Safe Transferable Regions

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.

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

Figure 1: `SELECT` dataflow operator

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

Region-based memory management, both manual as well as automatic, has been known for a long time. Manual region-based memory management suffers from the usual drawbacks, namely the potential for invalid references and the consequent lack of memory safety. Automatic region-based memory management systems guarantee memory safety, but impose various restrictions. MLKit, which implements the approach pioneered by Tofte and Talpin [16, 17], 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.

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

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 [16]-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 various regions, we can statically decide the safety of all references in the program.

Indeed, 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 checks. In particular, we

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

define an acceptable state transition discipline for transferable regions (Fig. 3), and check, at runtime, whether a given transition of a transferable region (*e.g.*, from *open* state to *freed* state) is valid or not. The check is lightweight since it only involves checking a single tag that captures the current state. We believe that this is a reasonable choice since regions are coarse-grained objects manipulated infrequently, when compared to the fine-grained objects that are present inside these regions, for which safety is enforced statically. An added advantage of delegating the enforcement of typestate invariants to runtime is that our region type system is simple[3], which made it possible to formulate a type inference that completely eliminates the need to write region type annotations. This, we believe, significantly reduces the impediment to adopt our approach in practical setting.

Contributions

The paper makes the following contributions:

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

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

2. An Informal Overview of BROOM

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

2.1 Using Regions in BROOM

Stack Regions The "`letregion R { S }`" construct creates a new stack region, with a static identifier `R`, whose scope is restricted to the statement `S`. The semantics of `letregion` is similar to Tofte and Talpin [16]'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`.

Transferable Regions Transferable 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<A>(() => 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 (possibly remote) consumer process.

Open and Closed Regions A transferable region must be explicitly *opened* using BROOM's `open` construct in or-

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

Figure 2: SELECT dataflow operator in BROOM

der to either read or update or allocate objects in the region. Specifically, the construct "`open rgn as v@R { S }`" does the following: (a). It opens the transferable region handled by `rgn` for allocation (i.e., makes it the current allocation context), (b). binds the identifier `R` to this open region, and (c). initializes the newly introduced local variable `v` to refer to the root object of the region. The `@R` part of the statement is optional and may be omitted. The `open` construct is intended to simplify the problem of ensuring memory safety, as will be explained soon. We refer to a transferable region that has not been opened as a *closed* region. A transferable region can only be transferred or freed when it is in closed state. The acceptable state transition discipline over the lifetime of a transferable region is described in Fig. 3. Enforcement of this discipline is done at runtime.

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

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

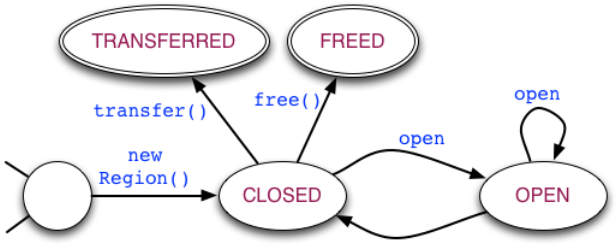

Figure 3: The lifetime of a dynamic (transferable) region in BROOM

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. We now formally develop this intuition via FEATHERWEIGHT BROOM (FB), our explicitly typed core language (with region types) that incorporates the features introduced in the previous section. FEATHERWEIGHT BROOM builds on the Featherweight Generic Java (FGJ) [9] formalism, and reuses notations and various definitions from [9], such as the definition of type well-formedness for the core (region-free) language.

3.1 Syntax

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

```
class Pair<a ◁ Object, b ◁ Object>
          <$\rho^a, \rho_1, \rho_2 \mid \rho_1 \succeq \rho^a \wedge \rho_2 \succeq \rho^a$> ◁ Object<$\rho^a$> {
  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;
  }
}
```

