Region Inference for an Object-Oriented Language

Technical Report

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[17]. 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 [22, 17, 16, 9]. Most of these projects (e.g. [17, 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 han-

dles object-oriented features such as class subtyping, method overriding, and downcast operations.

- **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 [17] 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.
- **Region Checking System and Safety:** We formulate and implement a region type system that can type check either hand-annotated Core-Java programs or programs generated by our inference algorithm. We formulate soundness proofs for our type system which guarantees that well-typed programs are region-safe and will not create dangling references in either its regions or stack.
- **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 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 presents a region type system for Core-Java, together with a set of safety guarantees. Sec 5 formalises the region inference rules, together with a correctness proof. Some examples of inference are then highlighted. Sec 6 formalises a global dependency graph that points to an ordering for performing region inference. Sec 7 describes how method override conflicts can be identified and rectified. Sec 8 proposes two compile-time solutions for the region-safety of downcast operations. Sec 9 reports on a preliminary experiment from our implementation of region inference. Sec 10 discusses related works, followed by some concluding remarks.

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[33]. 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.

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

(a) The Source Language

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

(b) The Target Language

Figure 1: The Syntax of Core-Java

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 $n \geq 0$.

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 $\forall r \cdot heap \succeq r$.

A constraint abstraction [29] 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 $inv.cn\langle r_1, .., r_n\rangle$. The region constraint for each method is also known as the *method precondition* and is denoted using $pre.m\langle r_1, .., r_n\rangle$, 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-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 [36, 37, 17, 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.

```
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

- 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[28], 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[36, 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 [17]. The second kind was introduced in [28]. 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:

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

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 [23, 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[36, 4], the performance of such a region-inferred program is comparable to that obtained by escape analysis[23]. 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⪰r3 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 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⪰r4∧r5⪰r4 of the resulting type Pair⟨r4,r5,r6⟩. 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 ⟨r1..rn⟩, to be an extension of A, namely ⟨r1..rm⟩, with n≥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 [13].

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 $r'_1, \ldots, r'_p$.

```
Y A.mn ⟨r'1,...,r'p⟩(X a) where φA.mn {...}
Y B.mn ⟨r'1,...,r'p⟩(X a) where φB.mn {...}
```

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 7, after the basic region inference method has been presented.

4. REGION CHECKING AND SAFETY

This section is devoted to a region type system for region-annotated Core-Java and its safety property with respect to the dynamic semantics given in Section 4.2.

4.1 Static Semantics

We shall formulate the static semantics for region-annotated Core-Java through the following set of typing relations.

- $def \in P$. It denotes a class within a given program P.
- $meth \in P$. It denotes a static method within program P.
- $P \vdash mbr \in ca$. It indicates that *mbr* is a member of class *ca*, either declared in current class *ca*, or inherited from its superclass. *mbr* can be a field or a method. Direct membership without inheritance is captured by $P \vdash mbr \in_D ca$.
- $P; R \vdash_{constr} t,\ \varphi$. The constraint that should be imposed on type t is φ.
- $P; R; \Psi \vdash_{type} t$. The type t is well-formed wrt program P, the set of alive regions R and the region constraint Ψ.
- $P; R; \Psi \vdash_{formal} t\ v$. A field $(t\ v)$ in a class or a formal parameter of a method is well formed.
- $P \vdash t <: t', \rho$. This captures a subtype relation between $\tau\langle x_{1..q}\rangle$ and $\tau'\langle x'_{1..q}\rangle$ with substitution ρ to enforce identical parameter regions.
- $P; R; \Psi \vdash t <: t'$. This states that t is a subtype of t' under the constraint Ψ.
- $P \vdash_{def} def$. It denotes that class declaration *def* is well-formed in program P.
- $P; \Gamma; R; \Psi \vdash_{meth} meth$. It specifies that method *meth* is well-defined with respect to program P, type environment Γ, live regions R and the region constraint Ψ.
- $P; \Gamma; R; \Psi \vdash e : t$. It indicates that expression e is of type t with respect to program P, type environment Γ, the set of alive regions R and the constraint Ψ.
- $\vdash P$. It indicates that program P is well-formed.

We shall begin with some auxiliary typing relations that are used to extract classes, methods and fields from a given program, together with checks on the well-formedness of types, parameters and subtyping.

