Region Inference for an Object-Oriented Language

Wei-Ngan Chin[1,2], Florin Craciun[2], Shengchao Qin[1,2], and Martin Rinard[3]
[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Laboratory for Computer Science, Massachusetts Institute of Technology
{chinwn,craciunm,qinsc}@comp.nus.edu.sg, rinard@lcs.mit.edu

ABSTRACT

Region-based memory management offers several important advantages over garbage-collected heap, including real-time performance, better data locality and efficient use of limited memory. The concept of regions was first introduced for a call-by-value functional language by Tofte and Talpin, and has since been advocated for imperative and object-oriented languages. Scope memory, a lexical variant of regions, is now a core feature in a recent proposal on Real-Time Specification for Java (RTSJ)[4].

Recent works in region-based programming for Java have focused on region-checking which requires manual effort in choosing regions with appropriate lifetimes. In this paper, we propose an automatic region-inference type system for a core subset of Java. To provide an inference method that is both precise and practical, we support classes and methods that are region-polymorphic; and with region-polymorphic recursion for methods. One challenging aspect of our inference rules is to ensure safe region programming in the presence of class subtyping, method overriding and downcast operations. Our set of region inference rules can handle these object-oriented features safely, without creating dangling references.

1. INTRODUCTION

In region-based memory management, each new object is added to a *region* with a designated lifetime. While objects may be added to their respective regions at different times, the entire set of objects in each region are freed simultaneously, when the region is deleted. Various studies have shown that region-based programming can provide safe memory management with good real-time performance. Data locality is also improved when related objects are placed together in the same region. By classifying objects into regions based on their lifetimes, better utilization of memory can be achieved when dead regions are recovered on a timely basis.

Recently, several works have investigated region-based programming for Java-based languages [17, 12, 11, 8]. Most of these works (e.g. [12, 8]) require programmers to manually annotate their programs with regions. Region type-checking then guarantees that well-typed programs never access dangling references, so that runtime tests for danglings can be omitted.

There are two main approaches to prevent unsafe accesses. One approach[12] uses a type-and-effect system to analyse regions that might be accessed, so that those regions that are not being accessed can be deallocated earlier, even if there are dangling references to these regions. Another approach[8] prevents all potential dangling references (from older to younger regions) by ensuring that each object always point-to other objects in regions with equal or longer lifetimes. The former approach (no-dangling-access) may yield more precise lifetimes[1], but the latter approach (no-dangling) is required by RTSJ and for co-existence with garbage collection.

In both approaches, region annotation poses considerable mental overheads for programmers, not to mention compatibility problems with legacy code. In addition, the quality of hand-annotation varies, with sub-optimal outcome for less experienced programmers.

In this paper, we provide a systematic attempt at formulating a region inference type system for a core subset of Java. We shall adopt the no-dangling approach and make use of lexically scoped regions, where the last region to be created is the first to be deallocated. The adoption of stack of scoped regions (with determinable lifetimes) is intended to simplify our inference method, and for conformance with RTSJ. Our main contributions are as follows:

- To the best of our knowledge, no one has successfully formalised a sound and complete region inference method for object-oriented languages. We propose a systematic solution to this problem using a core subset of the Java language.
- Our region type rules prevent dangling references by requiring that all field references outlive the current object. We formalise this explicitly through region lifetime constraints, without the need for an effect-based typing.
- We support classes and methods with region-polymorphism. In addition, region polymorphic recursion is supported for methods but not classes. This provides for an inference that is precise and yet efficient.
- Our inference scheme has been crafted to support class subtyping and method overriding. Previous work [12] required “phantom regions” to ensure soundness of class inheritance which may cause a loss in lifetime precision. We propose an improved solution without phantom regions, where modular compilation can be guided by a global dependency graph.
- We provide a theorem stating the correctness of our inference scheme which always leads to well-typed programs that are

[1]However, experiments done in [24] indicates no noticeable performance difference between the two approaches.

.

region-safe. Such well-typed programs are guaranteed not to create dangling references in either the store or stack.

- We also provide a compile-time analysis which ensures that downcast operations are region-safe. Previous proposal for a region-checking system (e.g. [6]) requires runtime checks for downcast operations.
- We describe a prototype implementation of our region inference system. Preliminary experiments suggest that our inference is competitive to hand annotation.

The remainder of this paper is organised as follows. Sec 2 introduces the language, called Core-Java, and its region-annotated target form. Sec 3 presents a set of guidelines for good region annotation. Sec 4 is devoted to region inference. It formalises the region inference rules and describes how method override conflicts can be rectified, together with correctness theorems. Sec 5 deals with the region-safety of downcast operations. Sec 6 reports on a preliminary experiment from our implementation of region inference. Sec 7 discusses related works, followed by some concluding remarks in Sec 8.

2. CORE-JAVA AND REGION TYPES

We present the syntax of a significant (subset of) Java-like language named *Core-Java* in Fig 1(a), which will be adopted as the source language for our region inference process. Core-Java is designed in the same minimalist spirit as Featherweight Java[26]. It supports assignments but is also an expression-oriented language where statements are expressions with the void type. Loops are omitted, as their effects can be captured by tail-recursion.

The target language, called region-annotated Core-Java, is given in Fig 1(b). This language is extended with region types and region constraints for each class and method. In addition, local regions with lexical scopes are introduced by the **letreg** declaration.

Note that r denotes a region variable, while v represents a data variable. The suffix notation s^* denotes a list of zero or more distinct syntactic terms that are separated by appropriate separators, while s^+ represents a list of one or more distinct syntactic terms. The syntactic terms, s, could be v, r, $(t\ v)$, *field*, etc. For example, $(t\ v)^*$ denotes $(t_1\ v_1, \ldots, t_n\ v_n)$ where $n \geq 0$.

The constraint $r_1 \succeq r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2. The constraint $r_1 = r_2$ denotes that r_1 and r_2 must be the same region, while $r_1 \neq r_2$ denotes the converse. Note that *this* is a reserved data variable referring to the current object, while *heap* is a reserved region to denote the global heap with unlimited lifetime, that is for any r: *heap* $\succeq r$.

Constraint abstraction from [23] of the form $q\langle r_1, .., r_n\rangle = rc$ denotes a parameterized constraint. For uniformity, the region constraint of every class and method will be captured with a constraint abstraction each. Region constraint for classes will also be known as *class invariant*, and be denoted using either $inv(cn)$ or $inv.cn\langle r_1, .., r_n\rangle$, with the latter in constraint abstraction form. Region constraint for methods will also be known as *method precondition*, and be denoted using either $pre(n)$ or $pre.n\langle r_1, .., r_m\rangle$, where n is either *cn.mn* or *mn*, depending on whether it is an instance or a static method. Note that *mn* denotes a method name. The set of constraint abstractions that is generated for a given program is denoted by Q. It is possible to inline the constraint abstractions, after fix-point iterations have been applied to resolve recursive constraints. In other words, constraint abstractions act as intermediate forms that can be later eliminated.

$$
\begin{aligned}
P &::= def^*\ meth^*\ eb \\
def &::= \textbf{class}\ cn_1\ \textbf{extends}\ cn_2 \{field^*\ meth^*\} \\
prim &::= \textbf{int} \mid \textbf{bool} \mid \textbf{void} \\
\tau &::= cn \mid prim \\
field &::= \tau\ f \\
meth &::= \tau\ mn((\tau\ v)^*)\ eb \\
eb &::= \{(\tau\ v)^*\ e\} \\
e &::= (cn)\ \textbf{null} \mid k \mid v \mid v.f \mid eb \\
&\quad \mid \textbf{new}\ cn(v^*) \mid v.f = e \mid v = e \\
&\quad \mid v.mn(v^*) \mid mn(v^*) \mid e_1\ ;\ e_2 \\
&\quad \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2
\end{aligned}
$$

(a) The Source Language

$$
\begin{aligned}
P &::= def^*\ meth^*\ eb\ Q \\
def &::= \textbf{class}\ ca_1\ \textbf{extends}\ ca_2\ \textbf{where}\ rc\ \{field^*\ meth^*\} \\
ca &::= cn\langle r^+\rangle \\
prim &::= \textbf{int} \mid \textbf{bool} \mid \textbf{void} \\
\tau &::= cn \mid prim \\
t &::= \tau\langle r^*\rangle \\
field &::= t\ f \\
meth &::= t\ mn\langle r^*\rangle((t\ v)^*)\ \textbf{where}\ rc\ eb \\
eb &::= \{(t\ v)^*\ e\} \\
e &::= (ca)\ \textbf{null} \mid k \mid v \mid v.f \mid eb \\
&\quad \mid \textbf{new}\ ca(v^*) \mid v.f = e \mid v = e \\
&\quad \mid v.mn\langle r^*\rangle(v^*) \mid mn\langle r^*\rangle(v^*) \mid e_1\ ;\ e_2 \\
&\quad \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \textbf{letreg}\ r\ \textbf{in}\ eb \\
rc &::= r_1 \succeq r_2 \mid r_1 = r_2 \mid r_1 \neq r_2 \mid \\
&\quad \mid rc_1 \wedge rc_2 \mid true \mid q\langle r_1, .., r_n\rangle \\
Q &::= \{(q\langle r_1, .., r_n\rangle = rc)^*\}
\end{aligned}
$$

(b) The Target Language

Figure 1: The Syntax of Core-Java

Each class definition in the target language is parameterized with one or more regions, to form a region type [28, 29, 12, 8]. For instance, $cn\langle r_1, \ldots, r_n\rangle$ is a class annotated with region parameters $r_1, \ldots, r_n$. Parameterization allows programmers to implement a region-polymorphic class whose components can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which a specified object of this class will be allocated. All other regions should *outlive* this region via the constraint $\wedge_{i:1..n}(r_i \succeq r_1)$. That is the regions of the components (possibly in $r_1, \ldots, r_n$) should have longer or equal lifetimes than the region (namely r_1) of its object. This condition, called *no-dangling* requirement, can prevent dangling references completely, as it guarantees that each object can never reference another object in a younger region. We do not impose region parameters on primitive types, since primitive values can be copied, and are directly placed in either the stack or inside its owner's space.