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

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

$$\begin{array}{llll}
cn & \in & \texttt{Class names} & ::= \texttt{Object} \mid \texttt{Region} \mid A \mid B \\
T & \in & \texttt{FGJ types} & ::= a \mid K \mid \texttt{unit} \mid \overline{T} \rightarrow T \\
D & \in & \texttt{Classes} & ::= \texttt{class}\ cn\langle \overline{a} \lhd \overline{K} \rangle \langle \rho^a \overline{\rho} \,|\, \phi \rangle \lhd N \{ \overline{\tau}\ \overline{f};\ k\ \overline{d} \} \\
k & \in & \texttt{Constructors} & ::= cn(\overline{\tau}\ \overline{x}) \{ \texttt{super}(\overline{x});\ \texttt{this}.\overline{f} = \overline{x}; \} \\
d & \in & \texttt{Methods} & ::= \tau\ m\langle \rho^a \overline{\rho} \,|\, \phi \rangle (\overline{\tau}\ \overline{x}) \{ \texttt{return}\ e; \}
\end{array}$$

$$\begin{array}{llll}
K & \in & \texttt{FGJ class types} & ::= cn\langle \overline{T} \rangle \\
N & \in & \texttt{FB class types} & ::= cn\langle \overline{T} \rangle \langle \pi^a \overline{\pi} \rangle \\
\tau & \in & \texttt{types} & ::= T@\pi \mid N \mid \texttt{unit} \\
 & & & \quad \mid \langle \rho^a \overline{\rho} \,|\, \phi \rangle \overline{\tau} \xrightarrow{\pi} \tau \\
\phi, \Phi & \in & \texttt{Constraints} & ::= true \mid \rho \succeq \rho \\
 & & & \quad \mid \rho = \rho \mid \phi \wedge \phi
\end{array}$$

$$\begin{array}{llll}
e & \in & \texttt{Expressions} & ::= () \mid x \mid e.f \mid e.m\langle \pi^a \overline{\pi} \rangle (\overline{e}) \mid \texttt{new}\ N(\overline{e}) \mid \lambda@\pi^a \langle \rho^a \overline{\rho} \,|\, \phi \rangle (\overline{x} : \overline{\tau}).e \mid e\langle \pi^a \overline{\pi} \rangle (\overline{e}) \\
 & & & \quad \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{letregion}\ \rho\ \texttt{in}\ e \mid \texttt{open}\ e\ \texttt{as}\ y@\pi\ \texttt{in}\ e
\end{array}$$

Figure 4: FEATHERWEIGHT BROOM: Syntax

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

To construct objects of the `Pair` class, its type and region parameters need to be instantiated with core types (T) and concrete region identifiers[6] (π), respectively. For example:

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

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

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

FB extends FGJ's expression language with a lambda expression and an application expression ($e\langle\pi^a\overline{\pi}\rangle(\overline{e})$) to define and apply functions[7]. Functions, like methods, exhibit constrained region polymorphism, as evident in their arrow region type ($\langle\rho^a\overline{\rho}\,|\,\phi\rangle\overline{\tau} \xrightarrow{\pi} \tau$). Since a function closure can escape the context in which it is created, it is important to keep track of the region in which it is created in order to avoid unsafe dereferences. The π annotation above the arrow in the arrow type denotes the allocation region of the corresponding closure. .

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. 5 contains an illustrative subset of such rules[8]. The rules refer to a context ($\mathcal{A}$), which is a tuple of:

- A set ($\Delta \in 2^{\pi}$) of static identifiers of regions that are estimated to be live,
- A finite map ($\Theta \in a \mapsto K$) of type variables to their bounds[9], and

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

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

[8]The full formal development can be found in the supplement

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

$$\boxed{\mathcal{A} \vdash \tau\ \mathsf{ok}, \quad \mathcal{A}, \pi^a, \Gamma \vdash e : \tau}$$

$$\frac{CT(B) = \texttt{class}\ B\langle \overline{a} \lhd \overline{K}\rangle\langle \rho^a, \overline{\rho} \,|\, \phi\rangle \lhd N\{...\} \quad \pi^a, \overline{\pi} \in \mathcal{A}.\Delta \qquad \mathcal{A}.\Theta \Vdash B\langle\overline{T}\rangle\ \mathsf{ok} \quad \mathcal{S} = [\overline{\pi}/\overline{\rho}, \pi^a/\rho^a, \overline{T}/\overline{a}] \quad \mathcal{A}.\Phi \vdash \mathcal{S}(\phi)}{\mathcal{A} \vdash B\langle \pi^a\overline{\pi}\rangle\langle\overline{T}\rangle\ \mathsf{ok}}$$

$$\frac{\mathcal{A}.\Theta \Vdash T\ \mathsf{ok} \quad \pi \in \mathcal{A}.\Delta \qquad \mathcal{A}.\Theta \Vdash T <: \texttt{Object}}{\mathcal{A} \vdash T@\pi\ \mathsf{ok}} \qquad \frac{\mathcal{A}.\Theta \Vdash T\ \mathsf{ok}}{\texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle}$$

