Safe Transferable Regions

Gowtham Kaki
Purdue University, USA[1]
gkaki@purdue.edu

G. Ramalingam
Microsoft Research, India
grama@microsoft.com

Abstract

There is an increasing interest in alternative memory management schemes that seek to combine the convenience of garbage collection and the performance of manual memory management in a single language framework. Unfortunately, ensuring safety in presence of manual memory management remains as great a challenge as ever. In this paper, we present a C#-like object-oriented language called Broom that uses a combination of region type system and lightweight runtime checks to enforce safety in presence of user-managed memory regions called *transferable regions*. Unsafe transferable regions have been previously used to contain the latency due to unbounded GC pauses. Our approach shows that it is possible to restore safety without compromising on the benefits of transferable regions. We prove the type safety of Broom in a formal framework that includes its C#-inspired features, such as higher-order functions and generics. We complement our type system with a type inference algorithm, which eliminates the need for programmers to write region annotations on types. The inference algorithm has been proven sound and relatively complete. We describe a prototype implementation of the inference algorithm, and our experience of using it to enforce memory safety in dataflow programs.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies, Software and its engineering → Object oriented languages, Software and its engineering → Data flow languages, Software and its engineering → Software verification and validation

Keywords and phrases Memory Safety, Formal Methods, Type System, Type Inference, Regions, Featherweight Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.11

Acknowledgements We are grateful to Kapil Vaswani, Dimitrios Vytiniotis, Michael Isard, and Steven Hand for their valuable inputs during the initial stages of this project. We would also like to thank multiple anonymous reviewers for their careful scrutiny and feedback on various versions of this work.

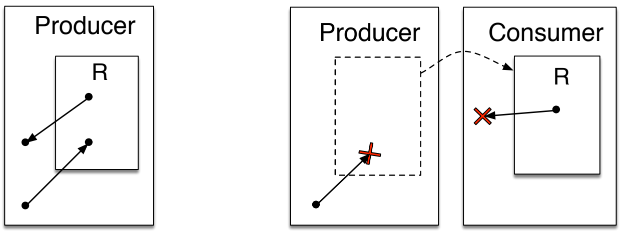
1 Introduction

Computations performed by a network of concurrent communicating actors often involve data transfer between producers and consumers. Consider for example the `SELECT` query operator shown in Fig. 1, which functions as an actor in a dataflow computation involving a network of other such query operators. `SELECT` 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 processed by applying a user-defined function to create a corresponding

[1] Work done during an internship at Microsoft Research, India.

```
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.



(a) Before transfer. **(b)** After transfer.

Figure 2 References in and out of a transferable region (R) become invalid after transfer.

output. Multiple messages with the same timestamp are permitted and messages with different timestamps may arrive out of order. 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.

Performance. When the transferred data structures are large, as is the case with many streaming big-data analysis systems, garbage collection overhead becomes significant [7]. Further, in a distributed dataflow system, the GC pause at one node can have a cascading adverse effect on the performance of other nodes [7, 14]: a GC pause at an upstream actor can block downstream actors that are waiting for messages. However, much of the GC overhead results from the collector performing avoidable or unproductive work. For the example in Fig. 1, GC might repeatedly traverse the `map`, although its objects cannot be collected until a suitable timing message arrives. There has been increasing interest in off-heap memory management for better performance in the context of several systems and languages (e.g., Spark [25], Scala [18], Rust). Several systems (e.g., [3]) resort to using "free object pools" to partially alleviate this problem, in the absence of language support.

Safety. Several of these systems are designed to allow the producer and consumer to execute in the same address space or in different address spaces (as determined by the compiler and runtime). We wish to ensure memory safety in the presence of such transferred data. A 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 transferred data structure become invalid post transfer. If the data structure contains references to objects outside (the transferred data), then such references become invalid in the context of the consumer. Both scenarios are depicted in Fig. 2. While references from within the transferable data to outside can be disallowed in the interest of safety, references into transferable data needs to be allowed before the data is transferred. Allowing such references is crucial, as any non-trivial program creates temporary references to the internal objects of a data-structure.

Safe Transferable Regions. In this paper, we present a language-based approach to cleanly encapsulate transferable data and a safe and efficient implementation of such data based on regions. A region is a block of memory that is allocated and freed in one shot, often in 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 transferable data-structure. 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`.

The use of regions alleviates the performance concern related to garbage collection described earlier. (It also enables more efficient serialization/deserialization of data-structures across address spaces, which is also a significant performance bottleneck in such systems.) However, manual region memory management introduces memory safety issues such as dangling pointers. We present a single type system that simultaneously addresses this safety concern, as well as those described above related to the use of *transferable data*.

Safe region-based memory management using types was pioneered by Tofte and Talpin [20, 21], who use only 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.

Our focus, in this paper, is on first-class *memory-safe* dynamic regions that can be safely transferred across address spaces. We refer to such dynamic regions as *transferable* regions. Our approach is based on a variation of ideas introduced by [23, 10] that combine linear types and regions to support dynamic regions. Unlike the prior work, we ensure memory safety in the presence of transferable regions through a *combination of a static typing discipline and lightweight runtime checks in place of linear types.*

Language. The cornerstone of this 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. By nesting a Tofte-Talpin 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 latter, and any references from the stack region to the transferable region are therefore safe. This is particularly useful, since such stack regions serve as temporary working storage while working with the (opened) transferable regions.

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.

Type System. Formally, our type system introduces *region parameters* (for both classes and methods) and uses constrained parametric polymorphism over these parameters, where the constraints capture outlives constraints between the region parameters. The type system may be seen as a form of ownership type system [5], with a region being the owner of all objects allocated in that region.

Lightweight Runtime Checks. Ensuring memory safety using this approach requires ensuring that the use of transferable regions satisfies certain temporal properties. Firstly, a transferable region should not be transferred/freed inside an open block of that region (i.e, while it is still open). Secondly, a transferred/freed region should not be opened. These are typestate invariants on the transferable region objects, which are hard to enforce statically in 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 define an acceptable state transition discipline for transferable regions (Fig. 4), 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.

Type Inference. One of the key contributions of this paper is a type inference algorithm that eliminates the need for users to write region type annotations. The users write programs in the underlying language that provides primitives for the users to manipulate regions (create, open, free, transfer) and to allocate objects in regions, but has no region type annotations.

Our inference algorithm proceeds in three stages: in the first stage, we elaborate the program by introducing region parameters (for classes and methods); in the second stage we generate a set of constraints that must be satisfied for the program to type check; in the third stage, we solve the set of constraints, inferring the preconditions of all classes and methods (in terms of the expected outlives-constraints between the region parameters). We show that the algorithm is sound, and the stages of constraint generation and solving are complete (i.e., the algorithm is complete relative to the first stage of elaboration).

Evaluation. Our work was inspired by [7], which presents evidence that realistic programs can be written using transferable regions and that this can yield significant performance improvements. While [7] addresses the engineering challenges in extending a managed runtime with transferable regions, it adopts an ad hoc approach insofar as language design is concerned, exposing the region functionality through an unsafe API, and in the process losing the safety guarantees. Our contribution is an alternative approach that is grounded in sound theory and restores the language safety guarantees. The utility of our approach is demonstrated by a prototype implementation of our type inference algorithm that was able to identify unsafe memory accesses among the benchmarks extracted from [7].

Contributions. The paper makes the following contributions:

- We present Broom, a C# -like typed object-oriented language that supports 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 free 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 latter 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. Our inference algorithm is based on a novel constraint solver that performs abduction in a partial-order constraint domain to infer weakest solutions to recursive constraints.
- We establish the soundness and relative completeness of the type inference algorithm.
- 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 [20]’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, as explained soon.

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 order to either read or update or allocate objects in the region. Specifically, the construct "`open rgn as v@R { S }`" does the following: (a). It opens the transferable region handled by `rgn` for allocation (i.e., makes it the current allocation context), (b). binds the identifier `R` to this open region, and (c). initializes the newly introduced local variable `v` to refer to the root object of the region. The `@R` part of the statement is optional and may be omitted. The `open` construct is intended to simplify the problem of ensuring memory safety, as will be explained soon. We refer to a transferable region that has not been opened as a *closed* region. 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. 4. Enforcement of this discipline is done at runtime.

Opening a region also makes it the current allocation context. Thus, given a C# library function `f` (that makes no use of BROOM's region constructs), opening a region `R` and invoking `f` has the effect that all objects created by this invocation are allocated in the region `R`.

Motivating Example. Fig 3 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). On Line 7, we create a new region to store the output for time `t`, initializing it to contain an empty list. On Lines 8 and 9, we open the input region `inRgn` followed by creating a new stack region `R0`. The temporary objects created by the iteration on line 10, for example, will be allocated in the stack region `R0` that lives just long enough. We open the desired output region on line 11, so that the new output objects created by the invocation of `selector` on line 12 are allocated in the output region. Finally, the input region is freed on line 14. 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.

Cloning. Note that in the example from Fig. 3 the object returned by the `selector` (on Line 12) 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

```
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>> (() => new List<TOut>());
    open inRgn as inList {
      letregion R0 {
        foreach (TIn input in inList) {
          open map[t] as outList {
            TOut output = selector(input);
            outList.add(output); } } } }
    inRgn.free();
  }
  void onNotify(Time t) {
     Region<List<TOut>> outRgn = map[t];
     map.Remove(t);
     outRgn.transfer(successorId);
  }
}
```

Figure 3 SELECT dataflow operator in BROOM.

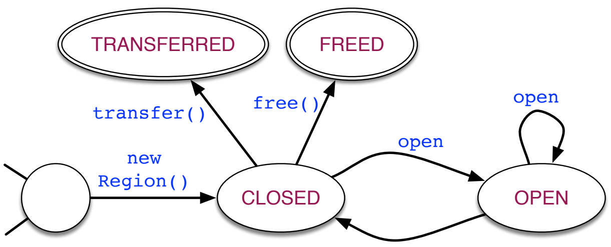


Figure 4 The lifetime of a dynamic (transferable) region in BROOM.

$$\pi \in \texttt{Region identifiers} \qquad \rho \in \texttt{Region variables} \qquad a, b \in \texttt{Type variables}$$
$$m \in \texttt{Method names} \qquad x, y, f \in \texttt{Variables and fields}$$

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

Figure 5 FEATHERWEIGHT BROOM: syntax.

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. While the invariant is easily stated, enforcing it in the presence of first-class dynamic regions, parametric polymorphism (generics), and higher-order functions requires new reasoning principles that we formally develop in this section. We introduce FEATHERWEIGHT BROOM (FB), an 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) [13] formalism, and reuses notations and various definitions from [13], such as the definition of type well-formedness for the core (region-free) language. (The language used in Section 2 is essentially a version of FGJ without region types and with some syntactic sugar.)

3.1 Syntax

Fig 5 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 (the symbol $\lhd$ should be read *extends*, and the symbol $\succeq$ stands for *outlives*):

