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 potential advantages over garbage collection, including real-time performance, better data locality, and more efficient use of limited memory. Researchers have advocated the use of regions for functional, imperative, and object-oriented languages. Lexically scoped regions are now a core feature of the Real-Time Specification for Java (RTSJ)[5].

Recent research in region-based programming for Java has focused on region checking, which requires manual effort to augment the program with region annotations. In this paper, we propose an automatic region inference 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, with region-polymorphic recursion for methods. One challenging aspect is to ensure region safety in the presence of features such as class subtyping, method overriding, and downcast operations. Our region inference rules can handle these object-oriented features safely without creating dangling references.

Categories and Subject Descriptors

D.3.3, D.3.4 [**Programming Languages**]: Language Constructs and Features, Processors; D.2.8 [**Software Engineering**]: Program Verification; F.3.2 [**Theory of Computation**]: Semantics of Programming Languages

General Terms

Languages, Verification, Theory

Keywords

Region Inference, Object-Oriented Languages, Type Systems, Memory Management, Downcasts, Method Overriding

1. INTRODUCTION

Region-based memory management systems allocate each new object into a *region* with a designated lifetime, with the entire set of objects in each region deallocated 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 may also improve when related objects are placed together in the same region. Classifying objects into regions based on their lifetimes may deliver better memory utilization if regions are deleted on a timely basis.

Dangling references are a safety issue for region-based memory management. A reference from (an object in) one region to (an object in) another region is said to be *dangling* if the latter region has a shorter lifetime than the former. A region has a *shorter lifetime* than another region if it is deleted before the latter. Using a dangling reference to access memory is unsafe because the accessed memory may have been reallocated to store a new object. Researchers have identified two approaches to eliminate this problem. The first approach allows the program to create dangling references, but uses a type and effect system to ensure that the program never uses a dangling reference to access memory[16]. The second approach uses a type sytem to prevent the program from creating dangling references at all[9]. The first approach (no-dangling-access) may yield more precise region lifetimes, but the latter approach (no-dangling) is required by the RTSJ and for co-existence with precise garbage collectors.

Several projects have recently investigated the use of region-based memory management for Java-based languages [21, 16, 15, 9]. Most of these projects (e.g. [16, 9]) require programmers to manually annotate their programs with region declarations. The type checker then uses these declarations to check that well-typed programs never access dangling references, ensuring safe memory management and enabling the omission of run-time tests for dangling references. An issue is that region annotations may impose a considerable mental overhead for the programmer and raise compatibility issues with legacy code. In addition, the quality of the annotation may vary, with potentially suboptimal outcomes for less experienced programmers.

In this paper, we provide a systematic formulation of a region inference system for a core subset of Java. We use the no-dangling approach and support programs that use a stack of lexically scoped regions in which the last region created is the first deleted. We adopt this approach for two reasons: to simplify our inference algorithm and to conform to the RTSJ. Our main contributions are:

- **Region Inference:** We present a new region inference algorithm for a core subset of Java. This algorithm (based on type inference) automatically augments unannotated programs with region type declarations; when the program runs it uses region-based memory management and is guaranteed to never create dangling references. Our algorithm fully handles object-oriented features such as class subtyping, method overriding, and downcast operations.

PLDI'04

- **Region Lifetime Constraints:** Our region type rules prevent dangling references by requiring the target object of each reference to live at least as long as the source object. We formalise this requirement explicitly through region lifetime constraints, with support for region subtyping.

- **Region Polymorphism:** We support classes and methods with region polymorphism. In addition, region-polymorphic recursion is supported for methods (but not for classes). These features provide an inference algorithm that is precise and yet efficient.

- **Class Inheritance:** Our inference scheme supports class subtyping and method overriding. Previous systems [16] require "phantom regions" to support inheritance, which causes a loss in lifetime precision. We propose an improved solution (without phantom regions) where modular compilation can be guided by a global dependency graph.

- **Correctness:** We state a correctness theorem that our inference scheme always leads to well-typed programs that are region-safe. Such well-typed programs are guaranteed not to create dangling references in either the store or the stack.

- **Downcast Safety:** We provide a compile-time analysis which ensures that downcast operations are region-safe. Previous proposals (e.g. [7]) require runtime checks for downcast operations.

- **Implementation:** We have built a prototype implementation of our region inference system. Preliminary experiments suggest that our inference is competitive with hand annotation.

The remainder of the paper is organised as follows. Sec 2 introduces the *Core-Java* language and its region-annotated target form. Sec 3 presents a set of guidelines for good region annotations; we use these guidelines to structure our region inference rules. Sec 4 is devoted to region inference. It formalises the region inference rules and describes how method override conflicts can be resolved. It also presents the key correctness theorem. Sec 5 deals with the inference of additional regions to ensure safe downcast operations. Sec 6 reports on preliminary experiments with our implementation of the region inference algorithm. Sec 7 discusses related work; Sec 8 contains the conclusion.

2. CORE-JAVA AND REGION TYPES

Fig 1(a) presents the syntax of a Java-like language named *Core-Java*. Core-Java is the source language for our region inference system. Core-Java is designed in the same minimalist spirit as Featherweight Java[30]. It supports assignments but remains an expression-oriented language. Loops are omitted in our syntax as they are dealt with through conversion to equivalent tail-recursive methods. To mimic the effects of loops, such converted methods differ from user-defined methods in that they pass their parameters by reference instead of by value. This conversion is for inference purposes only; the generated program executes the loop directly.

Fig 1(b) presents region-annotated Core-Java, the target language for our region inference system. This language extends Core-Java with region types and region constraints for each class and method. In addition, **letreg** declarations introduce local regions with lexical scopes.

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

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

(a) The Source Language

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

(b) The Target Language

Figure 1: The Syntax of Core-Java

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. Our algorithm will infer region constraints only of the form $r_1 \succeq r_2$ or $r_1 = r_2$ but never of the form $r_1 \neq r_2$. However, $r_1 \neq r_2$ is present in our syntax because a region checking system which we have also built supports it. Note that *this* is a reserved data variable referring to the current object, while *heap* is a reserved region to denote a global heap with unlimited lifetime, that is $\forall r \cdot \mathit{heap} \succeq r$.

A constraint abstraction [27] of the form $q\langle r_1, .., r_n\rangle = rc$ denotes a parameterized constraint. For uniformity, the region constraint of each class and method will be captured with a constraint abstraction. The region constraint for each class is also known as the *class invariant* and is denoted using $\mathit{inv.cn}\langle r_1, .., r_n\rangle$. The region constraint for each method is also known as the *method precondition* and is denoted using $\mathit{pre.m}\langle r_1, .., r_n\rangle$, where m is either *cn.mn* or *mn*, depending on whether it is an instance method or a static method. Note that *mn* denotes a method name. The set of constraint abstractions generated for a given program is denoted by Q. It is possible to inline the constraint abstractions, after fixed-

```
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 r4⪰r3 {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,r3⟩ extends Object⟨r1⟩
   where r3⪰r1∧r2⪰r3∧r2⪰r1 {
   Object⟨r2⟩ value
   List⟨r3,r2,r3⟩ next
   Object⟨r4⟩ getValue⟨r4⟩() where r2⪰r4  {value}
   List⟨r4,r5,r6⟩ getNext⟨r4,r5,r6⟩()
    where r5=r2∧r6=r3 {next}
   void setNext⟨r4,r5,r6⟩(List⟨r4,r5,r6⟩ o)
    where r5=r2∧r6=r3∧r4=r6  {next = o}
}
                 (b)
```

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

point analysis has been applied to obtain closed-form formulae for recursive constraints.