$$\frac{\mathcal{A} = (\Sigma, \Delta, \Theta, \Phi) \quad \mathcal{A} \vdash \tau\ \mathsf{ok} \qquad \Phi' = \Phi \wedge (\Delta \succeq \pi) \quad \Delta' = \Delta \cup \{\pi\} \qquad \pi \notin \Delta \quad \mathcal{A}' = (\Sigma, \Delta', \Theta, \Phi') \quad \mathcal{A}', \pi, \Gamma \vdash e : \tau}{\mathcal{A}, \pi^a, \Gamma \vdash \texttt{letregion}\ \pi\ \texttt{in}\ e : \tau}$$

$$\frac{\mathcal{A}, \pi^a, \Gamma \vdash \overline{e} : \overline{\tau} \qquad \mathcal{A} \vdash N\ \mathsf{ok} \quad \mathsf{fields}(N) = \overline{f} : \overline{\tau}}{\mathcal{A}, \pi^a, \Gamma \vdash \texttt{new}\ N(\overline{e}) : N}$$

$$\frac{\mathcal{A}.\Theta \Vdash T\ \mathsf{ok} \quad K = \texttt{Region}\,\langle T\rangle \qquad (\emptyset, \{\rho^a\}, \mathcal{A}.\Theta, true), \rho^a, \cdot \vdash e : T@\rho^a}{\mathcal{A}, \pi^a, \Gamma \vdash \texttt{new}\ K\langle\pi_\top\rangle(\lambda@\pi^a\langle\rho^a\rangle().e) : K\langle\pi_\top\rangle}$$

$$\frac{\mathcal{A}, \pi^a, \Gamma \vdash e_a : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle \qquad \mathcal{A} \vdash \tau\ \mathsf{ok} \quad \mathcal{A} = (\Sigma, \Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \Delta' = \Delta \cup \{\pi\} \qquad \mathcal{A}' = (\Sigma, \Delta', \Theta, \Phi) \quad \Gamma' = \Gamma[y \mapsto T@\pi] \quad \mathcal{A}', \pi, \Gamma' \vdash e_b : \tau}{\mathcal{A}, \pi^a, \Gamma \vdash \texttt{open}\ e_a\ \texttt{as}\ y@\pi\ \texttt{in}\ e_b : \tau}$$

$$\frac{\mathcal{A} = (\Sigma, \Delta, \Theta, \Phi) \quad \rho^a, \overline{\rho} \notin \Delta \qquad \Delta' = \Delta \cup \{\rho^a, \overline{\rho}\} \quad \Delta' \vdash \phi\ \mathsf{ok} \quad \mathcal{A}' = (\Sigma, \Delta', \Theta, \Phi \wedge \phi) \qquad \mathcal{A}' \vdash \overline{\tau^1}\ \mathsf{ok} \quad \mathcal{A}' \vdash \tau^2\ \mathsf{ok} \quad \mathcal{A}', \rho^a, \Gamma[\overline{x} \mapsto \overline{\tau^1}] \vdash e : \tau^2}{\mathcal{A}, \pi^a, \Gamma \vdash \lambda@\pi^a\langle\rho^a\overline{\rho} \,|\, \phi\rangle(\overline{x} : \overline{\tau^1}).e : \langle\rho^a\overline{\rho} \,|\, \phi\rangle\overline{\tau^1} \xrightarrow{\pi^a} \tau^2}$$

Figure 5: FEATHERWEIGHT BROOM: Type Well-formedness and Expression Typing

- A constraint formula (Φ) that captures the outlives constraints on regions in Δ.