```
class Pair<a ◁ Object, b ◁ Object>
          <ρ0, ρ1, ρ2 | ρ1 ⪰ ρ0 ∧ ρ2 ⪰ ρ0> ◁ Object<ρ0> {
  a@ρ1 fst;
  b@ρ2 snd;
  a@ρ1 getFst() { return this.fst; }
}
```

Note that we elide showing constructors since they are uninteresting from the type system's standpoint; their behavior in FB is same as in FGJ.

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 (ρ_0 in the above example) as the *allocation region* of the class: it serves to identify the region where an instance of the class is allocated[2]. An object in FB can contain fields referring to objects allocated in regions ($\overline{\rho}$) other than its own allocation region (ρ), provided that the former outlive the latter (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*.

[2] In general, $\overline{\rho}$ denotes the sequence $\rho_1\rho_2...$ of region parameters of a class or a method. In some cases, we also represent region parameters as $\rho\overline{\rho}$ to clearly distinguish between the allocation region parameter (ρ) and the rest.

To construct objects of the `Pair` class, its type and region parameters need to be instantiated with core types (T) and region annotations[3] (r), 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,π1,π0> (fst,snd);
```

In the above code, the instantiation of ρ_0 and ρ_1 with π_1, and ρ_2 with π_0 is allowed because (a) π_0 and π_1 are live during the instantiation, and (b). $\pi_0 \succeq \pi_1$ and $\pi_1 \succeq \pi_1$ (since outlives is reflexive). Observe that the region type of `p` conveys the fact that (a). it is allocated in region π_1, and (b). it holds references to objects allocated in region π_0 and π_1. In contrast, if we choose to allocate the `snd` object also in π_1, then `p` would be contained in π_1, and its region type would be `Pair<Object,Object>`$\langle\pi_1,\pi_1,\pi_1\rangle$, which we abbreviate as `Pair<Object,Object>`$@\pi_1$. In general, we treat $B\langle\overline{T}\rangle@\pi$ as being equivalent to $B\langle\overline{T}\rangle\langle\overline{\pi}\rangle$. Region annotation on type a, where a is a type variable, assumes the form $a@\pi$. If a is instantiated with `Pair<Object,Object>`, the result is the type of a `Pair` object contained in π.

Classes in FB are independently parameterized over types and regions. While this design decision has a downside in that it allows type variables to only denote the types of objects contained in a single region, it yields benefits that outweigh the costs. In particular, it lets us support region-polymorphic higher-order functions as class fields. This allows, for example, a generic class, whose type parameters are a and b, to contain a region-polymorphic function of type $\langle\rho_0, \rho_1, \rho_2 \,|\, \rho_1 \succeq \rho_0 \wedge \rho_2 \succeq \rho_0\rangle(a@\rho_1, b@\rho_2) \rightarrow$ `Pair` $\langle a, b\rangle\langle\rho_0, \rho_1, \rho_2\rangle$ as its field. Such region-polymorphic higher-order fields are used frequently by the dataflow operators, which apply them in the context of various regions (*e.g.,* see Fig. 3). Keeping type and region parameterizations separate also simplifies the type system so that inference becomes practical (Sec. 4). The need for polymorphism at the field level is also why FB treats function closures specially, rather than as objects of type `Func` as in C#.

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 (i.e., a method has at least one region parameter). Region parameters, like those on classes, are qualified with constraints (ϕ). 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 ($e\langle\overline{r}\rangle(\overline{e})$) to define and apply functions (lambdas). Functions, like methods, exhibit constrained region polymorphism, as evident in their arrow region type ($\langle\overline{\rho} \,|\, \phi\rangle\overline{\tau} \xrightarrow{r} \tau$). A function, like a method, is necessarily polymorphic w.r.t its allocation context. 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 r annotation above the arrow in the arrow type denotes the allocation region of the corresponding closure.

Note that mutable object fields are conspicuously absent from the FB model (and also the FGJ model in general), but this isn't a major shortcoming considering that constructor application, which happens during a new object creation, includes assignments to the object fields (see FGJ [13]). Thus the type system is already obligated to handle unsafe assignments.

[3] Region annotations (r) include region variables (ρ) and region identifiers (π). Region identifiers are to region variables, as types (T) are to type variables (a)

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 an illustrative subset of such rules[4]. The rules refer to a context ($\mathcal{A}$), which is a tuple of:

- A set ($\Delta \in 2^r$) of regions that are estimated to be live,
- A finite map ($\Theta \in a \mapsto K$) of type variables to their *bounds*, i.e., classes they are declared to extend (this artifact is inherited from FGJ), and
- A constraint formula (Φ) that captures the outlives constraints on regions in Δ.

In addition, the context for the expression typing judgment also includes (a). a type environment ($\Gamma \in x \mapsto \tau$) that contains the type bindings for variables in scope, and (b). the region (r) that serves as the allocation context for the expression being type checked. Like the judgments in FGJ [13], all the judgments defined by the rules in Fig. 6 are implicitly parameterized on a class table ($CT \in cn \mapsto D$) that maps class names to their definitions in FB.

The well-formedness judgment on region types ($\mathcal{A} \vdash \tau$ `ok`) makes use of the well-formedness and subtyping judgments on core types. We use a double-piped turnstile ($\Vdash$) for judgments in FGJ [13], and a simple turnstile ($\vdash$) for those in FB. The class table ($[\![CT]\!]$) for FGJ 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 ($B\langle\overline{T}\rangle\langle\overline{r}\rangle$) is responsible for enforcing the safety property that prevents objects from containing unsafe references. It does so by insisting that regions $\overline{r}$ satisfy the constraints (ϕ) imposed by the class on its region parameters. The latter is enforced by checking the validity of ϕ, with actual region arguments substituted for formal region parameters, under the conditions (Φ) 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, $T@r$ is a well-formed region type if r is a valid region. The type `Region`$\langle T\rangle\langle r\rangle$ is well-formed only if $r = \pi_\top$, 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 latter relies on an auxiliary definition[5] called fields (undefined for `Region` class) 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. The type rule for the field access expression ($e.f_i$) also uses fields and another definition called bound, which returns the bound of a type variable (bound is an identity function for concrete types).

The type rule for the `new Region` expression expects the `Region` class's constructor to be called with a nullary function that returns a value in its allocation context. This ensures that the value returned by the function stores no references to objects allocated

[4] Full formal development can be found in the appendix
[5] All auxiliary definitions we use in this exposition originate from the FGJ calculus.

Type Well-formedness $\boxed{\mathcal{A} \vdash \tau\ \texttt{ok}}$

$$\frac{CT(B) = \texttt{class}\ B\langle \overline{a} \triangleleft \overline{K}\rangle\langle \overline{\rho} \,|\, \phi\rangle \triangleleft N\{...\} \quad \overline{r} \in \Delta \quad \Theta \Vdash B\langle \overline{T}\rangle\ \texttt{ok} \quad \Phi \vdash [\overline{r}/\overline{\rho}](\phi)}{(\Delta, \Theta, \Phi) \vdash B\langle \overline{T}\rangle\langle \overline{r}\rangle\ \texttt{ok}}$$

$$\frac{\Theta \Vdash T\ \texttt{ok} \quad r \in \Delta \quad \Theta \Vdash T <: \texttt{Object}}{(\Delta, \Theta, \Phi) \vdash T@r\ \texttt{ok}} \qquad \frac{\Theta \Vdash T\ \texttt{ok} \quad \mathcal{A} = (\Delta, \Theta, \Phi)}{\mathcal{A} \vdash \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle\ \texttt{ok}}$$

Expression Typing $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau}$

$$\frac{\begin{array}{c}\mathcal{A}, \Gamma, r \vdash \overline{e} : \overline{\tau} \\ \mathcal{A} \vdash N\ \texttt{ok} \quad \mathsf{fields}(N) = \overline{f} : \overline{\tau}\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{new}\ N(\overline{e}) : N} \qquad \frac{\mathcal{A}, \Gamma, r \vdash e : \langle\rho\rangle\,\texttt{unit} \xrightarrow{r} T@\rho}{\mathcal{A}, \Gamma, r \vdash \texttt{new}\ \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle(e) : \texttt{Region}\,\langle T\rangle\langle \pi_\top\rangle}$$

$$\frac{\begin{array}{c}\mathcal{A}, \Gamma, r \vdash e : \tau' \\ \overline{f} : \overline{\tau} \ = \ \mathsf{fields}(\mathsf{bound}_{\mathcal{A}.\Theta}(\tau'))\end{array}}{\mathcal{A}, \Gamma, r \vdash e.f_i : \tau_i} \qquad \frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{\pi\}, \Theta, \Phi \wedge \phi) \\ \pi \notin \Delta \quad \phi = \Delta \succeq \pi \quad \mathcal{A}', \Gamma, \pi \vdash e : \tau \quad \mathcal{A} \vdash \tau\ \texttt{ok}\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{letregion}\ \pi\ \texttt{in}\ e : \tau}$$

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

$$\frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}, \Gamma, r \vdash e_0 : \tau \\ \mathsf{mtype}(m, \mathsf{bound}_\Theta(\tau)) = \langle \rho\overline{\rho} \,|\, \phi\rangle\overline{\tau^1} \rightarrow \tau^2 \\ r, \overline{r} \in \Delta \quad \mathcal{A} \vdash \langle \rho\overline{\rho} \,|\, \phi\rangle\overline{\tau^1} \rightarrow \tau^2\ \texttt{ok} \\ \mathcal{A}, \Gamma, r \vdash \overline{e} : [r\overline{r}/\rho\overline{\rho}]\,\overline{\tau^1} \quad \Phi \vdash [r\overline{r}/\rho\overline{\rho}]\,\phi\end{array}}{\mathcal{A}, \Gamma, r \vdash e_0.m\langle r\overline{r}\rangle(\overline{e}) : [r\overline{r}/\rho\overline{\rho}]\,\tau^2} \qquad \frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad r', r, \overline{r} \in \Delta \\ \mathcal{A}, \Gamma, r \vdash e : \langle \rho\overline{\rho} \,|\, \phi\rangle\overline{\tau^1} \xrightarrow{r'} \tau^2 \\ \Phi \vdash [r\overline{r}/\rho\overline{\rho}]\phi \quad \mathcal{A}, \Gamma, r \vdash \overline{e} : [r\overline{r}/\rho\overline{\rho}]\overline{\tau^1}\end{array}}{\mathcal{A}, \Gamma, r \vdash e\langle r\overline{r}\rangle(\overline{e}) : [r\overline{r}/\rho\overline{\rho}]\tau^2}$$

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

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

Figure 6 Featherweight Broom: select type rules.

elsewhere, including the top region ($\pi_\top$), thus preventing cross-region references originating from transferable regions. The body of the function might however create new regions while execution, but this is not a problem as long as such regions, and objects allocated in them, don't find their way into the result of its evaluation.

The type rule for the `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 existing outlives relationships on an eponymous region. The expression (e) under `letregion` is typechecked with the new region as its allocation context, under the assumption that the region is live ($\Delta \cup \{\pi\}$) 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.

The rule for the `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. The newly opened region becomes the allocation context for e, which is type checked under an environment (Γ) extended with the type binding for the root object.

The type rule for the 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 extended with the function's region parameters ($\rho\overline{\rho}$). Unlike a typical object, a function closure might capture references that may become unsafe if the closure escapes its allocation context. FB prevents this scenario by requiring the function closure to always be allocated in the current allocation context (r).

The type rule for method application uses an auxiliary definition mtype that gives the type of a class method. Method application involves instantiation of method's region parameters, and the type rule requires the first region parameter to be instantiated with the current allocation context (r). The type rule for function application does similarly. This requirement ensures the safety of closures as demonstrated by the following example.

