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.

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 ex-
changed 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., $\text{map}[t]$) can be a separate region that is allocated when the first message with timestamp $t$ arrives, and deallocated after $\text{map}[t]$ 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 [1] 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 run-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, 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.

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, 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:

```java
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 evaluated. Nonetheless, the propositions (a), R0 ⪰ 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 open 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.

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. 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 o1 in a region R1 is allowed to store a pointer to an object o2 in a region R2 only if R2 is guaranteed to outlive R1. (A similar restriction applies in the case where o1 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

```plaintext
1 class SelectVertex<TIn, TOut> {
  Func<TIn, TOut> selector;
  Dictionary<Time, Region<List<TOut>>> map;
  ...
  void onReceive(Time t, Region<List<TIn>> inRgn) {
    if (!map.ContainsKey(t)) {
      map[t] = new Region<List<TOut>>();
    }
    Region<List<TOut>> outRgn = map[t];
    outRgn.transfer();
  }
  void onNotify(Time t) {
    Region<List<TOut>> outRgn = map[t];
    map.Remove(t);
    outRgn.transfer();
  }
}
```

Figure 2: SELECT dataflow operator in BROOM

---

4 We omit @R annotation in open when we don’t need R.
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:

```java
class Pair<a Object, b Object> {<\( \mu \), \( \rho_1, \rho_2 \)| \( \rho_1 \geq \mu \land \rho_2 \geq \mu \)>({\( \rho \), Object}) { a@\( \rho_1 \) fst; b@\( \rho_2 \) snd;} Pair(a@\( \rho_1 \) fst, b@\( \rho_2 \) snd) { super(); this.fst = fst; this.snd = snd; } a@\( \rho_1 \) 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 \( \rho \). We refer to the first region parameter, usually denoted \( \rho^0 \), 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 (7) other than its own allocation region (\( \rho^0 \)), provided that the former outlive the later (i.e., \( \rho \geq \rho^0 \)). 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

---

The symbol \( \triangleleft \) should be read extends, and the symbol \( \triangleright \) stands for outlives.
\( \pi \in \text{Static region ids} \quad \rho \in \text{Region variables} \quad a, b \in \text{Type variables} \quad m \in \text{Method names} \quad x, y, f \in \text{Variables and fields} \)

cn \in \text{Class names} \quad ::= \text{Object} \mid \text{Region} \mid A \mid B

\( K \in \text{FGJ class types} \quad ::= \text{cn}(T) \)

\( T \in \text{FGJ types} \quad ::= a \mid K \mid \text{unit} \mid T \to T \)

\( N \in \text{Region - annotated class types} \quad ::= \text{cn}(\langle \pi^n \rangle) \)

\( \tau \in \text{types} \quad ::= T@\pi \mid N \mid \text{unit} \mid \langle\rho^p\rho\phi\rangle \to \tau \)

\( C \in \text{Class definitions} \quad ::= \text{class} \text{cn}(\langle T \rangle)(\rho^p\rho\phi) )\to N\langle T \rangle; k \overline{d} \}

\( k \in \text{Constructors} \quad ::= \text{cn}(\langle T \rangle)\{ \text{super} (\overline{T}); \text{this} . \overline{T} = \overline{T} \} \}

\( d \in \text{Methods} \quad ::= \tau m(\rho^p\rho\phi)(T)(\text{return} e; \}

\( \phi, \Phi \in \text{Region constraints} \quad ::= \text{true} \mid \rho \geq \rho \mid \rho = \rho \mid \phi \land \phi \)

\( e \in \text{Expressions} \quad ::= (\mid x \mid e.f \mid e.m(\pi^n\pi)(\overline{e}) \mid \text{new} \ N\langle T \rangle ) \mid \lambda \pi^n(\rho^p\rho\phi)(\overline{T}: \overline{e}).e \mid e(\pi^n\pi)(\overline{e}) \mid \text{let} x = e \in e \mid \text{letregion} \rho \in e \mid \text{open} x \text{ as } y@\pi \text{ in } e \)

Figure 4: \textsc{Featherweight Broom}: Syntax

\[ \text{allocRgn}(A(\pi^n\pi)(\overline{T})) = \pi^n \quad \text{bound}_a(a@\pi) = \Theta(a)@\pi \quad \text{ctype(}\text{Object}(\pi)) = \bullet \]

\[ \text{allocRgn}(\langle\rho^p\rho\phi\rangle\pi^n \to \tau^2) = \pi^n \quad \text{bound}_a(N) = N \quad \text{ctype}(B(T)(\pi^n\pi)) = \text{fields}(B(T)(\pi^n\pi)) \]

\[ \text{shape}(A(\rho^p\rho\phi)(\overline{T})) = A(\overline{T}) \quad \text{fields}(\text{Object}(\pi)) = \text{fields}(B(T)(\pi^n\pi)) \]

\[ \text{CT}(B) = \text{class} \ B(\pi \in \pi\langle \rho^p\rho\phi \rangle \in N\langle \overline{T} f \rangle; \ldots) \quad \text{fields}(S(N)) = \varphi : \overline{T} f \]

\[ \text{CT}(B) = \text{class} \ B(\pi \in \pi\langle \rho^p\rho\phi \rangle \in N\langle \overline{T} f \rangle; k \overline{d}) \quad \text{mtype}(m, B(T)(\pi^n\pi)) = \text{mtype}(m, S(N)) \]

\[ \text{CT}(B) = \text{class} \ B(\pi \in \pi\langle \rho^p\rho\phi \rangle \in N\langle \overline{T} f \rangle; k \overline{d}) \quad \text{mtype}(m, N) = (\rho^p\rho\phi)(\overline{\varphi} \overline{T} f) \quad \tau^{12} \]

\[ \text{override}(A, N, (\rho^p\rho\phi)(\overline{\varphi} \overline{T} f) \quad \tau^{22}) \]

Figure 5: \textsc{Featherweight Broom}: Auxiliary Definitions

\( (T) \) and concrete region identifiers\(^6\) (\( \pi \)), respectively. For example:

\begin{verbatim}
let region \( \pi_0 \) in
  let \( s nd = \text{new} \ \text{Object}<\pi_0>() \) in
  let region \( \pi_1 \) in
    let \( s nd = \text{new} \ \text{Object}<\pi_1>() \) in
    let p = \text{new} \ \text{Pair}<\text{Object}, \text{Object}>/<\pi_1, \pi_0, \pi_1> \;

In the above code, the instantiation of \( \rho^a \) and \( \rho_1 \) with \( \pi_0 \) and \( \rho_2 \) with \( \pi_1 \) is allowed because (a) \( \pi_0 \) and \( \pi_1 \) are live during the instantiation, and (b) \( \pi_0 \geq \pi_1 \) and \( \pi_1 \geq \pi_1 \) (since our lives are reflexive). Observe that the region type of \( p \) conveys the fact that (a), it is allocated in region \( \pi_1 \), and (b), it holds references to objects allocated in region \( \pi_0 \) and \( \pi_1 \). In contrast, if we choose to allocate the \( \text{snd} \) object also in \( \pi_1 \), then \( p \) would be contained in \( \pi_1 \), and its region type would be \( \text{Pair}<\text{Object}, \text{Object}>/<\pi_1, \pi_1, \pi_1> \), which we abbreviate as \( \text{Pair}<\text{Object}, \text{Object}>@\pi_1 \). In general,
\end{verbatim}

we treat \( B(T)@\pi \) as being equivalent to \( B(T)(\overline{\pi}) \). Region annotation on type \( a \), where \( a \) is a type variable, assumes the form \( a@\pi \). If \( a \) is instantiated with \( \text{Pair}<\text{Object}, \text{Object}> \), the result is the type of \( a \) \text{Pair} object contained in \( \pi \). The type \text{unit} is unboxed in \( \text{FB} \), hence it has no region annotations.

Like classes, methods can also exhibit constrained region polymorphism. A method definition in \( \text{FB} \) 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 (\( \phi \)). 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 \text{this} argument), then the required monomorphism can be captured as an equality constraint in \( \phi \).

\( \text{FB} \) extends FGJ’s expression language with a lambda expression and an application expression \( (e(\pi^n\pi)(\overline{e})) \) to define
and apply functions\footnote{We distinguish between functions and methods. The former result from lambda expressions, whereas the latter come from class definitions}. Functions, like methods, exhibit constrained region polymorphism, as evident in their arrow region type $\langle \rho^0, | \phi \rangle : \tau \triangleleft \tau$. Note that any of the $\tau$’s in the arrow type can themselves be region-parametric arrow types. In this respect, our region type system is comparable to System P’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 $\pi$ 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 ($\rho^0$) of a function and the allocation region ($\pi$) 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:

```java
let $f = \lambda(\rho^0). \text{new} \text{Object}(\rho^0)()$

in $f$
```

The type of $f$ is $\langle \rho^0 \rangle \text{unit} \vdash \text{Object}(\rho^0)$, covering that (a), $f$’s closure is allocated in $\pi$, and (b), when executed under an allocation context $\rho^0$, the closure returns an object allocated in $\rho^0$. 