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

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

Like the judgments in FGJ [9], all the judgments defined by the rules in Fig. 5 are implicitly parameterized on a class table ($CT \in cn \mapsto D$) that maps class names to their definitions in FB.
The well-formedness judgment on region types ($\mathcal{A} \vdash \tau\ \mathsf{ok}$) makes use of the well-formedness and subtyping judgments on core types [10]. The class table ($[\![CT]\!]$) for such judgments is derived from FB's class table (CT) by erasing all region annotations on types, and region arguments in expressions ($[\![\cdot]\!]$ denotes the region erasure operation). The well-formedness rule for class types ($B\langle\overline{T}\rangle\langle\pi^a\overline{\pi}\rangle$) is responsible for enforcing the safety property that prevents objects from containing unsafe references. It does so by insisting that regions $\pi^a\overline{\pi}$ satisfy the constraints (ϕ) imposed by the class on its region parameters. The later is enforced by checking the validity of ϕ, with actual region arguments substituted[11] for formal region parameters, under the conditions ($\mathcal{A}.\Phi$) guaranteed by the context. The semantics of this sequent is straightforward, and follows directly from the properties of outlives and equality relations. For any well-formed core type T, $T@\pi$ is a well-formed region type if π is a valid region. The type `Region` $\langle T\rangle\langle\pi\rangle$ is well-formed only if $\pi = \pi_\top$, where $\pi_\top$ is a special immortal region that outlives every other live region. This arrangement allows `Region` handlers to be aliased and referenced freely from objects in various regions, regardless of their lifetimes. On the flip side, this also opens up the possibility of references between transferable regions, which become unsafe in context of the recipient's address space. Fortunately, such references are explicitly prohibited by the type rule of `Region` objects, as described below.

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

The type rule for `new Region` expressions expects the `Region` class's constructor to be called with a nullary function that returns a value in its allocation context. It enforces this by typechecking the body (e) of the function under an empty context containing nothing but the allocation context of the function. This step ensures that the value returned by the function stores no references to objects allocated elsewhere, including the top region ($\pi_\top$), thus preventing cross-region references originating from transferable regions[12].

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

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

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

The type rule for `letregion` expression requires that the static identifier introduced by the expression be unique under the current context (i.e., $\pi \notin \Delta$). This condition is needed in order to prevent the new region from incorrectly assuming any existing outlives relationships on an eponymous region. Provided this is satisfied, the expression (e) under `letregion` is then typechecked assuming that the new region is live ($\pi \in \Delta$) and that it is outlived by all existing live regions ($\Delta \succeq \pi$). The result of a `letregion` expression must have a type that is well-formed under a context not containing the new region. This ensures that the value obtained by evaluating a `letregion` expression contains no references to the temporary objects inside the region[13].

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

The type rule for lambda expression typechecks the lambda-bound expression (e) under an extended type environment containing bindings for function's arguments, assuming that region parameters are live, and that declared constraints over region parameters hold. The constraints (ϕ) are required to be well-formed under Δ', which means that ϕ must only refer to the region variables in the set Δ'. Note that the closure is always allocated in the current allocation context (π^a). This prevents the closure from escaping the context in which it is created, thus trivially ensuring the safety of any dereferences inside the closure.

3.3 Operational Semantics and Type Safety

The supplement defines a small-step operational semantics for FEATHERWEIGHT BROOM via a five-place reduction relation:

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

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

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

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

[13] The supplement contains a formal proof of this claim

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

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

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

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

The above theorem states that if a $\texttt{new Region}\,\langle T\rangle\langle\pi_i\rangle(v)$ value is well-typed under one context, then it is also well-typed under every other context, whose Σ maps π_i to closed (`C`) state. Thus, a recipient of a transferable region only needs to add a binding for the region to its Σ in order to preserve its type safety.

4. Type Inference

BROOM's region type system imposes a heavy annotation burden, and the cost of manually annotating C# standard libraries with region types is prohibitive. We now present our region type inference algorithm that completely eliminates the need to write region type annotations, except on some higher-order functions. Formally, the type inference algorithm is an elaboration function from programs in $[\![FB]\!]$ (i.e., FB without region types, but with `letregion` and `open` expressions, similar to the language introduced in § 2) to programs in FB. The key aspects of this elaboration are described below.

4.1 Region Type Templates

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

is [14,15] $\langle\rho_0^a,\rho_{1-4}\,|\,\varphi_0\rangle$ `Pair` $\langle$ `A` , `B` $\rangle\langle\rho_{1-3}\rangle\rightarrow$ `A` $\langle\rho_4\rangle$, where ρ_0^a and ρ_{1-3} are fresh region variables, and φ_0 is a fresh predicate variable denoting unknown constraints over ρ_0^a and ρ_{1-3}. Region type template of a type variable `a` is `a`$@\rho$, where ρ is fresh. For a class, a template is computed based on the type templates of its fields, constructors and methods. Additional region and predicate variables are added as necessary. Following template of the `Pair` class illustrates this point:

```
class Pair⟨a ◁ Object , b ◁ Object ⟩
        ⟨ρ0^a, ρ0−4 | φ0⟩ ◁ Object ⟨ρ4⟩ {
  a@ρ0 fst; b@ρ1 snd;
  Pair(a@ρ2 fst, b@ρ3 snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  a@ρ5 getFst⟨ρ1^a, ρ5 | φ1⟩() { return this.fst; }
}
```