Each class declaration has a set of instance methods whose main purpose is to manipulate objects of the declared class, and where method overriding can occur. For completeness, we also provide a set of static methods with similar syntax as instance methods, but without overriding and without access to *this*. These two categories of methods are easily distinguished by their calling conventions.

Every method in the target language is decorated with zero or more region parameters. They are intended to capture the regions used by each method's parameters and result. Each method also has a region lifetime constraint that is consistent with the operations

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩
   where r2⪰r1 ∧ r3⪰r1 {
   Object⟨r2⟩ fst
   Object⟨r3⟩ snd
   Object⟨r4⟩ getFst⟨r4⟩() where r2=r4 {fst}
   void setSnd⟨r4⟩(Object⟨r4⟩ o) where r3=r4 {snd=o}
   Pair⟨r4,r5,r6⟩ cloneRev⟨r4,r5,r6⟩() where r2=r6∧r3=r5
     { Pair⟨r4,r5,r6⟩ tmp=new Pair⟨r4,r5,r6⟩(null,null);
     tmp.fst=snd; tmp.snd=fst; tmp }
   void swap() where r2=r3 {
     Object⟨r2⟩ tmp=fst; fst=snd; snd=tmp } }
               (a)
class List⟨r1,r2⟩ extends Object⟨r1⟩ where r2⪰r1 {
   Object⟨r2⟩ value
   List⟨r1,r2⟩ next
   Object⟨r3⟩ getValue⟨r3⟩() where r3=r2  {value}
   List⟨r3,r4⟩ getNext⟨r3,r4⟩() where r3=r1∧r4=r2 {next}
   void setNext⟨r3,r4⟩(List⟨r3,r4⟩ o)
    where r3=r1 ∧ r4=r2  {next = o} }
                 (b)
```

Figure 2: The `Pair` and `List` Classes

performed in the method body.

Two example programs are shown in Fig 2 for the `Pair` and `List` classes. We shall propose some guidelines for good region annotation in the next section.

3. REGION ANNOTATION GUIDELINES

To support region-based programming, we are required to add region parameters and constraints to each class and its methods. There are a number of ways to perform such region annotations. Our approach will be guided by the following principles:

- Keep regions of components in class (and the regions of parameters/results of methods) *distinct*, where possible.
- Keep region constraints on classes and methods *separate*, where the former is used to denote the expected invariant (including the no-dangling requirement) on the regions of each object of the class, while the latter denotes the pre-condition for method invocation.

The first principle allows more region polymorphism, where applicable. The second principle allows region constraints that are expected for each object to be placed in its class, while those that are specific to method invocation should be placed with the method. Take note that keeping region constraints with the methods where possible, allows these constraints to be selectively applied to only those objects which may invoke the methods. As we shall see later, this is helpful also to ensure the safety of method overriding. Let us consider how these two principles are applied.

3.1 Regions for Field Declarations

Consider the `Pair` class. As there are two fields (or components) in this class, we shall introduce a distinct region for each of them, as shown in the region-annotated counterpart in Fig 2(a). We shall add `r2⪰r1 ∧ r3⪰r1` to the class invariant, to impose the no-dangling requirement on every object of the class.

Consider also the `List` class with `next` as its recursive field. There are many different ways of annotating such recursive fields. While these ways may offer diversity, a key problem is that the best choice depends on how the objects are being manipulated. To keep matters simple, we shall use region *monomorphic recursion* for class declaration. With this, each recursive field would have the same annotation as its class, as illustrated in Fig 2(b). The class invariant, `r2⪰r1`, is solely for the no-dangling requirement.

Mutual recursive class declaration can be similarly handled. In this situation, all fields in the mutual recursive set will be used to provide the *same* set of regions for each mutual-recursive field. Mutual recursive class declarations can be identified through a global dependency graph on classes and methods. A technique to build such a dependency graph is described in Sec 4.3. For simplicity, we shall ignore mutual recursive class declarations in this paper.

The constraint abstractions for the `Pair` and `List` classes are:

```
inv.Pair<r1,r2,r3> = r2⪰r1 ∧ r3⪰r1
inv.List<r1,r2>    = r2⪰r1
```

3.2 Regions for Method Declarations

For each method declaration, we have to provide a set of regions to support the parameters and result of each method. For simplicity, no other regions will be made available for our methods. All regions used in the method will thus be mapped to these region parameters[2], or to the *heap*.

We must also provide region lifetime constraints over such region parameters (including the regions of `this` object). This naturally depends on how the objects are being manipulated in the method. Consider `getFst`, `setSnd` and `cloneRev` methods of the `Pair` class. We introduce a set of distinct region parameters for the methods' parameters and results, as shown in Fig 2(a). Moreover, the region (lifetime) constraints are based on the possible operations of the respective methods. For example, due to an assignment operation, we have `r2=r4` for the `getFst` method. Similarly, we have `r3=r4` for `setSnd`, while `r2=r6 ∧ r3=r5` are due to copying by the `cloneRev` method.

Consider the `swap` method. No region parameters are needed in this method as it does not have any parameters or result in its declaration. However, a region constraint `r2=r3` is still present due to the swapping operation on the current object itself. Though this constraint is exclusively on the regions of the current object, we shall keep it with the method. In this way, only those objects that might call this method will be subjected to this constraint.

Normally, the region constraints for methods also contain the region invariants of its parameters and result. For example, in `cloneRev`, we should also include the region invariant of the resulting type `Pair⟨r4,r5,r6⟩`, namely `r6⪰r4 ∧ r5⪰r4`.

For simplicity, we shall omit the presentation of such constraints in this paper. They can be recovered from the method's type signature. Except for this omission, the constraint abstractions for the various methods of the `Pair` class are as shown.

```
pre.Pair.getFst<r1,r2,r3,r4>         = r4=r2
pre.Pair.setSnd<r1,r2,r3,r4>         = r4=r3
pre.Pair.cloneRev<r1,r2,r3,r4,r5,r6> = r2=r6∧r3=r5
pre.Pair.swap<r1,r2,r3>              = r2=r3
```

3.3 Regions for Sub-class Declarations

Sub-classes typically have more fields and methods. Correspondingly, the regions of each sub-class are *extended* from its super-class, while its invariant represents a *strengthening* from the invariant of its super-class. These requirements are needed to support class subsumption. Consider:

```
class A⟨r1..rm⟩ extends Object⟨r1⟩ where φ_A
  ...
class B⟨r1..rn⟩ extends A⟨r1..rm⟩ where φ_B
  ...
```

We expect the regions of the sub-class `B`, namely ⟨`r1..rn`⟩, to be an extension of `A`, namely ⟨`r1..rm`⟩, with `n≥m`. Likewise, the

[2]This restriction may be lifted, if required.

region invariant of φ_B is a strengthening of φ_A, with the logical implication $\varphi_B \Rightarrow \varphi_A$. These requirements allow an object of the `B` class to be safely passed to any location where an `A` object is expected, as the invariant of the latter will hold by implication.

Method overriding poses another challenge which requires subtyping of functions to be taken into account. In general, the method of a sub-class is required to be a sub-type of the overridden method. According to [9], function sub-typing in object-oriented paradigm is sound if the normal parameters are contra-variant, while the selection parameters are co-variant.

Consider a method `mn` in class `A` that is being overridden by another method `mn` from the `B` sub-class. Let us assume that these two methods have the following method signatures, where `X,Y` denote some arbitrary classes with regions $r'_1, \ldots, r'_p$.