Consider a method m with two region parameters - ρ_0 and ρ_1 (i.e., $m\langle\rho_0, \rho_1\rangle(\ldots)$). Suppose the method immediately returns a function closure that contains a reference to (an object in) ρ_1. Since function closures are always allocated in the current allocation context, the closure is allocated in ρ_0, the allocation context for the method body (the rule for methods discussed below). Now, consider the following use of the method:

```
letregion R0 in
  let f = letregion R1 in m<R0,R1>()
  in f()
```

Observe that the first region parameter of `m` is instantiated with `R0`, while the allocation context for the method call is `R1`. The function `f` returned by `m` stores a reference to `R1`, which is unsafe when `f` is finally called. The type system fortunately disallows this scenario by enforcing certain restrictions during method and function applications as described above.

The method well-formedness rule makes use of an auxiliary definition override that judges whether the current method is a valid overriding of any eponymous method from the super class. The type environment is extended with a binding that binds the `this` keyword to the type of the current class. Note that the type of the current method as accessed via `this` (i.e., `this.m`) is region-parametric, thus admitting region-polymorphic recursion (i.e., recursive call to `m` can have different region arguments than `m`).

3.3 Operational Semantics and Type Safety

The operational semantics of FB describes a non-trivial runtime component that introduces memory regions and performs run-time verification of their typestate. Fig. 7 shows the additional language constructs of FB that manifest only at run-time. A location (`l`) abstracts

$$
\begin{array}{llllll}
\mathtt{l} & \in & \texttt{Memory locations} & & \\
r & \in & \texttt{Region annotations} & ::= & \mathtt{l} \mid \ldots \\
s & \in & \texttt{Region Typestate} & ::= & \square \mid \blacksquare \mid \times \\
e & \in & \texttt{Expressions} & ::= & \texttt{letd l in } e \mid \texttt{opened l}(s) \texttt{ in } e \mid \bot \mid \ldots
\end{array}
$$

Figure 7 Featherweight Broom: extended syntax to accommodate run-time constructs.

all the memory locations associated with a region (i.e., each memory region is associated with a single location). A transferable region can be in one of three possible states: closed ($\square$), open/live ($\blacksquare$), and transferred/freed ($\times$). Constructs `letd` and `opened` are the run-time manifestations of `letregion` and `open`; `letregion` reduces to `letd` while allocating a (static) region, and `open` reduces to `opened` while opening a (transferable) region. An `opened` expression is tagged with the typestate (s) of the transferable region before it is opened (as Fig. 4 shows, a transferable region can be `open`'d from either an open state or a closed state). Special value $\bot$ denotes an exception.

Operational semantics of FB defines a four-place small-step reduction relation of the form shown below:

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

The typestate of regions is tracked by a finite map (Σ) from locations to typestates (Fig. 4). The reduction relation relates an expression (e) and a typestate map (Σ) to a reduced expression (e') and an updated typestate map (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 that is not mapped to an appropriate typestate 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. 4, then it raises an exception value ($\bot$). An illustrative subset of operational semantics that formalize the intuitions described above is shown in Fig. 8. Fig. 15 of the appendix contains rest of the rules.

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

$$v \;\in\; \texttt{values} \;::=\; \texttt{new } B\langle\overline{T}\rangle\langle\overline{\mathtt{l}}\rangle(\overline{v}) \;\mid\; \lambda@\mathtt{l}\langle\overline{\rho}\,|\,\phi\rangle(\overline{\tau}\;\overline{x}).e \;\mid\; \texttt{new Region}\langle T\rangle\langle\mathtt{l}_{\top}\mathtt{l}\rangle(v)$$

The first two forms are obtained by using locations for region annotations in `new` and lambda expressions. The last form is the value that `new Region` expressions gets evaluated to; the location $\mathtt{l}_{\top}$ stands for the region $\pi_{\top}$, and $\mathtt{l}$ is the location of the newly allocated transferable region. The following type safety theorem shows that a well-typed program will never attempt to dereference an "invalid" reference (a reference to an object in a region that has been transferred or freed):

▶ **Theorem 1** (Type Safety). *$\forall e, \tau, \Delta, \Sigma$, such that* consistent$(\Delta, \Sigma)$ *and* $\Delta \vdash \Phi$ *ok, if* $(\Delta, \cdot, \Phi), \cdot \vdash e : \tau$*, then either e is a value, or e raises an exception ($(e, \Sigma) \longrightarrow \bot$), or there exists an e' and a Σ' such that* consistent(Δ, Σ') *and* $(e, \Sigma) \longrightarrow (e', \Sigma')$ *and* $(\Delta, \cdot, \Phi), \cdot \vdash e' : \tau$.

The relation consistent relates Δ and Σ only if both make consistent assumptions about liveness of regions. Since Δ is consistent with both Σ and Σ', the theorem also captures the key property of operational semantics that no live region is ever freed or transferred.

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

$$[\textsc{LetRegionBegin}] \quad \frac{\mathtt{l} \notin \mathit{dom}(\Sigma) \quad \Sigma' = \Sigma[\mathtt{l} \mapsto \blacksquare]}{(\mathtt{letregion}\ r\ \mathtt{in}\ e, \Sigma) \longrightarrow (\mathtt{letd}\ \mathtt{l}\ \mathtt{in}\ [\mathtt{l}/r]e, \Sigma')}$$

$$[\textsc{LetRegion}] \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(\mathtt{letd}\ \mathtt{l}\ \mathtt{in}\ e, \Sigma) \longrightarrow (\mathtt{letd}\ \mathtt{l}\ \mathtt{in}\ e', \Sigma')}$$

$$[\textsc{LetRegionEnd}] \quad \frac{\Sigma' = \Sigma[\mathtt{l} \mapsto \times]}{(\mathtt{letd}\ \mathtt{l}\ \mathtt{in}\ v, \Sigma) \longrightarrow (v, \Sigma')}$$

$$[\textsc{NewRegion}] \quad \frac{\mathtt{l} \notin \mathit{dom}(\Sigma) \quad \Sigma' = \Sigma[\mathtt{l} \mapsto \square]}{(\mathtt{new\ Region}\,\langle T\rangle\langle \pi_{\top}\rangle(v), \Sigma) \longrightarrow (\mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(v\langle \mathtt{l}\rangle()), \Sigma')}$$

$$[\textsc{NewRegion}] \quad \frac{(e, \Sigma[\mathtt{l} \mapsto \blacksquare]) \longrightarrow (e', \Sigma')}{(\mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(e), \Sigma) \longrightarrow (\mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(e'), \Sigma')}$$

$$[\textsc{Open}] \quad \frac{v_a = \mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(v) \quad \Sigma(\mathtt{l}) = \square\ \mathtt{or}\ \Sigma(\mathtt{l}) = \blacksquare \quad \Sigma' = \Sigma[\mathtt{l} \mapsto \blacksquare]}{(\mathtt{open}\ v_a\ \mathtt{as}\ x@r\ \mathtt{in}\ e_b, \Sigma) \longrightarrow (\mathtt{opened}\ \mathtt{l}(\Sigma(\mathtt{l}))\ \mathtt{in}\ [v/x][\mathtt{l}/r]e_b, \Sigma')}$$

$$[\textsc{Opened}] \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(\mathtt{opened}\ \mathtt{l}(s)\ \mathtt{in}\ e, \Sigma) \longrightarrow (\mathtt{opened}\ \mathtt{l}(s)\ \mathtt{in}\ e', \Sigma')}$$

$$[\textsc{OpenEnd}] \quad \frac{\Sigma' = \Sigma[\mathtt{l} \mapsto s]}{(\mathtt{opened}\ \mathtt{l}(s)\ \mathtt{in}\ v, \Sigma) \longrightarrow (v, \Sigma')}$$

$$[\textsc{OpenTransferred}] \quad \frac{v_a = \mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(v) \quad \Sigma(\mathtt{l}) \neq \square\ \mathtt{and}\ \Sigma(\mathtt{l}) \neq \blacksquare}{(\mathtt{open}\ v_a\ \mathtt{as}\ x@r\ \mathtt{in}\ e_b, \Sigma) \longrightarrow \bot}$$

$$[\textsc{Transfer}] \quad \frac{\Sigma(\mathtt{l}) = \square \quad \Sigma' = \Sigma[\mathtt{l} \mapsto \times]}{((\mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(v)).\mathsf{transfer}(\ldots), \Sigma) \longrightarrow (\,()\,, \Sigma')}$$

$$[\textsc{TransferOpened}] \quad \frac{\Sigma(\mathtt{l}) = \blacksquare}{((\mathtt{new\ Region}\,\langle T\rangle\langle \mathtt{l}_{\top}\mathtt{l}\rangle(v)).\mathsf{transfer}(\ldots), \Sigma) \longrightarrow \bot}$$

Figure 8 Featherweight Broom: a select subset of operational semantics.

Integrating regions with GC heap. FB stores the region handlers in a distinguished top region, which abstracts the GC heap. Type system prevents transferable regions from storing references into the GC heap. A static region however can store references to region handles, which need to be taken into account while GC'ing the heap. Traversing the region memory to identify references into the GC heap unfortunately defeats the purpose of regions. The solution we adopt is to disable collecting region handles as long as any static region is open. Since static regions are intended to store temporary objects while populating a dynamic region, their lifetimes are short enough to let region handles to be GC'd.

4 Type Inference

Broom's region type system imposes an annotation burden, and manually annotating C# standard libraries with region types can be tedious. We now present our region type inference algorithm that eliminates the need to write region type annotations. 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. For simplicity, we assume that each `letregion` and `open` construct in the input program introduces a distinct region identifier π.

Overview. We now present a high-level outline of the type inference algorithm. The algorithm consists of the following steps:

1. *Region Parameterization.* The first step elaborates the input program by introducing *formal region parameters* (for each class and method), and *region variables* (representing yet undetermined *actual region parameters*). We also introduce for each class and method, a *predicate variable* (φ) to denote an undetermined set of outlives-constraints over the region parameters of that class/method.
2. *Constraint Generation.* In the second step, we analyze the program to generate a set of constraints (over the region identifiers and the predicate variables) that must hold (as per the static semantics in Fig. 6).
3. *Constraint Solving.* We solve the generated set of constraints using our fixpoint constraint solving algorithm, which reduces the constraint solving problem to an abduction problem. If the original program in $[\![FB]\!]$ contains unsafe references, for example, a reference from a transferable region to a stack region, then the constraints generated during the elaboration are not satisfiable. In such a case, the solver fails to solve the constraints.
4. If the solver succeeds, it returns a solution η consisting of a pair of substitution functions η_R and η_P for free region and predicate variables, respectively, introduced in step 1. We apply these substitutions to the elaborated program to produce the final program.

We later on establish the soundness of the type inference algorithm, and the completeness of the constraint-generation and constraint-solving steps (steps 2 and 3 above).

4.1 Region Parameterization for Classes

Region parameterization is an iterative process involving the following three steps, the first two of which are mutually dependent on each other.

Introduction of Formal Region Parameters. For every class `C`, we identify a sequence of formal region parameters $\pi_0, \cdots \pi_n$ that `C` should be parametric over.