Figure 6: FEATHERWEIGHT BROOM: Static Semantics
Method Well-formedness \[ d \text{ ok in } B \]

\[
\Delta = \{ \rho^n, \overline{p}, \rho^n_m, \overline{p}_m \} \quad \Theta = [\overline{\pi} \mapsto K] \quad \Phi = \phi \land \phi_m \quad \pi = (\Sigma, \Delta, \Theta, \Phi) \quad \Delta \vdash \phi_m \text{ ok}
\]

\[
A \vdash \pi, \tau^2 \text{ ok} \quad \text{class } B(\pi \circ K)(\rho^n, \overline{p}) \circ N[\ldots] \quad A, \rho^n_m, \Gamma \vdash e : \tau \quad A \vdash \tau < \tau^2
\]

 overridden\( (m, N) \{ \text{return } e; \} \text{ ok in } B \)

Class Well-formedness \[ B \text{ ok} \]

\[
\Delta = \{ \rho^n, \overline{p} \} \quad \Theta = [\overline{\pi} \mapsto K] \quad \Phi = \phi \quad \pi = (\Sigma, \Delta, \Theta, \Phi) \quad \Delta \vdash \phi \text{ ok}
\]

\[
\Theta \land K \text{ ok} \quad A \vdash N, \tau^T \text{ ok} \quad \text{shape}(N) \neq \text{Region}(T) \quad \Phi \vdash \text{allocRgn}(\tau^T) \geq \rho^n
\]

\[
\text{ctype}(N) = \tau^N \quad k = B(\pi, \tau^T \pi, \tau^N \overline{\pi})\{ \text{super}(\pi); \text{this } f = \overline{\pi} \} \quad \overline{d} \text{ ok in } B
\]

\[ \text{class } B(\pi \circ K)(\rho^n, \overline{p}) \circ N[\tau^T \pi, k \overline{d}] \text{ ok} \]

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 \( (A) \), 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 \( (\Phi) \) that captures the outlives constraints on regions in \( \Delta \).

\( A \) also contains \( \Sigma \), 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 \( (\pi^n) \) of the allocation context for the expression 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 \( (A \vdash \tau \text{ ok}) \) makes use of the well-formedness and subtyping rules on core types \(^9\). The class table \( (\llbracket CT \rrbracket) \) for such judgments is derived from FB’s class table \( (CT) \) by erasing all region annotations on types, and region arguments in expressions \( (\llbracket \cdot \rrbracket) \)

\(^8\) A bound of a type variable \( (\alpha) \) 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 \( e \)”

\( \text{denotes the region erasure operation.} \) The well-formedness rule for class types \( (B(T)(\pi^n\pi)) \) is responsible for enforcing the safety property that prevents objects from containing unsafe references. It does so by insisting that regions \( \pi^n\pi \) satisfy the constraints \( (\Phi) \) imposed by the class on its region parameters. The later is enforced by checking the validity of \( \phi \), with actual region arguments substituted\(^{10}\) for formal region parameters, under the conditions \( (\Delta, \Phi) \) 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 \( T, T' \circ \pi \) is a well-formed region type if \( \pi \) is a valid region. The type \( \text{Region}(T)(\pi) \) is well-formed only if \( \pi = \pi_T \), where \( \pi_T \) is a special immortal region that outlives every other live region. This arrangement allows \( \text{Region} \) 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 \( \text{Region} \) objects, as described below.

The type rules distinguish between the new expressions that create objects of the \( \text{Region} \) class, and new expressions that create objects of other classes. The rule for the latter 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 \( \text{new} \) expressions involving \( \text{Region} \) class, as we do not define fields for \( \text{Region} \).

The type rule for \( \text{new} \) \( \text{Region} \) expressions expects the \( \text{Region} \) 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-
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 (\(\pi_T\)), thus preventing cross-region references originating from transferable regions.11

The type rule for \texttt{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 \texttt{letregion} is then typechecked assuming that the new region is live (\(\pi \in \Delta\)) and that it is outlived by all existing live regions (\(\Delta \geq \pi\)). The result of a \texttt{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 \texttt{letregion} expression contains no references to the temporary objects inside the region.12

The rule for \texttt{open} expression, unlike the rule for \texttt{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 \texttt{open}. This prevents new objects allocated inside the transferable region from storing references to those outside. Environment \(\Gamma\) 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 (\(\phi\)) are required to be well-formed under \(\Delta'\), which means that

---

11The 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 \(e\).

12The appendix contains a formal proof of this claim.
\(\phi\) must only refer to the region variables in the set \(\Delta'\). Note that the closure is always allocated in the current allocation context (\(n^a\)). 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:

\[
\Delta \vdash (\epsilon; \Sigma) \rightarrow (\epsilon'; \Sigma')
\]

The reduction judgment should be read as following: given a set \(\Delta\) of regions that are currently live, and a map \(\Sigma\) from unique identifiers of transferable regions to their current states (Fig. 3), the expression \(\epsilon\) reduces to \(\epsilon'\), while updating \(\Sigma\) to \(\Sigma'\). The semantics gets “stuck” if \(\epsilon\) attempts to access an object whose allocation region is not present in \(\Delta\), or if \(\epsilon\) tries to open a transferable region, whose identifier is not mapped to a state by \(\Sigma\). On the other hand, if \(\epsilon\) 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 (\(\bot\)).

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

\[v \in \text{values} ::= \text{new Region}(T)(\pi)(v)\]

Note that for \(\text{new Region}(T)(\pi)(v)\) to be considered a value, \(\pi \neq \pi_f\). The semantics reduces a new Region \(T\langle\pi\rangle(\epsilon)\) expression in the user program to a runtime new Region \(T\langle\pi\rangle(\epsilon)\) value that is tagged with a unique identifier (\(\pi\)) for this region. A binding is also added to \(\Sigma\), mapping \(\pi\) 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\footnote{Theorem 3.1 and A.7 are restated and proved in the appendix}:

**Theorem 3.1. (Type Safety)** \(\forall \epsilon, \pi, \Delta, \Sigma, \pi_f\), such that \(\pi \in \Delta\) and \(\Delta \vdash \Phi \phi \epsilon\), if \((\text{dom}(\Sigma), \Delta', \Phi)\), \(\pi_f \vdash e : \tau\), then either \(e\) is a value, or \(e\) raises an exception \((\Delta \vdash (\epsilon; \Sigma) \rightarrow \bot)\), or there exists an \(\epsilon'\) and a \(\Sigma'\) such that \(\Delta \vdash (\epsilon; \Sigma) \rightarrow (\epsilon'; \Sigma')\) and \((\text{dom}(\Sigma'), \Delta', \Phi)\), \(\pi_f \vdash e : \tau\).

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)** \(\forall v, \Delta, \Delta', \Sigma, \Sigma', \Phi, \Phi', \pi, \pi_f\), such that \(\pi \in \Delta\), \(\pi' \in \Delta'\), and \(\pi_f \notin \text{dom}(\Sigma') \cup \Delta \cup \{\pi_f\}\), if \((\text{dom}(\Sigma), \Delta, \Phi)\), \(\pi_f \vdash \text{new Region}(T)(\pi_f)(v)\):

\[
\text{Region}(T)(\pi_f), \text{then } (\Sigma'[\pi_i \rightarrow \mathbb{C}], \Delta', \Phi') \vdash \text{new Region}(T)(\pi_f)
\]

The above theorem states that if a new Region \(T\langle\pi_i\rangle(\epsilon)\) value is well-typed under one context, then it is also well-typed under every other context, whose \(\Sigma\) maps \(\pi_i\) to closed (C) state. Thus, a recipient of a transferable region only needs to add a binding for the region to its \(\Sigma\) 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.

### 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 let region 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 (\(\rho\)) to denote unknown region annotations, and predicate variables (\(\phi\)) 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 CSL†, which reduces the constraint solving problem to an abduction problem in a Herbrand constraint system, and then relies on CSL†, our abduction solver for that domain.