```
Y A.mn ⟨r'_1,...,r'_p⟩(X a) where φ_A.mn { ..  }
Y B.mn ⟨r'_1,...,r'_p⟩(X a) where φ_B.mn { ..  }
```

The constraint $\varphi_{A.mn}$ and $\varphi_{B.mn}$ are pre-conditions for the region parameters of `A.mn` and `B.mn` respectively. These parameters must be contra-variant for function sub-typing, requiring $\varphi_{A.mn} \Rightarrow \varphi_{B.mn}$. With the class invariant of `B` (as selection parameter), it is also safe to weaken this soundness check to $\varphi_B \wedge \varphi_{A.mn} \Rightarrow \varphi_{B.mn}$. The class invariant of `B` can be used as this method is only invoked when the current object is of the `B` class. Hence, strengthening φ_B may help comply with this soundness check. Its inclusion is critical to our approach for handling method overriding without phantom regions. Previous approaches (e.g. [8, 12]) did not have this provision.

Method overriding is particularly challenging for region inference. We shall introduce some techniques to ensure the compliance of the overriding checks in Section 4.4, after the basic region inference method has been presented.

4. REGION INFERENCE

We formalise a comprehensive set of type rules for region inference. Some examples are then used to illustrate the inference process. We also formalise the global dependency graph that guides the inference process, and describe how class subtyping and method overriding can be supported in the inference process. We then state the soundness of the region inference.

4.1 Region Inference Rules

Region inference aims to automatically derive region annotations. That is, given any program P, written in Core-Java, a program P' with appropriate region annotations can be derived via our region inference rules.

For simplicity, we assume the inputs to our region inference algorithm are *well-normal-typed* programs written in Core-Java. The normal type system for Core-Java can be derived from the region type system (given in our technical report [10]) by erasing all region stuffs (region annotations, region lifetime constraints, region environment). That is, if $\vdash P' : t$, then $\vdash_N$ **erase**(P') : **erase**(t). Notice that $\vdash_N$ denotes the well-normal-typedness of source programs written in Core-Java. The definition for the erasure function is straightforward and thus omitted here.

At the top-most level, the inference is captured by the following rule, where the region constraint φ is restricted to heap.

$$\frac{\begin{array}{c} P = \mathit{def}_{1..n}\ \mathit{meth}_{1..m}\ \mathit{eb} \quad P \vdash_{\mathit{def}} \mathit{def}_i \Rrightarrow \mathit{def}'_i; Q_i \ \ i \in 1..n \\ P; \{\} \vdash \mathit{meth}_i \Rrightarrow \mathit{meth}'_i; Q'_i \ \ i \in 1..m \quad \{\} \vdash \mathit{eb} \Rrightarrow \mathit{eb}' : t, \varphi \\ Q'' = \bigcup_{i \in 1..n} Q_i \cup \bigcup_{i \in 1..m} Q'_i \qquad P' = \mathit{def}'_{1..n}\ \mathit{meth}'_{1..m}\ \mathit{eb}'\ Q'' \end{array}}{\vdash P \Rrightarrow P'}$$

The above rule requires region inference to be performed on classes, methods and expressions, which are formalised as follows.

Class declaration: [CD]

For each class declaration, we must designate regions for allocating the objects of that class and their subcomponents. We also add region lifetime constraint to each class based on the constraints of its fields and methods. The overall rule is given below:

$$\frac{\begin{array}{c} \mathit{def}_2 = \textbf{class}\ \mathit{cn}_2\ \textbf{extends}\ \mathit{cn}_1\ \{(\tau_i\ f_i)_{1..p},\ (\mathit{cn}_2\ f_i)_{p+1..n},\ (\mathit{meth}_i)_{1..q}\} \\ \tau_i \neq \mathit{cn}_2 \quad P \vdash \tau_i \Rrightarrow \tau_i\langle r^*_i\rangle, \varphi_i \qquad i = 1,..,p \\ P \vdash \mathit{cn}_1 \Rrightarrow \mathit{cn}_1\langle r_{1..l}\rangle,\ \varphi_1 \qquad \Gamma = \mathit{this} : \mathit{cn}_2\langle r_{1..l}, r^*_{1..p}\rangle \\ P; \Gamma \vdash \mathit{meth}_i \Rrightarrow \mathit{meth}'_i; Q_i,\ i = 1,..,q \qquad \varphi = \mathit{inv.cn}_2\langle r_{1..l}, r^*_{1..p}\rangle \\ Q' = \{\varphi = (\varphi_1\ \wedge\ \bigwedge_{i:1..p} \varphi_i\ \wedge\ \bigwedge_{i:1..p}(r^*_i \succeq r_1))\} \\ \mathit{def}'_2 = \textbf{class}\ \mathit{cn}_2\langle r_{1..l}, r^*_{1..p}\rangle\ \textbf{extends}\ \mathit{cn}_1\langle r_{1..l}\rangle\ \textbf{where}\ \varphi \\ \{(\tau_i\langle r^*_i\rangle\ f_i)_{1..p},\ (\mathit{cn}_2\langle r_{1..l}, r^*_{1..p}\rangle\ f_i)_{p+1..n},\ (\mathit{meth}'_i)_{1..q}\} \end{array}}{P \vdash \mathit{def}_2 \Rrightarrow \mathit{def}'_2; Q' \cup \bigcup_{i \in 1..q} Q_i}$$

Region inference for class declaration is conducted in an inductive manner, by inferring regions for the fields first, and then collecting all necessary region variables and region constraints for the class. Recursive (and mutual recursive) fields should have the same region annotation as the class. Note that the region invariant of each class c is being captured by a constraint abstraction, named *inv.c*. Auxiliary rules used are presented in Fig 3.

Method declaration: [MD]

The following rule is used to infer necessary region variables for each method, and calculate lifetime constraint that should be imposed on them. Note that it gathers a fresh set of regions for the parameters and result of each method.

$$\frac{\begin{array}{c} \mathit{meth} = \tau_0\ \mathit{mn}((\tau_j\ v_j)_{j:2..p})\ \mathit{eb} \\ P \vdash \tau_i \Rrightarrow \tau_i\langle r^*_i\rangle,\ \varphi_i \quad i = 0, 2..p \\ \Gamma,\ (v_j : \tau_j\langle r^*_j\rangle)_{j:2..p} \vdash \mathit{eb} \Rrightarrow \mathit{eb}' : \tau'_0\langle x^*_0\rangle,\ \varphi \\ n = \mathit{if}\ (\mathit{this} : \mathit{cn}\langle r^+\rangle \in \Gamma)\ \mathit{then\ pre.cn.mn\ else\ pre.mn} \\ P \vdash \tau'_0\langle x^*_0\rangle <: \tau_0\langle r^*_0\rangle, \rho \quad y^* = r^*_2, .., r^*_p, r^*_0 \quad \varphi' = n\langle r^+, y^*\rangle \\ \mathit{meth}' = \tau_0\langle r^*_0\rangle\ \mathit{mn}\langle y^*\rangle((\tau_j\langle r^*_j\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi'\ (\mathit{eb}') \end{array}}{P;\ \Gamma\ \vdash \mathit{meth} \Rrightarrow \mathit{meth}'; \{\varphi' = (\varphi \wedge \bigwedge_{i:0,2..m} \varphi_i \wedge \mathit{ctr}(\rho))\}}$$

The method's body is inferred with a type $\tau'_0\langle x^*_0\rangle$ and region lifetime constraint, φ. This annotated type must be a sub-class of the expected output type, $\tau_0\langle r^*_0\rangle$. This inference rule can be used for both instance and static methods, with $\Gamma = \{\mathit{this} : \mathit{cn}\langle r^+\rangle\}$ for the former and Γ=$\emptyset$ for the latter. We also define a constraint abstraction for each method that is named as either *pre.cn.mn* or *pre.mn*, depending on whether it is an instance or a static method.Note that function $\mathit{ctr}(\rho)$ converts a substitution mapping to its corresponding set of equality constraints. For example, $\mathit{ctr}([x_1 \mapsto v, x_2 \mapsto w]) = (x_1 = v \wedge x_2 = w)$.

The inference rules for expressions have the following form:

$$\Gamma\ \vdash e \Rrightarrow e' : t, \varphi$$

Note that Γ is the type environment where types are annotated with regions. e, e' are resp. the un-annotated expression and the annotated counterpart. t is a region type, while φ is the derived (or gathered) region constraint. In what follows, we highlight the rules for instance method invocation and local region declaration which occurs in expression blocks. Other rules are presented in Fig 3.

Instance method invocation [EMI]

The inference rule for instance method invocation is given next. The rule gathers the respective method's region constraint, together with suitable equality constraint from its parameters' instantiations. The gathered region constraint is imposed as a pre-condition for each method invocation.

$$\frac{\begin{array}{c} P \vdash (\tau_0\langle x^*_0\rangle\ mn\langle y^*\rangle((\tau_j\langle x^*_j\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi\ \mathit{eb})\ \in\ \mathit{cn}\langle x^+_1\rangle \\ (v'_j : \tau'_j\langle x'^*_j\rangle) \in \Gamma \quad j = 2..p \qquad v'_1 : \mathit{cn}\langle x'^+_1\rangle \in \Gamma \\ P \vdash \tau_j\langle x'^*_j\rangle <: \tau_j\langle x^*_j\rangle, \rho'_j \qquad j = 2..p \\ \rho = \rho'_2 \ldots \rho'_p\ [x_0 \mapsto x'_0]^*\ [x_1 \mapsto x'_1]^+ \quad \mathit{fresh}\ x'^*_0 \end{array}}{\Gamma\ \vdash v'_1.mn(v'_2, ..., v'_p) \Rrightarrow v'_1.mn\langle \rho y^*\rangle(v'_2, ..., v'_p) : \tau_0\langle x'^*_0\rangle, \rho\varphi}$$

$$[\textbf{SUB1}]\quad \frac{\rho = [x'_i \mapsto x_i]_{i \in 1..n}}{P \vdash \tau\langle x_{1..n}\rangle <: \tau\langle x'_{1..n}\rangle, \rho}$$