Introduction of Actual Region Parameters. We then replace every instance of class `C` in the program by an instance `C`$\langle \rho_0, \cdots, \rho_n \rangle$, where $\rho_0, \cdots, \rho_n$ are fresh identifiers denoting actual region parameters.

Predicate Variable Introduction. For every class `C`, we introduce a fresh predicate variable φ, which represents the yet undetermined outlives-constraints between the formal region parameters of class `C`.

We identify the region parameters of classes as follows.

Non-Recursive Classes. The class `Object` is defined to have a single region parameter π_0 (the allocation region). The region parameters for any other non-recursive class `C` is determined only after the region parameters of any class that `C` depends on have been determined: this includes the base-class `B` of `C` and the class (type) of any of its fields. We replace every dependee type `T` in `C` by its instantiated type, using fresh region parameters as needed. The sequence of region parameters for `C` is defined to be the sequence of region parameters for the base class `B` concatenated with the list of all fresh region parameters introduced while instantiating the types of the fields in the class. (The class inherits its allocation region from its base class. Note that if a class does not specify an explicit base class, it has an implicit base class `Object`.)

This transformation is illustrated below, using a non-generic `Pair` class:

```
class Pair ◁ Object {
  Object fst;
  Object snd;
}
```

$\Rightarrow$

```
class Pair ⟨ρ0, ρ1, ρ2 | φ⟩ ◁ Object⟨ρ0⟩ {
  Object⟨ρ1⟩ fst;
  Object⟨ρ2⟩ snd;
}
```

Recursive Classes. The region parameters for a recursive class is computed in a similar fashion, with the following difference: any recursive field is ignored while instantiating region parameters for the fields of the class, and the region parameters of the recursive class are computed as before. We then do parameter instantiation for all recursive fields, such that their region annotations (the actual region parameters) are exactly the same as the (formal) region parameters of the class. The following example illustrates this for a non-generic `List` class. The resulting class represents a linked list with spine in the region ρ_0 and data objects in the region ρ_1.

```
class List ◁ Object {
  Object data;
  List next;
}
```

$\Rightarrow$

```
class List ⟨ρ0, ρ1 | φ⟩ ◁ Object⟨ρ0⟩ {
  Object⟨ρ1⟩ data;
  List⟨ρ0, ρ1⟩ next;
}
```

The above technique can be extended to mutually recursive classes in a straightforward manner, by simultaneously parameterizing them (and then instantiating them).

Type-Parametric Classes. The type parameter `T` of a class `C` is instantiated as `T`$@\rho$ using a single region parameter ρ. (This can be extended to use the bound specified for `T`, if any.)

Function Types. Since FB is higher-order, fields of function type are allowed.We explain how the parameter instantiation step instantiates function types below, after discussing parameterization for methods.

4.2 Region Parameterization for Methods and Function Types

As the next step, we introduce region parameters for every method. We do this by instantiating the types of all parameters and the return value (of the method) using fresh region identifiers (as explained previously), and then generalizing these region identifiers as formal region parameters of the method. In addition, a fresh region identifier is introduced to represent the allocation region. We also introduce a fresh predicate variable φ for every method, just as we did for each class. Thus, the method

```
Object m (List x) {...}
```

is instantiated as

`Object`$\langle\rho_3\rangle$ `m`$\langle\rho_0, \rho_1, \rho_2, \rho_3\rangle$ `(List`$\langle\rho_1, \rho_2\rangle$ `x) {...}`

We then consider every method invocation in the program, and introduce fresh region variables representing the (yet unknown) actual region parameters for this particular invocation. We similarly perform instantiation for every constructor invocation of the form `new@`π_0 `T(...)`, by instantiating the type `T` as before, turning it into `new T`$\langle\pi_0, \rho_1, \cdots, \rho_n\rangle$`(...)`, where $\rho_1, \cdots, \rho_n$ are fresh region variables. Note that the programmer typically specifies the

$$\pi \in R\ (\texttt{Region constants})\quad \nu \in V\ (\texttt{Region vars})\quad \varphi \in P\ (\texttt{Predicate vars})\quad \Delta \subseteq R$$

$$\begin{array}{rcl}
\rho \in \texttt{Region Identifiers} & ::= & \pi \mid \nu \\
\phi \in \texttt{Region Constraint} & ::= & \mathit{true} \mid \rho \succeq \rho \mid \phi \wedge \phi \\
F \in \texttt{Substitution} & ::= & \cdot \mid [\rho/\rho]F \\
\ell \in \texttt{Antecedent} & ::= & \phi \mid \varphi \mid \varphi \wedge \phi \\
r \in \texttt{Consequent} & ::= & \phi \mid F(\varphi) \\
\texttt{Constraint} & ::= & \ell \vdash r \mid \nu \in \Delta \mid \Delta \vdash \varphi\ \texttt{ok}
\end{array}$$

Figure 9 Syntax of constraints.

region π_0 where the object is to be allocated. If the programmer does not specify this, it is allocated in the allocation-context region by default. Region variables are introduced only for the other region parameters.

Function types (of fields and parameters) are instantiated just like methods above, (and the region where the closure is allocated is determined just as for other objects). For example, the function type `List` $\rightarrow$ `Object` is instantiated as $\langle \rho_0, \rho_1, \rho_2, \rho_3 \mid \varphi \rangle$ `List` $\langle \rho_1, \rho_2 \rangle \xrightarrow{\pi}$ `Object` $\langle \rho_3 \rangle$. Note that the newly introduced region identifiers are generalized as formal region parameters of the function type. This is a (heuristic) choice made in the case of higher order functions. Consider a higher order function f with a function typed parameter g. The fresh region identifiers introduced while instantiating the type of g could be alternatively generalized as formal region parameters of f, but we choose to generalize them as formal region parameters of g. We will discuss this aspect again later.

4.3 Constraint Generation

The constraint generation algorithm mimics the static type checker, but accumulates constraints that must hold for the type checking to succeed.

Syntax of Constraints. (See Fig. 9.) The constraints are expressed using a set R of region constants, a set V of region variables, and a set P of predicate variables.

Recall that a *region constant* may be either (a) a *formal region parameter* of a class or method, or (b) a *static region identifier* introduced by a `letregion` construct, or (c) an *open transferable region identifier* introduced by an `open` construct. A *region variable* is introduced to represent an unknown *actual* region parameter of a method invocation or object allocation, and the constraint-solver, if successful, will bind each region variable to a region constant.

A *region-constraint* ϕ consists of a conjunction of *outlives-constraints* of the form $\rho_1 \succeq \rho_2$. A predicate variable φ is introduced to represent, *e.g.*, the unknown precondition of a method. The constraint-solver will end up binding it to a *region-constraint* ϕ over a set of fixed formal region parameters. Our constraints also make uses of *pending substitutions* F: A pending substitution serves to bind formal region parameters in φ to the actual region parameters used in a particular context: E.g., in the validity constraint $\pi_1 \succeq \pi_2 \vdash [\pi_1/\rho_1][\pi_2/\rho_2]\varphi$, the pending substitution is $[\pi_1/\rho_1][\pi_2/\rho_2]$.

The constraints are primarily of the form $\varphi_i \wedge \phi_{cx} \vdash \phi_{cs}$ or $\varphi_i \wedge \phi_{cx} \vdash F_j(\varphi_j)$. Here, φ_i is a predicate variable (representing the precondition of a method to be determined), ϕ_{cs} is a

Expression Typing Constraint Generation $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau \lhd C}$

[NEW]
$$\frac{\mathcal{A}, \Gamma, r \vdash \overline{e} : \overline{\tau} \lhd C_1 \quad \mathcal{A} \vdash N\ \texttt{ok} \lhd C_2 \quad \mathsf{fields}(N) = \overline{f} : \overline{\tau}}{\mathcal{A}, \Gamma, r \vdash \texttt{new}\, N(\overline{e}) : N \lhd C_1 \cup C_2}$$

[NEWREGION]
$$\frac{\mathcal{A}, \Gamma, r \vdash e : \langle\rho\rangle\, \texttt{unit} \xrightarrow{r} T@\rho \lhd C}{\mathcal{A}, \Gamma, r \vdash \texttt{new Region}\, \langle T\rangle\langle\pi_\top\rangle(e) : \texttt{Region}\, \langle T\rangle\langle\pi_\top\rangle \lhd C}$$

[LETREGION]
$$\frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \mathcal{A}' = (\Delta \cup \{\pi\}, \Theta, \Phi \wedge (\Delta \succeq \pi)) \\ \mathcal{A}', \Gamma, \pi \vdash e : \tau \lhd C_1 \quad \mathcal{A} \vdash \tau\ \texttt{ok} \lhd C_2\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{letregion}\ \pi\ \texttt{in}\ e : \tau \lhd (C_1 \cup C_2)}$$

[FNAPPLY]
$$\frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad C_1 = \{r\overline{r} \in \Delta\} \quad \mathcal{A}, \Gamma, r \vdash e : \langle\rho\overline{\rho} \mid \phi\rangle\overline{\tau^1} \xrightarrow{r} \tau^2 \lhd C_2 \\ C_3 = \{\Phi \vdash [r\overline{r}/\rho\overline{\rho}]\phi\} \quad \mathcal{A}, \Gamma, r \vdash \overline{e} : [r\overline{r}/\rho\overline{\rho}]\overline{\tau^1} \lhd C_4\end{array}}{\mathcal{A}, \Gamma, r \vdash e\langle r\overline{r}\rangle(\overline{e}) : [r\overline{r}/\rho\overline{\rho}]\tau^2 \lhd \cup_{i=1}^{4} C_i}$$

[OPEN]
$$\frac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \mathcal{A}, \Gamma, r \vdash e_a : \texttt{Region}\, \langle T\rangle\langle\pi_\top\rangle \lhd C_1 \\ (\Delta \cup \{\pi\}, \Theta, \Phi), \Gamma[y \mapsto T@\pi] \vdash e_b : \tau \lhd C_2 \quad \mathcal{A} \vdash \tau\ \texttt{ok} \lhd C_3\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{open}\ e_a\ \texttt{as}\ y@\pi\ \texttt{in}\ e_b : \tau \lhd (C_1 \cup C_2 \cup C_3)}$$

Figure 10 Constraint generation rules part 1.

region-constraint that is *required* to hold at a particular program point (within the method), and ϕ_{cx} is a region-constraint that is *known* to hold at that program point. Constraints of the form $\varphi_i \wedge \phi_{cx} \vdash F_j(\varphi_j)$ are generated by an invocation of a method with precondition φ_j.

We use *well-formedness constraints* of the form $\rho \in \Delta$ to restrict the domain of unification for a region variable (ρ) to a constant set $\Delta = \{\pi_1, \cdots, \pi_n\}$ of regions in scope, and well-formedness constraints of form $\Delta \vdash \varphi\ \texttt{ok}$ to restrict the domain of a predicate variable (φ) to the set of all possible region-constraint formulas over a fixed set of regions ($\Delta = \{\pi_1, \cdots, \pi_n\}$) in scope.

Constraint Solution. We define an *assignment* η to be a pair of functions (η_R, η_P), where η_R is a map from V to R and η_P is a map from P to a region-constraint formula. Such an assignment is said to *satisfy* a set of constraints C if every sequent in C is valid after the substitutions η_P and η_R. We say that C is *satisfiable* if it has a satisfying assignment.