#### 4.1 Region Type Templates

Region type templates are FGJ types extended with fresh region variables (\(\rho\)) and predicate variables (\(\phi\)) 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 Object \(\langle\rho_0\rangle\), where \(\rho_0\) 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 Pair \(\langle A, B \rangle \rightarrow A\) is
The template elides Pair’s methods, whose region type templates contain no free variables. Among the region parameters of the class template, $\rho_{0-4}$ are obtained by generalizing free region variables in the types of its class’s fields, constructor arguments, and its superclass type. The remaining parameter ($\rho^n_0$) is a fresh region variable denoting the allocation region argument. Fresh predicate variable $\varphi_0$ denotes unknown constraints over $\rho^n_0$ and $\rho_{0-4}$ 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 occurrences of the class among the types of its fields are ignored and the class is templatezized as if it is a non-recursive class. Next, all the recursive occurrences are templatezized 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 (a) class containing two fields: data of type a and next of type ListNode (a). where a is the type of the data stored in the linked list node. To templatezize the ListNode (a) class, we first ignore its recursive occurrence in the type of next field, and templatezize the type a of data field as a $\mathcal{0}_{\rho_0}$. Based on this type template of data field, we compute the class’s template as following:

\[14, 15\ \langle\rho^n_0, \rho_1, \rho_2, \rho_3 | \varphi_0\rangle\ Pair(\mathcal{A}, \mathcal{B})(\rho_1, \rho_2, \rho_3) \rightarrow \text{unit},\]
where $\rho^n_0$ and $\rho_{1-3}$ are fresh region variables, and $\varphi_0$ is a fresh predicate variable denoting unknown constraints over $\rho^n_0$ and $\rho_{1-3}$. Region type template of a type variable a is a $\mathcal{0}_{\rho_0}$, where $\rho_0$ 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 occurring in the templates computed in the first step as region parameters of the class. Finally, we add a fresh allocation region parameter ($\rho^n$) to the list of parameters, and introduce a new predicate variable ($\varphi$) to denote unknown constraints on region parameters. For example, consider the standard FGI definition of Pair (a, b) class, where a and b are the types of fst and snd fields, respectively. It can be templatized as following:

\[
\text{class } Pair(a \triangleleft \text{Object}, b \triangleleft \text{Object}) \{ \rho^n_0, \rho_{0-4} | \varphi_0 \triangleleft \text{Object}(\rho_4) \} \\ \begin{array}{l}
\text{a} \mathcal{0}_{\rho_0} \ \text{fst}; \\
b \mathcal{0}_{\rho_1} \ \text{snd}; \\
\text{Pair}(a \mathcal{0}_{\rho_2} \ \text{fst}, b \mathcal{0}_{\rho_3} \ \text{snd}) \{ \ldots \} \\
\end{array}
\]

The type template for a recursively defined class is computed in two steps. First, all recursive occurrences 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 occurrences 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 (a) class containing two fields: data of type a and next of type ListNode (a). where a is the type of the data stored in the linked list node. To templatezize the ListNode (a) class, we first ignore its recursive occurrence in the type of next field, and templatezize the type a of data field as a $\mathcal{0}_{\rho_0}$. Based on this type template of data field, we compute the class’s template as following:

\[14\text{In our exposition, we assume that classes } \mathcal{A} \text{ and } \mathcal{B} \text{ are trivial subclasses of } \text{Object} \text{ with no fields/methods. Like } \text{Object}, \text{ they accept one region parameter - the allocation region of their objects.}
15\text{We abuse arrow notation to also represent types of methods, but unlike function types, there is no allocation region annotation atop the arrow in a method type.} \]
contains region type bindings for all the arguments of the method, including the implicit \( \text{this} \) argument. The region type template returned by \( \text{elabExpr} \) 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.

\( \text{elabClass} \) elaborates the definition of a class \( B \). It relies on \( \text{elabCons} \) and \( \text{elabMeth} \) functions to elaborate \( B \)’s constructor \( (k) \) and method definitions \( (\bar{d}) \), respectively. To the set of constraints returned by these functions, \( \text{elabClass} \) 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 outline 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 \( \text{elabClassTable} \) (Fig. 10) elaborates every definition in the class table \( CT \), while accumulating constraints. The constraints are finally solved by solve \((4.3)\), which returns substitution functions \( S_\rho \) and \( S_\phi \) 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 \( \rho \) to the set \( \Delta \) of regions in scope,

- Well-formedness constraints of form \( \Delta \vdash \phi \) \( \text{ok} \), restricting the domain of a predicate variable \( \phi \) to the set of all possible constraint formulas over region variables \( \Delta \) in scope, and

- Validity constraints of form \( \Phi \vdash \phi \), where \( \Phi \) and \( \phi \) are region constraints (Fig. 4) extended with predicate variables and pending substitutions\(^{17}\):\n
\[
\Phi, \phi \ ::= \ \text{true} \mid \rho \geq \rho' \mid \rho = \rho' \mid F(\phi) \mid \phi \land \phi \\
F \ ::= \cdot \mid [\rho/\rho']F
\]

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

\(^{16}\)The definition of \( \text{elabCons} \) is straightforward, hence not shown.

\(^{17}\)we borrowed this terminology from \([12]\).
tions that need to be carried out when a predicate variable \( (\varphi) \) is replaced by a concrete formula in a validity constraint. For instance, in the validity constraint \( \pi_1 \geq \pi_2 \vdash [\pi_1/\rho_1][\pi_2/\rho_2] \varphi \), pending substitution is \( [\pi_1/\rho_1][\pi_2/\rho_2] \).

Any concrete formula (call it \( \phi_{\text{sol}} \)) over variables \( \rho_1 \) and \( \rho_2 \) is a solution to \( \varphi \) if and only the formula obtained by substituting \( \pi_1 \) and \( \pi_2 \) for \( \rho_1 \) and \( \rho_2 \) (resp.) in \( \phi_{\text{sol}} \) is deducible from \( \pi_1 \geq \pi_2 \).

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

\[
\phi_{\text{ex}} \land \bigwedge_i \varphi_i \vdash \phi_{\text{cs}} \quad \phi_{\text{ex}} \land \bigwedge_i \varphi_i \vdash F_j(\varphi_j)
\]

Where \( \varphi_i \)'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 \( \varphi_i \)'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 \( \phi_{\text{ex}} \) and \( \phi_{\text{cs}} \) are concrete, i.e., free of predicate variables and pending substitutions. While \( \phi_{\text{ex}} \) captures relationships that are known to hold between concrete region identifiers (i.e., \( \pi \)'s) when the constraint was generated, \( \phi_{\text{cs}} \) captures relationships that are required to hold among region variables (i.e., \( \rho \)'s), or relationships between region variables and identifiers. Each region variable occurring in \( \phi_{\text{cs}} \) 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 occurring in the constraint.

**Constraints Example 1** Consider the `Pair` class template from [47]. Following constraints are generated during its elaboration (Constraints are identified with \( c \)'s). Some trivial constraints, such as \( \rho_4 \in \Delta_0 \) and \( \rho_5 \in \Delta_1 \), where \( \Delta_0 = \{ \rho_0^0, \rho_0^{-4} \} \) and \( \Delta_1 = \Delta_0 \cup \{ \rho_5 \} \), have been elided:

\[
\begin{align*}
[c_1] & : \Delta_0 \vdash \varphi_0 \land r \quad [c_2] : \varphi_0 \land r \vdash \rho_0 & \leq \rho_4 \land \rho_4 \vdash \rho_0 \\
[c_3] & : \varphi_0 \land \rho_2 = \rho_0 & \quad [c_4] : \varphi_0 \land \rho_3 = \rho_1 \\
[c_5] & : \varphi_0 \land \rho_1 \vdash \rho_5 = \rho_0 & \quad [c_6] : \Delta_1 \vdash \varphi_1 \land r
\end{align*}
\]

**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):

```plaintext
unit alt<\rho_0^0, \rho_0^{-4} | \varphi_2>(\text{Region}<Pair<A,A>> \times \pi_1) \rightarrow r,
Pair<A,A> <q_8 \ldots q_{12} \ldots q > q, (q <p_9> y) \rightarrow
open r as \pi_0 with root p in
let x = new Pair<A,A> <\pi_1, \pi_0, \pi_0> in
alt<\pi_1, \pi_1, \pi_0, \pi_0, \pi_0>(r, x, p.snd)
```

Constraints generated during the elaboration are shown below (let \( \Delta_2 = \{ \rho_0^0, \rho_0^{-4}, \rho_2^0, \rho_6^{-9} \} \) :

\[
\begin{align*}
[c_7] & : \Delta_2 \vdash \varphi_2 \land r \quad [c_8] : \Delta_2 \vdash \rho_7 \geq \rho_6 \land \rho_8 \geq \rho_6 \\
[c_9] & : \varphi_1 \land \varphi_2 \land \rho_7 = \rho_9 & \quad [c_10] : \varphi_1 \land \varphi_2 \land \rho_8 = \rho_9 \\
[c_11] & : \varphi_1 \land \varphi_2 \land \pi_0 \geq \pi_1 & \quad \Delta_1', \Delta_1 \vdash \varphi_2
\end{align*}
\]

4.5 Constraint Solving

Our constraint generation algorithm traverses the entire program, performing elaboration and collecting constraints, which are subsequently solved by mass. 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 preempt 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 there are no mutually recursive definitions in the source program. Recursive definitions are nonetheless allowed.