$$[\textbf{SUB2}]\quad \frac{\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle\ \cdots\ \in P \qquad P \vdash cn'\langle x_{1..m}\rangle <: cn''\langle x'_{1..p}\rangle,\ \rho}{P \vdash cn\langle x_{1..n}\rangle <: cn''\langle x'_{1..p}\rangle,\ \rho}$$

$$[\textbf{MC}]\quad \frac{\vdash P \Rrightarrow P' \quad def \in P'}{def \in P} \qquad [\textbf{MSM}]\quad \frac{\vdash P \Rrightarrow P' \quad meth \in P'}{meth \in P}$$

$$[\textbf{CPT}]\quad \frac{}{P \vdash prim \Rrightarrow prim\langle\rangle, true} \qquad [\textbf{OBJ}]\quad \frac{fresh\ r}{P \vdash \text{Object} \Rrightarrow \text{Object}\langle r\rangle,\ true}$$

$$[\textbf{CCT}]\quad \frac{\textbf{class}\ cn\langle r_{1..n}\rangle \ldots\ \textbf{where}\ \varphi\ \{\ldots\} \in P \qquad fresh\ a_{1..n}}{P \vdash cn \Rrightarrow cn\langle a_{1..n}\rangle,\ ([r_{1..n} \mapsto a_{1..n}]\,\varphi)}$$

$$[\textbf{EV}]\quad \frac{v : \tau\langle r^*\rangle \in \Gamma}{\Gamma \vdash v \Rrightarrow v : \tau\langle r^*\rangle,\ true} \qquad [\textbf{EF}]\quad \frac{(v : cn\langle x^+\rangle) \in \Gamma \qquad P \vdash (\tau\langle x'^*\rangle\ f) \in cn\langle x^+\rangle}{\Gamma \vdash v.f \Rrightarrow v.f : \tau\langle x'^*\rangle,\ true}$$

$$[\textbf{EC1}]\quad \frac{P \vdash cn \Rrightarrow cn\langle r^+\rangle, \varphi}{\Gamma \vdash (cn)\textbf{null} \Rrightarrow (cn\langle r^+\rangle)\textbf{null} : cn\langle r^+\rangle, true} \qquad [\textbf{EC2}]\quad \frac{}{\Gamma \vdash k \Rrightarrow k : prim_k\langle\rangle, true}$$

$$[\textbf{ES}]\quad \frac{\Gamma \vdash e_1 \Rrightarrow e'_1 : t_1, \varphi_1 \qquad \Gamma \vdash e_2 \Rrightarrow e'_2 : t_2, \varphi_2}{\Gamma \vdash e_1\,;e_2 \Rrightarrow e'_1\,;e'_2 : t_2, \varphi_1 \wedge \varphi_2}$$

$$[\textbf{EA}]\quad \frac{lhs = v \mid v.f \qquad \Gamma \vdash e \Rrightarrow e' : t', \varphi' \qquad \Gamma \vdash lhs \Rrightarrow lhs : t, true \qquad P \vdash t' <: t, \rho \quad \varphi = \varphi' \wedge ctr(\rho)}{\Gamma \vdash lhs = e \Rrightarrow lhs = e' : \textbf{void}, \varphi}$$

$$[\textbf{EN}]\quad \frac{\begin{array}{c} P \vdash cn \Rrightarrow cn\langle x^+\rangle, \varphi_0 \\ fieldlist(cn\langle x^+\rangle) = (t_i\ f_i)_{i:1..p} \\ (v_i : t'_i) \in \Gamma \quad P \vdash t'_i <: t_i,\ \rho_i\ i = 1..p \\ \varphi = \bigwedge_{i:0..p} ctr(\rho_i) \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{new}\ cn(v^*) \Rrightarrow \\ \qquad \textbf{new}\ cn\langle x^+\rangle(v^*) : cn\langle x^+\rangle, \varphi \end{array}}$$

$$[\textbf{EIF}]\quad \frac{\begin{array}{c} (v_0 : \textbf{bool}\langle\rangle) \in \Gamma \qquad t = msst(t_1, t_2) \\ \Gamma \vdash e_1 \Rrightarrow e'_1 : t_1, \varphi_1 \quad P \vdash t_1 <: t,\ \rho_3 \\ \Gamma \vdash e_2 \Rrightarrow e'_2 : t_2, \varphi_2 \quad P \vdash t_2 <: t,\ \rho_4 \\ \varphi = \varphi_1 \wedge \varphi_2 \wedge ctr(\rho_3) \wedge ctr(\rho_4) \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{if}\ v_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Rrightarrow \\ \qquad \textbf{if}\ v_0\ \textbf{then}\ e'_1\ \textbf{else}\ e'_2 : t, \varphi \end{array}}$$

$$[\textbf{EMS}]\quad \frac{\begin{array}{c} \vdash (\tau_0\langle x_0^*\rangle\ mn\langle y^*\rangle((\tau_i\langle x_i^*\rangle\ v_i)_{i:1..p})\ \textbf{where}\ \varphi\ eb) \in P \\ (v'_i : \tau'_i\langle x'^*_i\rangle) \in \Gamma \quad i = 1..p \\ P \vdash \tau'_i\langle x'^*_i\rangle <: \tau_i\langle x_i^*\rangle, \rho'_i \quad i = 1..p \\ \rho = \rho'_1 \ldots \rho'_p\,[x_0 \mapsto x'_0]^* \qquad fresh\ x'^*_0 \end{array}}{\begin{array}{l} \Gamma \vdash mn(v'_1, \ldots, v'_p) \Rrightarrow \\ \qquad mn\langle \rho y^*\rangle(v'_1, \ldots, v'_p) : \tau_0\langle x'^*_0\rangle, \rho\varphi \end{array}}$$

Figure 3: Subtyping Rules, Auxiliary Rules and Other Region Inference Rules

Expression block $[\textbf{EB}]$

A key inference rule concerns how regions may be localised at the expression block. This rule attempts to identify regions that are effectively dead thereafter.

We introduce the function *reg* (shown below) to extract region variables from a type environment. We also use $reg(\varphi)$ to get all regions from a given constraint φ.

$$reg(\{\}) = \{\} \qquad reg(v : \tau\langle r^*\rangle, \Gamma) = \{r^*\} \cup reg(\Gamma)$$

The following set $ors(\varphi, s_1, s_2)$ is used to gather those regions out of s_1 that have equal-or-longer lifetime than some region of s_2 with respect to φ.

$$ors(\varphi, s_1, s_2) = \{r \mid r \in s_1 \wedge \exists r' \in s_2 \cdot (\varphi \Rightarrow r \succeq r')\}$$

Its complementary set is $\overline{ors}(\varphi, s_1, s_2) = s_1 - ors(\varphi, s_1, s_2)$. This set consists of regions that can be *localised* as they do not escape the expression block. These localised regions enjoy the same lifetime and are coalesced into a single region, as shown in the rule below.

$$\frac{\begin{array}{c} P \vdash \tau_j \Rrightarrow \tau_j\langle x_j^*\rangle, \varphi_j,\ j = 1..p \\ \Gamma, \{v_j : \tau_j\langle x_j^*\rangle\}_{j:1..p} \vdash e \Rrightarrow e' : \tau\langle r^*\rangle, \varphi \\ \rho = st(\varphi \wedge \bigwedge_{j:1..p} \varphi_j,\ \bigcup_{j:1..p}\{x_j^*\},\ reg(\Gamma)) \\ rs = \overline{ors}(\varphi \wedge \bigwedge_{j:1..p} \varphi_j,\ reg(\varphi) \cup \bigcup_{j:1..p}\{x_j^*\},\ \{r^*\} \cup reg(\Gamma)) \\ rs \neq \emptyset \qquad fresh\ r \qquad \rho' = \{x \mapsto r \mid \forall x \in rs\} \\ \varphi' = \rho((\varphi \wedge \bigwedge_{j:1..p} \varphi_j) \backslash rs) \end{array}}{\begin{array}{l} \Gamma \vdash \{(\tau_j\ v_j)_{j:1..p}\ e\} \Rrightarrow \\ \qquad \{\textbf{letreg}\ r\ \textbf{in}\ \rho'\rho\,\{(\tau_j\langle x_j^*\rangle\ v_j)_{j:1..p}\ e'\}\} : \rho\tau\langle r^*\rangle, \varphi' \end{array}}$$

Those regions that may escape the block can be traced to regions that exist in either the type environment or the result type. All regions that outlive these also escape.

We provide a definition that maps the regions from s_1 to their equal counterparts in s_2 with respect to the constraint φ. Note that $sel(s_x)$ makes an arbitrary choice of one element from a non-empty set of equal regions s_x.

$$\begin{array}{l} st(\varphi, s_1, s_2) = \{x \mapsto sel(s_x) \mid x \in s_1\ \wedge \\ \qquad\qquad s_x = \{y \mid y \in s_2\ \wedge\ \varphi \Rightarrow (x = y)\}\ \wedge\ s_x \neq \emptyset\} \end{array}$$

This definition was used to obtain a mapping ρ that help identify regions that are equivalent to regions in the type environment, Γ. As stated, these and older regions escape the expression block.

It may also be possible that none of the regions can be localised. This is signified by $rs = \emptyset$, where we would just return the annotated expression block without **letreg**.

4.2 Examples

Let us show the workings of our region inference rules via some examples.

4.2.1 Localised Regions

Consider an example with four `Pair` objects that are connected as shown in Fig 4(a). Its code fragment is given in Fig 4(b). As shown in Fig 4(c), our inference rules would initially decorate each local variable and constructor with new distinct regions, and proceed to gather the constraints from each statement. A set of equality and outlive constraints will be collected, and this can be applied to reduce the number of distinct regions, to obtain the simplified region-annotated code in Fig 4(d).

The result type of this block is $\texttt{Pair}\langle\texttt{r2,r2a,r4}\rangle$, with constraints $\texttt{r4a}\succeq\texttt{r4}\wedge\texttt{r4b}\succeq\texttt{r4}$, $\texttt{r4}\succeq\texttt{r3}\wedge\texttt{r3b}\succeq\texttt{r3}$, $\texttt{r2a}\succeq\texttt{r2}\wedge\texttt{r4}\succeq\texttt{r2}$ and $\texttt{r2}\succeq\texttt{r1}\wedge\texttt{r3}\succeq\texttt{r2}$. Based on this type and region lifetime constraints, our rule can deduce that all the regions escape this block, except for regions `r1`,`r3` and `r3b`. These non-escaping regions can be localised by $[\textbf{EB}]$ to a single region (say `r`), as shown in Fig 4(e).

4.2.2 Recursive Methods

We shall illustrate how region annotations and constraints are inferred for recursive methods. Fig 5(a) contains a static method which merges two lists of objects. Due to the swapping parameters at its recursive call, the resulting list contains alternating elements from both lists.

Region inference would introduce a constraint abstraction, called `pre.join`, and build a definition for it, as shown in Fig 5(c) (after simplification). As the constraint abstraction is recursive, we shall apply fix-point iteration to obtain a closed form formula for it. Starting with the initial version of $\texttt{pre.join}_0$, we progressively refine the definition of `pre.join` until a fix-point is reached, as highlighted in Fig 5(d).

$$\begin{tikzcd}
& p1 \arrow[dl] \arrow[dr] & \\
p2 \arrow[dr] & & p3 \arrow[dl] \\
& p4 &
\end{tikzcd}$$

(a) Acyclic data structure