- The following two rules are for extracting the classes and static methods from a given program.

$$\frac{P = \ldots def \ldots\ meth^*\ Q}{def \in P} \qquad \frac{P = def^*\ \ldots meth \ldots\ Q}{meth \in P}$$

- Methods/fields defined in a class.

  Let $mbr = field \mid meth$ be a class member (i.e. field or instance method). We first define direct membership using the following rules:

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle \ldots \{\ldots mbr \ldots\} \in P}{P \vdash mbr \in_D cn\langle r_{1..n}\rangle}$$

  Fields/methods may be inherited through the superclass. This could be dealt by the following membership with inheritance relation.

$$\frac{P \vdash mbr \in_D cn\langle r_{1..n}\rangle}{P \vdash mbr \in cn\langle r_{1..n}\rangle}$$

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle\ \cdots \in P \quad P \vdash mbr \in cn'\langle r_{1..m}\rangle \quad \neg(P \vdash mbr \in_D cn\langle r_{1..n}\rangle)}{P \vdash mbr \in cn\langle r_{1..n}\rangle}$$

  For convenience, we may drop P from our relations and use $mbr \in cn\langle r_{1..n}\rangle$. Also, we introduce $(cn', mbr) \in cn\langle r_{1..n}\rangle$ that returns the class cn' that *mbr* is inherited from.

- Well-formedness of types and fields/parameters.

  The first three rules derive the invariant associated with each type. For each class type, there is some invariant describing the lifetime relation on regions involved. However, for primitive and *Object* types, no invariant is imposed since they have at most one region parameter.

$$\frac{}{P; R \vdash_{constr} prim\langle\rangle,\ true} \qquad \frac{r \in R}{P; R \vdash_{constr} Object\langle r\rangle,\ true}$$

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\ \textbf{where}\ \varphi\ \{...\}\ \in\ P \qquad R \supseteq \{x_1, ..., x_n\}}{P; R \vdash_{constr} cn\langle x_{1..n}\rangle, [r_1 \mapsto x_1, ..., r_n \mapsto x_n]\varphi}$$

The next two rules are used to check the well-formedness of types and fields, under a region constraint Ψ.

$$\frac{P; R \vdash_{constr} t,\ \varphi \qquad \Psi \Rightarrow \varphi}{P; R; \Psi \vdash_{type} t} \qquad \frac{P; R; \Psi \vdash_{type} t}{P; R; \Psi \vdash_{formal} t\ v}$$

- Subtyping relation.

  The three subtyping rules are already given in Section 3.2. The following is another checking rule that will be used later.

$$\frac{P \vdash t <: t',\ \varphi \qquad \Psi \Rightarrow \varphi \quad P; R; \Psi \vdash_{type} t \quad P; R; \Psi \vdash_{type} t'}{P; R; \Psi \vdash t <: t'}$$

We shall now describe typing relations for classes and methods.

- Well formed class definitions: $P \vdash_{def} def$

  This relation states that class declaration *def* is well-formed in program *P*.

$$\frac{\varphi \Rightarrow \forall i : 2..n \cdot r_i \succeq r_1 \qquad R = \{r_1, \ldots, r_n, heap\} \quad \forall i : 1..q \cdot P; \{this : cn\langle r_{1..n}\rangle\}; R; \varphi \vdash_{meth} meth_i \quad \forall i : 1..p \cdot P; R; \varphi \vdash_{formal} field_i}{P \vdash_{def} \textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\ \textbf{where}\ \varphi\ \{field_{1..p}\ meth_{1..q}\}}$$

  A class is well defined if all its fields and methods are well-formed and the constraint specifies necessary lifetime relations among its regions.

- Well formed methods: $P; \Gamma; R; \Psi \vdash_{meth} meth$

  This relation specifies that method *meth* is well-defined with respect to program *P*, type environment Γ, the set of live regions *R* and the region constraint Ψ.