Each class definition in the target language is parameterized with one or more regions to form a region type [32, 33, 16, 9]. For instance, $cn\langle r_1, ..., r_n\rangle$ is a class annotated with region parameters $r_1 ... r_n$. Parameterization allows us to obtain a region-polymorphic type for each 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. The components (or fields) of the objects, if any, are allocated in the other regions $r_2 ... r_n$ which should *outlive* the region of the object. This is expressed by the constraint $\wedge_{i=2}^{n}(r_i \succeq r_1)$, which captures the property that the regions of the components (in $r_2 ... r_n$) should have lifetimes no shorter than the lifetime of the region (namely r_1) of the object that refers to them. This condition, called *no-dangling* requirement, prevents dangling references completely, as it guarantees that each object will never reference another object in a younger region. We do not require region parameters for primitive types, since primitive values are copied and therefore always stored directly in either the stack or inside its owner object.

Each class declaration has a set of instance methods whose main purpose is to manipulate objects of the declared class. The instance methods of a subclass can override the instance methods of the superclass. For completeness, we also provide a set of static methods with similar syntax as instance methods, but without overriding and without access to the *this* object. 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; these parameters 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 performed in the method body. Fig 2 presents two example classes, the `Pair` and `List` classes, in the target language.

3. REGION ANNOTATION GUIDELINES

To support region-based programming, our region inference algorithm adds region parameters and constraints to each class and its methods. There are a number of ways to perform such region annotations. The following principles guide our approach:

- Keep the regions of components in each class (and the regions of the parameters and results of each method) *distinct*, where possible.
- Keep region constraints on classes and methods *separate*. Region constraints on classes capture the expected class invariant (including the no-dangling requirement) on the regions of each object of the class. Region constraints on methods denote the precondition for invoking the method.
- Use *region subtyping*, where applicable, to improve the precision of the lifetimes of the regions.

The first principle allows more region polymorphism, where applicable. The second principle places the region constraints that must hold for every object of a given class in the class, while placing the region constraints for each method invocation in the method. Placing region constraints with 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 principle is helpful also to ensure the safety of method overriding. The third principle allows an object from a region with longer lifetime to be assigned to a location where a region with a shorter lifetime is expected. This concept was pioneered in a safe dialect of C, called Cyclone[26], and is implemented as region subtyping. We shall see how this idea improves the precision of region lifetimes and how it can be further enhanced.

3.1 Regions for Field Declarations

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

Next consider the `List` class with `next` as its recursive field. There are many different ways of annotating such recursive fields; the best choice depends on how the objects are manipulated. To keep matters simple, we use a special form of region-monomorphic recursion for class declarations, similar to Tofte/Birkedal's handling of data structures[32, 4], but with support for region subtyping. We introduce a distinct region especially for all the recursive fields. This approach ensures that each recursive field will have the same annotation as its class, except for its first region. Given a recursive class declaration with region type $cn\langle r_1, r^*, r_n\rangle$, we shall annotate each of the fields as $cn\langle r_n, r^*, r_n\rangle$. In the case of the `List` class, the region `r3` is reserved specially for the recursive `next` field, as illustrated in Fig 2(b). We handle mutually recursive class declarations similarly. For simplicity, we ignore mutual recursive class declarations in this paper, even though our implementation supports them.

Based on the above guidelines, the constraint abstractions for the `Pair` and `List` classes are:

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

3.2 Region Subtyping Principle

We investigate three versions of the region subtyping rules; the versions differ in the precision of the inferred region lifetimes:

- no (region) subtyping
- object (region) subtyping
- field (region) subtyping

The first kind of subtyping was used in [9] and [16]. The second kind was introduced in [26]. In this paper, we advocate the third kind of region subtyping (which subsumes the second kind) with enhanced support for the regions of immutable fields.

Object region subtyping relies on the fact that once an object is allocated in a particular region, it stays within the same region and never migrates to another region. This immutability property allows us to apply covariant subtyping to the region of the current object. However, the object fields are mutable (in general) and must therefore use equivariant subtyping to ensure the soundness of subsumption. By reserving the first region exclusively for the region of each object, we can therefore use the following two subtype rules.

$$\frac{\varphi = (x_1 \succeq \hat{x}_1) \wedge \bigwedge_{i=2}^{n}(x_i = \hat{x}_i) \qquad \rho = [\hat{x}_i \mapsto x_i]_{i=1}^{n}}{\vdash \tau\langle x_{1..n}\rangle <: \tau\langle \hat{x}_{1..n}\rangle,\ \varphi \mid \rho}$$

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle\ \cdots\ \in P' \\ \vdash cn'\langle x_{1..m}\rangle <: cn''\langle x'_{1..p}\rangle,\ \varphi \mid \rho\end{array}}{\vdash cn\langle x_{1..n}\rangle <: cn''\langle x'_{1..p}\rangle,\ \varphi \mid \rho}$$

Note that $x_1 \succeq \hat{x}_1$ allows an object in a region with a longer lifetime to be assigned to a location that expects objects in a region with a shorter lifetime. For the other regions (that are used by the fields), a stronger equivariant constraint $\bigwedge_{i=2}^{n}(x_i = \hat{x}_i)$ would be used instead. The second rule is for the class subtype hierarchy. Both rules return a region constraint φ and a region substitution ρ. The latter may be used if we are interested in only equivariant subtyping. One example where object region subtyping is useful is the following:

```
void foo (Object a, Object b)
{ Object tmp;
  if ...  then tmp=a else tmp=b;
  ...
}
```

Without object subtyping, the dual assignments of both `a` and `b` to `tmp` cause their regions to be coalesced together and generate the constraint $r_a = r_b$ (where r_a and r_b are the regions for `a` and `b`). With object subtyping, regions of `a` and `b` may be different, as long as they both outlive the region of `tmp`.

This concept of region subtyping can be further extended to selected fields if they are immutable after object initialization. The fields of recursive structures are particularly important as they may involve many objects that are typically grouped into the same region. We use an *isRecReadOnly* function to check if a class has immutable recursive fields or not. With this information, we can support a more precise region subtyping rule, namely:

$$\frac{isRecReadOnly(\tau) \qquad \varphi = (x_1 \succeq \hat{x}_1) \wedge \bigwedge_{i=2}^{n-1}(x_i = \hat{x}_i) \wedge (x_n \succeq \hat{x}_n)}{\vdash \tau\langle x_1, .., x_n\rangle <: \tau\langle \hat{x}_1, .., \hat{x}_n\rangle,\ \varphi \mid [\hat{x}_i \mapsto x_i]_{i=1}^{n}}$$

One advantage of this field region subtyping rule is that it allows each recursive object to be placed in a region that is different (and may have a longer lifetime) from that of the prior object in the recursive chain. Such a feature is important for recursive methods that build temporary data structures during recursive invocations. An example is the following function, called *Reynolds3*, that was highlighted in [22, 4].

```
Bool search (RList p, Tree t)
{ if isNull(t) then false
  else { Object x; x=t.value;
         if member(x,p) then true
         else { RList p2; p2=new RList(x,p);
                if search(p2,t.left) then true
                else search(p2,t.right) } } }
```

We use `RList` to denote a list structure with an immutable recursive field. Applying region inference with field subtyping, we are able to obtain a target program where the region for variable `p2` is localised. Unlike previous work[32, 4], the performance of such a region-inferred program is comparable to that obtained by escape analysis[22]. We thus advocate for region subtyping to be used, where possible, to obtain better region annotations.

3.3 Regions for Method Declarations

For each method declaration, we must 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 each method will thus be mapped to these region parameters or to the *heap*.

We must also provide region lifetime constraints over such region parameters (including the regions of `this` object). These constraints naturally depend on how the method manipulates the objects. Consider the `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 and region subtyping, we have `r4`$\succeq$`r3` for `setSnd`, while `r2`$\succeq$`r6`$\wedge$`r3`$\succeq$`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 associate the constraint with the method. In this way, only those objects that might call the method will be required to satisfy this constraint.