Constraint Generation. The constraint generation algorithm is a direct adaption of the type checker: each type checking judgment is modified to produce a set of constraints that must hold for the type checker to succeed.

Fig. 10 illustrates this for selected language constructs. (The appendix contains the remaining constraint generation rules.) These rules use the same context as the corresponding typing judgment in Fig. 6, except that this is generalized to permit the use of region variables and predicate variables. Symbols $\mathcal{A}$ and Γ retain their meaning, modulo this extension. Thus, the component Φ of $\mathcal{A}$ will now be a *symbolic expression* of the form $\varphi \wedge \phi$, where φ is the predicate variable representing the undetermined precondition of the method being analyzed.

The algorithm proceeds top-down, analyzes an expression e in a context $\mathcal{A}, \Gamma, r$, and returns a type τ and a set of constraints C (expressed in the rules as $\mathcal{A}, \Gamma, r \vdash e : \tau \lhd C$), indicating that the expression will have a type τ provided the constraints C hold.

When we adapt a type-checking judgement rule to produce a corresponding constraint-generation rule, we treat the antecedent conditions of the type-checking rule in one of two ways. Many of these antecedent conditions are converted into constraints which are accumulated in the set C. However, some of the antecedent conditions are expected to be trivially satisfied and are expressed as antecedent conditions in the constraint-generation rule as well. E.g., our frontend ensures that the region constants introduced by `open` and `letregion` constructs are unique (and the elaboration phase ensures this by alpha-renaming them as needed). Thus, the precondition $\rho \notin \Delta$ in rules LetRegion and Open is expected to hold true at constraint-generation time (and does not produce any constraints).

The rules for generating constraints from a method and class definition first build a context ($\mathcal{A}$) containing a set (Δ) denoting regions that are currently live, a map (Θ) mapping type variables to their bounds, and a constraint formula (Φ) capturing constraints over live region variables. We use predicate variables (φ and φ_m) to capture constraints over variables in Δ that are yet to be inferred.

Let GenConstraint(q) denote the set of constraints generated from an elaborated program q. The following theorem states that constraint-generation is sound and complete:

▶ **Theorem 2.** *Let $C =$ GenConstraint(q). An assignment η (for the region and predicate variables in q) satisfies C iff $q[\eta]$ is well-typed.*

4.4 The Constraint Solver

We refer to any validity constraint whose antecedent contains a predicate variable (*i.e.*, is of the form $\varphi \wedge \phi_{cx}$) as an *abduction constraint*. Here, ϕ_{cx} is either *true* (in which case, we call the constraint a trivial abduction constraint) or a conjunction of one or more outlives-constraints (in which case, we call the constraint a non-trivial abduction constraint).

Non-trivial abduction constraints are a key challenge in solving the constraints. We now describe some special properties of the set of constraints C generated by our algorithm, which allow us to handle abduction constraints efficiently.

For any predicate variable φ used in C, C has exactly one constraint of the form $\Delta \vdash \varphi$ `ok`. We will refer to this Δ as $\Delta_P^C(\varphi)$. (Our constraint-generation algorithm guarantees the preceding property. Even otherwise, a set of constraints $\Delta_i \vdash \varphi$ `ok` can be replaced by the single equivalent constraint $(\cap_i \Delta_i) \vdash \varphi$ `ok`.) We say that an abduction constraint is *C-decomposable* if its antecedent is of the form $\varphi \wedge \phi_{cx}$ where φ is a predicate variable and ϕ_{cx} is a conjunction of zero or more outlives-constraints of the form $\pi_1 \succeq \pi_2$ satisfying the following conditions: (1) $\pi_2 \notin \Delta_P^C(\varphi)$. (2) if $\pi_1 \in \Delta_P^C(\varphi)$, then for every $\pi_f \in \Delta_P^C(\varphi)$, $\pi_f \succeq \pi_2$ is a conjunct in ϕ_{cx}. We will omit the reference to C in the above notation if no confusion is likely.

▶ **Lemma 3.** *Every abduction constraint in C, where $C =$ GenConstraint(q), is C-decomposable.*

▶ **Lemma 4.** *Consider any C-decomposable constraint $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ where both π_i and π_j are region constants. Let η satisfy C.*
(a) If $\{\pi_i, \pi_j\} \subseteq \Delta_P(\varphi)$: η satisfies $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ iff η satisfies $\varphi \vdash \pi_i \succeq \pi_j$.
(b) If $\{\pi_i, \pi_j\} \not\subseteq \Delta_P(\varphi)$: η satisfies $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ iff η satisfies $\phi \vdash \pi_i \succeq \pi_j$

The above lemma shows how we can reduce a non-trivial abduction constraint to either a trivial abduction constraint or a non-abduction constraint, provided that the consequent is an outlives-constraint without region variables.

Constraint Solver. The first step in our algorithm for solving a set of constraints C computes a set $C^* \supseteq C$ of constraints by iteratively applying the following rules until a fixed point is reached:

1. (Initialization) $\ell \vdash r \in C \Rightarrow \ell \vdash r \in C^*$
2. (Transitivity)
   a. $\ell \vdash \rho_1 \succeq \rho_2 \in C^*$, $\ell \wedge \phi \vdash \rho_2 \succeq \rho_3 \in C^* \Rightarrow \ell \wedge \phi \vdash \rho_1 \succeq \rho_3 \in C^*$
   b. $\ell \wedge \phi \vdash \rho_1 \succeq \rho_2 \in C^*$, $\ell \vdash \rho_2 \succeq \rho_3 \in C^* \Rightarrow \ell \wedge \phi \vdash \rho_1 \succeq \rho_3 \in C^*$
3. (Substitution) $\ell \vdash F(\varphi) \in C^*$, $\varphi \vdash \phi \in C^* \Rightarrow \ell \vdash F(\phi) \in C^*$
4. (Abduction Decomposition)
   a. $\varphi \wedge \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*$, $\{\pi_i, \pi_j\} \subseteq \Delta_P(\varphi) \Rightarrow \varphi \vdash \pi_i \succeq \pi_j \in C^*$
   b. $\varphi \wedge \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*$, $\{\pi_i, \pi_j\} \not\subseteq \Delta_P(\varphi)$, $\{\pi_i, \pi_j\} \subseteq R \Rightarrow \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*$

We can show that every new constraint added by the above rules is implied by existing constraints:

▶ **Theorem 5.** *Let $C = \textsc{GenConstraint}(q)$. An assignment η satisfies C iff η satisfies C^*.*

A key goal of this step is to identify the value every region variable must have in any solution of the set of constraints, as explained below. Consider a set of constraints D. We say that a region variable ρ *occurs* in a *context* ℓ (in D) if D contains some constraint $\ell \vdash r$ where ρ occurs in r. We say that a region variable ρ is *bound* to a region constant π in a context ℓ if $\{\ell \vdash \rho \succeq \pi, \ell \vdash \pi \succeq \rho\} \subseteq D$. We say that a region variable ρ is *bound* to a region constant π (in D) if ρ is bound to π in every context ℓ in which it occurs. We say that a region variable ρ is *somewhere-bound* to a region constant π (in D) if ρ is bound to π in some context ℓ in which it occurs.

The set C^* makes it easy to identify a solution $\hat{\eta}$ (if one exists), as below. For any predicate variable φ, $\hat{\eta}_P(\varphi)$ is defined to be $\wedge\{\pi_1 \succeq \pi_2 \mid \pi_1, \pi_2 \in R, \varphi \vdash \pi_1 \succeq \pi_2 \in C^*\}$. For any region variable ρ, we define $\hat{\eta}_R(\rho)$ to be any element of the set $\{\pi \in R \mid \rho \text{ is somewhere-bound to } \pi \text{ in } C^*\}$, if this set is non-empty. If this set is empty for any ρ, $\hat{\eta}$ is undefined.

The set C^* also makes it easy to check if C is satisfiable. Let C^*_g denote the subset of all ground constraints (*i.e.*, constraints without any region variable or predicate variable) in C^*. Let WF_R denote the subset of all well-formedness constraints for region variables in C^*. Define $\textsc{Solve}(C)$ as below.

$$\textsc{Solve}(C) = \begin{cases} \textsc{Some}(\hat{\eta}) & \text{if } C^*_g \text{ is valid and } \hat{\eta} \text{ is defined and satisfies } \mathrm{WF}_R \\ \textsc{None} & \text{otherwise} \end{cases}$$

▶ **Theorem 6.** *Let $C = \textsc{GenConstraint}(q)$. (Soundness) If $\textsc{Solve}(C) = \textsc{Some}(\eta)$, then η satisfies C. (Completeness) If $\textsc{Solve}(C) = \textsc{None}$, then C is unsatisfiable.*

Checking the validity of a ground constraint is straightforward. A ground constraint is of the form $\wedge_{i \in I} \ell_i \vdash r$, where each ℓ_i and r is an outlives-constraint. Let D denote $\{\vdash \ell_i \mid i \in I\}$. The given constraint is valid iff r belongs to D^*.

Algorithmic Aspects. Note that checking the validity of a ground constraint can be realized using a simple graph reachability algorithm. Given a set S of outlives constraints, define the directed graph $G(S) = (V(S), E(S))$ as follows. Every distinct region identifier ρ in S is represented by a vertex, which we will also refer to as ρ. Every outlives constraint $\rho_1 \succeq \rho_2$ is represented by an edge from ρ_1 to ρ_2. It is easy to see that $\wedge S \vdash \rho_1 \succeq \rho_2$ iff there exists a path from ρ_1 to ρ_2 in $G(S)$. Thus, a simple graph reachability algorithm can be used to check the validity of ground constraints.

This idea generalizes. Extending the simple graph reachability algorithm to incorporate the Substitution rule (in computing C^*) turns the problem into a context-free reachability problem in graphs [16] (as usual for context-sensitive interprocedural analysis).

Algorithms for context-free reachability can be adapted to incorporate the Abduction Decomposition step. Alternatively, the iterative process described above is a standard fixed point computation and can be encoded using a set of Datalog rules, allowing us to compute the closure using any Datalog engine.

4.5 Soundness and Completeness: Discussion

Theorems 2 and 6 show that the second and third steps of the type-inference algorithm are sound. It is easy to verify the soundness of the first step (the elaboration phase): For any program $p \in [\![FB]\!]$, the elaboration phase produces a q such that for any assignment η, we have $[\![q[\eta]]\!] = p$. The soundness of the type inference algorithm follows.

Theorem 2 and 6 also establish the completeness of the constraint-generation and constraint-solving steps. The only source of incompleteness in the type inference algorithm is the set of heuristic choices made during the first step, as explained below.