**Terminology** In a validity constraint, a predicate variable occurring on the left side of the function is said to occur negatively, or with *negative polarity*. In contrast, a predicate variable occurring 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 ($\varphi$) represents the unknown precondition in this case. The dependency from $c_2$ to $c_1$ means that $c_1$ must be solved to compute $\varphi$ 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 idiosyncrasies 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:

$$C_2: \{c_5\} \rightarrow C_1: \{c_2-4\} \leftarrow C_3: \{c_8-11\}$$

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 $c_{10}$, 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 ($\varphi$) 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 ($\varphi$), and also all the region variables ($\rho$) that occur free in $C$, such that the solution satisfies well-formedness constraints on $\varphi$ and $\rho$. To simplify presentation, we think of $C$ as being parameterized on $\varphi$ and $\rho$, and write it as $C[\varphi, \rho]$. We now formalize the constraint satisfaction problem, and its solution.

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

- $\phi_{cx} \land \varphi \vdash \phi_{cs} \land \phi_{cx} \land \varphi \vdash F_i(\varphi)$

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

- $\Phi_{sol}$ is well-formed under $\Delta_{\varphi}$ (i.e., $\Delta_{\varphi} \vdash \Phi_{sol} \land \varphi$).
- Every sequent in $C[\Phi_{sol}, S(\varphi)]$ is valid.
- $\Phi_{sol}$ is maximally weak. That is, $\exists \phi_{sol}’$ such that $\Phi_{sol}$ is well-formed, $C[\Phi_{sol}, S(\varphi)]$ is valid, and $\Phi_{sol}$’ is strictly weaker than $\Phi_{sol}$ (i.e., $\Phi_{sol} \Rightarrow \Phi_{sol}’$ but $\varphi \Phi_{sol}’$). 

### 4.6 Solving the CSP

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

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

Where $\Phi_{cx} = \bigwedge_i \phi_{cx}^i$ and $\Phi_{cs} = \bigwedge_i \phi_{cs}^i$. To see why both CSPs are equivalent, consider two distinct constraints, $c_i[\varphi, \rho_i]$ and $c_j[\varphi, \rho_j]$. Since $i \neq j$, $\rho_i \neq \rho_j$. 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, $\phi_{cx}^i$ either describes the same relationships that are already described by $\phi_{cx}^i$ 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:

\[
\begin{align*}
[c_{12}]: & & \varphi_0 \vdash p_0 \geq p_0^3 \land p_1 \geq p_0^3 \land p_4 = p_0^3 \land p_2 = p_0 \land p_3 = p_1 \\
[c_{13}]: & & \varphi_0 \land \varphi_1 \vdash p_5 = p_0 \\
[c_{14}]: & & \varphi_1 \land \varphi_2 \land \pi_0 \geq \pi_1 \land p_3 \geq p_0 \land p_8 \geq p_6 \land p_3 = p_0 \land \pi_0 = \pi_0 / p_0^2 \land \pi_1 / p_0^2 \land \pi_2 = \varphi_2
\end{align*}
\]

### 4.6.1 Non-Recursive Constraints

We first describe how we solve a non-recursive CSP $\{\mathcal{c}[\varphi, S], \Delta_\varphi\}$, where $\mathcal{c}[\varphi, S]$ is of the form $\Phi_{\text{ex}} \land \varphi \vdash \Phi_{\text{es}}$.

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

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

$$\Phi_{\text{ex}} \land \varphi \vdash \Phi'_{\text{ex}}$$

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

- **Case $\Delta_\varphi \vdash \phi$ ok:** This means that $\phi$ is a solution to $\varphi$.
- **Case $\Delta_\varphi \not\vdash \phi$ ok:** In this case, $\phi$ contains at least one equality or outlives constraint on two region identifiers, $\pi_i$ and $\pi_j$, 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 $\Phi_{\text{ex}}$ (if it is implied, then $\phi$ is not maximally weak). Let us denote such constraint on $\pi_i$ and $\pi_j$ as $\phi_{ij}$. Now, let us consider a solution $\varphi_{sol}$ to $\varphi$, which means that $\Phi_{\text{ex}} \land \varphi_{sol} \vdash \Phi'_{\text{ex}}$. Since $\Phi_{\text{ex}} \vdash \phi \iff \Phi'_{\text{ex}}$, we have $\Phi_{\text{ex}} \land \varphi_{sol} \vdash \phi$. Since $\phi \vdash \phi_{ij}$, we have $\Phi_{\text{ex}} \land \varphi_{sol} \vdash \phi_{ij}$, although $\Phi_{\text{ex}} \not\vdash \phi_{ij}$. But this is impossible. To see why, recall that all the constraints on identifiers in $\Delta_\varphi$ occurring in $\Phi_{\text{ex}}$ are subsumed by $(\Delta - \Delta_\varphi) \geq \Delta_\varphi$. Therefore, it is impossible to derive $\phi_{ij}$ from $\Phi_{\text{ex}}$ by only adding constraints on $\pi \in \Delta_\varphi$.

Hence, such a solution $\varphi_{sol}$ cannot exist.

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

**Definitions** Given a constraint formula $\phi$, 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 $\phi$, and whose edges ($E(\phi)$) denote outlives constraints in $\phi$. That is, if $\phi$ contains a constraint $\rho_1 \geq \rho_2$, 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 $\rho_1, \rho_2 \in \Delta_\varphi$, if there exists a path between $\rho_1$ and $\rho_2$ in $G_2$, then there must exist a path between same pair of vertices in $G_1$.

A maximally weak $\varphi_{sol}$ that satisfies $\Phi_{\text{ex}} \vdash \varphi_{sol} \iff \Phi_{\text{es}}$ is a constraint encoding of the smallest graph $G$ (i.e., $\Phi(G)$) such that $\Phi(G_{\text{ex}}) \cup G$ is as connected as $\Phi(\varphi_{es})$. The problem of computing such a $G$ is equivalent to the problem finding of finding minimum number of edges to add to $G(\varphi_{es})$ such that it is as connected as $G(\varphi_{ex})$. Algorithms to solve the later problem are known to exist.

Solving the constraints $c_{12}$ and $c_{13}$ by reducing them to graph augmentation problems, as described above, results in the following solutions for $\varphi_0$ and $\varphi_1$ (symmetric outlives constraints are replaced with equalities):

$\varphi_1 \equiv p_0 \geq p_0^3 \land p_1 \geq p_0^3 \land p_4 = p_0^3 \land p_2 = p_0 \land p_3 = p_1$

$\varphi_2 \equiv p_5 = p_0$

### 4.6.2 Solving Recursive Constraints

Substituting the above solutions for $\varphi_0$ and $\varphi_1$ in $c_{14}$ gives us a recursive constraint of the following form:

$$\Phi_{\text{ex}} \land \varphi \vdash \Phi_{\text{es}} \land F(\varphi)$$

Where $\Phi_{\text{ex}}$ and $\Phi_{\text{es}}$ are concrete constraint formulas, and $F$ is a substitution function, not necessarily idempotent. We now extend constraint solving to recursive CSP $\{c[\varphi, S], \Delta_\varphi\}$, where $c[\varphi, S]$ assumes the form $\Phi_{\text{ex}} \land \varphi \vdash \Phi_{\text{es}} \land \bigcup_i F_i(\varphi)$.

For the sake of brevity, we define $G(\varphi) = \Phi_{\text{ex}} \land \bigcup_i F_i(\varphi)$, and use $F$ in $c[\varphi, S]$:

$$\Phi_{\text{ex}} \land \varphi \vdash G(\varphi)$$

To solve the above recursive constraint, we start with the observation that the set of all constraints (\phi) over $\Delta \cup \{\pi\}$ is a lattice, where $\phi_1 \leq \phi_2 \iff \phi_1 \Rightarrow \phi_2$. Note that $G$ is a monotone over the lattice:

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

From Knaster-Tarski’s theorem, we know that $G$ has a greatest fixed point ($\phi_f$), with following properties:

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

---

We alternate between viewing a constraint formula (\phi) as a set of constraints and a conjunction of constraints in this description.
We therefore compute the greatest fixed point ($\phi_f$) of $G$, and convert the recursive constraint to the following non-recursive constraint:

$$\Phi_{ct} \land \varphi \vdash \phi_f$$

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 ($\phi_f$) is shown below:

$$\rho_7 \geq \rho_6 \land \rho_8 \geq \rho_6 \land \rho_7 = \rho_9 \land \rho_8 = \rho_9 \land \pi_0 \geq \pi_1 \land \pi_0 = \pi_0$$