The region constraints for methods also contain the class invariants of its parameters and result. For example, the region constraint for `cloneRev` implicitly includes the class invariant `r6`$\succeq$`r4`$\wedge$`r5`$\succeq$`r4` of the resulting type `Pair`$\langle$`r4,r5,r6`$\rangle$. For simplicity, we omit the presentation of such constraints in this paper; these constraints can be easily 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 below:

```
pre.Pair.getFst<r1,r2,r3,r4>          = r2⪰r4
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
```

3.4 Regions for SubClass Declarations

Subclasses typically augment the superclass with additional fields and methods. Correspondingly, the regions of each subclass are *extended* from its superclass, while its invariant represents a *strengthening* from the invariant of its superclass. These requirements are needed to support class subsumption. Consider:

```
class A⟨r1..rm⟩ extends Object⟨r1⟩ where φA...
class B⟨r1..rn⟩ extends A⟨r1..rm⟩ where φB...
```

We expect the regions of the subclass `B`, namely $\langle$`r1..rn`$\rangle$, to be an extension of `A`, namely $\langle$`r1..rm`$\rangle$, with `n`$\geq$`m`. Likewise, the 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 that expects an `A` object, 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 subclass is required to be a subtype of the overridden method.

Function subtyping in object-oriented programs is sound if the normal parameters are contravariant and the selection parameters are covariant [12].

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

$$\texttt{Y A.mn } \langle r'_1,\ldots,r'_p\rangle \texttt{(X a) where } \varphi_{A.mn}\ \{\ldots\}$$
$$\texttt{Y B.mn } \langle r'_1,\ldots,r'_p\rangle \texttt{(X a) where } \varphi_{B.mn}\ \{\ldots\}$$

The constraints $\varphi_{A.mn}$ and $\varphi_{B.mn}$ are preconditions for the region parameters of `A.mn` and `B.mn`, respectively. These parameters must be contravariant for function subtyping, 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 the method satisfy this soundness check. Its inclusion is critical to our approach for handling method overriding without phantom regions.

Method overriding is particularly challenging for region inference. We shall introduce some techniques to ensure the compliance of the overriding checks in Sec 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. We then present some examples that illustrate the inference process. We also formalise the analysis of the global dependency graph to guide the inference process, and describe how class subtyping and method overriding can be supported. We then state a theorem on the correctness of our region inference.

4.1 Region Inference Rules

The goal of region inference is 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. The normal type system for Core-Java can be derived from the region type checking system (given in a companion report [14]) by erasing all region-related notations. That is, if $\vdash P'$, then $\vdash_N$ **erase**(P'). 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.

Fig 3 presents the complete set of region inference rules. Our rules assume that source program P is globally available. Some of our rules also assume that the target program P' is also available. This is possible as we perform region inference in stages, in accordance with the calling hierarchy with the help of the global dependency graph.

4.1.1 Class Declarations

The inference rule [CD] for class declarations has the form:

$$\vdash \mathit{def} \Rrightarrow \mathit{def}', \mathcal{Q}$$

Note that *def* is the source code, while *def'* is the region-annotated target with $\mathcal{Q}$ to capture the constraint abstractions obtained during inference. For each class declaration, we must designate regions for the objects of that class and their components. We also add region lifetime constraints to each class based primarily on the constraints of its fields.

Region inference for class declaration is conducted in an inductive manner, by inferring regions for the fields first, and then separately applying region inference to the instance methods. The rule uses $\mathit{split}(\mathit{fds}, \mathit{cn}_2)$ to separate out the non-recursive fields from the recursive fields. Recursive (and mutual recursive) fields should have the same region annotation as the class, except for the first region. Note that the region invariant of each class *cn* is captured by a constraint abstraction, named *inv.cn*.

4.1.2 Method Declarations

The inference rule [MD] for method declarations has the form:

$$\Gamma \vdash \mathit{meth} \Rrightarrow \mathit{meth}',\ \mathcal{Q}$$

This rule can be used for both instance and static methods, with $\Gamma = \{\mathit{this} : \mathit{cn}\langle r^+\rangle\}$ for the former and $\Gamma = \emptyset$ for the latter. It infers necessary region variables for each method, and calculates lifetime constraints that should be imposed on them. Note that this rule generates a fresh set of regions for the parameters and result of each method. 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 subtype of the expected output type, $\tau_0\langle r^*_0\rangle$.

4.1.3 Expressions

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 unannotated expression and the region annotated counterpart. t is a region type, while φ is the derived region constraint. We next discuss the rules for instance method invocations and local region declarations.

Rule [EMI] is the region inference rule for instance method invocations. The rule gathers the respective method's region constraint suitably modified by a region substitution obtained from equivariant instantiation. We use a substitution to ensure that the region of each actual argument is mapped to the region of its corresponding parameter. We implicitly apply the region subtyping principle as we assign each parameter to its corresponding local variable in our implementation of the call-by-value parameters. The gathered region constraint is then imposed as a precondition on each method invocation.

The rule [EB] is a key inference rule that governs how regions may be localised at each expression block. It attempts to identify regions that are effectively dead thereafter. 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 regions also escape. 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

We next illustrate 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 annotate each local variable and constructor with new distinct regions and proceed to gather the constraints from each sub-expression. A set of equality and outlive constraints will be collected and simplified; these constraints can be applied to reduce the number of distinct regions.

[CPT]
$$\frac{}{\vdash \mathit{prim} \Rrightarrow \mathit{prim}\langle\rangle,\, \mathit{true}}$$

[OBJ]
$$\frac{r{=}\mathit{fresh}()}{\vdash \text{Object} \Rrightarrow \text{Object}\langle r\rangle,\ \mathit{true}}$$

[CCT]
$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle \ldots\ \textbf{where}\ \varphi\,\{\ldots\} \in P' \quad a_{1..n}{=}\mathit{fresh}()}{\vdash cn \Rrightarrow cn\langle a_{1..n}\rangle,\ ([r_i \mapsto a_i]_{i=1}^{n}\,\varphi)}$$

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

[EV]
$$\frac{v : \tau\langle r^*\rangle \ \in\ \Gamma}{\Gamma \vdash v \Rrightarrow v : \tau\langle r^*\rangle,\ \mathit{true}}$$

[EC1]
$$\frac{\vdash cn \Rrightarrow cn\langle r^+\rangle, \varphi}{\Gamma \vdash (cn)\textbf{null} \Rrightarrow (cn\langle r^+\rangle)\textbf{null} : cn\langle r^+\rangle, \mathit{true}}$$

[EC2]
$$\frac{}{\Gamma \vdash k \Rrightarrow k : \mathit{prim}_k\langle\rangle, \mathit{true}}$$

[ES]
$$\frac{\Gamma \vdash e_i \Rrightarrow e_i' : t_i, \varphi_i \quad i = 1, 2}{\Gamma \vdash e_1\,\textbf{;}\,e_2 \Rrightarrow e_1'\,\textbf{;}\,e_2' : t_2, \varphi_1 {\wedge} \varphi_2}$$

[EA]
$$\frac{\begin{array}{c} \mathit{lhs} \ =\ v \mid v.f \\ \Gamma \vdash e \Rrightarrow e' : t', \varphi' \\ \Gamma \vdash \mathit{lhs} \Rrightarrow \mathit{lhs} : t, \mathit{true} \\ \vdash t' <: t, \phi \quad \varphi = \varphi' \wedge \phi \end{array}}{\Gamma \vdash \mathit{lhs} = e \Rrightarrow \mathit{lhs} = e' : \textbf{void}, \varphi}$$

[EN]
$$\frac{\begin{array}{c} \vdash cn \Rrightarrow cn\langle x^+\rangle, \varphi_0 \\ \mathit{fieldlist}(cn\langle x^+\rangle) = [(t_i\, f_i)]_{i=1}^{p} \\ (v_i : t_i') \in \Gamma \quad \vdash t_i' <: t_i,\ \phi_i \quad i{=}1..p \\ \varphi = \varphi_0 \wedge \bigwedge_{i=1}^{p} \phi_i \end{array}}{\Gamma \vdash \textbf{new}\ cn(v_{1..p}) \Rrightarrow \textbf{new}\ cn\langle x^+\rangle(v_{1..p}) : cn\langle x^+\rangle, \varphi}$$