```
{Pair p4=new Pair(null,null);
 Pair p3=new Pair(p4,null);
 Pair p2=new Pair(null,p4);
 Pair p1=new Pair(p2,null);
 p1.setSnd(p3);
 p2 }
```

(b) Original code fragment

```
{Pair⟨r4,r4a,r4b⟩ p4 =
   new Pair⟨r4,r4a,r4b⟩(null,null);
   //r4a⪰r4 ∧ r4b⪰r4
 Pair⟨r3,r3a,r3b⟩ p3 =
   new Pair⟨r3,r3a,r3b⟩(p4,null);
   // r3a⪰r3 ∧ r3b⪰r3 ∧ r3a=r4
 Pair⟨r2,r2a,r2b⟩ p2 =
   new Pair⟨r2,r2a,r2b⟩(null,p4);
   // r2a⪰r2 ∧ r2b⪰r2 ∧ r2b=r4
 Pair⟨r1,r1a,r1b⟩ p1 =
   new Pair⟨r1,r1a,r1b⟩(p2,null);
   // r1a⪰r1 ∧ r1b⪰r1 ∧ r1a=r2
 p1.setSnd⟨r3⟩(p3); // r1b=r3
 p2 }
```

(c) Region-annotated target program.

```
{Pair⟨r4,r4a,r4b⟩ p4 =
   new Pair⟨r4,r4a,r4b⟩(null,null);
   //r4a⪰r4 ∧ r4b⪰r4
 Pair⟨r3,r4,r3b⟩ p3 =
   new Pair⟨r3,r4,r3b⟩(p4,null);
   // r4⪰r3 ∧ r3b⪰r3
 Pair⟨r2,r2a,r4⟩ p2 =
   new Pair⟨r2,r2a,r4⟩(null,p4);
   // r2a⪰r2 ∧ r4⪰r2
 Pair⟨r1,r2,r3⟩ p1 =
   new Pair⟨r1,r2,r3⟩(p2,null);
   // r2⪰r1 ∧ r3⪰r1
 p1.setSnd⟨r3⟩(p3);
 p2 }
```

(d) Simplified target program.

```
{letreg r in
 { Pair⟨r4,r4a,r4b⟩ p4 =  new Pair⟨r4,r4a,r4b⟩(null,null); //r4a⪰r4 ∧ r4b⪰r4
   Pair⟨r,r4,r⟩ p3     =  new Pair⟨r,r4,r⟩(p4,null);  // r4⪰r
   Pair⟨r2,r2a,r4⟩ p2  =  new Pair⟨r2,r2a,r4⟩(null,p4);  // r2a⪰r2 ∧ r4⪰r2
   Pair⟨r,r2,r⟩ p1     =  new Pair⟨r,r2,r⟩(p2,null);  // r2⪰r
   p1.setSnd⟨r⟩(p3);  p2 } }
```

(e) Final target program.

Figure 4: Example Inference with Localised Region

Fix-point analysis always terminates for our constraint abstraction. This is due to the finite set of possible constraints made up from a bounded set of regions. Note that this example relies on region-polymorphic recursion, without which some loss in lifetime precision occurs.

4.3 Global Dependency Graph

Due to a fairly complex inter-dependency between classes and methods, our region inference system is required to process the classes and methods in some particular order, chiefly to support program hierarchy and class inheritance. This ordering is important for modular compilation.

For this purpose, we propose a set of inference rules that attempt to identify the following kinds of dependencies:

- $cn_i \rightarrow cn_j$: denotes cn_j is a component of cn_i or that cn_i is a sub-class of cn_j.
- $mn_i \rightarrow cn_j$: denotes mn_i made use of class cn_j in its body.
- $mn_i \rightarrow mn_j$: denotes mn_i calls mn_j.
- $cn'.mn \rightarrow cn.mn$: from override check.
- $cn \rightarrow cn.mn$: from override check.

The first three dependencies arise from the constituents of each class and method (static and instance), while the last two dependencies are induced by method overriding check of the form:

$$inv.cn\langle..\rangle \wedge pre.cn'.mn\langle..\rangle \Rightarrow pre.cn.mn\langle..\rangle$$

The inference mechanism for these dependencies is straightforward. The details are in a companion report [10]. The final dependency graph has the classes and methods organised into a hierarchy of strongly connected components (SCCs). Each set of classes in a SCC will be regarded as a mutual-recursive class declaration, and be processed according to region monomorphic recursion principle that we have adopted for classes. Correspondingly, each set of methods in a SCC is regarded as a mutual-recursive set for fix-point iteration. Through a bottom-up processing of each SCC, we can perform region inference in a modular and systematic fashion.

4.4 Subtyping and Inheritance

As mentioned in Section 3, class subtyping and method overriding must comply with their respective checks to ensure the soundness of subsumption.

The class subtype check is relatively easy to comply. The existing [CD] rule already accumulates the invariant from each class to its sub-class in order to ensure:

`inv(subclass) ⇒ inv(superclass)`

In contrast, the method overriding check is more complex. Consider a class `A` and its sub-class `B`, where a method `A.mn` is being overridden by `B.mn`. For method overriding to be sound, we require the following check to be satisfied:

`inv(B) ∧ pre(A.mn) ⇒ pre(B.mn)`

Such a check may not hold initially. To rectify this, region inference can perform selective strengthening of the above premises, with the following considerations:

1. We can strengthen either `inv(B)` or `pre(A.mn)`, or both.
2. Strengthening `pre(A.mn)` can be problematic as some regions are present in class `B` but not `A`.

These two issues can be considered in a systematic way. The next section describes our formalisation.

4.4.1 *Override Conflict Resolution*

Recall that each method override check is of the form:

$$inv.cn\langle r_1..r_n\rangle \wedge pre.cn'.mn\langle r_1..r_m, r'_1..r'_p\rangle$$
$$\Rightarrow pre.cn.mn\langle r_1..r_n, r'_1..r'_p\rangle$$

where *cn.mn* is a method in class *cn* that overrides a corresponding method in its super-class, cn'. Such a check is in conflict if the pre-condition of the overridden method is not strong enough. We can strengthen either $inv.cn\langle r_1..r_n\rangle$, or $pre.cn'.mn\langle r_1..r_m, r'_1..r'_p\rangle$ or both. This resolution can be achieved by examining each basic constraint of $pre.cn.mn\langle r_1..r_n, r'_1..r'_p\rangle$ to determine if (i) it is already valid, or (ii) can be added to $pre.cn'.mn$, or (iii) can be added to *inv.cn*, or (iv) can be split into a region equality in *inv.cn* and a modified constraint in $pre.cn'.mn$. We formalise this conflict resolution as the following inference rule:

$$I, A, B \vdash I', A'$$

