtype isn't defined for Region. Using these facts, and inverting $H6$, we get $(dom(\Sigma), \Delta, \cdot, \Phi) \vdash N\ \texttt{ok}$. Inverting it again:

$$\mathsf{allocRgn}(N) \in \Delta \quad H13$$

Now, since $\mathsf{mtype}(m, N)$ is defined if and only if $\mathsf{mbody}(m, N)$ is defined, we know that:

$$\mathsf{mbody}(m, N) = \overline{x}.\, e_m \quad H14$$

From $H13$ and $H14$, we know that $\Delta \vdash ((v_a.m\langle\pi^a, \overline{\pi}\rangle(\overline{v}), \Sigma) \longrightarrow ([\overline{v}/\overline{x}][\texttt{new}\ N(\overline{v'})/\,\texttt{this}\,]e_m, \Sigma)$

- Case ($e = e_a\langle\pi^a\overline{\pi}\rangle(\overline{e})$). Proof closely follows the proof for $e_a.m\langle\pi^a\overline{\pi}\rangle(\overline{e})$. The only difference is that when e_a evaluates to a lambda $\lambda@\pi_0\langle\rho^a\overline{\rho}\,|\,\phi\rangle(\overline{\tau}\ \overline{x}).e_b$, we need a proof that $\pi_0 \in \Delta$. This can be obtained by inverting the type judgment for the lambda.
- Case($e = \texttt{new}\ N(\overline{e})$): Intros. Hypotheses:

$$\begin{array}{cl} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new}\ N(\overline{e}) : \tau & H4 \end{array}$$

Inverting $H4$ leads to two cases:

- SCase ($\mathsf{shape}(N) \neq \texttt{Region}\,\langle T\rangle$): Hypotheses:

$$\begin{array}{cl} (dom(\Sigma), \Delta, \cdot, \Phi) \vdash N\ \texttt{ok} & H5 \\ \mathsf{fields}(N) = \overline{f} : \overline{\tau} & H7 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \overline{e} : \overline{\tau} & H8 \end{array}$$

Inverting $H5$:

$$\mathsf{allocRgn}(N) \in \Delta \quad H10$$

Three cases:

- SSCase ($\exists i$ such that e_i takes a step): In this case, e also takes a step.
- SSCase ($\exists i$ such that e_i reduces to $\bot$). In this case, e also reduces to $\bot$.
- SSCase ($\forall i\ e_i$ is a value v_i): In this case, e is also a value $\texttt{new}\ N(\overline{v})$.

- SCase ($\mathsf{shape}(N) = \texttt{Region}\,\langle T\rangle$ and $e = \texttt{Region}\,\langle T\rangle\langle\pi_i\rangle(e_0)$): From $H4$:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(e_0) : \tau \quad H4$$

Inversion on $H4$ gives two cases:

- SSCase($\pi_i = \pi_\top$): In this case, $e_0 = \lambda@\pi\langle\rho^a\rangle().e_1$. In this case, $\Delta \vdash (\texttt{new Region}\,\langle T\rangle\langle\pi_\top\rangle(v_0), \Sigma) \longrightarrow (\texttt{new Region}\,\langle T\rangle\langle\pi_j\rangle([\pi_j/\rho^a]e_1), \Sigma[\pi_j \mapsto \texttt{C}])$, where π_j is fresh ($\pi_j \notin \Delta \cup dom(\Sigma)$).
- SSCase ($\pi_i \neq \pi_\top$): Hypotheses:

$$\begin{array}{cl} \pi_i \in dom(\Sigma) & H16 \\ \pi_i \notin \Delta \cup \{\pi_\top\} & H18 \\ (dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi_i, \cdot \vdash e_0 : T@\pi_i & H20 \end{array}$$

From $IH1$ and $H20$, we have three cases:

- SSSCase (e_0 is a value v_0): In this case, $e = \texttt{Region}\,\langle T\rangle\langle\pi_i\rangle(v_0)$ is also a value.
- SSSCase (e_0 raises $\bot$): In this case, e also raises $\bot$.
- SSSCase ($\Delta\cup\{\pi_i\} \vdash (e_0, \Sigma) \longrightarrow (e_0', \Sigma')$): In this case, $\Delta \vdash (\texttt{Region}\,\langle T\rangle\langle\pi_i\rangle(e_0), \Sigma) \longrightarrow (\texttt{Region}\,\langle T\rangle\langle\pi_i\rangle(e_0'), \Sigma')$