[EIF]
$$\frac{\begin{array}{c} (v_0 : \textbf{bool}\langle\rangle) \in \Gamma \\ \Gamma \vdash e_1 \Rrightarrow e_1' : t_1, \varphi_1 \\ \Gamma \vdash e_2 \Rrightarrow e_2' : t_2, \varphi_2 \\ (t, \varphi_3) = \mathit{msst}(t_1, t_2) \quad \varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \end{array}}{\Gamma \vdash \textbf{if}\ v_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Rrightarrow \textbf{if}\ v_0\ \textbf{then}\ e_1'\ \textbf{else}\ e_2' : t, \varphi}$$

[EMS]
$$\frac{\begin{array}{c} (\tau_0\langle x_0^*\rangle\ mn\langle y^*\rangle((\tau_i\langle x_i^*\rangle\ v_i)_{i:1..p})\ \textbf{where}\ \varphi\ eb) \in P' \\ (v_i' : \tau_i'\langle x_i'^*\rangle) \in \Gamma \quad i = 1..p \\ \vdash \tau_i'\langle x_i'^*\rangle <: \tau_i\langle x_i^*\rangle, \rho_i \quad i{=}1..p \\ \rho = \rho_1 \ldots \rho_p\, [x_0 \mapsto x_0']^* \qquad x_0'^*{=}\mathit{fresh}() \end{array}}{\Gamma \vdash mn(v_1', \ldots, v_p') \Rrightarrow mn\langle \rho y^*\rangle(v_1', \ldots, v_p') : \tau_0\langle x_0'^*\rangle, \rho\varphi}$$

[EMI]
$$\frac{\begin{array}{c} (\tau_0\langle x_0^*\rangle\ mn\langle y^*\rangle((\tau_j\langle x_j^*\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi\ eb) \ \in\ \mathit{methlist}(cn\langle x_1^+\rangle) \\ (v_j' : \tau_j'\langle x_j'^*\rangle) \in \Gamma \quad j = 2..p \qquad v_1' : cn\langle x_1'^+\rangle \in \Gamma \\ \vdash \tau_j\langle x_j'^*\rangle <: \tau_j\langle x_j^*\rangle, \rho_j \qquad j = 2..p \\ \rho = \rho_2 \ldots \rho_p\, [x_0 \mapsto x_0']^*\, [x_1 \mapsto x_1']^+ \quad x_0'^*{=}\mathit{fresh}() \end{array}}{\Gamma \vdash v_1'.mn(v_2', \ldots, v_p') \Rrightarrow v_1'.mn\langle \rho y^*\rangle(v_2', \ldots, v_p') : \tau_0\langle x_0'^*\rangle, \rho\varphi}$$

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

[CD]
$$\frac{\begin{array}{c} \mathit{def}_2 = \textbf{class}\ cn_2\ \textbf{extends}\ cn_1\ \{\mathit{fd}^*\ (\mathit{meth}_i)_{1..q}\} \\ \mathit{split}(\mathit{fd}^*, cn_2) = ([(\tau_i\, f_i)]_{i=1}^{p},\ [(cn_2\, f_i)]_{i=p+1}^{n}) \qquad \tau_i{\neq}cn_2 \quad i{=}1..p \\ \vdash \tau_i{\Rrightarrow}\tau_i\langle r_i^*\rangle, \varphi_i \quad i{=}1..p \quad \varphi_0 = (r{\succeq}r_1 \lhd p{<}n \rhd \mathit{true}) \\ ca_2 = (cn_2\langle r_{1..l}, r_{1..p}^*, r\rangle \lhd p{<}n \rhd cn_2\langle r_{1..l}, r_{1..p}^*\rangle) \quad \Gamma = \{\mathit{this} : ca_2\} \\ \vdash cn_1 \Rrightarrow cn_1\langle r_{1..l}\rangle, \varphi_1 \quad \Gamma \vdash \mathit{meth}_i \Rrightarrow \mathit{meth}_i'; Q_i,\ i{=}1..q \\ Q' = \{\mathit{inv}.ca_2 = (\varphi_1\ \wedge\ \bigwedge_{i=1}^{p} \varphi_i\ \wedge\ \bigwedge_{i=1}^{p}(r_i^* \succeq r_1)\ \wedge\ \varphi_0)\} \\ \mathit{def}_2' = \textbf{class}\ ca_2\ \textbf{extends}\ cn_1\langle r_{1..l}\rangle\ \textbf{where}\ \varphi \\ \{(\tau_i\langle r_i^*\rangle\, f_i)_{1..p},\ (cn_2\langle r, r_{2..l}, r_{1..p}^*, r\rangle\, f_i)_{p+1..n},\ (\mathit{meth}_i')_{1..q}\} \end{array}}{\vdash \mathit{def}_2 \Rrightarrow \mathit{def}_2'; Q' \ \cup\ \bigcup_{i=1}^{q} Q_i}$$

[MD]
$$\frac{\begin{array}{c} \mathit{meth} = \tau_0\ mn((\tau_j\ v_j)_{j:2..p})\ eb \quad \vdash \tau_i \Rrightarrow \tau_i\langle r_i^*\rangle,\ \varphi_i \quad i{=}0, 2..p \\ \Gamma,\ (v_j : \tau_j\langle r_j^*\rangle)_{j:2..p} \vdash eb \Rrightarrow eb' : \tau_0'\langle x_0^*\rangle,\ \varphi \\ (m, r_1^*) = ((\mathit{pre.cn.mn}, r^+) \lhd (\mathit{this} : cn\langle r^+\rangle \in \Gamma) \rhd (\mathit{pre.mn}, [\,])) \\ \vdash \tau_0'\langle x_0^*\rangle <: \tau_0\langle r_0^*\rangle, \phi \quad y^* = r_1^*, .., r_p^*, r_0^* \quad \varphi' = m\langle y^*\rangle \\ \mathit{meth}' = \tau_0\langle r_0^*\rangle\ mn\langle y^*\rangle((\tau_j\langle r_j^*\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi'\ (eb') \end{array}}{\Gamma \vdash \mathit{meth} \Rrightarrow \mathit{meth}'; \{\varphi' = (\varphi \wedge \varphi_0 \wedge \bigwedge_{i=2}^{p} \varphi_i \wedge \phi)\}}$$

[PROG]
$$\frac{\begin{array}{c} P = \mathit{def}_{1..n}\ \mathit{meth}_{1..m} \\ \vdash_{\mathit{def}} \mathit{def}_i \Rrightarrow \mathit{def}_i'; Q_i \ \ i \in 1..n \\ \{\} \vdash \mathit{meth}_i \Rrightarrow \mathit{meth}_i'; Q_i' \ \ i \in 1..m \\ Q'' = \bigcup_{i=1}^{n} Q_i \cup \bigcup_{i=1}^{m} Q_i' \\ P' = \mathit{def}_{1..n}'\ \mathit{meth}_{1..m}'\ Q'' \end{array}}{\vdash P \Rrightarrow P'}$$

$$\frac{\vdash P \Rrightarrow P' \quad \mathit{def} \in P'}{\mathit{def} \in P} \qquad \frac{\vdash P \Rrightarrow P' \quad \mathit{meth} \in P'}{\mathit{meth} \in P} \qquad \frac{(t\ f) \in \mathit{fieldlist}(cn\langle x^*\rangle)}{\vdash (t\ f) \in cn\langle x^*\rangle} \qquad \frac{\vdash t_1 <: t_2,\ \varphi \mid \rho}{\vdash t_1 <: t_2,\ \varphi} \qquad \frac{\vdash t_1 <: t_2,\ \varphi \mid \rho}{\vdash t_1 <: t_2,\ \rho} \qquad \frac{\vdash t_1 <: t_2,\ \varphi \mid \rho}{t_1 <: t_2}$$