```
List join(List xs, List ys)
{if isNull(xs) then
  if isNull(ys) then (List)null
  else join(ys,xs)
 else {
  x=xs.getValue();
  r=join(ys,xs.getNext());
  new List(x,r)
  }
}
```

(a) Source Program

```
List⟨r5,r6⟩ join⟨r1,..,r6⟩(List⟨r1,r2⟩ xs, List⟨r3,r4⟩ ys)
  where pre.join⟨r1,..,r6⟩
{if isNull⟨r1,r2⟩(xs) then
  if isNull⟨r3,r4⟩(ys) then (List⟨r5a,r6a⟩)null // r5=r5a ∧ r6a=r6
  else join⟨r3,r4,r1,r2,r5b,r6b⟩(ys,xs) // r5=r5b ∧ r6b=r6
 else {
  Object⟨r7⟩ x; List⟨r8,r9⟩ r;
  x=xs.getValue⟨r2⟩(); // r7=r2
  xs=xs.getNext⟨r1,r2⟩();
  r=join⟨r3,r4,r1,r2,r5c,r6c⟩(ys,xs); //r8=r5 ∧ r6=r9 ∧ r5=r5c ∧ r6c=r6
  new List<r10,r11>(x,r) //r10=r5 ∧ r6=r11 ∧ r2=r11
  } }
```

(b) Region-Annotated Target Program

```
List⟨r5,r6⟩ join⟨r1,..,r6⟩(List⟨r1,r2⟩ xs, List⟨r3,r4⟩ ys)
  where pre.join⟨r1,..,r6⟩
{if isNull⟨r1,r2⟩(xs) then
  if isNull⟨r3,r4⟩(ys) then (List⟨r5,r6⟩)null
  else join⟨r3,r4,r1,r2,r5,r6⟩(ys,xs)
 else {
  Object⟨r2⟩ x; List⟨r5,r6⟩ r;
  x=xs.getValue⟨r2⟩();
  xs=xs.getNext⟨r1,r2⟩();
  r=join⟨r3,r4,r1,r2,r5,r6⟩(ys,xs);
  new List<r5,r6>(x,r) // r2=r6
  } }
```

(c) Final Target Program

```
Q = {pre.join⟨r1,..,r6⟩=(r2=r6)∧pre.join⟨r3,r4,r1,r2,r5,r6⟩}

pre.join0⟨r1,...,r6⟩ = True

pre.join1⟨r1,...,r6⟩ = r2=r6 ∧ pre.join0⟨r3,r4,r1,r2,r5,r6⟩
                     = r2=r6

pre.join2⟨r1,...,r6⟩ = r2=r6 ∧ pre.join1⟨r3,r4,r1,r2,r5,r6⟩
                     = r2=r6 ∧ r4=r6

pre.join3⟨r1,...,r6⟩ = r2=r6 ∧ pre.join2⟨r3,r4,r1,r2,r5,r6⟩
                     = r2=r6 ∧ r4=r6
```

(d) Fix-Point Analysis

Figure 5: Region Inference for Recursive Method

```
class Triple⟨r1,r2,r3,r3a⟩
  extends Pair⟨r1,r2,r3⟩ where r2⪰r1∧r3⪰r1∧r3a⪰r1 {
  Object⟨r3a⟩ thd
  Pair⟨r4,r5,r6⟩ cloneRev⟨r4,r5,r6⟩() where r2=r6∧r3a=r5
    { Pair⟨r4,r5,r6⟩ tmp =new Pair⟨r4,r5,r6⟩(null,null);
    tmp.fst=thd; tmp.snd=fst; tmp}
```

Figure 6: The `Triple` Class

Note that I denotes the class invariant of the subclass, A denotes the pre-condition of the overridden method (from superclass), while B represents the pre-condition of the overriding method (from subclass). The results I', A' are strengthened version of I, A which satisfies the soundness of overriding. Each constraint is expressed as a set of atomic constraints in conjunctive form. In the following rules, we assume that $R_I = \{r_1..r_n\}$, $R_A = \{r_1..r_m, r'_1..r'_p\}$ and $R_m = \{r_1..r_m\}$.

$$\frac{I \wedge A \Rightarrow B}{I, A,\ B \vdash I, A} \qquad \frac{\begin{array}{c} c \in B \quad \neg(I \wedge A \Rightarrow c) \\ reg(c) \subseteq R_A \qquad I, A \wedge c,\ B \vdash I', A' \end{array}}{I, A,\ B \vdash I', A'}$$

$$\frac{\begin{array}{c} c \in B \quad \neg(I \wedge A \Rightarrow c) \\ reg(c) \subseteq R_I \\ I \wedge c, A,\ B \vdash I', A' \end{array}}{I, A,\ B \vdash I', A'} \qquad \frac{\begin{array}{c} c \in B \quad \neg(I \wedge A \Rightarrow c) \\ \exists \rho : reg(c) \cap (R_I - R_A) \rightarrow R_m \\ I \wedge ctr(\rho), A \wedge \rho c,\ B \vdash I', A' \end{array}}{I, A,\ B \vdash I', A'}$$

Let us illustrate this resolution mechanism through an example in Fig 6. For the overriding `cloneRev` method there are two basic constraints present: `r2=r6` and `r3a=r5`. The first basic constraint is already satisfiable. However, the second constraint cannot be placed in the class invariant of `Triple`, nor in the pre-condition of `Pair.cloneRev`. Nonetheless, we could still split it into two constraints `r3=r3a ∧ r3=r5` that can be added to `inv.Triple` and `pre.Pair.cloneRev`, respectively. Note that we have a choice of using either ρ_1=`(r3a=r3)` or ρ_2=`(r3a=r2)`. We chose the former since `(r3=r5)` exists in `pre.Pair.cloneRev` but not `(r2=r5)`. While multiple solutions exist, we shall always opt for a solution which minimises on the number of new equality constraints added.

4.5 Correctness

This subsection is devoted to the correctness of our region inference. We have given a comprehensive set of region type checking rules and proven the safety properties in our technical report [10]. In our type system for region-annotated Core-Java, $P \vdash_{def} def$ denotes the class declaration *def* is well-formed, $P; \Gamma; R; \Psi \vdash_{meth} meth$ indicates the method *meth* is well-formed, while $P; \Gamma; R; \Psi \vdash e : t$ represents that the expression e is well-typed. These notations will be used in the following theorem.

THEOREM 1. *Suppose* $\vdash P \Rrightarrow P'$.

(a). If $\Gamma \vdash e \Rrightarrow e' : t,\ \varphi$, *and all classes and static methods that* e' *depends are well-formed in* P', *then there exists* $R \supseteq reg(\Gamma) \cup reg(t)$, *such that* $P'; \Gamma; R; \varphi \vdash e' : t$.

(b). If $P; \Gamma \vdash meth \Rrightarrow meth'; Q$, *and all classes and static methods that* $meth'$ *depends are well-formed in* P', *then* $P'; \Gamma; R; \Psi \vdash_{meth} meth'$ *where* $R = \{r_{1..n}, heap\}, \Gamma = \{this : cn\langle r_{1..n}\rangle\}, \Psi = inv.cn\langle r_{1..n}\rangle$, *if* $meth' \in cn\langle r_{1..n}\rangle$; $R = \{heap\}, \Gamma = \emptyset, \Psi = true$, *otherwise.*

(c). If $P \vdash def \Rrightarrow def'; Q$, *and all classes and static methods that* def' *depends are well-formed in* P', *then* $P' \vdash_{def} def'$.

THEOREM 2 (CORRECTNESS). *Given any well-normal-typed source program* P *in Core-Java.*

(a). There exists a program P' *in region-annotated Core-Java, such that* $\vdash P \Rrightarrow P'$.

(b). If $\vdash\ P \Rrightarrow P'$, *then there exists a region-annotated type* t', *such that* $\vdash P' : t'$.

The proofs for these theorems can be found in [10].