(1) We determine the set of region parameters for a recursive class using the heuristic that a recursive occurrence of the class has the same parameters, in the same order, as the class itself. This heuristic fails, for example, if the program uses a recursive list type whose elements alternatively come from two different regions. Such a program would require the following elaboration, which is beyond the scope of our approach:

```
class List ◁ Object
{
  Object data;
  List next;
}
```

⇒

```
class List ⟨ρ0, ρ1, ρ2 | φ⟩ ◁ Object⟨ρ0⟩
{
  Object⟨ρ1⟩ data;
  List⟨ρ0, ρ2, ρ1⟩ next;
}
```

(2) Our technique for region parameterization also uses a heuristic in the case of higher order programs. The following examples illustrates that principal types may not exist for higher order functions.

```
unit apply ( T → unit f, T x, T y) { f(x); f(y); }
```

This method may be typed assuming either that `f` is polymorphic over the region that its parameter is allocated in (permitting `x` and `y` to be allocated in any regions), or by assuming that `x` and `y` are allocated in the same region ρ_1 that `f` expects its parameters to be allocated in. Neither type subsumes the other. Our algorithm heuristically chooses the first option, as it appears to be the more likely and useful candidate.

If users provide partial region annotations, especially in situations (such as above) where elaboration makes a heuristic choice, the elaboration procedure can use the user-provided choices instead. This can help the type-inference overcome these limitations.

4.6 Modularity Aspects of Type Inference

The type inference algorithm, as presented, traverses the entire program to generate the set of constraints, which are solved en masse, using an iterative fixed point computation. However, the type inference can be realized in a modular and compositional fashion, subject only to the restrictions imposed by recursion.

In the elaboration phase, we can process a class `C` only after any class `B` that `C` depends on has been processed: class `C` depends on class `B` if `B` is either `C`'s base class or the type of any field of `C` depends on `B`. In effect, this means that any collection of mutually recursive classes must be processed together. Non-recursive dependences can be handled in a compositional fashion: if class `C` depends on `B` non-recursively, then the elaboration can be done for `B` first, and then `C` can be processed.

The same idea applies to the constraint-solving phase as well. Given a set of constraints, we say that a predicate variable φ_1 *directly-depends* on another predicate variable φ_2 if the set of constraints includes a constraint $\varphi_1 \wedge \phi_{cx} \vdash F(\varphi_2)$. We say that φ_1 *depends* on φ_2 if φ_1 transitively depends on φ_2. The constraint solver needs to process any collection of mutually dependent predicate variables together. In effect, this requires the type inference to process any collection of mutually recursive methods together. However, methods that are not mutually recursive can be processed separately.

5 Implementation and Evaluation

As mentioned earlier, our work is a continuation of the work reported in [7], which provides users with a C#-based implementation of transferable regions (the features described in Section 2). This system has no type system and provides users with no safety guarantees. [7] presents evidence that realistic programs can be implemented using transferable regions and that this can yield significant performance gains. In particular, it reports speedups up to 34% for typical big-data analytics jobs. We now describe our implementation and experience with the type system and type inference algorithm presented in the current paper.

We have implemented BroomC, a prototype of Broom compiler frontend, including its region type system and type inference, in 3k+ lines of OCaml. 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.