The type template for for a recursively defined class is computed in two steps. First, a template is computed by ignoring the recursive occurrences. Next, all the recursive occurrences are templatized based on the template computed in first step. This approach can be extended to mutually recursive definitions in a straightforward manner. Using the techniques outlined above, we compute region type templates for all classes bound in the class table of the $[\![FB]\!]$ program before we proceed to elaborate expressions.

4.2 Elaboration and Constraint Generation

Elaborating $[\![FB]\!]$ expressions to FB expressions involves (a). replacing core types occurring in expressions with fresh region type templates, and (b). explicitly instantiating region parameters with fresh region variables in method calls and function applications. Our expression elaboration function[16] traverses expressions in a syntax-directed manner of a type checker, introducing fresh region type templates for unknown region types, while generating constraints over region and predicate variables. The precise nature of generated constraints is explained in §4.3, but in summary, they capture the relationships between the type templates of various subexpressions and their well-formedness.

Expression elaboration can be trivially lifted to the level of a method, and method elaboration to the level of a class, while accumulating constraints. Besides such constraints, elaboration functions for methods and classes also generate new constraints by checking well-formedness and subtype relationships among the types of their constituents. For instance, method elaboration generates a new constraint by typechecking the method body against its return type. Likewise, the elaboration function for a class generates a fresh constraint that captures a couple of safety conditions: first, the allocation regions of objects referred by the instance variables should outlive the allocation region of the instance itself, and second, the allocation regions of a class type and its superclass type must be the same.

Elaborating a class table (CT) entails elaborating every class definition in CT, while collecting constraints. The constraints thus collected are finally solved to obtain assignments for free region and predicate variables introduced during templatization and elaboration stages. The solutions 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.

4.3 Constraints

Three kinds of constraints are generated during elaboration:

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

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

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

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

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

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

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

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

[16]Supplement contains the pseudocode for all the elaboration functions.

[17]we borrowed this terminology from [12]

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):

```
unit alt<ρ^a_2,ρ_{6−9} | φ_2>(Region<Pair<A,A>><π_⊤> r,
                         Pair<A,A><ρ_{6−8}> q, A<ρ_9> y) {
  q.fst := y; q.snd := y;
  open r as p@π_0 in
    letregion π_1 in
      let x = new Pair<A,A><π_1,π_0,π_0>
                            (p.fst,p.fst) in
        alt<π_1,π_1,π_0,π_0,π_0>(r,x,p.snd)
}
```

Constraints generated during the elaboration are shown below (let $\Delta_2 = \{\rho^a_0, \rho_{0-4}, \rho^a_2, \rho_{6-9}\}$):

$$[\mathbf{c_7}] : \ \Delta_2 \vdash \varphi_2 \ \mathtt{ok} \qquad [\mathbf{c_8}] : \ \varphi_1 \wedge \varphi_2 \vdash \rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6$$
$$[\mathbf{c_9}] : \ \varphi_1 \wedge \varphi_2 \vdash \rho_7 = \rho_9 \qquad [\mathbf{c_{10}}] : \ \varphi_1 \wedge \varphi_2 \vdash \rho_8 = \rho_9$$
$$[\mathbf{c_{11}}] : \ \varphi_1 \wedge \varphi_2 \wedge \pi_0 \succeq \pi_1 \vdash [\pi_1/\rho^a_2][\pi_1/\rho_6][\pi_0/\rho_{7-9}]\varphi_2$$

4.4 Constraint Solving

Although we collect constraints for the entire program during elaboration, we solve the constraints in a compositional fashion. Concretely, 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 constraint solving algorithm by demonstrating its performance over the example constraints shown in the previous section. A formal account of the algorithm can be found in the supplement.

Terminology In a validity constraint, a predicate variable occurring on the left side (resp. right side) of the turnstile is said to occur negatively (resp. positively). 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:

- **Type 1** c_1 constrains a predicate variable that c_2 uses.
- **Type 2** c_1 constrains a (non-strict) subset of predicate variables that c_2 constrains.