Computing a maximally weak $\phi_{sol}$ such that $\varphi_1 \land \pi_0 \geq \pi_1 \vdash \phi_{sol} \Rightarrow \phi_f$ results in the following solution for $\varphi_2$ that meets its well-formedness requirement ($\{\rho_0^6, \rho_0 - 4, \rho_2^5, \rho_6 - 9\} \vdash \varphi_2 \circ \Omega$):

$$\rho_7 \geq \rho_6 \land \rho_8 \geq \rho_6 \land \rho_7 = \rho_9 \land \rho_8 = \rho_9$$

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 $[\lceil FB^+\rceil]$, an extended version of $[\lceil FB\rceil]$ 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 $[\lceil FB^+\rceil]$ 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 referent 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 $[\lceil FB^+\rceil]$, and used our prototype compiler to assign region types to these programs, and thus proved their safety. During the process, we found multiple instances of potential memory safety violations in the $[\lceil FB^+\rceil]$ 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 referent from the outgoing message (a transferable region) to the payload of the incoming message. For example, the implementation of RegionSelectVertex contains the following:

```java
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. 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.

[Cyclone] 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-
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, shouldn’t 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. Cyclone 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 [15] by generalizing [15]’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. [13] 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 origins 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 Talpins-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


A. Appendix

**Lemma A.1. (Substitution Preserves Typing)** ∀v, z, τ₁, τ₂, Σ, Δ, Γ, Φ, if (dom(Σ), Δ, ·, Φ), π, Γ[z ↦ τ₁] ⊢ e : τ₂ and (dom(Σ), Δ, ·, Φ), π, Γ ⊢ v : τ₁, then (dom(Σ), Δ, ·, Φ), π, Γ ⊢ [v/ż]e : τ₂

**Proof** Intros e. Induction on e. For every subexpression e₀, inductive hypothesis says the following:

∀(v, τ₁, τ₂, Σ, Δ, Γ, Φ). (dom(Σ), Δ, ·, Φ), π, Γ[z ↦ τ₁] ⊢ e₀ : τ₂ ∧ (dom(Σ), Δ, ·, Φ), π, Γ ⊢ v : τ₁ H1

⇒ (dom(Σ), Δ, ·, Φ), π, Γ ⊢ [v/ż]e₀ : τ₂

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

(dom(Σ), Δ, ·, Φ), π, Γ[z ↦ τ₁] ⊢ e : τ₂ H2
(dom(Σ), Δ, ·, Φ), π, Γ ⊢ v : τ₁ H4

In each case, proof strategy is the same: invert on H2, apply H1, and then construct the proof term for the goal by applying type rules.

**Lemma A.2. (Weakening)** ∀v, τ, Δ₀, Σ, Φ, π, π₀, such that value(v), π ∈ Δ, π₀ ⊈ Δ, and Δ₀ ⊆ Δ, if (dom(Σ), Δ ∪ {π₀}, ·, Φ ∧ Δ₀ ≥ π₀), π₀, ⊢ v : τ and (dom(Σ), Δ, ·, Φ) ⊢ τ ⊥ k, then (dom(Σ), Δ, ·, Φ), π, ⊢ v : τ.

**Proof** Intros. Hypotheses:

π ∈ Δ H1
π₀ ⊈ Δ H2
Δ₀ ⊆ Δ H4
(dom(Σ), Δ ∪ {π₀}, ·, Φ ∧ Δ₀ ≥ π₀), π₀, ⊢ v : τ H6
(dom(Σ), Δ, ·, Φ) ⊢ τ ⊥ k H8

Proof by induction on H6. Cases:

- Case (v = new N₀(τ) and τ = N₀): Inversion on H6:

fields(N₀) = T : τ H10
(dom(Σ), Δ, ·, Φ) ⊢ N₀ ⊥ k H11
(dom(Σ), Δ ∪ {π₀}, ·, Φ ∧ Δ₀ ≥ π₀), π₀, ⊢ τ : τ H12

Inductive hypothesis on τ:

(dom(Σ), Δ, ·, Φ) ⊢ τ ⊥ k ⇒ (dom(Σ), Δ, ·, Φ), π, ⊢ τ : τ H1

Inversion on H8 tells us that N₀ is of the form B{T} (πτ), where, class B{T} (πτ) {ρ₀, p | φ} ⊥ N {τφ, τ₆, ...} is a well-formed class definition. Furthermore, we get:

π, π ∈ Δ H13
® ⊥ T ⊥ k H14
Φ ⊢ [π/ρ₀][π/φ] H16

Let fields(N) = g : τφ. Inverting the well-formedness of class B, we get the following:

(θ, {ρ₀, p}, g ↦ τ₆), φ ⊢ τφ ⊥ k H18

Since ® ⊥ B{T} ⊥ k, H18 gives:

(θ, {ρ₀, p}, π, φ) ⊢ [T/π](τφ) ⊥ k H20

And, since π ≠ ρ₀ and {π} ∩ {p} = 0:

(θ, {π, π}, π/ρ₀)[π/φ] ⊢ [T/π](τφ) ⊥ k H22

From the definition of fields, we have:

fields(B{T} (πτ)) = g : [π/φ][π/ρ₀][T/π](τφ), g : [π/φ][π/ρ₀][T/π](τφ) ⊥ k H24

From H10 and H24, we know that τ = [π/φ][π/ρ₀][T/π](τφ). Substituting this in H22:

(θ, {π, π}, π/ρ₀)[π/φ] ⊢ τ ⊥ k H26
Using $H_{13}$ and $H_{16}$, from $H_{26}$, we derive:

$$(\text{dom}(\Sigma), \Delta, \cdot, \Phi) \vdash \pi \ ok \quad H_{28}$$

From $H_{28}$ and $IH_{1}$, we get:

$$(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \pi : \tau \quad H_{30}$$

From $H_{10}$, $H_{11}$, and $H_{30}$, we prove the required goal:

$$(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new } N_0(\pi) : N_0 \quad H_{30}$$

- Case ($v = \text{new } \text{Region } \langle T \rangle \langle \pi_1 \rangle (v_0)$ and $\tau = \text{Region } \langle T \rangle \langle \pi_\tau \rangle$): Inversion on $H_6$:

  $$\pi_i \in \text{dom}(\Sigma) \quad H_{10}$$
  $$\pi_i \notin \Delta \cup \{\pi_0, \pi_\tau\} \quad H_{12}$$

  $$(\text{dom}(\Sigma), \Delta \cup \{\pi_i, \cdot, \Phi) \vdash T^{\uplus \pi_i} \ ok \quad H_{14}$$

  $$(\text{dom}(\Sigma), \Delta \cup \{\pi_i, \cdot, \Phi) \vdash v_0 : T^{\uplus \pi_i} \quad H_{16}$$

Inductive hypothesis on $v_0$:

$$(\text{dom}(\Sigma), \Delta \cup \{\pi_i, \cdot, \Phi)) \vdash T^{\uplus \pi_i} \ ok \quad H_{22}$$

$H_{22}$ and $IH_{1}$ gives:

$$(\text{dom}(\Sigma), \Delta \cup \{\pi_i, \cdot, \Phi) \vdash v_0 : T^{\uplus \pi_i} \quad H_{24}$$

$H_{12}$ implies $\pi_i \notin \Delta \cup \{\pi_\tau\}$. Using this, and $H_{10}$, $H_{22}$ and $H_{24}$, we conclude:

$$(\text{dom}(\Sigma), \Delta, \cdot, \Phi) \vdash \text{Region } \langle T \rangle \langle \pi_1 \rangle : \text{Region } \langle T \rangle \langle \pi_\tau \rangle \quad H_{24}$$

- Case ($v = \lambda x^{\pi_a} (\rho^a \{\phi(\tau^T) \} \cdot \pi$ and $\tau = \rho^a \{\phi(\tau^T) \} \xrightarrow{\pi_a} \tau^2$): By inversion on $H_6$:

  $$\pi_a = \pi_0 \quad H_{10}$$

  $$(\text{dom}(\Sigma), \Delta \cup \{\pi_0, \cdot, \Phi) \vdash \lambda x^{\pi_0} (\rho^a \{\phi(\tau^T) \} \cdot \pi : \rho^a \{\phi(\tau^T) \} \xrightarrow{\pi_a} \tau^2 \quad H_{12}$$

From $H_{8}$:

$$(\text{dom}(\Sigma), \Delta, \cdot, \Phi) \vdash \rho^a \{\phi(\tau^T) \} \xrightarrow{\pi_a} \tau^2 \ ok \quad H_{14}$$

Inverting $H_{14}$ tells us that $\pi_0 \in \Delta$. But $H_{22}$ 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 $(\text{dom}(\Sigma), \Delta, \cdot, \Phi)$. Intuitively, this means that a let-expression or an open expression can never return a closure, thereby ensuring 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 fresh($\pi$) (i.e., $\pi \notin \Delta \cup \Sigma \cup \text{RVars}(e) \cup \text{RVars}(CT)$), if $(\Sigma, \Delta \cup \{\rho, \cdot, \Phi) \vdash e : \tau$, then $(\Sigma, \Delta \cup \{\pi, \cdot, \Phi) \vdash [\pi/\rho]e : [\pi/\rho]\tau$.

**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:

  $$\forall \Sigma, \tau, \Delta, \Phi, \rho, \pi$, such that $\Delta \vdash \Phi \ ok, \rho \notin \Delta \cup \Sigma$ and fresh($\pi$), if $(\Sigma, \Delta \cup \{\rho, \cdot, \Phi) \vdash e : \tau$, then $IH_1$$
  $$(\Sigma, \Delta \cup \{\pi, \cdot, \Phi) \vdash [\pi/\rho]e : [\pi/\rho]\tau$$