$$\frac{\Gamma' = \Gamma, (v_j : t_j)_{j:1..p} \qquad R' = R \cup \{r_1, \ldots, r_m\} \quad P; R'; \Psi' \vdash_{type} t_j,\ j = 0, .., p \qquad \Psi' = \Psi \wedge \varphi_2 \quad P; \Gamma'; R'; \Psi' \vdash eb : t_0' \qquad P; R'; \Psi' \vdash t_0' <: t_0}{P; \Gamma; R; \Psi \vdash_{meth} t_0\ mn\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p})\ \textbf{where}\ \varphi_2\ eb}$$

  Region constraint of variables are checked at declaration block. For each type, all regions used must be alive in *R* and has an invariant that is satisfied by the expected context, Ψ'.

Our next typing relation $P; \Gamma; R; \Psi \vdash e : t$ indicates that expression e is of type t with respect to program *P*, type environment Γ, the set of alive regions *R* and the constraint Ψ. The relation is defined in a syntax-directed fashion, as follows.

- Constants.

$$\frac{P; R; \Psi \vdash_{type} cn\langle r_{1..n}\rangle}{P; \Gamma; R; \Psi \vdash (cn\langle r_{1..n}\rangle)\ \textbf{null} : cn\langle r_{1..n}\rangle} \qquad \frac{}{P; \Gamma; R; \Psi \vdash k : prim_k\langle\rangle}$$

- Variables.

$$\frac{(v : t) \in \Gamma}{P; \Gamma; R; \Psi \vdash v : t}$$

  Take note that region constraints of variables are checked at declaration sites, rather than at their uses.

- Expression block.

$$\frac{P; R; \Psi \vdash_{type} t_i \quad i = 1, .., p \quad \Gamma' = \Gamma + (v_i : t_i)_{i:1..p} \quad P; \Gamma'; R; \Psi \vdash e : t}{P; \Gamma; R; \Psi \vdash \{(t_i\ v_i)_{i:1..p}\ e\} : t}$$

- Object creation.

$$\frac{P; R; \Psi \vdash_{type} cn\langle x_{1..n}\rangle \quad fieldlist(cn\langle x_{1..n}\rangle) = (t_i\ f_i)_{i:1..p} \quad (v_i : t_i') \in \Gamma \quad P; R; \Psi \vdash t_i' <: t_i \quad i = 1, .., p}{P; \Gamma; R; \Psi \vdash \textbf{new}\ cn\langle x_{1..n}\rangle(v_1, .., v_p) : cn\langle x_{1..n}\rangle}$$

  Note that function $fieldlist(cn\langle x_{1..n}\rangle)$ returns a list comprising all available fields in class $cn\langle x_{1..n}\rangle$, including the fields of its superclasses. They are organised in an order determined by the constructor function.

- Object field access and assignment.

  Let $lhs = v \mid v.f$ be either a variable v or an object field $v.f$. The rule on the right covers both object field update and general assignment.

$$\frac{(v : cn\langle a_{1..n}\rangle) \in \Gamma \quad P \vdash (tf) \in cn\langle r_{1..n}\rangle \quad t' = [r_1 \mapsto a_1, ..., r_n \mapsto a_n]t}{P; \Gamma; R; \Psi \vdash v.f : t'} \qquad \frac{P; \Gamma; R; \Psi \vdash lhs : t \quad P; \Gamma; R; \Psi \vdash e : t' \quad P; R; \Psi \vdash t' <: t}{P; \Gamma; R; \Psi \vdash lhs = e : \textbf{void}}$$

- Method invocation.

  The first rule deals with instance method invocation, while the second rule is for static method invocation.

$$\frac{(v_0 : cn\langle a^+\rangle) \in \Gamma \quad P \vdash meth \in cn\langle r^+\rangle \quad meth = t\ mn\langle r'^*\rangle((t_i\ v_i)_{i:1..n})\ \textbf{where}\ \varphi_0\ eb \quad (v_i' : t_i') \in \Gamma,\ i = 1, .., n \quad \rho =_{df} [r^+ \mapsto a^+,\ r'^* \mapsto a'^*] \qquad \Psi \Rightarrow \rho\varphi_0 \quad P; R; \Psi \vdash t_i' <: \rho t_i,\ i = 1, .., n}{P; \Gamma; R; \Psi \vdash v_0.mn\langle a'^*\rangle(v'_{1..n}) : \rho t}$$

$$\frac{meth\ \in\ P \quad meth = t\ mn\langle r'^*\rangle((t_i\ v_i)_{i:1..n})\ \textbf{where}\ \varphi_0\ eb \quad (v_i' : t_i') \in \Gamma,\ i = 1, .., n \quad \rho =_{df} [r'^* \mapsto a'^*] \qquad \Psi \Rightarrow \rho\varphi_0 \quad P; R; \Psi \vdash t_i' <: \rho t_i,\ i = 1, .., n}{P; \Gamma; R; \Psi \vdash mn\langle a'^*\rangle(v'_{1..n}) : \rho t}$$

- Sequential composition, conditional and region declaration.