- Case ($e = e_0.\mathsf{transfer}\langle\pi^a\rangle()$): Intros. Hypotheses:

$$\begin{array}{cl} \pi \in \Delta & H2 \\ \pi^a = \pi & H3 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0.\mathsf{transfer}\langle\pi^a\rangle() : \tau & H4 \end{array}$$

By inversion on $H4$:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0 : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle \quad H6$$

Now, if e_0 can take a step, so can e, hence there is progress. Else, if e_0 raises an exception, so does e. The only non-trivial case is when e_0 is a value. But, only values of type $\texttt{Region}\langle T\rangle\langle\pi_\top\rangle$ is $\texttt{new Region}\langle T\rangle\langle\pi_i\rangle(...)$, where $\pi_i \neq \pi_\top$. By inversion on $H6$, we get:

$$\pi_i \in dom(\Sigma) \quad H8$$

We have two cases:

- ($\Sigma(\pi_i) \neq \texttt{O}$): In this case, $\Delta \vdash (e, \Sigma) \longrightarrow (\,()\,, \Sigma[\pi_i \mapsto \texttt{X}])$.
- ($\Sigma(\pi_i) = \texttt{O}$): In this case, evaluation of e raises $\perp$.

- Case (e is a lambda abstraction): e is already a value.
- Case ($e = e_1;\ e_2$): Proof trivial.

THEOREM A.5. (**Preservation**) *$\forall e, \tau, \Delta, \Sigma, \Phi, \pi$, such that $\pi \in \Delta$, if $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$ and $\Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma')$, then $(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e' : \tau$.*

Proof Intros e. Induction on e. For every subexpressions e_0, inductive hypothesis gives us the following:

$$\begin{array}{c} \forall(\tau_0, \Delta_0, \Sigma_0, \Phi_0, \pi_0). \quad (\pi_0 \in \Delta_0) \wedge ((dom(\Sigma_0), \Delta_0, \cdot, \Phi_0), \pi_0, \cdot \vdash e_0 : \tau_0) \wedge (\Delta \vdash (e_0, \Sigma_0) \longrightarrow (e_0', \Sigma_0')) \quad IH1 \\ \Rightarrow (dom(\Sigma_0'), \Delta_0, \cdot, \Phi_0), \pi_0, \cdot \vdash e_0' : \tau_0 \end{array}$$

Cases from induction

- Case ($e = ()$ or $e = x$): Proof is trivial.
- Case ($e = e_0.f_i$): Intros. Hypothesis:

$$\begin{array}{cc} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H4 \\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') & H6 \end{array}$$

Inverting $H4$:

$$\begin{array}{cc} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0 : \tau' & H7 \\ \overline{f} : \overline{\tau} = \mathsf{fields}(\mathsf{bound}.(\tau')) & H9 \end{array}$$

Inverting $H6$, we get two cases:

- SCase ($\Delta \vdash (e_0, \Sigma) \longrightarrow (e_0', \Sigma')$): In this case, $\Delta \vdash (e_0.f_i, \Sigma) \longrightarrow (e_0'.f_i, \Sigma')$. $H7$ and $IH1$ gives:

$$(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0' : \tau' \quad H11$$

  Proof follows from $H11$ and $H9$.
- SCase (e_0 is a value $\texttt{new}\ N(\overline{v})$): Hypotheses:

$$\begin{array}{cc} \mathsf{allocRgn}(N) \in \Delta & H13 \\ \Delta \vdash (e, \Sigma) \longrightarrow (v_i, \Sigma) & H15 \end{array}$$

We need to prove that $((dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot) \vdash v_i : \tau_i$. From $H7$, since $e_0 = \texttt{new}\ N(\overline{v})$:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new}\ N(\overline{v}) : \tau' \quad H16$$

Inverting $H16$ and using $H9$ gives us the proof.