• If $\rho \not\in \text{RVars}(e)$ and $e_0$ is a subexpression of $e$, then $\rho \not\in \text{RVars}(e_0)$

• If $\pi \not\in \text{fv} (\phi_1 \land \phi_2)$, 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, \text{ if } \pi \in \Delta \text{ and } (\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau, \text{ then one of the following holds:}

(i) $\exists (e', \Sigma'). \Delta \vdash (e, \Sigma) \rightarrow (e', \Sigma')$

(ii) value($e$)

(iii) $\Delta \vdash (e, \Sigma) \rightarrow \bot$

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

$\forall (\tau_0, \Delta_0, \Sigma_0, \Phi_0, \pi_0). \pi_0 \in \Delta_0 \land (\text{dom}(\Sigma_0), \Delta_0, \cdot, \Phi_0), \pi_0, \cdot \vdash e_0 : \tau_0 \Rightarrow IH1$

$\exists (e_0', \Sigma_0'). \Delta \vdash (e_0, \Sigma_0) \rightarrow (e_0', \Sigma_0') \lor (\text{value}(e_0)) \lor (\Delta \vdash (e_0, \Sigma_0) \rightarrow \bot)$

Cases from the induction:

• Cases ($e = \emptyset$ and $e = x$): proof trivial.

• Case ($e = e_0.f_i$): Intros. Hypothesis:

\[
\pi \in \Delta \quad H2 \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau \quad H5
\]

Inverting $H5$:

\[
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_0 : \tau' \quad H7 \\
\overline{\tau} : \tau = \text{fields}(\text{bound}(\tau')) \quad H9
\]

Applying $H7$ in $IH1$, we have three cases:

• SCase ($e_0$ takes a step): Hypotheses:

\[
\Delta \vdash (e_0, \Sigma) \rightarrow (e_0', \Sigma_0') \quad H11
\]

Therefore ($e_0.f_i, \Sigma$) takes a step to ($e_0', f_i, \Sigma_0'$) under $\Delta$.

• SCase ($e_0$ is a value): Since $e_0$ has type $\tau'$ and bound is defined for $\tau'$, it follows that $e_0$ is $\text{new } N(\overline{\tau})$. From $H7$:

\[
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new } N(\overline{\tau}) : \tau' \quad H14
\]

Inverting $H14$:

\[
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash N \circ \overline{\tau} \quad H16
\]

Inverting $H16$:

\[
\text{allocRgn}(N) \in \Delta \quad H18
\]

From $H9, H18$, we know that ($e_0.f_i, \Sigma$) takes a step to ($v_i, \Sigma$)

• SCase ($e_0$ raises $\bot$): $e_0.f_i$ also raises $\bot$.

• Case ($e = \text{let region } \pi_0$ in $e_0$): Intros. Hypothesis:

\[
\pi \in \Delta \quad H2 \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau \quad H5
\]

Inverting $H5$:

\[
\pi_0 \not\in \Delta \quad H7 \\
(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \land \Delta \uplus \pi_0), \pi_0, \cdot \vdash e_0 : \tau \quad H9
\]

From $H9$ and $IH1$, we have three cases:

• SCase ($e_0$ takes a step). Hypotheses:

\[
\Delta \cup \{\pi_0\} \vdash (e_0, \Sigma) \rightarrow (e_0', \Sigma_0') \quad H11
\]

From $H7$ and $H11$, $\Delta \vdash (e, \Sigma) \rightarrow (\text{let region } \pi_0 \text{ in } e_0', \Sigma_0')$.

• SCase ($e_0$ is a value $v_0$): From $H7$, $\Delta \vdash (e, \Sigma) \rightarrow (v_0, \Sigma)$

• SCase ($e_0$ raises $\bot$): $e$ raises $\bot$ too.
• Case \((e = \text{open } a \text{ as } y@\pi_0 \text{ in } e_b)\): Intros. Hypotheses:

\[
\begin{align*}
\pi \in \Delta & \quad \text{H2} \\
(dom(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot \vdash e : \tau) & \quad \text{H5}
\end{align*}
\]

Inverting \(H5\):

\[
\begin{align*}
(dom(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot \vdash e : \text{Region } T\langle \pi \rangle) & \quad \text{H7} \\
\pi_0 \not\in \Delta & \quad \text{H9} \\
(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi, \pi_0, [y \mapsto T@\pi_0] \vdash e_b : \tau) & \quad \text{H11}
\end{align*}
\]

We have three cases. First case deals with \((e_a, \Sigma)\) taking a step to \((e'_a, \Sigma')\) under \(\Delta\). Under this context, \((e, \Sigma)\) takes a step to \((\text{open } e'_a \text{ as } y@\pi_0 \text{ in } e_b, \Sigma'_a)\). So, there is progress. Second case deals with \(e_a\) raising \(\perp\). In this case, execution of \(e\) also raises \(\perp\). So, we again have progress. Third case deals with \(e_a\) being a value \(\text{new } \bar{N}(\pi)\). 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 Region values. The first rule does not apply because \(\text{fields(Region } T\langle \pi \rangle)\) is undefined. The only rule that applies is the special type rule for Region values. Hence, \(e_a\) is \(\text{new Region } T\langle \pi \rangle(v)\), where:

\[
\begin{align*}
\pi_i \not\in \Delta & \quad \text{H12} \\
\pi_i \in \text{dom}(\Sigma) & \quad \text{H13} \\
(dom(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi, \pi_i, \cdot \vdash v : T@\pi_i) & \quad \text{H16}
\end{align*}
\]

From \(H9, H13, H16\), and Lemma [A.3]

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

From \(H11, H18\) and Lemma [A.1]

\[
(dom(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi, \pi_0, \cdot \vdash [[\pi_0/\rho^0]v/y]e_b : \tau) \quad \text{H19}
\]

From \(H12\) and \(H13\), we know that \(\pi_i \in \text{dom}(\Sigma)\). We now have three cases:

• SCase \((\Sigma(\pi_i) \neq \lambda \text{ and } e_b \text{ is not a value})\): By inductive hypothesis, \([[\pi_0/\rho^0]v/y]e_b, \Sigma[\rho \mapsto 0]\) can either (a) take a step to \((e'_b, \Sigma')\) under \(\Delta \cup \{\pi_0\}\), or (b) \(e_b\) evaluates to \(\perp\). In the first case, \((e, \Sigma)\) itself evaluates to:

\[
\text{(open new Region } T\langle \rho \rangle(\lambda 0\pi(\rho^0)(\cdot),v)) \text{ as } e'_b@y \text{ in } \Sigma'[\rho \mapsto \Sigma(\rho)]
\]

In the second case, the evaluation of \(e\) also raises \(\perp\).

• SCase\((\Sigma(\pi_i) \neq \lambda \text{ and } e_b \text{ is a value } v_b)\): Trivially, \(\Delta \vdash (e, \Sigma) \rightarrow (v_b, \Sigma)\).

• SCase\((\Sigma(\pi_i) = \lambda)\): \(e\) raises \(\perp\).

• \((e = e_a.m(\pi^a, \bar{\pi})(\bar{\tau}))\): Intros. Hypotheses:

\[
\begin{align*}
\pi \in \Delta & \quad \text{H2} \\
(dom(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot \vdash e : \tau) & \quad \text{H4}
\end{align*}
\]

By inversion on \(H4\):

\[
\begin{align*}
(dom(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot \vdash e_a : \tau_a) & \quad \text{H6} \\
\pi^a = \pi & \quad \text{H7} \\
\pi \in \Delta & \quad \text{H8} \\
\text{mtype}(m, \text{bound } (\tau_a)) = \langle \rho^\alpha \bar{\rho} | \phi \rangle \rightarrow \tau^2 & \quad \text{H10} \\
(dom(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot \vdash \bar{\tau} : \bar{\pi}/\bar{\rho}[\pi/\rho^0]\bar{\tau}) & \quad \text{H11} \\
\Phi \vdash \bar{\pi}/\bar{\rho}[\pi/\rho^0]\phi & \quad \text{H11}
\end{align*}
\]

Three cases:

• SCase \((e_a\text{ isn’t a value})\): From \(H16\) and \(IH\), we know that either (a) \(e_a\) can take a step, or (b) \(e_a\) reduces to \(\perp\). In first case, \(e\) can also take a step, and in second case, \(e\) also reduces to \(\perp\).