$$\frac{\begin{array}{c} \vdash t_i{<:}t, \phi_i, \quad i{=}1, 2 \\ (\forall \hat{t} \cdot (t_1{<:}\hat{t}) \wedge (t_2{<:}\hat{t}) \Rightarrow (t{<:}\hat{t}) \end{array}}{\mathit{msst}(t_1, t_2) =_{\mathit{df}} (t, \phi_1{\wedge}\phi_2)} \qquad \frac{\begin{array}{c} \textbf{class}\ cn_1\langle r_{1..n}\rangle\ \textbf{extends}\ cn_2\langle r_{1..m}\rangle\ ..\ \{(t_i\ f_i)_{i:1..p}\ \mathit{meth}_{j:1..q}\} \in P \\ x_{1..n}{=}\mathit{fresh}() \quad (\ell_1, \ell_2) = \mathit{mbrlist}(cn_2\langle x_{1..m}\rangle) \quad \rho = [r_i{\mapsto}x_i]_{i=1}^{n} \end{array}}{\mathit{mbrlist}(cn_1\langle x_{1..n}\rangle) {=}_{\mathit{df}} (\ell_1 {+\!\!+} [(\rho\ t_i)\ f_i]_{i=1}^{p},\ \ell_2 {+\!\!+} [(\rho\ \mathit{meth}_j)]_{j=1}^{q})} \qquad \frac{}{\mathit{mbrlist}(\textbf{Object}\langle r\rangle) {=}_{\mathit{df}} ([\,], [\,])}$$

$$\frac{\mathit{mbrlist}(cn\langle x_{1..n}\rangle) = (\ell_1, \ell_2)}{\mathit{fieldlist}(cn\langle x_{1..n}\rangle) =_{\mathit{df}} \ell_1} \qquad \frac{\mathit{mbrlist}(cn\langle x_{1..n}\rangle) = (\ell_1, \ell_2)}{\mathit{methlist}(cn\langle x_{1..n}\rangle) =_{\mathit{df}} \ell_2} \qquad \frac{}{\mathit{meth} {\in} (\ell {+\!\!+} [\mathit{meth}])} \qquad \frac{\mathit{name}(\mathit{meth}) {\neq} \mathit{name}(\mathit{meth}') \quad \mathit{meth} {\in} \ell}{\mathit{meth} {\in} (\ell {+\!\!+} [\mathit{meth}'])}$$

$$\frac{}{\mathit{split}([\,], cn) =_{\mathit{df}} ([\,], [\,])} \qquad \frac{\mathit{split}(\mathit{fdl}, cn) = (l_1, l_2) \quad \tau{\neq}cn}{\mathit{split}([(\tau\ f)] {+\!\!+} \mathit{fdl}, cn) =_{\mathit{df}} ([(\tau\ f)] {+\!\!+} l_1, l_2)} \qquad \frac{\mathit{split}(\mathit{fdl}, cn) = (l_1, l_2) \quad \tau{=}cn}{\mathit{split}([(\tau\ f)] {+\!\!+} \mathit{fdl}, cn) =_{\mathit{df}} (l_1, [(\tau\ f)] {+\!\!+} l_2)} \qquad \frac{x {\in} s}{\mathit{sel}(s) =_{\mathit{df}} x}$$

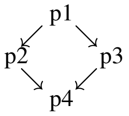
$$\mathit{reg}(\{\}) {=}_{\mathit{df}} \{\} \quad \mathit{reg}(\{(v : \tau\langle r^*\rangle)\} {\cup} \Gamma) {=}_{\mathit{df}} \{r^*\} {\cup} \mathit{reg}(\Gamma) \quad \mathit{reg}(\mathit{true}) {=}_{\mathit{df}} \{\} \quad \mathit{reg}(r_1{=}r_2) {=}_{\mathit{df}} \{r_1, r_2\} \quad \mathit{reg}(r_1{\succeq}r_2) {=}_{\mathit{df}} \{r_1, r_2\} \quad \mathit{reg}(c{\wedge}\varphi) {=}_{\mathit{df}} \mathit{reg}(c) {\cup} \mathit{reg}(\varphi)$$

$$\mathit{name}(t_0\ mn(..)..) =_{\mathit{df}} mn \qquad st(\varphi, s_1, s_2) {=}_{\mathit{df}} \{x {\mapsto} \mathit{sel}(s_x) \mid x {\in} s_1 \wedge s_x {=} \{y \mid y {\in} s_2 {\wedge} \varphi {\Rightarrow} (x{=}y)\} \wedge s_x {\neq} \emptyset\}$$

$$\mathit{ors}(\varphi, s_1, s_2) =_{\mathit{df}} \{r \mid r \in s_1 \wedge \exists r' \in s_2 \cdot (\varphi \Rightarrow r \succeq r')\} \qquad \overline{\mathit{ors}}(\varphi, s_1, s_2) =_{\mathit{df}} s_1 - \mathit{ors}(\varphi, s_1, s_2)$$

fresh() returns one or more new/unused region names $\qquad \xi_1 \lhd b \rhd \xi_2 =_{\mathit{df}} \begin{cases} \xi_1, & \text{if } b \\ \xi_2, & \text{otherwise} \end{cases}$

Figure 3: Region Inference Rules



(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) Source program

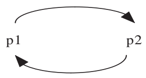
```
{Pair⟨r4,r4a,r4b⟩ p4 =
    new Pair⟨r4,r4a,r4b⟩(null,null);
    //r4a⪰r4 ∧ r4b⪰r4
 Pair⟨r3,r3a,r3b⟩ p3 =
    new Pair⟨r3,r3a,r3b⟩(p4,null);
    // r3a⪰r3 ∧ r3b⪰r3 ∧ r4⪰r3a
 Pair⟨r2,r2a,r2b⟩ p2 =
    new Pair⟨r2,r2a,r2b⟩(null,p4);
    // r2a⪰r2 ∧ r2b⪰r2 ∧ r4⪰r2b
 Pair⟨r1,r1a,r1b⟩ p1 =
    new Pair⟨r1,r1a,r1b⟩(p2,null);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

(c) Initial region-annotated program.

```
letreg r in
 { Pair⟨r4,r4a,r4b⟩ p4 =
     new Pair⟨r4,r4a,r4b⟩(null,null);
     //r4a⪰r4 ∧ r4b⪰r4
   Pair⟨r,r,r⟩ p3      =
     new Pair⟨r,r,r⟩(p4,null);
   Pair⟨r2,r2a,r2b⟩ p2 =
     new Pair⟨r2,r2a,r2b⟩(null,p4);
     // r2a⪰r2 ∧ r2b⪰r2 ∧ r4⪰r2b
   Pair⟨r,r,r⟩ p1      =
     new Pair⟨r,r,r⟩(p2,null);
   p1.setSnd⟨r⟩(p3);
   p2
 }
```

(d) Final region-annotated program.

Figure 4: Example with Localised Region



(a) Cyclic data structure

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

(b) Source program

```
{Pair⟨r1,r1a,r1b⟩ p1 =
    new Pair⟨r1,r1a,r1b⟩(null,null);
    //r1a⪰r1 ∧ r1b⪰r1
 Pair⟨r2,r2a,r2b⟩ p2 =
    new Pair⟨r2,r2a,r2b⟩(p1,null);
    // r2a⪰r2 ∧ r2b⪰r2 ∧ r1⪰r2a
 p1.setSnd⟨r2⟩(p2); // r2⪰r1b
 p2
}
```

(c) Initial region-annotated program