- Case ($e = \texttt{letregion}\ \pi_0\ \texttt{in}\ e_0$): Intros. Hypothesis:

$$\begin{array}{cc} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H4 \\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') & H6 \end{array}$$

Inverting $H4$:

$$\begin{array}{cc} \pi_0 \notin \Delta & H7 \\ (dom(\Sigma), \Delta, \cdot, \Phi) \vdash \tau\ \texttt{ok} & H8 \\ (dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta \succeq \pi_0), \pi_0, \cdot \vdash e_0 : \tau & H9 \end{array}$$

Inverting $H6$, we get two cases:

- SCase ($\Delta \cup \{\pi_0\} \vdash (e_0, \Sigma) \longrightarrow (e_0', \Sigma')$): In this case, $\Delta \vdash (\texttt{letregion}\ \pi_0\ \texttt{in}\ e_0, \Sigma) \longrightarrow (\texttt{letregion}\ \pi_0\ \texttt{in}\ e_0', \Sigma')$. $H9$ and $IH1$ gives:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \wedge \Delta \succeq \pi_0), \pi, \cdot \vdash e_0' : \tau \quad H11$$

From $H7$ and $H11$, we can conclude that $(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{letregion}\ \pi_0\ \texttt{in}\ e_0' : \tau$.

- SCase (e_0 is a value v_0): In this case, $\Delta \vdash (\texttt{letregion}\ \pi_0\ \texttt{in}\ e_0, \Sigma) \longrightarrow (v_0, \Sigma')$. From $H2$, $H7-9$, and Lemma A.2, we have:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash v_0 : \tau \quad H13$$

Thus, type is preserved.

- Case ($e = \texttt{open}\ e_a\ \texttt{as}\ y@\pi_0\ \texttt{in}\ e_b$): Intros. Hypotheses:

$$\begin{array}{cl} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & H4 \\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') & H6 \end{array}$$

Inverting $H4$:

$$\begin{array}{cl} (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_a : \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle & H7 \\ (dom(\Sigma), \Delta, \cdot, \Phi) \vdash \tau\ \texttt{ok} & H8 \\ \pi_0 \notin \Delta & H9 \\ (dom(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, [y \mapsto T@\pi_0] \vdash e_b : \tau & H11 \end{array}$$

Inverting $H6$, we get many cases:

- SCase ($\Delta \vdash (e_a, \Sigma) \longrightarrow (e_a', \Sigma')$): Since the domain of Σ monotonically increases during the evaluation, we have:

$$dom(\Sigma) \subseteq dom(\Sigma') \quad H13$$

Since strenthening the context trivially preserves typing and well-formedness, from $H7-11$ and $H13$, we have:

$$\begin{array}{cl} (dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_a : \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle & H15 \\ (dom(\Sigma'), \Delta, \cdot, \Phi) \vdash \tau\ \texttt{ok} & H17 \\ \pi_0 \notin \Delta & H19 \\ (dom(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, [y \mapsto T@\pi_0] \vdash e_b : \tau & H20 \end{array}$$

From $H15-20$, we have $(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$.

- SCase (e_a is a value v_a, and e_b steps to e_b'): Hypotheses:

$$\begin{array}{cl} v_a = \texttt{new Region}\,\langle T\rangle\langle \pi_i\rangle(v_r) & H22 \\ \pi_i \neq \pi_\top & H23 \\ \Sigma(\pi_i) \neq \texttt{X} & H24 \\ \pi_0 \notin \Delta & H26 \\ \Delta \cup \{\pi_0\} \vdash ([[\pi_0/\pi_i]v_r/y]e_b, \Sigma[\pi_i \mapsto \texttt{0}]) \longrightarrow (e_b', \Sigma') & H27 \\ \Sigma'' = \Sigma'[\pi_i \mapsto \Sigma(\pi_i)] & H29 \end{array}$$

We need to prove that $(dom(\Sigma''), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{open}\ v_a\ \texttt{as}\ y@\pi_0\ \texttt{in}\ e_b' : \tau$. Note that $dom(\Sigma'') = dom(\Sigma')$. Hence, the proof obligation is $(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{open}\ v_a\ \texttt{as}\ y@\pi_0\ \texttt{in}\ e_b' : \tau$. First, since the domain of Σ monotonically increases during the evaluation, we have:

$$dom(\Sigma) \subseteq dom(\Sigma') \quad H31$$

Next, since $e_a = v_a$, from $H7$ and $H22$, we have:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new Region}\,\langle T\rangle\langle \pi_i\rangle(v_r) : \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle \quad H33$$