$$\frac{P; \Gamma; R; \Psi \vdash e_1 : t_1 \quad P; \Gamma; R; \Psi \vdash e_2 : t_2}{P; \Gamma; R; \Psi \vdash e_1\ ;\ e_2 : t_2} \qquad \frac{v : \textbf{bool}\langle\rangle \in \Gamma \quad P; \Gamma; R; \Psi \vdash e_1 : t_1 \quad P; R; \Psi \vdash t_1 <: t \quad P; \Gamma; R; \Psi \vdash e_2 : t_2 \quad P; R; \Psi \vdash t_2 <: t}{P; \Gamma; R; \Psi \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : t}$$

  Conditional branches are checked separately before a most specific super-class is identified through subtyping.

- Region declaration.

$$\frac{fresh\ a \qquad \varphi = (R \succ a) \quad freereg(t) \subseteq R \quad P; \Gamma; R \cup \{a\}; \Psi \wedge \varphi \vdash [r \mapsto a]eb : t}{P; \Gamma; R; \Psi \vdash \textbf{letreg}\ r\ \textbf{in}\ eb : t}$$

  Local blocks are checked with an extra live region, and the constraint $R \succ a$, which is an abbreviation for $\bigwedge_{a' \in R}(a' \succeq a \wedge a' \neq a)$. Also, $freereg(t)$ returns all free region variables in *t*.

Lastly, we give the typing relation for the entire program using $\vdash P : t$, together with some auxiliary definitions. It indicates that program P is well-typed (with type t).

$$\frac{\begin{array}{c}\mathit{WFClasses}(P) \quad \mathit{FieldsOnce}(P) \quad \mathit{MethodsOnce}(P) \quad \mathit{InheritanceOK}(P) \\ P = \mathit{def}_{1..n}\ \mathit{meth}_{1..m}\ Q \qquad \forall i : 1..n \cdot P \vdash_{\mathit{def}} \mathit{def}_i \\ \forall i : 1..m \cdot P; \emptyset; \{\mathit{heap}\}; \mathit{true} \vdash_{\mathit{meth}} \mathit{meth}_i\end{array}}{\vdash P}$$

A region-annotated program is well typed if all declared classes and static methods are well-formed and the body of the program is well-typed. The predicates in the premise are used to capture standard well-formedness conditions for object-oriented programs. That is:

- no duplicate definitions of classes and no cycle in class hierarchy.
- no duplicate definitions of fields.
- no duplicate definitions of methods.
- soundness of class subtyping and method overriding.

Some auxiliary definitions follow. Note that IR^* is the transitive closure of the relation IR, whereas ID is the identity relation of class names.