```
{Pair⟨r1,r1a,r1⟩ p1 =
    new Pair⟨r1,r1a,r1⟩(null,null);
    //r1a⪰r1
 Pair⟨r1,r1,r2b⟩ p2 =
    new Pair⟨r1,r1,r2b⟩(p1,null);
    // r2b⪰r1
 p1.setSnd⟨r1⟩(p2);
 p2
}
```

(d) Final region-annotated program

Figure 5: Example with Circular Structure

The result type of this block is `Pair⟨r2,r2a,r2b⟩`, with the constraints `r4a⪰r4∧r4b⪰r4`, `r3a⪰r3∧r3b⪰r3∧r4⪰r3a`, `r2a⪰r2∧r2b⪰r2∧r4⪰r2b` , `r1a⪰r1∧r1b⪰r1∧r2⪰r1a`, and `r3⪰r1b`. Based on these type and region lifetime constraints, our rule can deduce that all the regions escape this block, except for regions `r1,r1a,r1b,r3,r3a,r3b`. These non-escaping regions will be localised to a single region (say `r`), as shown in Fig 4(d).

4.2.2 Circular Structures

Because of the outlives constraint from the no-dangling requirement, every cyclic structure must be placed in the same region. Consider the code fragment in Fig 5(b), which constructs a cyclic structure involving two `Pair` nodes, `p1` and `p2`, as shown in Fig 5(a). The initial region-annotated program is shown in Fig 5(c). After constraint simplification to coalesce equal regions together, we obtain the target program in Fig 5(d). Notice that `p1` and `p2` are initially placed in regions `r1` and `r2`, respectively. However, the region constraint gathered, namely `r2⪰r1b∧r1b⪰r1∧r1⪰r2a∧r2a⪰r2`, implies that `r1=r2=r1b=r2a`. Applying this extra constraint causes the two objects to be located in the same region.

The resulting type of this block is `Pair⟨r1,r1,r2b⟩` with the region constraint `r2b⪰r1∧r1a⪰r1`. As all declared regions escape from this expression block, the [EB] rule does not introduce any localised regions.

4.2.3 Recursive Methods

We discuss the inference of region annotations and constraints for recursive methods. Fig 6(a) contains a static method which merges two lists of objects. Because the method swaps its parameters at the 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 6(c) (after simplification). As the constraint abstraction is recursive, we apply a fixed-point analysis to obtain its closed-form formula. Starting with the initial version of $\texttt{pre.join}_0$, we progressively refine the definition of `pre.join` until a fixed-point is reached, as highlighted in Fig 6(d).

Fixed-point analysis always terminates for our constraint abstraction — the finite set of possible constraints is made up from a bounded set of regions. This example relies on region-polymorphic recursion, without which some loss in lifetime precision occurs. Note that each recursive call has a different region type (c.f region parameters) from its caller.

4.3 Global Dependency Graph

Because of the inter-dependency between classes and methods, our region inference system must process the classes and methods in a specific order. This order supports hierarchically structured programs and inheritance and is important for modular compilation. For this purpose, we propose a set of rules that identify the following kinds of dependencies:

- $cn_i \rightarrow cn_j$: denotes cn_j is a component or superclass of cn_i.
- $mn_i \rightarrow cn_j$: 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 a method override 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 our technical report [14]. 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 for simultaneous processing. Correspondingly, each set of methods in a SCC is regarded as a mutual-recursive set for fixed-point analysis. Through a bottom-up processing of each SCC, we are able to perform region inference in a modular and systematic fashion.

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

(a) Source program

```
List⟨r7,r8,r9⟩ join⟨r1,..,r9⟩(List⟨r1,r2,r3⟩ xs, List⟨r4,r5,r6⟩ ys)
  where pre.join⟨r1,..,r9⟩
{if isNull⟨r1,r2,r3⟩(xs) then
  if isNull⟨r4,r5,r6⟩(ys) then (List⟨r7a,r8a,r9a⟩)null // r7a⪰r7∧r8a⪰r8∧r9a⪰r9
  else join⟨r4,r5,r6,r1,r2,r3,r7b,r8b,r9b⟩(ys,xs) // r7b⪰r7∧r8b⪰r8∧r9b⪰r9
 else {
  Object⟨r10⟩ x; List⟨r11,r12,r13⟩ res;
  x=xs.getValue⟨r2⟩(); // r2⪰r10
  xs=xs.getNext⟨r1,r2,r3⟩();
  res=join⟨r4,r5,r6,r1,r2,r3,r7c,r8c,r9c⟩(ys,xs); // r7c⪰r11∧r8c⪰r12∧r9c⪰r13
  new List<r14,r15,r16>(x,res)
  // r10⪰r15∧r11⪰r14∧r12⪰r15∧r13⪰r16∧r14⪰r17∧r15⪰r8∧r16⪰r9 } }
```

(b) Initial region-annotated program

```
List⟨r7..r9⟩ join⟨r1..r9⟩(List⟨r1..r3⟩ xs,List⟨r4..r6⟩ ys)
  where pre.join⟨r1..r9⟩
{if isNull⟨r1,r2,r3⟩(xs) then
  if isNull⟨r4,r5,r6⟩(ys) then (List⟨r7,r8,r9⟩)null
  else join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩(ys,xs)
 else {
  Object⟨r8⟩ x; List⟨r9,r8,r9⟩ res;
  x=xs.getValue⟨r2⟩();
  xs=xs.getNext⟨r1,r2,r3⟩();
  res=join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩(ys,xs);
  new List<r7,r8,r9>(x,res) // r2⪰r8
  } }
```

(c) Final region-annotated program

```
Q={pre.join⟨r1..r9⟩=(r2⪰r8)∧pre.join⟨r4..r6,r1..r3,r7..r9⟩}

pre.join0⟨r1..r9⟩ = True

pre.join1⟨r1..r9⟩ = r2⪰r8 ∧ pre.join0⟨r4..r6,r1..r3,r7..r9⟩
                  = r2⪰r8

pre.join2⟨r1..r9⟩ = r2⪰r8 ∧ pre.join1⟨r4..r6,r1..r3,r7..r9⟩
                  = r2⪰r8 ∧ r5⪰r8

pre.join3⟨r1..r9⟩ = r2⪰r8 ∧ pre.join2⟨r4..r6,r1..r3,r7..r9⟩
                  = r2⪰r8 ∧ r5⪰r8
```

(d) Fixed-point analysis

Figure 6: Region Inference for a Recursive Method

4.4 Override Conflict Resolution

As mentioned in Sec 3.4, class subtyping and method overriding must comply with their respective checks to ensure the soundness of subsumption. The class subtyping check is relatively easy to enforce. The existing [CD] rule already accumulates the invariant from each class A to its subclass B in order to ensure:

$$inv.B\langle r_1..r_n\rangle \Rightarrow inv.A\langle r_1..r_m\rangle$$

In contrast, the method overriding check is more complex. Consider a class A, its subclass B, and a method $A.mn$ overridden by $B.mn$. For method overriding to be sound, we require the following property to be satisfied:

$$\begin{array}{l} inv.B\langle r_1..r_n\rangle \wedge pre.A.mn\langle r_1..r_m, r'_1..r'_p\rangle \\ \quad \Rightarrow pre.B.mn\langle r_1..r_n, r'_1..r'_p\rangle \end{array}$$

This property may not hold initially. To rectify this, the region inference can selectively augment the premise of each overriding check, with the following considerations:

1. We can strengthen either the premise $inv.B\langle r_1..r_n\rangle$ or the premise $pre.A.mn\langle r_1..r_m, r'_1..r'_p\rangle$ or both.
2. Strengthening $pre.A.mn\langle r_1..r_m, r'_1..r'_p\rangle$ can be problematic as some regions, namely $r_{m+1}..r_n$, are present in class B but not A.

These two issues can be considered systematically by examining each basic constraint of $pre.B.mn\langle r_1..r_n, r'_1..r'_p\rangle$ to determine if it (i) is already valid, or (ii) can be added to $pre.A.mn$, or (iii) can be added to $inv.B$, or (iv) can be split into an equality constraint for $inv.B$ and a modified constraint for $pre.A.mn$. We formalise this conflict resolution as the following inference rule:

$$I, X,\ Y \vdash I', X'$$

Note that I denotes the class invariant of the subclass, X denotes the precondition of the overridden method (from the superclass), while Y represents the precondition of the overriding method (from the subclass). The results I', X' are strengthened versions of I, X which satisfy the soundness of overriding. Each constraint is expressed as a set of atomic constraints in conjunctive form $\bigwedge c$, where $c \;\equiv\; r_1{\succeq}r_2 \mid r_1{=}r_2$. In the following rules, we assume that $R_B = \{r_1..r_n\}$, $R_X = \{r_1..r_m, r'_1..r'_p\}$ and $R_A = \{r_1..r_m\}$. Note that $\rho : R_1{\rightarrow}R_2$ denotes a region substitution with $R_1(R_2)$ as its domain (co-domain). $ctr(\rho)$ transforms the substitution ρ into an equality constraint. For example, $ctr([r_1{\mapsto}r_2, r_3{\mapsto}r_4]) = (r_1{=}r_2{\wedge}r_3{=}r_4)$.

$$\frac{I \wedge X \Rightarrow Y}{I, X,\ Y \vdash I, X} \qquad \frac{\begin{array}{c} c \in Y \quad \neg(I \wedge X \Rightarrow c) \\ reg(c) \subseteq R_X \qquad I, X \wedge c,\ Y \vdash I', X' \end{array}}{I, X,\ Y \vdash I', X'}$$

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

We use the following extension of the `Pair` class to illustrate this override resolution mechanism:

```
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}
```

Two basic constraints are present in an overriding `cloneRev` method, namely `r2⪰r6` and `r3a⪰r5`. The first constraint is already satisfiable, but the second constraint cannot be directly placed in the class invariant of `Triple`, nor in the precondition of `Pair.cloneRev`. Nonetheless, we can still split it into two constraints `r3a=r3` and `r3⪰r5` that can be added to `inv.Triple` and `pre.Pair.cloneRev`, respectively. We have a choice of mapping the extra region `r3a` to either `r3` or `r2` using `[r3a↦r3]` or `[r3a↦r2]`, respectively. We choose the former since `(r3⪰r5)` exists in `pre.Pair.cloneRev` but not `(r2⪰r5)`. While multiple solutions exist, we choose a solution which minimises the number of new constraints.

```
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 = lb:new B⟨r1,r2,r5⟩(..)  // B upcast to A
else ..
  a = lc:new C⟨r1,r2,r6⟩(..)  // C upcast to A
else ..
  a = le: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

4.5 Correctness

The correctness of the type inference algorithm is often defined in relation to a checking system. We have formalised a comprehensive set of region type checking rules and proven their safety properties in our technical report [14]. In our type system for region-annotated Core-Java, $P \vdash_{def} def$ denotes that class declaration *def* is well-region-typed (or well-typed in short), $P;\Gamma;R;\Psi \vdash_{meth} meth$ indicates that a method *meth* is well-typed, $P;\Gamma;R;\Psi \vdash e : t$ indicates that an expression *e* is well-typed, while $\vdash P$ denotes that a region-annotated program *P* is well-typed.

The following theorem states the correctness of the region inference algorithm. It guarantees the existence of a well-region-typed target program P' for each well-normal-typed source program *P*. By region erasure, we can show that both programs have the same observable behaviour (through bisimulation) in the absence of dangling accesses. The main safety property (proven in [14]) is that no dangling reference is ever created during the execution of any well-region-typed expression.

THEOREM 1 (CORRECTNESS). *Given any well-normal-typed source program P in Core-Java, there exists a region-annotated program P', such that $\vdash P \Rrightarrow P'$ and P' is well-region-typed.*

The proof of Theorem 1 relies on a global dependency graph and the following Lemma. The details of the proof can be found in [14]. The Lemma states that each part of the inferred program is well-typed, assuming those parts it depends on are well-typed.

LEMMA 2. *Suppose $\vdash P \Rrightarrow P'$.*

(a). If $\Gamma \vdash e \Rrightarrow e' : t,\ \varphi$,
and all classes and static methods that e' depends on 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 $\Gamma \vdash meth \Rrightarrow meth'; Q$,
and all classes and static methods that $meth'$ depends on are well-typed 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 $\vdash def \Rrightarrow def'; Q$,
and all classes and static methods that def' depends on are well-typed in P', then $\vdash_{def} def'$.

5. HANDLING DOWNCASTS

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 [7], 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 providing an incorrect region annotation. With automatic region inference, it is the responsibility of the region inference system to (at compile time) prevent such a situation. Let us see how this problem can be resolved.

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

To illustrate the problem, consider the program fragment in Fig 7. Note that every `new` statement is labelled with a unique program point to identify its source location. 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 originally lost during the upcast operation. We propose two techniques to preserve these regions.

Our first technique preserves 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 = lb:new B⟨r1,r2,r5⟩(..)  // r5=r1
else ..
  a = lc:new C⟨r1,r2,r6⟩(..)  // r6=r1
else ..
  a = le: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 and modular, some lifetime precision are lost due to the additional region equality constraints.

Another solution is to maintain extra regions during upcasting for objects that may be subsequently downcasted. Specifically, all objects that may be downcasted (to some subclasses) must be padded in advance with sufficient number of extra regions to support subsequent region-safe downcasting. A global 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 subclass (`D`) has the maximum number of regions, we shall pad both these types with up to 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, the objects `a2` and `b` are never downcasted and hence we

Programs	Size (lines)		Compile-Time (sec)		Param.	Space Usage/Total Allocation				Diff. in
	Source	Ann.	Inference	Checking	Input	No Sub	Object Sub	Field Sub	RegJava	`letreg`
Sieve of Eratosthenes	80	12	0.08	0.14	50000	1	1	1	1	0
Ackermann	67	5	0.02	0.04	(4,7)	0.004	0.004	0.004	0.004	0
Merge Sort	170	16	0.35	0.47	50000	0.179	0.179	0.179	0.179	0
Mandelbrot	110	14	0.05	0.09	100	0.002	0.002	0.002	0.002	0
Naive Life	114	14	0.08	0.23	10	1	1	1	1	0
Optimized Life (array)	121	15	0.09	0.25	10	0.196	0.196	0.196	0.196	0
Optimized Life (dangling)	35	5	0.01	0.04	10	1	1	1	1	-1
Optimized Life (stack)	80	10	0.04	0.08	10	1	1	1	1	0
Reynolds3	59	12	0.11	0.29	10	1	1	0.004	-	-
foo-sum	65	10	0.11	0.24	100	0.340	0.010	0.010	-	-