Since $H23$, inversion on $H33$ gives:

$$\begin{array}{cl} (dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi_i, \cdot \vdash v_r : T@\pi_i & H34 \\ \pi_i \notin \Delta & H35 \end{array}$$

From $H23$, $H26$, $H34$, $H35$, and Lemma A.3, we have:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [\pi_0/\pi_i]v_r : T@\pi_0 \quad H36$$

From $H11$, $H36$ and Lemma A.1, we get:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [[\pi_0/\pi_i]v_r/y]e_b : \tau \quad H38$$

$H24$ says that $\pi_i \in dom(\Sigma)$. Hence, from $H38$:

$$(dom(\Sigma[[\pi_i \mapsto \mathtt{O}]]), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [[\pi_0/\pi_i]v_r/y]e_b : \tau \quad H40$$

From $H40$, $H27$ and $IH1$:

$$(dom(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash e_b' : \tau \quad H42$$

By strengthening the type context:

$$(dom(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot[y \mapsto T@\pi_0] \vdash e_b' : \tau \quad H48$$

Since $dom(\Sigma) \subseteq dom(\Sigma')$ (from $H31$), we get the following by strengthening the context in $H7-11$:

$$\begin{array}{rl} (dom(\Sigma''), \Delta, \cdot, \Phi), \pi, \cdot \vdash v_a : \mathtt{Region}\,\langle T\rangle\langle\pi_\top\rangle & H50 \\ (dom(\Sigma''), \Delta, \cdot, \Phi) \vdash \tau \;\mathtt{ok} & H51 \end{array}$$

From $H48$, $H50$, $H51$, we have the required goal:

$$(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \mathtt{open}\; v_a \;\mathtt{as}\; y@\pi_0 \;\mathtt{in}\; e_b' : \tau$$

- SCase (e_a is a value v_a, and e_b is a value v_b): In this case, $\Delta \vdash (\mathtt{open}\; v_a \;\mathtt{as}\; y@\pi_0 \;\mathtt{in}\; v_b, \Sigma) \longrightarrow (v_b, \Sigma)$. From $H11$:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, [y \mapsto T@\pi_0] \vdash v_b : \tau \quad H53$$

Since $\mathsf{value}(v_b)$, it has not free variables. Consequently:

$$(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash v_b : \tau \quad H55$$

From $H2$, $H8$, $H9$, $H55$ and Lemma A.2, we prove the required goal:

$$(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash v_b : \tau$$

- Case ($e = \mathtt{new\ Region}\,\langle T\rangle\langle\pi_\top\rangle(e_0)$): Hypotheses:

$$\begin{array}{cl} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \mathtt{new\ Region}\,\langle T\rangle\langle\pi_\top\rangle(e_0) : \tau & H4 \\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') & H6 \end{array}$$

Inverting $H4$, we know that $e_0 = \lambda@\pi\langle\rho^a\rangle().e_1$, and:

$$\begin{array}{cl} (\emptyset, \{\rho^a\}, \cdot, true), \rho^a, \cdot \vdash e_1 : T@\rho^a & H7 \\ \cdot \Vdash T \;\mathtt{ok} & H8 \end{array}$$

From $H7$ and Lemma A.3:

$$(\emptyset, \{\pi_i\}, \cdot, true), \pi_i, \cdot \vdash [\pi_i/\rho^a]e_1 : T@\pi_i \quad H10$$

Inverting $H6$:

$$\begin{array}{cl} \pi_i \notin dom(\Sigma) \cup \Delta & H17 \\ \Sigma' = \Sigma[\pi_i \mapsto \mathtt{C}] & H19 \\ \Delta \vdash (\mathtt{new\ Region}\,\langle T\rangle\langle\pi_\top\rangle(\lambda@\pi\langle\rho^a\rangle().e_1)) \longrightarrow (\mathtt{new\ Region}\,\langle T\rangle\langle\pi_i\rangle([\pi_i/\rho^a]e_1), \Sigma') & H21 \end{array}$$