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 (§ 4.3) is shown below:

$$\begin{tikzcd} \mathbf{C_2}:\{c_5\} \arrow[r] & \mathbf{C_1}:\{c_{2-4}\} & \mathbf{C_3}:\{c_{8-11}\} \arrow[l] \end{tikzcd}$$

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

A dependency DAG makes the dependencies between constraints explicit. Constraints in each set are mutually dependent, and need to be solved simultaneously, whereas constraints in different sets can be solved as per any valid topological ordering of the graph's transpose. Accordingly, we obtain a topological ordering of nodes in the graph G^T_C (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. 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 a set (C) of constraints involves finding assignments for free region (ρ) and predicate (φ) variables in the set. Assignments to region variables can be found via unification (for the examples in § 4.3, we have already performed the unification, so C_{1-3} do not contain free region variables). Once the region variables have been unified, we solve C to find an assignment for free predicate variables. The first step is to reduce the set of constraints into an equivalent single constraint by conjoining antecedents and consequents of all constraints in the set[19]. For the constraint sets C_1, C_2, and C_3, the equivalent constraint are c_{12}, c_{13}, and c_{14}, respectively, shown below:

$$[\mathbf{c_{12}}] : \ \varphi_0 \vdash \rho_0 \succeq \rho^a_0 \wedge \rho_1 \succeq \rho^a_0 \wedge \rho_4 = \rho^a_0 \wedge \rho_2 = \rho_0 \wedge \rho_3 = \rho_1$$
$$[\mathbf{c_{13}}] : \ \varphi_0 \wedge \varphi_1 \vdash \rho_5 = \rho_0$$
$$[\mathbf{c_{14}}] : \ \varphi_1 \wedge \varphi_2 \wedge \pi_0 \succeq \pi_1 \vdash \rho_7 \succeq \rho_6 \wedge \rho_8 \succeq \rho_6 \wedge \rho_7 = \rho_9 \wedge$$
$$\rho_8 = \rho_9 \wedge [\pi_1/\rho^a_2][\pi_1/\rho_6][\pi_0/\rho_{7-9}]\varphi_2$$

The constraint c_{12} is a non-recursive constraint of the form $\phi_{cx} \wedge \varphi \vdash \phi_{cs}$, where ϕ_{cx} and ϕ_{cs} are concrete constraint formulas (ϕ_{cx} is $true$ in this case). Note that the constraint describes an abduction problem, whose solution is the weakest ϕ_{sol} such that $\phi_{cx} \wedge \phi_{sol} \vdash \phi_{cs}$. We solve the problem by casting it as an instance of graph augmentation problem [3]. Our intuition is based on the observation that a constraint can be viewed as a directed graph with region variables as vertices and outlives relationships between them as edges (equality is encoded as a conjunction of symmetric outlives relationships). Let $G(\phi_{cx})$ and $G(\phi_{cs})$ represent such graphs for ϕ_{cx} and ϕ_{cs}, respectively. Now, $\phi_{cx} \vdash \phi_{cs}$ if and only if every pair (ρ_1, ρ_2) of connected vertices in $G(\phi_{cs})$ are also connected in $G(\phi_{cx})$. If ρ_1 and ρ_2 are not connected in

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

[19] Supplement contains formal justification for this step

$G(\phi_{cx})$, then it means that $\phi_{cs} \vdash \rho_1 \succeq \rho_2$ but $\phi_{cx} \nvdash \rho_1 \succeq \rho_2$. Since ϕ_{sol} needs to satisfy $\phi_{cx} \wedge \phi_{sol} \vdash \phi_{cs}$, it must be the case that ρ_1 and ρ_2 are connected in $G(\phi_{cx}) \cup G(\phi_{sol})$. Since ϕ_{sol} must be weakest such formula, it can be obtained by computing the minimum number of edges that need to be added to $G(\phi_{cx})$ such that it is "as connected as" $G(\phi_{cs})$. Using this technique, we obtain the following solutions to φ_0 and φ_1 for constraints in the running example:

$$\varphi_0 \Leftrightarrow \rho_0 \succeq \rho_0^a \wedge \rho_1 \succeq \rho_0^a \wedge \rho_4 = \rho_0^a \wedge \rho_2 = \rho_0 \wedge \rho_3 = \rho_1$$
$$\varphi_1 \Leftrightarrow \rho_5 = \rho_0$$