$$\frac{\begin{array}{c}P = \mathit{def}_{1..n}\ \mathit{meth}^*\ Q \\ \mathit{def}_i = \textbf{class}\ cn_i\langle\ldots\rangle\ \textbf{extends}\ cn_{i'}\langle\ldots\rangle\ \ldots \\ \mathit{IR} = \{(cn_i, cn_{i'}) \mid 1 \le i \le n\} \\ \mathit{IR}^* \cap \mathit{ID} = \emptyset \\ \forall i, j : i \neq j \cdot cn_i \neq cn_j\end{array}}{\mathit{WFClasses}(P)}$$

$$\frac{\begin{array}{c}P = \mathit{def}_{1..n}\ \mathit{meth}^*\ Q \\ \mathit{def}_i = \textbf{class}\ cn_i\langle\ldots\rangle\ \textbf{extends}\ cn_{i'}\ \ldots\{(\mathit{fd}_{ij})_{j:1..p_i}\ (\mathit{meth}_{ij})_{j:1..q_i}\} \\ \forall j, l : j \neq l \cdot \textbf{name}(\mathit{fd}_{ij}) \neq \textbf{name}(\mathit{fd}_{il}),\ i = 1, .., n\end{array}}{\mathit{FieldsOnce}(P)}$$

$$\frac{\begin{array}{c}P = \mathit{def}_{1..n}\ \mathit{meth}_{1..k}\ Q \\ \mathit{def}_i = \textbf{class}\ cn_i\langle\ldots\rangle\ \textbf{extends}\ cn_{i'}\ \ldots\{(\mathit{fd}_{ij})_{j:1..p_i}\ (m_{ij})_{j:1..q_i}\} \\ \forall j, l : j \neq l \cdot \textbf{name}(m_{ij}) \neq \textbf{name}(m_{il}),\ i = 1, .., n \\ \forall j, l : j \neq l \cdot \textbf{name}(\mathit{meth}_j) \neq \textbf{name}(\mathit{meth}_l)\end{array}}{\mathit{MethodsOnce}(P)}$$

For the soundness of class subtyping and method overriding, we check every class definition through the following rules. Take note that $\varphi \Rightarrow \varphi'$ is to support the soundness of class subtyping, while $\Psi \wedge \varphi' \Rightarrow \varphi$ is to ensure the soundness of method overriding.

$$\frac{P = \mathit{def}_{1..n}\ \mathit{meth}^*\ Q \qquad \forall i : 1..n \cdot P \vdash \mathit{InheritanceOK}(\mathit{def}_i)}{\mathit{InheritanceOK}(P)}$$

$$\frac{\begin{array}{c}\mathit{def} = \textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle\ \textbf{where}\ \varphi\ \{\mathit{fd}_{1..p}\ \mathit{meth}_{1..q}\} \\ n \ge m \quad P, \{r_1, \ldots, r_m\} \vdash_{\mathit{constr}} cn'\langle r_{1..m}\rangle, \varphi' \quad \varphi \Rightarrow \varphi' \\ \exists \mathit{meth}' \cdot P \vdash \mathit{meth}' \in cn'\langle r_{1..m}\rangle \wedge \mathit{name}(\mathit{meth}') = \mathit{name}(\mathit{meth}_j) \\ \Rightarrow (P; \varphi \vdash \mathit{OverridesOK}(\mathit{meth}_j, \mathit{meth}')),\ j = 1, .., q\end{array}}{P \vdash \mathit{InheritanceOK}(\mathit{def})}$$

$$\frac{\begin{array}{c}\mathit{meth} = t_0\ mn\langle r_{1..n}\rangle((t\,v)_{1..m})\ \textbf{where}\ \varphi\ \ldots \\ \mathit{meth}' = t_0\ mn\langle r_{1..n}\rangle((t\,v)_{1..m})\ \textbf{where}\ \varphi'\ldots \\ \Psi \wedge \varphi' \Rightarrow \varphi\end{array}}{P; \Psi \vdash \mathit{OverridesOK}(\mathit{meth}, \mathit{meth}')}$$

4.2 Safety

This subsection is devoted to the safety properties of our region type system. We shall give the dynamic semantics for region-annotated Core-Java, and demonstrate the soundness of our region type system with respect to the dynamic semantics.

The dynamic semantics is described in small step. The following notations shall be used.

Locations :	ℓ	$\in$	$\mathit{Location}$
primitive values :	$k \mid (ca)\texttt{null}$	$\in$	prim
Values :	δ	$\in$	$\mathit{Value} = \mathit{prim} \uplus \mathit{Location}$
Store :	ϖ	$\in$	$\mathit{Store} = \mathit{Location} \rightharpoonup_{\text{fin}} \mathit{ObjVal}$
Variable env. :	Π	$\in$	$\mathit{VEnv} = \mathit{Var} \rightharpoonup_{\text{sfin}} \mathit{Value}$
Outlives relation :	η	$\subseteq$	$\mathit{RRel} = \{a \succ a' \mid a, a' \in \mathit{Region}\}$
Object values :	$cn\langle a^*\rangle\langle\delta'^*\rangle$	$\in$	$\mathit{ObjVal} = \mathit{CN} \times \mathit{Region}^* \times \mathit{Value}^*$

Notice that null values are of object type, but can be regarded as primitive values. They can be freely copied and are not placed in specific locations of the store.