Since strengthening the context preserves typing, strengthening the context for type judgment in $H10$ gives us the following:

$$(dom(\Sigma'), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi_i, \cdot \vdash [\pi_i/\rho^a]e : T@\pi_i \quad H24$$

$H19$ implies $\pi_i \in dom(\Sigma')$. This, and $H17$, $H8$, and $H24$ entail the required goal:

$$(dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \mathtt{new\ Region}\,\langle T\rangle\langle\pi_i\rangle([\pi_i/\rho^a]e_1) : \mathtt{Region}\,\langle T\rangle\langle\pi_\top\rangle$$

- Case ($e =$ `new Region` $\langle T\rangle\langle\pi_i\rangle(e_0)$, where $\pi_i \neq \pi_\top$):Hypotheses:

$$\begin{array}{cr} \pi \in \Delta & H2 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(e_0) : \tau & H4 \\ \Delta \vdash (\texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(e_0), \Sigma) \longrightarrow (e', \Sigma') & H6 \end{array}$$

  Since e isn't a value, inverting $H6$ tells us that `new Region` $\langle T\rangle\langle\pi_i\rangle(e_0)$ takes a step to `new Region` $\langle T\rangle\langle\pi_i\rangle(e_0')$ when e_0 takes a step to e_0'. The proof for this case is similar to the previous case; we invert $H4$ and $H6$, apply inductive hypothesis to derive typing judgment for e_0 under a context containing π_i, and finally apply the type rule for `new Region` $\langle T\rangle\langle\pi_i\rangle(e_0')$ (where $\pi_i \neq \pi_\top$) to prove the preservation.
- Case (e is a lambda expression): e is a value, hence cannot take a step. Preservation trivially holds.
- Case (e is a method/function call, or a `let` expression): Proof follows directly from the inductive hypothesis, substitution lemma (A.1) and renaming lemma (A.3).

THEOREM A.6. (**Type Safety**) $\forall e, \tau, \Delta, \Sigma, \pi$, *if* $\pi \in \Delta$ *and* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau$*, then either* e *is a value, or one of the following holds:*

$$\begin{array}{rl} (i) & \exists(e', \Sigma').\ \textit{such that}\ \Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma')\ \textit{and}\ (dom(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau \\ (ii) & \Delta \vdash (e, \Sigma) \longrightarrow \bot \end{array}$$

Proof Directly follows from Theorems A.4 and A.5.

THEOREM A.7. (**Transfer Safety**) $\forall\ v, \Delta, \Delta', \Sigma, \Sigma', \Phi, \Phi', \pi, \pi'$*, such that* $\pi \in \Delta$, $\pi' \in \Delta'$*, and* $\pi_i \notin dom(\Sigma') \cup \{\pi_\top\}$*, if* $(dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash$ `new Region` $\langle T\rangle\langle\pi_i\rangle(v)$: `Region` $\langle T\rangle\langle\pi_\top\rangle$*, then* $(\Sigma'[\pi_i \mapsto \texttt{C}], \Delta', \cdot, \Phi') \vdash$ `new Region` $\langle T\rangle\langle\pi_i\rangle(v)$: `Region` $\langle T\rangle\langle\pi_\top\rangle$

Proof Intros. Hypothesis:

$$\begin{array}{cr} \pi_i \notin dom(\Sigma') \cup \Delta' \cup \{\pi_\top\} & H6 \\ (dom(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(v) : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle & H8 \end{array}$$

By inversion on H8, we observe that v is well-typed under an empty environment containing nothing but π_i. Hence, $\mathsf{frv}(v) \in \{\pi_i\}$. Since v is a value, it means that v preserves its type under any context that contains a binding for π_i in Σ. Since $\pi_i \notin dom(\Sigma') \cup \Delta'$, it means that v preserves its type under a context with $\Sigma[\pi_i \mapsto \texttt{C}]$. Applying the type rule for `new Region` $\langle T\rangle\langle\pi_i\rangle(v)$ expression, where $\pi_i \neq \pi_\top$, we get the proof. ∎