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

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

Where ϕ_{cx} and ϕ_{cs} are concrete constraint formulas, and F is a substitution function, not necessarily idempotent. Our solution to the constraints of the above form is based on the observation that $H(\varphi) = \phi_{cs} \wedge F(\varphi)$ is a monotone over the complete meet semi-lattice of all possible constraints over the domain of region variables in the program. Hence, by Tarski's fixpoint theorem [15], H has a greatest fixed point. To solve the recursive constraint, we first compute the greatest fixed point (ϕ_f) for H, and then solve the non-recursive constraint $\phi_{cx} \wedge \varphi \vdash \phi_f$ using the graph approach described above. Using this approach, we solve the recursive constraint c_{14}, to obtain the following solution to φ_2:

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

4.5 Higher-Order Type Inference

Our type inference algorithm does not straightforwardly generalize to the higher-order case, due to the lack of principal typing in presence of higher-rank polymorphism. Fortunately, higher-order type inference, in its full generality, is rarely needed in the context of BROOM, where higher-order control flow is predominantly due to the second-order arguments to constructors (For *e.g.*, user-defined functions in dataflow queries, and lambda arguments to the `Region` class constructor). We therefore extend our inference algorithm with a heuristics tailor-made for the aforementioned second-order cases. The simplistic approach seems to suffice in our experience, but, it is always possible that it makes the type checker reject a program that is otherwise valid. In such cases, programmer can help the type checker by writing region annotations only over higher-order arguments. Nonetheless, we note that it is possible to infer most general (if not principal) types for higher-order arguments, by extending our constraint solver with an approach modeled after [14], which lets it compute maximally weak/strong solutions to multiple competing φ's simultaneously.

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. Our implementation is called BROOMC. The input to BROOMC is a program in $[\![FB^+]\!]$, an extended version of $[\![FB]\!]$ that includes assignments, conditionals, loops, more primitive datatypes (*e.g.*, integers), and a null value. Our implementation of region type inference and constraint solving closely follows the description given in Sec. 4. The one difference is that our constraint solver uses an approximation algorithm to compute the minimal graph augmentation (§ 4.4).

```
class LinkedList<T><R5,R4 | R4⪰R5> {
  ListNode<T><R5,R4> head;
  ..
  List<T><R17,R4> rev<R17,R4 | R4⪰R17>(unit u) {
    List<T><R17,R4> xs =
        new List<T><R17,R4>(this.head.val);
    ListNode<T><R5,R4> cur = this.head.next;
    while (!cur == Null) {
      xs.add<R17>(cur.val)
      cur = cur.next;
    }
    return xs;
  }
```

Figure 6: Region-annotated definition of `rev` computed by BROOMC

To evaluate the practical utility of our region type system and type inference, we performed two kinds of experiments. First, we implemented some of the microbenchmarks ($\leq$100 LOC), which are the standard libraries such as pairs, lists, list iterators, etc., in $[\![FB^+]\!]$, and used our inference engine to infer their principal region types. Since a library class is region-oblivious, if it is well-typed as per the core type system, then BROOMC must be able to automatically construct its region-type-annotated definition without fail. As expected, BROOMC was able to infer principal region types for all the library classes, under 10ms. Fig 6 shows the region-type-annotated definition computed for the list reverse method. Observe that BROOMC was able to infer that the list and its data (of type `T`) can be allocated in different regions, as long as the later outlives the former. This allows, for instance, a `preOrder` method to traverse a tree in a transferable region, and return a list of its nodes, where the list itself is allocated in the stack region.

Next, we translated some of the Naiad streaming query operator benchmarks (Naiad vertices) used in [4] to $[\![FB^+]\!]$, and used BROOMC to verify their safety. During the process, we found multiple instances of potential memory safety violations in the $[\![FB^+]\!]$ translation of benchmarks, which we verified to be present in the original C# implementation as well. The cause of all safety violations is the creation of a reference from the outgoing message (a transferable region) to the payload of the incoming message. For example, the implementation of `SelectVertex` contains the following:

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

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

Our experience with Naiad benchmarks suggests that, although our approach cannot statically enforce memory safety, it is nonetheless useful, particularly because the static verification comes at no additional cost to the developer. If the developer is satisfied with the confidence gained by static verification, she may even choose to turn the runtime safety checks off. Since we did not implement code generation in BROOMC, we could not measure the runtime overhead of checks needed to enforce safety. However, the number of LOC peforming operations on `Region` objects relative to the total LOC never exceeds 8% in Naiad benchmarks.