Note that $f : A \rightharpoonup_{\text{fin}} B$ denotes a partial function from A to B with a finite domain, written $A = \mathit{dom}(f)$. While $f : A \rightharpoonup_{\text{sfin}} B$ denotes a *stackable* mapping from A to B. We write $f\{a \mapsto b\}$ for the function like f but mapping a to b (for stackable mapping f, it only updates the most recently included mapping of a by b), and $\{\}$ for the function always undefined.

A stackable mapping from A to B is defined formally as follows:

$$A \rightharpoonup_{\text{sfin}} B =_{\mathit{df}} (A^{\mathbf{Int}} \rightharpoonup_{\text{fin}} B,\ \mathbf{Int})$$

where **Int** denotes the next frame number of the mapping, while $A^{\mathbf{Int}}$ denotes the domain that has been marked with frame numbers. Each time a new frame is added, the number is increased by 1. We shall use the following operations:

Lookup into a frame : $(f, n)(v^i) = f(v^i)$, where $i < n$
Lookup for the latest :
$(f, n)(v) = f(v^i)$, *where* $v^i \in \mathit{dom}(f) \wedge \forall j > i \cdot v^j \notin \mathit{dom}(f)$
Next frame : $\mathit{frame}(f, n) = n$
Push a new frame : $(f, n) + \{(v \mapsto u)^*\} = (f + \{(v^n \mapsto u)^*\},\ n{+}1)$
Pop a frame : $(f, n{+}1)\backslash\{v^*\} = (f\backslash\{(v^n)^*\},\ n)$
Update : $(f, n)\{v \mapsto u\} = (f\{v^i \mapsto u\},\ n)$, *where*
$v^i \in \mathit{dom}(f) \wedge \forall j > i \cdot v^j \notin \mathit{dom}(f)$
Domain : $\mathit{dom}((f, n)) = \mathit{dom}(f)$
Membership : $v \in \mathit{dom}((f, n)) = \exists i < n \cdot v^i \in \mathit{dom}(f)$

The variable environment Π is such a stackable mapping: $\mathit{Var} \rightharpoonup_{\text{sfin}} \mathit{Value}$. The type environment Γ that corresponds to the runtime variable environment is also a stackable mapping: $\mathit{Var} \rightharpoonup_{\text{sfin}} \mathit{Type}$.

The domain of the store, $\mathit{dom}(\varpi)$, represents all global locations that have been allocated at a given moment. For a given location $\ell \in \mathit{dom}(\varpi)$, the region to which it belongs is recorded in the object value that ℓ refers.

We require some intermediate expressions for the dynamic semantics to follow through. The syntax of intermediate expressions is thus extended from the original expression syntax as follows:

$$e ::= \cdots \mid cn\langle r^*\rangle\langle\delta^*\rangle \mid \ell \mid \texttt{ret}(v^*, e) \mid \texttt{retr}(r, e) \mid \texttt{nullerr}$$

The expression $\texttt{ret}(v^*, e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The list of variables associated with `ret` is the local variables declared and used by the block. This set of variables is popped from the stack at the end of the block's evaluation. The expression $\texttt{retr}(r, e)$ is used to capture the result of evaluating an expression using a local region r. We use the special value `nullerr` to represent a null pointer error.

We extend the static semantics of the language to include these new expressions. In the process, we require introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with that captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics (eg. [35]). In our case, it is denoted by:

$$\Sigma \in \mathit{StoreType} = \mathit{Location} \rightharpoonup_{\text{fin}} \mathit{ca}$$

Judgements in the static semantics will be extended with store typing, as follows:

$$P; \Gamma; R; \Psi; \Sigma \vdash e : t$$

The static semantics for the new intermediate expressions are shown here:

$$\frac{r_{1..n} \subseteq R}{P; \Gamma; R; \Psi; \Sigma \vdash cn\langle r_{1..n}\rangle\langle\delta^*\rangle : cn\langle r_{1..n}\rangle} \qquad \frac{}{P; \Gamma; R; \Psi; \Sigma \vdash \ell : \Sigma(\ell)}$$

$$\frac{v^* \subseteq dom(\Gamma) \quad P; \Gamma; R; \Psi; \Sigma \vdash e : t}{P; \Gamma; R; \Psi; \Sigma \vdash \texttt{ret}(v^*, e) : t} \qquad \frac{P; \Gamma; R; \Psi; \Sigma \vdash e : t \quad r \in R \quad r \notin freereg(t)}{P; \Gamma; R; \Psi; \Sigma \vdash \texttt{retr}(r, e) : t}$$

The evaluation rules are of the following form.