We performed two kinds of experiments to evaluate our region type system and type inference. First, we implemented some microbenchmarks ($\leq$100 LOC) consisting of standard classes such as pairs, lists, list iterators, etc., in $[\![FB^+]\!]$, and used our inference engine to infer their region types. These classes are region-oblivious. Hence, as long as they are well-typed as per the core type system, BroomC must be able to automatically construct its region-type-annotated definition without fail. BroomC was able to infer the expected region types for all these classes under 10ms. Fig 11 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 latter 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 4 out of the 6 Naiad streaming query operator benchmarks (Naiad vertices) used in [7] to $[\![FB^+]\!]$, and used BroomC to verify their safety. The 2 remaining benchmarks were left out because their region behavior (from the perspective of the type system) is subsumed by the included benchmarks. The number of LOC performing operations

```
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 11 Region-annotated definition of `rev` computed by Broomc.

on `Region` objects relative to the total LOC is 8% or under in the Naiad benchmarks. During the process, we found multiple instances of potential memory safety violations in the $[\![FB^{+}]\!]$ translation of all the 4 Naiad vertices, 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[6]. 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 shows the utility of our type inference/checking tool, particularly because it comes at no additional cost to the developer.

6 Related Work

Following Tofte and Talpin's seminal work in [15, 20, 21], static type systems for safe region-based memory management have been extensively studied in the context of various languages and problem settings [8, 10, 24, 2, 1, 9, 23, 17, 11]. Our work differs from the existing proposals in one or more of the following respects.

1. Our design choice focuses on ensuring memory-safety while giving programmers control over region management and allocation of objects in regions. In contrast, some systems automate all aspects of memory management. This is a convenience-performance trade-off.
2. We support both lexically scoped (stack) regions and dynamic transferable regions (both programmer-managed).
3. We exploit a combination of a simple static type discipline and lightweight runtime checks to ensure memory safety. In particular, our approach circumvents the need for restrictive static mechanisms (e.g., linear types and unique pointers) or expensive runtime mechanisms (e.g., garbage collection and reference counting) in order to guarantee safety.

[6] This unsafe reference could have gone unnoticed during experiments in [7] because their experimental setup included only one actor.

4. We present a full (interprocedural) type inference algorithm that eliminates the need to write region annotations on types.
5. Our underlying language is an object-oriented programming language, equipped with higher-order functions and parameterized (generic) types. These language features necessitate some non-trivial choices in the design of the region-parametricity aspect of the language, which also have an impact on aspects such type inference.

Tofte and Talpin's approach [21] uses compiler-managed lexically scoped (stack) regions (as a replacement for GC). Our type inference is analogous to theirs in some respects, while differing in others. Their inference algorithm only generates equality constraints, solvable via unification. Our type inference algorithm generates partial order outlives constraints. Consequently, our constraint solving algorithm is more sophisticated, and is capable of inferring unknown outlives constraints over region arguments of polymorphic recursive functions.

Walker and Watkins [23] extend lambda calculus with first-class regions with dynamic lifetimes, and impose linear typing to control accesses to regions. Our open/close lexical block for transferable regions traces its origins to the `let!` expression in [23] and [22], which safely relaxes linear typing restrictions, allowing variables to be temporarily aliased. We don't use linear typing (for references to regions), thus admit unrestricted aliasing, but use lightweight runtime checks for safety. Moreover, [23]'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.

Cyclone [8] equips C with programmer-managed stack regions, and a typing discipline that statically guarantees the safety of all pointer dereferences. Later proposals [10, 19] extends Cyclone with dynamic regions. Broom differs from Cyclone in its non-intrusiveness design principle, which requires its safety mechanisms to not intrude on 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. Cyclone 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 [19], 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. Similar contrast can be made of region type inference in both the languages. Cyclone's type inference was only ever described as being similar to Tofte and Talpin's, and its effectiveness in presence of tracked pointers is not clear. In contrast, a detailed type inference algorithm is one of our core contributions.

Our region type system can also be thought of as a specialized ownership type system [5], where each region is the owner of all objects allocated in the region. An ownership type system for safe region-based memory management in real-time Java has been proposed by Boyapati *et al.* [2]. Their language permits only lexically-scoped (stack) regions. In contrast, we permit regions with dynamically determined lifetimes. Our language also admits generics and higher-order functions. We also establish type safety and transfer safety results that formalize the guarantees provided by our system. While Boyapati *et al.*'s language is explicitly typed, our language comes equipped with full type inference. However, several inference algorithms have been proposed in the context of other ownership type systems. Our type inference algorithm is also novel compared to these existing ownership inference algorithms, which are based on, e.g., pointer analysis [12] or boolean satisfiability [6]. (See Section 5.2 of Clark *et al.*'s survey of ownership type systems [5] for a more comprehensive discussion of ownership inference algorithms.) Some distinguishing characteristics of our algorithm is that it is customized to our problem, does not use any pointer analysis algorithm (which can be a source of imprecision) or SAT solvers (which can be a source of inefficiency), and comes with relative completeness guarantees.

Henglein *et al.* [9] propose a flow-sensitive approach for first-order programs to generalize Tofte and Talpin's approach to dynamic regions. Cherem and Rugina [4] describe a flow-insensitive and context-sensitive analysis that transforms Java programs to use (dynamic) regions. However, neither of themr supports dynamic regions as first-class objects; they cannot be stored in data structures or passed to methods. Furthermore, while Henglein *et al.* [9] require reference counting to ensure memory safety, Cherem and Rugina's analysis [4] comes with no formal safety guarantees. Holk *et al.* [11] use regions to safely transfer data between the CPU and GPU in the context of Scheme. However, their setting only includes lexically-scoped regions for which Tofte and Talpin-style analysis suffices. In contrast, we provide first-class support for transferable regions with dynamic lifetimes.

References

1 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen 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. doi:10.1145/1640089.1640097.

2 Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin 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. doi:10.1145/781131.781168.

3 Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014. URL: http://www.vldb.org/pvldb/vol8/p401-chandramouli.pdf.

4 Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 85–96, New York, NY, USA, 2004. ACM. doi:10.1145/1029873.1029884.

5 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.

6 Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 333–357, 2011. doi:10.1007/978-3-642-22655-7_16.

7 Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael 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.

8 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James 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. doi:10.1145/512529.512563.

9 Fritz Henglein, Henning Makholm, and Henning 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. doi:10.1145/773184.773203.

10 Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor 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. doi:10.1145/1029873.1029883.

11 Eric Holk, Ryan Newton, Jeremy Siek, and Andrew 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. doi:10.1145/2660193.2660244.

12 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 181–206, 2012. doi:10.1007/978-3-642-31057-7_9.

13 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

14 Martin Maas, Tim Harris, Krste Asanovic, and John 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.

15 Mads Tofte and Jean-Pierre 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.

16 Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998. doi:10.1016/S0950-5849(98)00093-7.

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

18 Type-safe off-heap memory, 2016. Accessed: 2016-06-6 13:21:00. URL: https://github.com/densh/scala-offheap.

19 Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Science of Computer Programming*, 62(2):122–144, 2006. Special Issue: Five perspectives on modern memory management - Systems, hardware and theory. `doi:10.1016/j.scico.2006.02.003`.

20 Mads Tofte and Jean-Pierre 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. `doi:10.1145/174675.177855`.

21 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997. `doi:10.1006/inco.1996.2613`.

22 Philip 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, 1990. North-Holland.

23 David Walker and Kevin 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. `doi:10.1145/507635.507658`.

24 Bennett Norton 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.

25 Matei Zaharia. New developments in spark, 2015. URL: `http://www.slideshare.net/databricks/new-developments-in-spark`.

A Appendix

A.1 Static Semantics

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

$$
\frac{CT(B) = \texttt{class}\ B\langle\overline{a}\triangleleft\overline{K}\rangle\langle\overline{\rho}\,|\,\phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ ...\} \quad \mathcal{S} = [\overline{r}/\overline{\rho},\overline{T}/\overline{a}] \qquad \mathsf{fields}(\mathcal{S}(N)) = \overline{g}:\overline{\tau^g}}{\mathsf{fields}(B\langle\overline{T}\rangle\langle\overline{r}\rangle)\ =\ \overline{g}:\overline{\tau^g},\ \overline{f}:\mathcal{S}(\overline{\tau^f})}
$$

$$
\frac{CT(B) = \texttt{class}\ B\langle\overline{a}\triangleleft\overline{K}\rangle\langle\overline{\rho}\,|\,\phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ \overline{d}\} \quad m \notin \overline{d} \qquad \mathcal{S} = [\overline{r}/\overline{\rho},\overline{T}/\overline{a}]}{\mathsf{mtype}(m, B\langle\overline{T}\rangle\langle\overline{r}\rangle)\ =\ \mathsf{mtype}(m,\mathcal{S}(N))}
$$

$$
\frac{CT(B) = \texttt{class}\ B\langle\overline{a}\triangleleft\overline{K}\rangle\langle\overline{\rho}\,|\,\phi\rangle \triangleleft N\{\overline{\tau^f}\ \overline{f};\ \overline{d}\} \quad \tau^2\ m\langle\overline{\rho_m}\,|\,\phi_m\rangle(\overline{\tau^1\ x})\{...\} \in \overline{d} \quad \mathcal{S} = [\overline{r}/\overline{\rho},\overline{T}/\overline{a}]}{\mathsf{mtype}(m, B\langle\overline{T}\rangle\langle\overline{r}\rangle)\ =\ \mathcal{S}(\langle\overline{\rho_m}\,|\,\phi_m\rangle\overline{\tau^1} \rightarrow \tau^2)}
$$

$$
\frac{\begin{array}{c}\mathsf{mtype}(m,N) = \langle\overline{\rho_1}\,|\,\phi_1\rangle\overline{\tau^{11}} \rightarrow \tau^{12} \quad \texttt{implies} \\ \mathcal{A}.\Phi\ \vdash\ \phi_2 \Leftrightarrow [\overline{\rho_2}/\overline{\rho_1}](\phi_1) \quad \texttt{and} \quad \overline{\tau^{21}} = [\overline{\rho_2}/\overline{\rho_1}](\overline{\tau^{11}}) \quad \texttt{and} \quad \mathcal{A}\ \vdash\ \tau^{22} <: [\overline{\rho_2}/\overline{\rho_1}](\tau^{12})\end{array}}{\mathcal{A} \vdash \mathsf{override}(m, N, \langle\overline{\rho_2}\,|\,\phi_1\rangle\overline{\tau^{21}} \rightarrow \tau^{22})}
$$

Figure 12 FEATHERWEIGHT BROOM: auxiliary definitions.

Subtyping $\boxed{\mathcal{A} \vdash \tau_1 <: \tau_2}$

$$\begin{array}{c}\mathcal{A} \vdash \tau <: \tau \\ (\Delta, \Theta, \Phi) \vdash a@\rho <: \Theta(a)@\rho\end{array} \qquad \dfrac{CT(B) = \texttt{class}\ B\langle\overline{a} \lhd \overline{K}\rangle\langle\overline{\rho}\,|\,\phi\rangle \lhd N\{...\}}{\mathcal{A} \vdash B\langle\overline{T}\rangle\langle\overline{r}\rangle <: [\overline{r}/\overline{\rho}, \overline{T}/\overline{a}](N)}$$