```
class A⟨r1,r2⟩ ...;
class B⟨r1,r2,r3⟩ extends A⟨r1,r2⟩ ...;
class C⟨r1,r2,r3⟩ extends A⟨r1,r2⟩ ...;
class D⟨r1,r2,r3,r4⟩ extends C⟨r1,r2,r3⟩ ...;
class E⟨r1,r2,r3,r4,r5⟩ extends A⟨r1,r2⟩ ...;
 :
A⟨r1,r2⟩ a;
A⟨r3,r4⟩ a2;
if ..  then
  a = new B⟨r1,r2,r5⟩(..)  // B upcast to A
else ..
  a = new C⟨r1,r2,r6⟩(..)  // C upcast to A
else ..
  a = new E⟨r1,r2,r7,r8,r9⟩(..)  // E upcast to A
B⟨r1,r2,r10⟩ b = (B) a; // downcast to B
C⟨r1,r2,r11⟩ c = (C) a; // downcast to C
D⟨r1,r2,r11,r12⟩ d = (D) c; //downcast to D
```

Figure 7: Program Fragment with Downcasts

5. HANDLING DOWNCAST

One important feature that is missing from Core-Java is the *downcast* operation. In general, this operation may be type unsafe if the object in question is not the expected subtype. Unless both the subtype and supertype have the same set of regions, it may also be possible for the downcasted regions to be wrong. In [6], a type-passing approach was extended to carry ownership information to allow this property to be checked at runtime. If a region error is detected at runtime, the blame can still be pinned on the programmer for wrong region annotation. With automatic region inference, the onus will be on the type inference system to prevent such a situation; moreover at compile-time. Let us see how this problem can be resolved.

Downcast and upcast represent opposite operations. In our present formulation, regions may be lost during upcast operations. As a consequence, we are unable to carry out region-safe downcast, as the lost regions cannot be recovered.

To illustrate the problem, consider a program fragment with class hierarchy in Fig 7. During the upcast operations, regions `r5,r6,r7,r8,r9` are lost. These lost regions cannot be recovered when subsequent downcast operations are performed, leading to unknown regions `r10,r11,r12`.

To support region-safe downcasting, a key technique is to preserve the regions that were supposedly lost during upcasting. We propose two techniques to ensure this.

In our first solution, we preserve lost regions during upcasting by equating them with the first region. In this way, downcasting can always be achieved through this first region. For example, the following upcast operation forces region `r3` to be equivalent to `r1`:

```
A⟨r1,r2⟩ a = new B⟨r1,r2,r3⟩(..)  // r3=r1
```

As a consequence, we can easily recover the lost region during a downcast operation, as follows:

```
···(B⟨r4,r5,r6⟩) a···// r4=r1∧r5=r2∧r6=r1
```

Applying this technique to our earlier program fragment results in the following, where the imposed region constraints are shown as comments.

```
A⟨r1,r2⟩ a;
A⟨r3,r4⟩ a2;
if ..  then
  a = new B⟨r1,r2,r5⟩(..)  // r5=r1
else ..
  a = new C⟨r1,r2,r6⟩(..)  // r6=r1
else ..
  a = new E⟨r1,r2,r7,r8,r9⟩(..)  // r7=r8=r9=r1
(B⟨r1,r2,r10⟩) b = (B) a; // r10=r1
(C⟨r1,r2,r11⟩) c = (C) a; // r11=r1
(D⟨r1,r2,r11,r12⟩) d = (D) c; // r12=r1
```

While this solution is simple to implement, some lifetime precision are lost due to the region equality constraints imposed.

Another solution is to maintain extra regions during upcasting, if they may be downcasted later. Specifically, all objects that may be downcasted (to some sub-classes) must be padded in advance with sufficient number of extra regions to support region-safe downcasting later. A flow-based analysis is required to determine the scope to which each object and its components may be downcasted.

Based on the earlier program fragment, we can determine that the object `a` may be downcasted to `B,C,D`, while the object `c` may be downcasted to `D`. As the sub-class (`D`) has the maximum number of regions, we shall pad both these types with upto four regions, namely `A⟨r1,r2⟩[r3,r4]` for `a`, and `C⟨r1,r2,r3⟩[r4]` for `c`, to support region-safe downcast to either `B,C` or `D`. Note that `[r3,r4]` and `[r4]` denote the padded regions for `a` and `c`, respectively. In contrast, object `a2` and `b` are never downcasted, hence we do not impose any extra regions on their class types. Our earlier program fragment can now be transformed to:

```
A⟨r1,r2⟩[r3,r4] a;
A⟨r1′,r2′⟩ a2;
if ..  then
  a = new B⟨r1,r2,r5⟩(..)  //r5=r3
else ..
  a = new C⟨r1,r2,r6⟩(..)  //r6=r3
else ..
  a = new E⟨r1,r2,r7,r8,r9⟩(..)  // not in downcast
(B⟨r1,r2,r10⟩) b = (B) a; // r10=r3
(C⟨r1,r2,r11⟩[r12]) c = (C) a; //r11=r3 ∧ r12=r4
(D⟨r1,r2,r11,r12⟩) d = (D) c; // r12=r4
```

Take note that the extra regions of `E`, namely `r7,r8,r9` are not made equal to the padded regions of `a`. The reason is that the `E` class is not in the set to which this object may be downcasted. Hence, any downcast on this object will fail, regardless of the padded regions.

To support our new approach to region-safe downcast, we propose a backward analysis technique to find potential downcast set for each object in our program.

Details of the flow analysis is given in the technical report [10]. In the case of the example, the gathered set of flows is:

$$\{a \dashrightarrow lb,\ a \dashrightarrow lc,\ a \dashrightarrow le,\ b \xrightarrow{B} a,\ c \xrightarrow{C} a,\ d \xrightarrow{D} c\}$$

where $l_1 \dashrightarrow l_2$ and $l_1 \xrightarrow{C} l_2$ both denote the flow of an object from location l_2 to l_1, with the latter arrow also capturing an explicit downcast to C operation. This set is initially converted to:

$$a[B,C] \ \wedge\ c[D] \ \wedge\ \{a \dashrightarrow lb,\ a \dashrightarrow lc,\ a \dashrightarrow le,\ b \dashrightarrow a,\ c \dashrightarrow a,\ d \dashrightarrow c\}$$

Note that $l[C_1, .., C_n]$ captures that possible downcast set, $[C_1, .., C_n]$.

Applying the closure of backward flows gives:

$$\{b \dashrightarrow lb,\ b \dashrightarrow lc,\ b \dashrightarrow le,\ c \dashrightarrow lb,\ c \dashrightarrow lc,\ c \dashrightarrow le,\ d \dashrightarrow lb,\ d \dashrightarrow lc,\ d \dashrightarrow le,\ d \dashrightarrow a\}$$

Applying the downcast closure rule gives:

$$a[B,C,D],\ c[D],\ lb[B,C,D],\ lc[B,C,D],\ le[B,C,D]$$

This outcome can guide the padding of extra regions for each variable declaration and object creation site. Moreover, the analysis can sometimes tell if downcast is bound to fail. For example, all possible downcast at point *le* will fail since an *E* object is created that is not a sub-class of *B*, or *C* or *D*. Under this scenario, we need not instantiate the padded regions, as region preservation is only required when downcast succeeds.

6. IMPLEMENTATION

We have constructed a prototype region inference system for Core-Java. The output from region inference can be verified by a separate type-checking system that we have also built. The entire

Program	Size (bytes)			Compile-Time (sec)		Diff. in
	Source	Target	Annotation (%)	Inference	Checking	`letreg`
Sieve of Eratosthenes	1616	2147	32.9	0.094	0.083	0
Ackermann	1523	1837	20.6	0.016	0.042	0
Merge Sort	3720	4671	25.6	0.224	0.157	0
Mandelbrot	2802	3389	20.9	0.067	0.090	0
Naive Life	2367	3353	41.7	0.237	0.119	0
Optimized Life (array)	2420	3477	43.7	0.208	0.132	0
Optimized Life (dangling)	466	1013	117.4	0.014	0.034	-1
Optimized Life (stack)	1202	1603	33.4	0.038	0.048	0

Figure 8: Comparative Statistics

prototype system was built using Glasgow Haskell compiler[27] where we have also added a library to solve region outlive constraints.

The primary objective of our initial experiments is to show that our proposed region inference produces region annotation that are comparable to those done by hand. We tested our system on a set of RegJava benchmark programs from [12] that have been hand-annotated for their region-checking system. Figure 8 summarises the statistics obtained for each program that we inferred.

The last column indicates the difference in the number of localised regions between our inferred annotations, and that which were hand-annotated in [12]. All annotations are identical, except for *optimized life (with dangling)*. Our inference produces one less local region, since we use *no-dangling* rather than *no-dangling access* that was assumed by RegJava checker. Our region inference is thus comparable in performance to human experts. Take note that the type annotations are on average around 33.3% of the program size. This represents a sizeable mental effort for a programmer with only a region type-checker. Note also that the analysis time for region inference and region checking are comparable for the benchmark we tested. We plan to scale up our experiments in the near future. To do that, we will have to extend our language and inference mechanism to handle other Java language features. A preliminary discussion on how this may be achieved can be found in our technical report.

7. RELATED WORK

Tofte and Talpin [28, 29] proposed a region inference approach for a typed call-by-value λ-calculus, and tested their approach in a region-based implementation of Standard ML. In their approach, all values (including function values) are put into regions at runtime, and all points of region placement can be inferred automatically using a type and effect based program analysis. The treatment of reference type is similar to that for objects; as all values stored in a given reference must be placed in the same region which in turn outlives the region where the reference is located. Their outlive requirement is specified *indirectly* through region effects and on what regions must not escape. Other than this, their method is tailored mainly to functional language features.

Christiansen and Velschow proposed a similar region-based approach to memory management in Java [12]. They call their system RegJava, in which a stack of lexically scoped regions are introduced for the allocation of objects. They proposed a region type system and demonstrated its soundness by linking the static semantics with the dynamic semantics. However, their system requires programmers to manually decorate programs with region annotations, in addition to the use of phantom regions and close-world assumption on the class hierarchy.

A number of recent works[20, 21, 25, 30] have advocated for non-lexical regions to support tighter region lifetimes. Most of these approaches require programmers to at least indicate when regions are to be created, allocated and released. The only approach that is fully automatic is [1] which uses as starting point a program with lexical regions, before proceeding to allow late allocation and early deallocation of regions, where possible. This approach is complementary to our approach to region inference, as it could be considered as a post-phase. Nevertheless, lexical regions are more amenable to inference. With an explicit outlive relation on the lexical regions, the possibility of region subtyping can be further exploited, as pioneered in [22].

Beebee and Rinard [3] gave an early implementation of scoped memory for Real-Time Java in the MIT Flex compiler infrastructure. They relied on both static analysis and dynamic debugging to help locate incorrect uses of scoped memory. Later, Boyapati *et. al.* [8] combined region types [28, 29, 15, 22, 12] and ownership types [14, 13, 5, 7] in a unified framework to capture object encapsulation and yet prevent dangling references. Their static type system guaranteed that scope-memory runtime checks will never fail for well-typed programs. It also ensured that real-time threads do not interfere with the garbage collector. Using object encapsulation, an object and all components it *owns* are put into the same region; in order to optimize on region lifetimes. Our region type system is quite similar to theirs, but we separate out object encapsulation and RTSJ issues, and prefer to infer region types automatically.

To the best of our knowledge, ours is the first sound and complete inference system that systematically inserts region annotations for Java-like object-oriented programs. Two other recent works in this direction are applicable in specialised contexts. Deters and Cytron [17] automatically translated Java code into Real-Time Java using dynamic analysis to determine the lifetime of an object. However, their method incurs a runtime overhead and is not totally safe since dynamic analysis may miss some execution paths. Dhurjati *et. al.* [18] proposed a compiler technique, based on *type homogeneity* principle, which tolerated dangling references as long as each freed object is being overwritten by another object of the same type. Their approach requires explicit malloc/free operation to be correctly supplied by programmer. While type safety is preserved, any logical errors caused by premature deallocation of objects is not detected by their system – neither at compile time, nor at runtime.

8. CONCLUDING REMARKS

Our aim is to provide a fully-automatic region inference type system for a core subset of Java. We achieved this by allowing classes and methods to be region-polymorphic, with polymorphic recursion for methods. As shown by the examples, the region constraints inferred allow us to obtain fairly precise region annotations. We have seen how appropriate region instantiations are decided by the region lifetime constraints, which are meant exclusively at preventing dangling references. There remains a number of areas where improvements are possible.

Several directions can be taken to improve memory utilization. The key idea is to put objects into regions with shorter lifetimes, whenever it is safe to do so. As an example, component objects that are *owned* by another object can be placed in the same region as the latter, since no references exist from outside the owner. This idea has been explored in [8]. Coupled with alias (including ownership) annotations that can be automatically inferred, as described in [2], we believe that ownership information can be derived to make this optimization fully automatic.

We presently give a distinct region type to each occurrence of null values, just like normal objects. However, null values are more akin to primitive values as they can be freely moved between regions and stack. To take advantage of this, we could introduce a fictitious region, denoted by $\top$, for each null value. Such a region is non-existent since null values need not reside in a fixed region. The following axioms hold at all times for any r: $\top \succeq r$, $r \succeq \top$, $r = \top$ and $r \neq \top$. These axioms can be used to improve region lifetime constraint when it is combined with an analysis that tracks definite occurrences of null values (e.g. [19]).

Class subtyping is currently supported in our region type rules. However, region subtyping itself is not supported due to the invariant requirement imposed on the object fields that may be updated. Nevertheless, it is possible to explore region subtyping for read-only fields, and this could further improve on region lifetimes.

Our region type rules are flow insensitive (within each method) but context-sensitive (across methods). The latter is due to our use of region polymorphism at method boundary. Flow insensitivity may cause some loss in region lifetime precision when objects from distinct regions are equated through shared local variables. To rectify this, we could make use of the SSA form[16]. Conversion of programs to SSA form can be handled in a preprocessing step, prior to region inference.

At the present moment, our rules may introduce localised regions at each expression block. These are presently mandated at the bodies of procedures, though in practice they can also occur in any sub-expression. Effective placement of local variable declarations, object allocations and expression blocks can affect the extent to which memory are reused; because they affect where the regions can be placed. Another future work is to explore suitable liveness analysis, as well as its associated restructuring transformation, to further improve on memory utilization.

Acknowledgments

The authors would like to thank William Beebee, Chandrasekar Boyapati, Mong Leng Sin, Siau-Cheng Khoo, Corneliu Popeea, Dana Xu and Huu Hai Nguyen for helpful discussions and various pointers. Special thanks to Fritz Henglein for insightful comments on an earlier draft.

9. REFERENCES

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM PLDI*, pages 174–185, 1995.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[3] W. Beebee and M. Rinard. An Implementation of Scoped Memory for Real-Time Java. In *Proceedings of Embedded Software, First International Workshop (EMSOFT '01)*, Tahoe City, California, October 2001.

[4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000. The latest version is available at http://www.rtj.org.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[6] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. In *Proceedings of the 2003 ECOOP Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *ACM POPL*, New Orleans, Louisiana, January 2003.

[8] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *ACM PLDI*, San Diego, California, June 2003.

[9] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. on Programming Languages and Systems*, 17(3):431–447, May 1995.

[10] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region Inference for an Object-Oriented Language. Technical report, School of Computing, National Univ. of Singapore, November 2003. avail. at ⟨http://www.comp.nus.edu.sg/~qinsc/papers/reginf.ps.gz⟩.

[11] M. V. Christiansen, F. Henglein, H. Niss, and P. Velschow. Safe Region-Based Memory Management for Objects. Technical Report D-397, DIKU, University of Copenhagen, October 1998.

[12] M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, DIKU, University of Copenhagen, 1998.

[13] D. G. Clarke and S. Drossopoulou. Ownership, Encapsulation and Disjointness of Type and Effect. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *ACM OOPSLA*, October 1998.

[15] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *ACM POPL*, January 1999.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[17] M. Deters and R. Cytron. Automated Discovery of Scoped Memory Regions for Real-Time Java. In *Proceedings of the International Symposium on Memory Management (ISMM '02)*, June 2002.

[18] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, San Diego, California, June 2003.

[19] M. Fahndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, Anaheim, CA, October 2003.

[20] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *ACM PLDI*, June 1998.

[21] D. Gay and A. Aiken. Language support for regions. In *ACM PLDI*, pages 70–80, 2001.

[22] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *ACM PLDI*, June 2002.

[23] J Gustavsson and J Svenningsson. Constraint abstractions. In *Programs as Data Objects (PADO II)*, pages 63–83, Aarhus, Denmark, May 2001.

[24] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *ACM PLDI*, Berlin, Germany, June 2002.

[25] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, Montréal, Canada, 2001. ACM.

[26] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM OOPSLA*, Denver, Colorado, November 1999.

[27] S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[28] M. Tofte and J. Talpin. Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In *ACM POPL*, January 1994.

[29] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

[30] D. Walker and K. Watkins. On regions and linear types (extended abstract). In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional programming*, pages 181–192. ACM Press, 2001.