6. Related Work

Tofte and Talpin in [11, 16, 17] introduce the concept of a region type system to statically enforce the safety of their region-based memory management scheme 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, 19, 20]. 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 eliminates the need to write region annotations on types to convince the type checker that the program is safe.

[17] elaborates SML programs to introduce stack regions and do away with GC in a transparent fashion. 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 regions. Our type inference is analogous to their region inference. 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.

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

Cyclone [5] equips C with programmer-managed stack regions, and a typing discipline that statically guarantees the safety of all pointer dereferences. However, types need to be written explicitly. While some region annotations can be inferred, type inference is purely intraprocedural, and cannot infer region types for functions. Later proposal [7] extends Cyclone with dynamic regions, imposes unique pointer restrictions to guarantee safety. Objects that need to be shared within data structures are allowed to have multiple references, and safety for such objects was ensured via runtime reference counting.

An ownership type system for safe region-based memory management in real-time Java has been proposed by [2]. Their source language admits various kinds of regions with lexically-scoped lifetimes (i.e., stack regions) in order to support shared-memory concurrency. 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 theirs. However, our language also admits parametric polymorphism (generics) and higher-order functions. Moreover, we establish type safety and transfer safety results that formalize the guarantees provided by our system. While [2]'s language is explicitly typed, our language comes equipped with full type inference.

[6] proposes a flow-sensitive region-based memory management for first-order programs that proposes to overcome some of the drawbacks of [17] by generalizing [17]'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. [19] extends lambda calculus with first-class regions with dynamic lifetimes, and imposes linear typing to control accesses to regions. Our open/close lexical block for transferable regions traces its orgins to the `let!` expression in [19] and [18], which safely relaxes linear typing restrictions, allowing variables to be temporarily aliased. We don't have linear typing, thus admit unrestricted aliasing. Moreover, [19]'s linear type system is insufficient to enforce the invariants needed to ensure safety under region transfers, such as the absence of references that escape a transferable region.

The idea of using region-based memory management to facilitate the safe transfer of rich data structures between computational nodes has been previously explored by [8] in the context of Scheme language. However, their setting only includes lexically-scoped regions for which Tofte and Talpin-style analysis [17] suffices. In contrast, our language provides first-class support for transferable regions with dynamic lifetimes. We require this generality in order to support streaming query operators, such as the one shown in Fig. 1.

References

[1] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097. URL http://doi.acm.org/10.1145/1640089.1640097. 6

[2] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 324–337, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781168. URL http://doi.acm.org/10.1145/781131.781168. 2, 6

[3] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM Journal on Discrete Math*, 5(1):25–53, 1992. 4.4

[4] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015. URL https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog. 1, 1, 5, 20

[5] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL http://doi.acm.org/10.1145/512529.512563. 6

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

[7] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029883. URL http://doi.acm.org/10.1145/1029873.1029883. 6

[8] E. Holk, R. Newton, J. Siek, and A. Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 141–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660244. URL http://doi.acm.org/10.1145/2660193.2660244. 6

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL http://doi.acm.org/10.1145/503502.503505. 3, 9, 3.2, 10

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

[11] Mads Tofte and J.-P. Talpin. A Theory of Stack Allocation in Polymorphically Typed Languages. Technical Report DIKU-report 93/15, University of Copenhagen, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6564. 6

[12] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602. URL http://doi.acm.org/10.1145/1375581.1375602. 17

[13] Rust. The Rust Programming Language, 2015. URL https://doc.rust-lang.org/book. Accessed: 2015-11-7 13:21:00. 6

[14] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 223–234, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542501. URL http://doi.acm.org/10.1145/1542476.1542501. 4.5

[15] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. 4.4

[16] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: 10.1145/174675.177855. URL http://doi.acm.org/10.1145/174675.177855. 1, 2.1, 6

[17] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997. ISSN 0890-5401. doi: 10.1006/inco.1996.2613. URL http://dx.doi.org/10.1006/inco.1996.2613. 1, 6

[18] P. Wadler. Linear Types Can Change the World! In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, Apr. 1990. North-Holland. 6

[19] D. Walker and K. Watkins. On regions and linear types (extended abstract). In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 181–192, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507658. URL http://doi.acm.org/10.1145/507635.507658. 6

[20] B. N. Yates. A type-and-effect system for encapsulating memory in java. Master's thesis, Department of Computer Science and Information Science, University of Oregon, 1999. 6