• SCase \((e_a = v_a, \text{ but } \exists i\text{ such that } e_i\text{ isn’t a value})\): From \(H11\), we know that \((e_i, \Sigma)\) can either (a) take a step to \((e'_i, \Sigma')\) under \(\Delta\), or (b) \(e_i\) reduces to \(\perp\). In the first case, we have \(\Delta \vdash (v_a.m(\pi^a, \bar{\pi})(...,e_i,...),\Sigma) \rightarrow (v_a.m(\pi^a, \bar{\pi})(...,e'_i,...),\Sigma')\).
• SCase (eₐ = vₐ and ∀i. eᵢ = vᵢ): H10 says that bound for τₐ is defined under empty Θ. This is possible only if τₐ = N and vₐ = new N(¯v). Furthermore, N cannot be of form Region(T)(πₗ) because mtype isn’t defined for Region.

Using these facts, and inverting H6, we get (dom(Σ), Δ, ·, Φ) ⊢ N ok. Inverting it again:

\[ \text{allocRgn}(N) ∈ Δ \quad H13 \]

Now, since mtype(m, N) is defined if and only if mbody(m, N) is defined, we know that:

\[ \text{mbody}(m, N) = \overline{\text{e}} \quad H14 \]

From H13 and H14, we know that Δ ⊢ ((vₐ,m(πⁿ),π),(Σ) → (π PURPOSE new N(¯v))|this|eₐ,Σ) →

• Case (e = eₐ(πⁿ,π)¯v). Proof closely follows the proof for eₐ.m(πⁿ,π)¯v. The only difference is that when eₐ evaluates to a lambda λ®π₀(ρ®π)(σ,τ).eₐ, we need a proof that π₀ ∈ Δ. This can be obtained by inverting the type judgment for the lambda.

• Case (e = new N(π)): Intros. Hypotheses:

\[ \pi ∈ Δ \quad H2 \]
\[ (\text{dom}(Σ), Δ, ·, Φ), π, · ⊢ \text{new N}(\pi) : τ \quad H4 \]

Inverting H4 leads to two cases:

• SCase (shape(N) ≠ Region(T)): Hypotheses:

\[ (\text{dom}(Σ), Δ, ·, Φ), π, · ⊢ \text{new Region}(T)(\pi₀) : τ \quad H4 \]

Inverting H5:

\[ \text{allocRgn}(N) ∈ Δ \quad H10 \]

Three cases:

• SSSCase (∃i such that eᵢ takes a step): In this case, e also takes a step.

• SSSCase (∃i such that eᵢ reduces to ⊥). In this case, e also reduces to ⊥.

• SSSCase (∀i eᵢ is a value vᵢ): In this case, e is also a value new N(π).

• SCase (shape(N) = Region(T) and e = Region(T)(πᵢ)(e₀)): From H4:

\[ (\text{dom}(Σ), Δ, ·, Φ), π, · ⊢ \text{new Region}(T)(π₀)(e₀) : τ \quad H4 \]

Inversion on H4 gives two cases:

• SSSCase (πᵢ = πₗ): In this case, e₀ = λ®π₀(ρ®π)(e₁). In this case, Δ ⊢ (new Region(T)(πₗ)(v₀), Σ) → (new Region(T)(πₗ)(πₗ)(πₗ), Σ[πₗ → C]), where πₗ is fresh (πₗ \notin Δ ∪ dom(Σ)).

• SSSCase (∀i.eᵢ ≠ πₗ): Hypotheses:

\[ \piᵢ ∈ \text{dom}(Σ) \quad H16 \]
\[ πᵢ \notin Δ ∪ \{πₗ\} \quad H18 \]
\[ (\text{dom}(Σ), Δ ∪ \{πᵢ\}, ·, Φ), πᵢ, · ⊢ e₀ : T@πᵢ \quad H20 \]

From IH1 and H20, we have three cases:

• SSSCase (e₀ is a value v₀): In this case, e = Region(T)(πᵢ)(v₀) is also a value.

• SSSCase (e₀ raises ⊥): In this case, e also raises ⊥.

• SSSCase (∆∪{πᵢ} → (e₀, Σ)): In this case, Δ ⊢ (Region(T)(πᵢ)(e₀), Σ) → (Region(T)(πᵢ)(e₀), Σ′)

• Case (e = e₀.transfer(πⁿ)()): Intros. Hypotheses:

\[ \pi ∈ Δ \quad H2 \]
\[ πⁿ = π \quad H3 \]
\[ (\text{dom}(Σ), Δ, ·, Φ), π, · ⊢ e₀ \text{. transfer}(πⁿ)() : τ \quad H4 \]

By inversion on H4:

\[ (\text{dom}(Σ), Δ, ·, Φ), π, · ⊢ e₀ : \text{Region}(T)(πₗ) \quad H6 \]
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 \( \text{Region} \{ T \} \langle \pi \rangle \) is \textit{new} \( \text{Region} \{ T \} \langle \pi \rangle \langle \pi \rangle \), where \( \pi \neq \pi \). By inversion on \( H6 \), we get:

\[
\pi_i \in \text{dom}(\Sigma) \quad H8
\]

We have two cases:
- \((\Sigma(\pi_i) \neq 0)\): In this case, \( \Delta \vdash (e, \Sigma) \rightarrow (e, \Sigma[\pi_i \mapsto X]) \).
- \((\Sigma(\pi_i) = 0)\): In this case, evaluation of \( e \) raises \( \bot \).

**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 \( (\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) \vdash e : \tau \) and \( \Delta \vdash (e, \Sigma) \rightarrow (e', \Sigma') \), then \( (\text{dom}(\Sigma'), \Delta, \cdot, \Phi, \pi, \cdot) \vdash e' : \tau \).

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

\[
\forall (\pi_0, \Delta_0, \Sigma_0, \Phi_0, \pi_0). \quad (\pi_0 \in \Delta_0) \wedge ((\text{dom}(\Sigma_0), \Delta_0, \cdot, \Phi_0, \pi_0, \cdot) \vdash e_0 : \pi_0) \wedge (\Delta \vdash (e_0, \Sigma_0) \rightarrow (e'_0, \Sigma'_0)) \quad IH1
\]

Cases from induction
- Case \( (e = \) \( \) or \( e = x \)): Proof is trivial.
- Case \( (e = e_0.f_i) \): Intros. Hypothesis:

\[
\begin{align*}
\pi & \in \Delta & H2 \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) & \vdash e : \tau & H4 \\
\Delta & \vdash (e, \Sigma) \rightarrow (e', \Sigma') & H6
\end{align*}
\]

Inverting \( H4 \):

\[
(\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) \vdash e_0 : \tau' & H7 \\
\bar{f} : \tau = \text{fields(bound } (\tau')) & H9
\]

Inverting \( H6 \), we get two cases:
- \( \text{SCase} (\Delta \vdash (e_0, \Sigma) \rightarrow (e'_0, \Sigma')) \): In this case, \( \Delta \vdash (e_0.f_i, \Sigma) \rightarrow (e'_0.f_i, \Sigma') \). \( H7 \) and \( IH1 \) gives:

\[
(\text{dom}(\Sigma'), \Delta, \cdot, \Phi, \pi, \cdot) \vdash e'_0 : \tau' \quad H11
\]

Proof follows from \( H11 \) and \( H9 \).
- \( \text{SCase} \ (e_0 \text{ is a value } \text{new } N(\pi)) \): Hypotheses:

\[
\begin{align*}
\text{allocRgn}(N) & \in \Delta & H13 \\
\Delta & \vdash (e, \Sigma) \rightarrow (v_1, \Sigma) & H15
\end{align*}
\]

We need to prove that \(((\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) \vdash v_1 : \tau_1)\). From \( H7 \), since \( e_0 = \text{new } N(\pi) \):

\[
(\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) \vdash \text{new } N(\pi) : \tau' \quad H16
\]

Inverting \( H16 \) and using \( H9 \) gives us the proof.
- Case \( (e = \text{letregion } \pi_0 \text{ in } e_0) \): Intros. Hypothesis:

\[
\begin{align*}
\pi & \in \Delta & H2 \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi, \pi, \cdot) & \vdash e : \tau & H4 \\
\Delta & \vdash (e, \Sigma) \rightarrow (e', \Sigma') & H6
\end{align*}
\]

Inverting \( H4 \):

\[
\begin{align*}
\pi_0 & \notin \Delta & H7 \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi) & \vdash \tau \circ k & H8 \\
(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \land \Delta \geq \pi_0), \pi_0, \cdot & \vdash e_0 : \tau & H9
\end{align*}
\]

Inverting \( H6 \), we get two cases:
\item SCase \((\Delta\cup\{\pi_0\}) \vdash (e_0, \Sigma) \longrightarrow (e'_0, \Sigma')\): In this case, \(\Delta \vdash (\text{let\ region}\ \pi_0\ \text{in}\ e_0, \Sigma) \longrightarrow (\text{let\ region}\ \pi_0\ \text{in}\ e'_0, \Sigma')\). 

H9 and H11 gives:

\[(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi \land \Delta \succeq \pi_0), \pi, \cdot \vdash e_0 : \tau\quad\text{H11}\]

From H7 and H11, we can conclude that \((\text{dom}(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{let\ region}\ \pi_0\ \text{in}\ e'_0 : \tau\).

\item SCase \((e_0)\): In this case, \(\Delta \vdash (\text{let\ region}\ \pi_0\ \text{in}\ e_0, \Sigma) \longrightarrow (v_0, \Sigma')\). From H2, H7, and Lemma \[A.2\] we have:

\[(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash v_0 : \tau\quad\text{H13}\]

Thus, type is preserved.

\item Case \((e = \text{open}\ e_a\ \text{as}\ y \# \pi_0\ \text{in}\ e_b)\): Intros. Hypotheses:

\[
\begin{align*}