Figure 8: Comparative Statistics on Inference/Checking and Region Subtyping

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 = lb:new B⟨r1,r2,r5⟩(..)  //r5=r3
else ..
  a = lc:new C⟨r1,r2,r6⟩(..)  //r6=r3
else ..
  a = le: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
```

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 this approach to region-safe downcast, we propose a backwards analysis technique to find a potential downcast set for each object in our program. We first provide a set of inference rules to gather a set of backward flows. The inference rule is expressed using the relation:

$$\Gamma,\ x \vdash e,\ C$$

Note that x is the receiver that may capture the result of e under type environment, Γ. The output C denotes a set of backward flows that occur in e and its receiver x.

Each backward flow is represented using either $v_1 \dashrightarrow v_2$ or $v_1 \xrightarrow{D} v_2$, where the arrows indicate that v_1 captures a value from v_2. In addition, the second arrow is annotated with a D-class to indicate that its source may be subjected to a downcast-to-D operation.

The rule for downcast operation is defined as:

$$\Gamma, x \vdash (D)\, v,\ \{x \xrightarrow{D} v\}$$

The value of v may flow into its outer receiver x and be subjected to a downcast operation. This is captured by $x \xrightarrow{D} v$.

Our technical report provides the details of this flow inference[14]. For our example, the initial set of flows is:

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

We can proceed to perform a transitive closure to gather all program points of variable declarations and object allocations that could be downcasted. The goal of our analysis is to find a set of classes that could be subsequently downcasted for each object at a given location. For each variable, v, or object field, $v.f$, we associate a set of casts D using $v[D]$ or $v.f[D]$.These sets are initially empty and are initialised by the following rule:

$$v \xrightarrow{D} w\ \wedge\ w[S]\ \Rightarrow\ v \dashrightarrow w\ \wedge\ w[S \cup \{D\}]$$

For our example, the initial flow set is converted to:

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

Applying a closure of backward flows (see [14]) 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 a downcast closure operation 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 the downcast is bound to fail. For example, all possible downcasts at point *le* will fail since an *E* object is created that is not a subclass 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.

Our approach to downcast safety is achieved at compile-time through global flow analysis. Another approach is to make use of type polymorphism (advocated in Generic Java [10]). This approach is expected to alleviate the need for downcast operations but requires a fundamental change in the design of the language.

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 system was built using the Glasgow Haskell compiler[31]; we have also added a library to solve region outlive constraints.

The primary objective of our initial experiments is to evaluate the quality of our automatically inferred region annotations as compared to region annotations produced by hand. We tested our system on a set of eight RegJava benchmark programs from [16] that have been hand-annotated for their region checking system. Figure 8 summarises the statistics for each program.

The last column indicates the difference in the number of localised regions between our inferred annotations and those which were hand-annotated in [16]. All annotations are identical, except for *optimized life (with dangling)*. Our inference produces one less local region, since we use the *no-dangling* policy rather than the *no-dangling-access* policy of the RegJava checker. For this set of programs, our region inference is therefore comparable in performance to human experts. Take note that the region annotations occur in around 12.3% of the programs' lines. This indicates that manually generating the region annotations may represent a sizeable mental effort for a programmer with only a region type checker. Note also that the region inference and region checking times are reasonable for this set of programs, running in less than a second for all of the programs in this benchmark set. We have also compiled the programs to run on a region-based custom allocator, called Titanium[29], and measured the resulting space utilization as compared with the total allocation space of the program. Due to the lack of space reuse, four of the programs have a ratio of one, including *optimized life (with dangling)* despite an extra localized region. Four other examples have significant space reuse but have the same performance for all three kinds of region subtyping. However, we achieved significantly better space reuse for *Reynolds3* and *foo-sum* when our inference is augmented with object/field region

Programs	bisort	em3d	health	mst	power	treeadd	tsp	perim.	n-body	voronoi
Source (lines)	340	462	562	473	765	195	545	745	1128	1000
Ann. (lines)	7	32	24	34	35	7	12	21	38	50
Inference (seconds)	0.14	0.61	3.58	0.48	0.4	0.07	0.28	1.38	2.88	4.63

Figure 9: Region Inference Times for the Olden Benchmark Programs

subtyping. To check the scalability of our region inference, we converted a set of ten programs from the Olden benckmark set[11] to Core-Java, and measured their inference times, as shown in Fig 9.

7. RELATED WORK

Tofte and Talpin [32, 33] proposed a region inference approach for a typed call-by-value λ-calculus. 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 types can be considered as a special case of objects, as it has an outlives requirement for its values when compared to its location. This requirement is specified indirectly through region effects. Apart from this, their method is tailored mainly to functional languages.

Christiansen and Velschow proposed a similar region-based approach to memory management in Java [16]. They call their system RegJava and use a stack of lexically scoped regions for memory management. 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 annotate programs with region annotations. In their system, each class is augmented with the full set of regions from the entire class hierarchy, including those from its subclasses and its sibling classes. While this makes downcast operations trivially safe, it uses phantom regions and has a closed-world assumption on the class hierarchy.

Researchers have recently advocated non-lexical regions to support tighter region lifetimes[24, 25, 28, 34]. Most of these approaches require programmers to at least indicate when regions are to be created, allocated and released. One technique [1] accepts a program with lexically scoped regions, then transforms the program to allow, when possible, late creation and early deletion of these regions. This technique is complementary to our approach to region inference, as it could be used as a post-phase. With an explicit outlives relation on the lexical regions, we have also exploited the concept of region subtyping, as pioneered in [26].

Beebee and Rinard present an early implementation of scoped memory for Real-Time Java in the MIT Flex compiler infrastructure[3]. They rely on both static analysis and dynamic debugging to help locate incorrect uses of scoped memory. Later, Boyapati *et. al.* [9] combined region types [32, 33, 19, 26, 16] and ownership types [18, 17, 6, 8] in a unified framework to capture object encapsulation and prevent dangling references. The 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, each object and all components it *owns* are put into the same region; in order to optimize on region lifetimes. Our region type system is similar to theirs, but we separate out object encapsulation and RTSJ issues. We also infer region types automatically across procedures, whilst they have limited support through intra-procedure inference and the use of defaults in region types.

Deters and Cytron [21] automatically translated Java code into Real-Time Java using dynamic analysis to determine the lifetime of an object. Because the analysis is dynamic, it may not be sound — it may miss some execution paths that create and use dangling references given their extracted object lifetime information.

In research performed concurrently with ours, Cherem and Rugina have developed a region inference algorithm for Java [13]. This region inference algorithm handles all of the features of Java, including inheritance, dynamic dispatch, downcasts, and multithreading. Unlike our inference system, their system uses the no-dangling-access principle and produces programs that use non-lexically scoped regions. Our inference system is designed to produce an augmented program with region type annotations; perhaps because non-lexically scoped regions are less amenable to formalisation in a region type system, their system produces a program with region handles and region-based memory management, but without region types nor region subtyping.

8. CONCLUDING REMARKS

Our aim is to provide a fully-automatic region inference system for a core subset of Java. We achieved this by allowing classes and methods to be region-polymorphic, with region-polymorphic recursion for methods. As shown by the examples, the inferred region constraints allow us to obtain fairly precise region annotations. We have seen how the region lifetime constraints prevent dangling references and generate appropriate region instantiations. There remain 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 [9]. 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 currently 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 the stack. To take advantage of this, we could introduce a fictitious region, denoted by $\top$, for each null value. Such a region would not exist during the execution of the program since null objects do not occupy space. The following axioms hold at all times for any r: $\top \succeq r$, $r \succeq \top$, $r = \top$ and $r \neq \top$. In combination with an analysis that tracks definite occurrences of null values (e.g. [23]), these axioms can improve the region lifetime constraints.

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 boundaries. Flow-insensitivity may cause some loss in region lifetime precision when the same local variable is used for objects with different lifetime requirements. To rectify this, we could use Static Single Assignment (SSA) intermediate form[20] which is known to give better flow-sensitive analysis results. Conversion of programs to SSA form can be handled in a preprocessing step, prior to region inference.

Our current set of rules may introduce localised regions at each expression block. These are presently mandated at the bodies of procedures, though in practice they can occur in any subexpression. Effective placement of local variable declarations, object allocations and expression blocks can therefore affect region placement and the extent to which memory is effectively reused. Another future work is to explore suitable liveness analysis and restructuring transformations to further improve the 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 various pointers. Huu Hai also helped with some coding. Special thanks to Fritz Henglein for insightful comments on an earlier draft. The referees of PLDI04 provided many invaluable suggestions, including the use of region subtyping which we have now adopted. We acknowledge the support of Singapore-MIT Alliance and grant R252-000-092-112.

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] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *ACM POPL*, pages 171–183. ACM Press, January 1996.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

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

[7] 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.

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

[9] 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.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *ACM OOPSLA*, October 1998.

[11] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *ACM PPoPP*, pages 29–38, Santa Barbara, California (ACM Press), May 1993.

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

[13] S. Cherem and R. Rugina. Region Analysis and Transformation for Java Programs. Technical Report, Computer Science Dept, Cornell University, April 2004.

[14] W.N. Chin, F. Craciun, S.C. 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⟩.

[15] 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.

[16] M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

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

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

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

[20] 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 Trans. on Programming Lang. and Systems*, 13(4):451–490, 1991.

[21] 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.

[22] A. Deutsch. On the complexity of escape analysis. In *ACM POPL*, pages 358–371. ACM Press, 1997.

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

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

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

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

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

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

[29] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual v1.11. Computer Science Division (EECS), University of California Berkeley, California, Mar 2004.

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

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

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

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

[34] D. Walker and K. Watkins. On regions and linear types (extended abstract). In *ACM ICFP*, pages 181–192. ACM Press, 2001.