$$\dfrac{\begin{array}{c}\mathcal{A} \vdash \tau_1 <: \tau_2 \\ \mathcal{A} \vdash \tau_2 <: \tau_3\end{array}}{\mathcal{A} \vdash \tau_1 <: \tau_3} \qquad \dfrac{\begin{array}{c}\mathcal{A}.\Phi \vdash \phi_1 \Rightarrow \phi_2 \\ \mathcal{A} \vdash \overline{\tau^{11}} <: \overline{\tau^{21}} \quad \mathcal{A} \vdash \tau^{22} <: \tau^{12}\end{array}}{\mathcal{A} \vdash \langle\overline{\rho}\,|\,\phi_2\rangle\overline{\tau^{21}} \xrightarrow{r} \tau^{22} <: \langle\overline{\rho}\,|\,\phi_1\rangle\overline{\tau^{11}} \xrightarrow{r} \tau^{12}}$$

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

$$\dfrac{r \in \Delta}{(\Delta, \Theta, \Phi) \vdash \texttt{Object}\,\langle r\rangle\ \texttt{ok}} \qquad \dfrac{r_0, r_1 \in \Delta}{\Delta \vdash r_0 \succeq r_1\ \texttt{ok}} \qquad \dfrac{\Delta \vdash \phi_0\ \texttt{ok} \quad \Delta \vdash \phi_1\ \texttt{ok}}{\Delta \vdash \phi_0 \wedge \phi_1\ \texttt{ok}}$$

$$\dfrac{r \in \Delta \quad \overline{\rho} \notin \Delta \quad \Delta' = \Delta \cup \{\overline{\rho}\} \quad \mathcal{A}' = (\Delta', \Theta, \Phi \wedge \phi) \quad \Delta' \vdash \phi\ \texttt{ok} \quad \mathcal{A}' \vdash \overline{\tau^1}\ \texttt{ok} \quad \mathcal{A}' \vdash \tau^2\ \texttt{ok}}{(\Delta, \Theta, \Phi) \vdash \langle\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{r} \tau^2\ \texttt{ok}}$$

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

$$\dfrac{\begin{array}{c}\Delta = \{\rho, \overline{\rho}\} \quad \Theta = [\overline{a} \mapsto \overline{K}] \quad \Phi = \phi \quad \mathcal{A} = (\Delta, \Theta, \Phi) \quad \Delta \vdash \phi\ \texttt{ok} \quad \Theta \Vdash \overline{K}\ \texttt{ok} \quad \overline{d}\ \texttt{ok in}\ B \\ \mathcal{A} \vdash N, \overline{\tau^f}\ \texttt{ok} \quad \textsf{shape}(N) \neq \texttt{Region}\,\langle T\rangle \quad \Phi \vdash \textsf{allocRgn}(\overline{\tau^f}) \succeq \rho \quad \textsf{allocRgn}(N) = \rho\end{array}}{\texttt{class}\ B\langle\overline{a} \lhd \overline{K}\rangle\langle\rho\overline{\rho}\,|\,\phi\rangle \lhd N\{\overline{\tau^f}\ \overline{f};\ \overline{d}\}\ \texttt{ok}}$$

Figure 13 Featherweight Broom: subtyping and well-formedness rules (contiunuation of Fig. 6).

Expression Typing $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau}$

$$\dfrac{\begin{array}{c}\mathcal{A}, \Gamma, r \vdash e_1 : \tau_1 \\ \mathcal{A}, \Gamma[x \mapsto \tau_1], r \vdash e_2 : \tau_2\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \dfrac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathtt{l} \notin \Delta \\ \mathcal{A}' = (\Delta \cup \{\mathtt{l}\}, \Theta, \Phi \wedge \Delta \succeq \mathtt{l}) \\ \mathcal{A}', \Gamma, \mathtt{l} \vdash e : \tau \quad \mathcal{A} \vdash \tau\ \texttt{ok}\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{letd}\ \mathtt{l}\ \texttt{in}\ e : \tau}$$

$$\dfrac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{\mathtt{l}\}, \Theta, \Phi) \quad \mathcal{A}' \vdash T@\mathtt{l}\ \texttt{ok} \quad \mathcal{A}', \Gamma, \mathtt{l} \vdash e : T@\mathtt{l}}{\mathcal{A}, \Gamma, r \vdash \texttt{new Region}\,\langle T\rangle\langle \mathtt{l}_\top \mathtt{l}\rangle(e) : \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle}$$

$$\dfrac{\begin{array}{c}\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{\mathtt{l}\}, \Theta, \Phi) \\ \mathtt{l} \in \Delta\ \texttt{implies}\ s = \blacksquare \quad \mathcal{A}', \Gamma, \mathtt{l} \vdash e : \tau \quad \mathcal{A} \vdash \tau\ \texttt{ok}\end{array}}{\mathcal{A}, \Gamma, r \vdash \texttt{opened}\ \mathtt{l}(s)\ \texttt{in}\ e : \tau} \qquad \dfrac{\begin{array}{c}\mathcal{A}, \Gamma, r \vdash e : \tau \\ \mathcal{A} \vdash \tau <: \tau'\end{array}}{\mathcal{A}, \Gamma, r \vdash e : \tau'}$$

Figure 14 Featherweight Broom: expression typing (continuation of Fig. 6).

A.2 Operational Semantics

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

$$[\textsc{EvalOrder}] \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(E[e], \Sigma) \longrightarrow (E[e'], \Sigma')}$$

$$[\textsc{Exception}] \quad \frac{(e, \Sigma) \longrightarrow \bot}{(E[e], \Sigma) \longrightarrow \bot}$$

$$[\textsc{FieldAccess}] \quad \frac{\Sigma(\mathtt{l}) = \blacksquare \quad \mathsf{fields}(A\langle\overline{T}\rangle\langle\mathtt{l}\overline{\mathtt{l}}\rangle) = \overline{f} : \overline{\tau}}{((\,\mathtt{new}\ A\langle\overline{T}\rangle\langle\mathtt{l}\overline{\mathtt{l}}\rangle(\overline{v})).f_i, \Sigma) \longrightarrow (v_i, \Sigma)}$$

$$[\textsc{MethodInv}] \quad \frac{\Sigma(\mathtt{l}) = \blacksquare \quad \mathsf{mbody}(m, A\langle\overline{T}\rangle\langle\mathtt{l}\overline{\mathtt{l}}\rangle) = \rho\overline{\rho}.\overline{x}.e}{\begin{array}{c}((\,\mathtt{new}\ A\langle\overline{T}\rangle\langle\mathtt{l}\overline{\mathtt{l}}\rangle(\overline{v})).m\langle\mathtt{l}'\overline{\mathtt{l}'}\rangle(\overline{v'}), \Sigma) \longrightarrow \\ ([\mathtt{l}'\overline{\mathtt{l}'}/\rho\overline{\rho}][\overline{v'}/\overline{x}][\,\mathtt{new}\ A\langle\overline{T}\rangle\langle\mathtt{l}\overline{\mathtt{l}}\rangle(\overline{v})/\,\mathtt{this}\,]\,e, \Sigma)\end{array}}$$

$$[\textsc{FnApply}] \quad \frac{v_a = \lambda @\mathtt{l}'\langle\rho\overline{\rho}\rangle(\overline{\tau}\ \overline{x}).e \quad \Sigma(\mathtt{l}') = \blacksquare}{(v_a\langle\mathtt{l}\overline{\mathtt{l}}\rangle(\overline{v}), \Sigma) \longrightarrow ([\overline{v}/\overline{x}][\mathtt{l}\overline{\mathtt{l}}/\rho\overline{\rho}]\,e, \Sigma)}$$

$$[\textsc{Exception}] \quad \frac{\Sigma(\mathtt{l}) = \square \quad (e, \Sigma[\mathtt{l} \mapsto \blacksquare]) \longrightarrow \bot}{(\,\mathtt{new}\ \ \mathtt{Region}\,\langle T\rangle\langle\mathtt{l}_\top\mathtt{l}\rangle(e), \Sigma) \longrightarrow \bot}$$

$$[\textsc{LetExp}] \quad \frac{}{(\,\mathtt{let}\ x\ =\ v\ \mathtt{in}\ e, \Sigma) \longrightarrow ([v/x]e, \Sigma)}$$

Evaluation Context $\boxed{E}$

$$\begin{array}{rcl} E & ::= & \bullet \mid (\bullet).f \mid \bullet.m\langle\overline{\mathtt{l}}\rangle(\overline{e}) \mid v.m\langle\overline{\mathtt{l}}\rangle(\ldots,\bullet,\ldots) \mid \mathtt{new}\ N(\ldots,\bullet,\ldots) \\ & & \mid\ \mathtt{new}\ \ \mathtt{Region}\,\langle T\rangle\langle\pi_\top\rangle(\bullet) \mid \bullet\langle\overline{\mathtt{l}}\rangle(\overline{e}) \mid v\langle\overline{\mathtt{l}}\rangle(\ldots,\bullet,\ldots) \\ & & \mid\ \mathtt{let}\ x\ =\ \bullet\ \mathtt{in}\ e \mid \mathtt{open}\ \ \bullet\ \ \mathtt{as}\ y@r\ \mathtt{in}\ e \end{array}$$

Figure 15 FEATHERWEIGHT BROOM: operational semantics (continuation of Fig. 8).

A.3 Constraint Generation Rules

Expression Typing Constraint Generation $\boxed{\mathcal{A},\Gamma,r \vdash e : \tau \lhd C}$

[Unit]
$$\mathcal{A},\Gamma,r \vdash () : \texttt{unit} \lhd \{\}$$

[Var]
$$\mathcal{A},\Gamma,r \vdash x : \Gamma(\tau) \lhd \{\}$$

[FieldAccess]
$$\frac{\mathcal{A},\Gamma,r \vdash e : \tau' \lhd C \quad \overline{f} : \overline{\tau} = \mathsf{fields}(\mathsf{bound}_{\mathcal{A}.\Theta}(\tau'))}{\mathcal{A},\Gamma,r \vdash e.f_i : \tau_i \lhd C}$$

[Let]
$$\frac{\mathcal{A},\Gamma,r \vdash e_1 : \tau_1 \lhd C_1 \quad \mathcal{A},\Gamma[x \mapsto \tau_1],r \vdash e_2 : \tau_2 \lhd C_2}{\mathcal{A},\Gamma,r \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \lhd C_1 \cup C_2}$$

[MethodInv]
$$\frac{\begin{array}{c}\mathcal{A},\Gamma,r \vdash e_0 : \tau \lhd C_1 \quad C_2 = \{r\overline{r} \in \mathcal{A}.\Delta\} \\ \mathsf{mtype}(m, \mathsf{bound}_{\mathcal{A}.\Theta}(\tau)) = \langle\rho\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \to \tau^2 \\ \mathcal{A} \vdash \langle\rho\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \to \tau^2\ \mathsf{ok} \lhd C_3 \quad \mathcal{A},\Gamma,r \vdash \overline{e} : [r\overline{r}/\rho\overline{\rho}](\overline{\tau^1}) \lhd C_4 \\ C_5 = \{\mathcal{A}.\Phi \vdash [r\overline{r}/\rho\overline{\rho}](\phi)\}\end{array}}{\mathcal{A},\Gamma,r \vdash e_0.m\langle r\overline{r}\rangle(\overline{e}) : [r\overline{r}/\rho\overline{\rho}](\tau^2) \lhd C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5}$$

[Lambda]
$$\frac{\begin{array}{c}r \in \mathcal{A}.\Delta \quad \rho\overline{\rho} \notin \mathcal{A}.\Delta \quad \mathcal{A}' = (\mathcal{A}.\Delta \cup \{\rho\overline{\rho}\}, \mathcal{A}.\Theta, \mathcal{A}.\Phi \wedge \phi) \\ \mathcal{A}'.\Delta \vdash \phi\ \mathsf{ok} \quad \mathcal{A}' \vdash \overline{\tau^1}\ \mathsf{ok} \lhd C_1 \quad \mathcal{A}' \vdash \tau^2\ \mathsf{ok} \lhd C_2 \\ \mathcal{A}',\Gamma[\overline{x} \mapsto \overline{\tau^1}],\rho \vdash e : \tau^2 \lhd C_3\end{array}}{\mathcal{A},\Gamma,r \vdash \lambda@r\langle\rho\overline{\rho}\,|\,\phi\rangle(\overline{x} : \overline{\tau^1}).e : \langle\rho\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{r} \tau^2 \lhd \cup_{i=1}^{4} C_i}$$

[SubTyping]
$$\frac{\mathcal{A},\Gamma,r \vdash e : \tau \lhd C_1 \quad \mathcal{A} \vdash \tau <: \tau' \lhd C_2}{\mathcal{A},\Gamma,r \vdash e : \tau' \lhd C_1 \cup C_2}$$

Method Well-formedness Constraint Generation $\boxed{\vdash d\ \mathsf{ok\ in}\ B \lhd C}$

[Method]
$$\frac{\begin{array}{c}CT(B) = \texttt{class}\ B\langle\overline{a} \lhd \overline{K}\rangle\langle\overline{\rho}\,|\,\varphi\rangle \lhd N\{\cdots\} \\ \mathcal{A} = (\Delta,\Theta,\Phi) = (\{\overline{\rho},\rho_m,\overline{\rho_m}\},[\overline{a} \mapsto \overline{K}],\varphi_m) \qquad C_1 = \{\Delta \vdash \varphi_m\ \mathsf{ok}\} \\ \Gamma = \cdot[\texttt{this} \mapsto B\langle\overline{a}\rangle\langle\overline{\rho}\rangle][\overline{x} \mapsto \overline{\tau^1}] \quad \mathsf{mtype}(m,N) = \langle\overline{\rho_m}\,|\,\phi_m\rangle\overline{\tau^1} \to \tau^2 \\ \mathcal{A},\Gamma,\rho_m \vdash e : \tau^2 \lhd C_2 \quad \mathcal{A} \vdash \overline{\tau^1}\ \mathsf{ok} \lhd C_3 \quad \mathcal{A} \vdash \tau^2\ \mathsf{ok} \lhd C_4\end{array}}{\vdash \tau^2\ m\langle\rho_m\overline{\rho_m}\,|\,\varphi_m\rangle(\overline{\tau^1\ x})\{\texttt{return}\,e;\}\ \mathsf{ok\ in}\ B \lhd (C_1 \cup C_2 \cup C_3 \cup C_4)}$$

Class Well-formedness Constraint Generation $\boxed{\vdash B\ \mathsf{ok} \lhd C}$

[Class]
$$\frac{\begin{array}{c}\mathcal{A} = (\Delta,\Theta,\Phi) = (\{\rho,\overline{\rho}\},[\overline{a} \mapsto \overline{K}],\varphi) \\ C_1 = \{\Delta \vdash \varphi\ \mathsf{ok}\} \qquad \Theta \Vdash \overline{K}\ \mathsf{ok} \qquad \mathcal{A} \vdash N\ \mathsf{ok} \lhd C_2 \qquad \mathcal{A} \vdash \overline{\tau^f}\ \mathsf{ok} \lhd C_3 \\ C_4 = \{\Phi \vdash \mathsf{allocRgn}(\overline{\tau^f}) \succeq \rho \wedge \mathsf{allocRgn}(N) = \rho\} \\ \vdash \overline{d}\ \mathsf{ok\ in}\ B \lhd C_5\end{array}}{\vdash \texttt{class}\ B\langle\overline{a} \lhd \overline{K}\rangle\langle\rho\overline{\rho}\,|\,\varphi\rangle \lhd N\{\overline{\tau^f\ x};\ \overline{d}\}\ \mathsf{ok} \lhd \bigcup_{i=1}^{5} C_i}$$

Figure 16 Featherweight Broom: Constraint generation rules part 2 (Continuation of Fig. 10).

Subtyping Constraint Generation $\boxed{\mathcal{A} \vdash \tau_1 <: \tau_2 \lhd C}$

[Reflexivity]
$$\mathcal{A} \vdash \tau <: \tau \lhd \{\}$$

[Unify]
$$\mathcal{A} \vdash \tau <: [\pi/\rho](\tau) \lhd \{\pi \succeq \rho, \rho \succeq \pi\}$$

[Transitivity]
$$\frac{\mathcal{A} \vdash \tau_1 <: \tau_2 \lhd C_1 \quad \mathcal{A} \vdash \tau_2 <: \tau_3 \lhd C_2}{\mathcal{A} \vdash \tau_1 <: \tau_3 \lhd C_1 \cup C_2}$$

[FnSubtyping]
$$\frac{\begin{array}{c} C_1 = \{\mathcal{A}.\Phi \vdash \phi_1 \Rightarrow \phi_2\} \\ \mathcal{A} \vdash \overline{\tau^{11}} <: \overline{\tau^{21}} \lhd C_2 \quad \mathcal{A} \vdash \tau^{22} <: \tau^{12} \lhd C_3 \end{array}}{\mathcal{A} \vdash \langle\overline{\rho}\,|\,\phi_2\rangle\overline{\tau^{21}} \xrightarrow{r} \tau^{22} <: \langle\overline{\rho}\,|\,\phi_1\rangle\overline{\tau^{11}} \xrightarrow{r} \tau^{12} \lhd C_1 \cup C_2 \cup C_3}$$

Type Well-formedness Constraint Generation $\boxed{\mathcal{A} \vdash \tau \; \texttt{ok} \lhd C}$

[ObjectType]
$$\frac{C = \{r \in \Delta\}}{(\Delta, \Theta, \Phi) \vdash \texttt{Object}\,\langle r\rangle \; \texttt{ok} \lhd C}$$

[ClassType]
$$\frac{\begin{array}{c} CT(B) = \texttt{class} \; B\langle\overline{a \lhd K}\rangle\langle\overline{\rho}\,|\,\phi\rangle \lhd N\{\ldots\} \quad \Theta \Vdash B\langle\overline{T}\rangle \; \texttt{ok} \\ C = \{\overline{r} \in \Delta, \Phi \vdash [\overline{r}/\overline{\rho}](\phi)\} \end{array}}{(\Delta, \Theta, \Phi) \vdash B\langle\overline{T}\rangle\langle\overline{r}\rangle \; \texttt{ok} \lhd C}$$

[TypeParam]
$$\frac{\Theta \Vdash T \; \texttt{ok} \quad \Theta \Vdash T <: \texttt{Object} \qquad C = \{r \in \Delta\}}{(\Delta, \Theta, \Phi) \vdash T@r \; \texttt{ok} \lhd C}$$

[FnType]
$$\frac{\begin{array}{c} C_1 = \{r \in \Delta\} \\ \overline{\rho} \notin \Delta \quad \Delta' = \Delta \cup \{\overline{\rho}\} \quad \mathcal{A}' = (\Delta', \Theta, \Phi \wedge \phi) \\ \Delta' \vdash \phi \; \texttt{ok} \lhd C_2 \quad \mathcal{A}' \vdash \overline{\tau^1} \; \texttt{ok} \lhd C_3 \quad \mathcal{A}' \vdash \tau^2 \; \texttt{ok} \lhd C_4 \end{array}}{(\Delta, \Theta, \Phi) \vdash \langle\overline{\rho}\,|\,\phi\rangle\overline{\tau^1} \xrightarrow{r} \tau^2 \; \texttt{ok} \lhd C_1 \cup C_2 \cup C_3 \cup C_4}$$

[RegionType]
$$\frac{\Theta \Vdash T \; \texttt{ok}}{(\Delta, \Theta, \Phi) \vdash \texttt{Region}\,\langle T\rangle\langle\pi_\top\rangle \; \texttt{ok} \lhd \{\}}$$

Figure 17 Featherweight Broom: Constraint generation rules part 3 (Continuation of Fig. 16).