$$\langle \varpi, \Pi, R, \eta\rangle[e] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[e']$$

The following are dynamic rules. Note that all evaluation rules that yield errors other than `nullerr` are omitted, and assumed to occur if no other rules are applicable.

$$\frac{v \in dom(\Pi)}{\langle \varpi, \Pi, R, \eta\rangle[v] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[\Pi(v)]}\ \textbf{[D-VAR]}$$

$$\frac{\Pi(v) = \ell \quad \varpi(\ell) = cn\langle a^*\rangle\langle\delta_{1..p}\rangle \quad i = mapName(cn, f)}{\langle \varpi, \Pi, R, \eta\rangle[v.f] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[\delta_i]}\ \textbf{[D-FR]}$$

Notice that $mapName(cn, f)$ returns an index of field f in the field list contained by the class cn.

$$\frac{\langle \Pi, \varpi, R, \eta\rangle[e] \hookrightarrow \langle \Pi', \varpi', R, \eta\rangle[e'] \quad lhs = v \mid v.f}{\langle \Pi, \varpi, R, \eta\rangle[lhs = e] \hookrightarrow \langle \Pi', \varpi', R, \eta\rangle[lhs = e']}\ \textbf{[D-AS1]}$$

$$\frac{v \in dom(\Pi) \quad \Pi' = \Pi\{v \mapsto \delta\}}{\langle \Pi, \varpi, R, \eta\rangle[v = \delta] \hookrightarrow \langle \Pi', \varpi, R, \eta\rangle[0]}\ \textbf{[D-AS2]}$$

$$\frac{\Pi(v) = \ell \quad \varpi(\ell) = cn\langle a^*\rangle\langle\delta'_{1..p}\rangle \quad i = mapName(cn, f) \quad \varpi' = \varpi\{\ell \mapsto cn\langle a^*\rangle\langle\delta'_1, .., \delta'_{i-1}, \delta, \delta'_{i+1}, .., \delta'_p\rangle\}}{\langle \Pi, \varpi, R, \eta\rangle[v.f = \delta] \hookrightarrow \langle \Pi, \varpi', R, \eta\rangle[0]}\ \textbf{[D-AS3]}$$

$$\frac{(\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ ca'\ \textbf{where}\ \varphi\ \{\ldots\}) \in P \quad \eta \Rightarrow [(r \mapsto a)_{1..n}]\varphi \quad \mathit{fresh}\ \ell \quad \delta_i = \Pi(v_i),\ i = 1, .., p \quad \varpi' = \varpi + \{\ell \mapsto cn\langle a_{1..n}\rangle\langle\delta_{1..p}\rangle\}}{\langle \Pi, \varpi, R, \eta\rangle[\textbf{new}\ cn\langle a_{1..n}\rangle(v_{1..p})] \hookrightarrow \langle \Pi, \varpi', R, \eta\rangle[\ell]}\ \textbf{[D-NEW]}$$

$$\frac{\Pi' = \Pi + \{(v_i \mapsto \textbf{init}(t_i))_{1..p}\}}{\langle \varpi, \Pi, R, \eta\rangle[\{(t\,v)_{1..p}\ e\}] \hookrightarrow \langle \varpi, \Pi', R, \eta\rangle[\texttt{ret}(v_{1..p}, e)]}\ \textbf{[D-EB]}$$

Note that in rule [**D-EB**], each of the locally declared variables is assigned, with the help of the function **init**, an initial value according to its type.