\pi &\in \Delta & &\text{H2} \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau & &\text{H4} \\
\Delta \vdash (e, \Sigma) \longrightarrow (e', \Sigma') & &\text{H6}
\end{align*}
\]

Inverting H4:

\[
\begin{align*}
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash e_a : \text{Region}\ (T)\langle \pi_\tau \rangle & &\text{H7} \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi) \vdash \tau \circ k & &\text{H8} \\
\pi_0 \notin \Delta & &\text{H9} \\
(\text{dom}(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, [y \mapsto T @ \pi_0] \vdash e_b : \tau & &\text{H11}
\end{align*}
\]

Inverting H6, we get many cases:

\item SCase \((\Delta \vdash (e_a, \Sigma) \longrightarrow (e'_a, \Sigma'))\): Since the domain of \(\Sigma\) monotonically increases during the evaluation, we have:

\[\text{dom}(\Sigma) \subseteq \text{dom}(\Sigma')\quad\text{H13}\]

Since strengthening the context trivially preserves typing and well-formedness, from H7, H11, and H13, we have:

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

From H15, H20, we have \((\text{dom}(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau\).

\item SCase \((e_a = \text{open}\ v_a, \text{and} \ e_b\ \text{steps to} \ e'_b)\): Hypotheses:

\[
\begin{align*}
v_a = \text{new Region} &\langle T\rangle\langle \pi_r \rangle(v_r) & &\text{H22} \\
\pi_i \neq \pi_r & &\text{H23} \\
\Sigma(\pi_i) \neq \chi & &\text{H24} \\
\pi_0 \notin \Delta & &\text{H26} \\
\Delta \cup \{\pi_0\} \vdash \{[\pi_0/\pi_i]v_r/y[v_b, \Sigma(\pi_i) \mapsto 0]\} \longrightarrow (e'_b, \Sigma') & &\text{H27} \\
\Sigma'' = \Sigma''[\pi_i \mapsto \Sigma(\pi_i)] & &\text{H29}
\end{align*}
\]

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

\[\text{dom}(\Sigma) \subseteq \text{dom}(\Sigma')\quad\text{H31}\]

Next, since \(e_a = v_a\), from H7 and H22, we have:

\[(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new Region} &\langle T\rangle\langle \pi_r \rangle(v_r) : \text{Region} &\langle T\rangle\langle \pi_\tau \rangle\quad\text{H33}\]

Since H23, inversion on H33 gives:

\[
\begin{align*}
(\text{dom}(\Sigma), \Delta \cup \{\pi_i\}, \cdot, \Phi), \pi_i, \cdot \vdash v_r : T @ \pi_i & &\text{H34} \\
\pi_i \notin \Delta & &\text{H35}
\end{align*}
\]

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

\[(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [\pi_0/\pi_i]v_r : T @ \pi_0\quad\text{H36}\]
From $H11$, $H36$ and Lemma $A.1$ we get:

$$(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [[\pi_0/\pi_t]v_0]e_0 : \tau \quad H38$$

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

$$(\text{dom}(\Sigma[[\pi_i \mapsto 0]]), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash [[\pi_0/\pi_t]v_0]e_0 : \tau \quad H40$$

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

$$(\text{dom}(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash e'_0 : \tau \quad H42$$

By strengthening the type context:

$$(\text{dom}(\Sigma'), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash y \mapsto T@\pi_0 \vdash e'_0 : \tau \quad H48$$

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

$$(\text{dom}(\Sigma''), \Delta, \cdot, \Phi), \pi, \cdot \vdash v_a : \text{Region}(T)\langle \pi \tau \rangle \quad H50$$

$$(\text{dom}(\Sigma''), \Delta, \cdot, \Phi) \vdash \tau \circ \kappa \quad H51$$

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

$$(\text{dom}(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{open } v_0 \text{ as } y \mapsto \pi_0 \text{ in } e'_0 : \tau \quad \text{SCase (} e_a \text{ is a value } v_a, \text{ and } e_b \text{ is a value } v_b): \text{In this case, } \Delta \vdash (\text{open } v_0 \text{ as } y \mapsto \pi_0 \text{ in } v_b, \Sigma) \rightarrow (v_b, \Sigma). \text{ From } H11:

$$(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash v_0 : \tau \quad H53$$

Since value($v_b$), it has no free variables. Consequently:

$$(\text{dom}(\Sigma), \Delta \cup \{\pi_0\}, \cdot, \Phi), \pi_0, \cdot \vdash v_0 : \tau \quad H55$$

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

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

• Case ($e = \text{new Region}(T)\langle \pi \tau \rangle(e_0)$): Hypotheses:

$$(\pi \in \Delta$$

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

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

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

$$(\emptyset, \{\rho^a\}, \cdot, \text{true}), \rho^a, \cdot \vdash e_1 : T@\rho^a \quad H7$$

\$\vdash T \circ \kappa \quad H8$$

From $H7$ and Lemma $A.3$

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

Inverting $H6$:

$$\pi_i \notin \text{dom}(\Sigma) \cup \Delta$$

$$\Sigma' = \Sigma[\pi_i \mapsto \rho]$$

$$\Delta \vdash (\text{new Region}(T)\langle \pi \tau \rangle(\lambda@\pi(\rho^a)(\cdot), e_1)) \rightarrow (\text{new Region}(T)\langle \pi \tau \rangle(\pi_i/\rho^a)e_1), \Sigma')$$

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

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

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

$$(\text{dom}(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new Region}(T)\langle \pi \tau \rangle(\pi_i/\rho^a)e_1) : \text{Region}(T)\langle \pi \tau \rangle$$
• Case \( e = \text{new Region}(T)(\pi_i)(e_0) \), where \( \pi_i \neq \pi_\tau \): Hypotheses:

\[
\begin{align*}
\pi & \in \Delta \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new Region}(T)(\pi_i)(e_0) : \tau & \quad \text{H2} \\
\Delta \vdash (\text{new Region}(T)(\pi_i)(e_0), \Sigma) \rightarrow (e', \Sigma') & \quad \text{H4} \\
\end{align*}
\]

Since \( e \) isn’t a value, inverting \( H6 \) tells us that \( \text{new Region}(T)(\pi_i)(e_0) \) takes a step to \( \text{new Region}(T)(\pi_i)(e'_0) \) 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 \( \pi_i \), and finally apply the type rule for \( \text{new Region}(T)(\pi_i)(e'_0) \) (where \( \pi_i \neq \pi_\tau \)) 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 \( \text{let} \) expression): Proof follows directly from the inductive hypothesis, substitution lemma \( \text{(A.1)} \) and renaming lemma \( \text{(A.3)} \).

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

\[
\begin{align*}
(i) & \quad \exists (e', \Sigma'). \text{ such that } \Delta \vdash (e, \Sigma) \rightarrow (e', \Sigma') \text{ and } (\text{dom}(\Sigma'), \Delta, \cdot, \Phi), \pi, \cdot \vdash e : \tau \\
(ii) & \quad \Delta \vdash (e, \Sigma) \rightarrow \bot
\end{align*}
\]

**Proof** Directly follows from Theorems \( \text{(A.4)} \) and \( \text{(A.5)} \).

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

**Proof** Intros. Hypothesis:

\[
\begin{align*}
\pi_i \notin \text{dom}(\Sigma') \cup \Delta' \cup \{\pi_\tau\} & \quad \text{H6} \\
(\text{dom}(\Sigma), \Delta, \cdot, \Phi), \pi, \cdot \vdash \text{new Region}(T)(\pi_i)(v) : \text{Region}(T)(\pi_\tau) & \quad \text{H8}
\end{align*}
\]

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