$$\frac{\langle \varpi, \Pi, R, \eta\rangle[e] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[e']}{\langle \varpi, \Pi, R, \eta\rangle[\texttt{ret}(v^*, e)] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[\texttt{ret}(v^*, e')]}\ \textbf{[D-RET1]}$$

$$\frac{}{\langle \varpi, \Pi, R, \eta\rangle[\texttt{ret}(v^*, \delta)] \hookrightarrow \langle \varpi, \Pi \backslash \{v^*\}, R, \eta\rangle[\delta]}\ \textbf{[D-RET2]}$$

$$\frac{(t_0\ mn\langle r'^*\rangle((t\,v)_{1..p})\ \textbf{where}\ \varphi\ eb) \in cn\langle r_{1..n}\rangle \quad \ell = \Pi(v'_0) \quad \varpi(\ell) = cn\langle a_{1..n}\rangle\langle\delta^*\rangle \quad \delta'_i = \Pi(v'_i),\ i = 1, .., p \quad \Pi' = \Pi + \{(v_i \mapsto \delta'_i)_{1..p}, \mathit{this} \mapsto \ell\} \quad \eta \Rightarrow [r'^* \mapsto a'^*]\varphi \quad e' = \texttt{ret}((\mathit{this}, v_{1..p}), [r'^* \mapsto a'^*]eb)}{\langle \varpi, \Pi, R, \eta\rangle[v'_0.mn\langle a'^*\rangle(v'_{1..p})] \hookrightarrow \langle \varpi, \Pi', R, \eta\rangle[e']}\ \textbf{[D-IM]}$$

$$\frac{(\mathit{meth} = t_0\ mn\langle r'^*\rangle((t\,v)_{1..p})\ \textbf{where}\ \varphi\ eb) \in P \quad \delta'_i = \Pi(v'_i),\ i = 1, .., p \quad \Pi' = \Pi + \{(v_i \mapsto \delta'_i)_{1..p}\} \quad e' = \texttt{ret}(v_{1..p}, [r'^* \mapsto a'^*]eb)}{\langle \varpi, \Pi, R, \eta\rangle[mn\langle a'^*\rangle(v'_{1..p})] \hookrightarrow \langle \varpi, \Pi', R, \eta\rangle[e']}\ \textbf{[D-SM]}$$

$$\frac{\mathit{fresh}\ a \quad R' = R \cup \{a\} \quad \eta' = (\eta \wedge (R \succ a))}{\langle \varpi, \Pi, R, \eta\rangle[\textbf{letreg}\ r\ \textbf{in}\ eb] \hookrightarrow \langle \varpi, \Pi, R', \eta'\rangle[\texttt{retr}(a, [r \mapsto a]eb)]}\ \textbf{[D-LETR]}$$

$$\frac{\langle \varpi, \Pi, R, \eta\rangle[e] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[e']}{\langle \varpi, \Pi, R, \eta\rangle[\texttt{retr}(a, e)] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[\texttt{retr}(a, e')]}\ \textbf{[D-RETR1]}$$

$$\frac{R \succeq a \quad (\delta = \ell) \Rightarrow (\mathit{curreg}(\ell) \neq a) \quad \forall v \in \Pi \cdot (\Pi(v) = \ell') \Rightarrow (\mathit{curreg}(\ell') \neq a)}{\langle \varpi, \Pi, R, \eta\rangle[\texttt{retr}(a, \delta)] \hookrightarrow \langle \varpi, \Pi, R - \{a\}, \eta \backslash a\rangle[\delta]}\ \textbf{[D-RETR2]}$$

Take note that we check the result and runtime stack to see if the region to be deleted escapes. This helps prevent dangling references. Due to our use of lexically scoped regions, the lifetime check $R \succeq a$ always succeeds.

$$\frac{\langle \varpi, \Pi, R, \eta\rangle[e_1] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[e'_1]}{\langle \varpi, \Pi, R, \eta\rangle[e_1\,;e_2] \hookrightarrow \langle \varpi', \Pi', R', \eta'\rangle[e'_1\,;e_2]}\ \textbf{[D-SEQ1]}$$

$$\frac{}{\langle \varpi, \Pi, R, \eta\rangle[\delta_1\,;e_2] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[e_2]}\ \textbf{[D-SEQ2]}$$

$$\frac{\Pi(v) = \mathit{true}}{\langle \varpi, \Pi, R, \eta\rangle[\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[e_1]}\ \textbf{[D-IF1]}$$

$$\frac{\Pi(v) = \mathit{false}}{\langle \varpi, \Pi, R, \eta\rangle[\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[e_2]}\ \textbf{[D-IF2]}$$

$$\frac{\Pi(v) = (ca)\texttt{null} \quad e\ =\ v.f \mid (v.f = \delta) \mid v.mn\langle a^*\rangle(v'_{1..p})}{\langle \varpi, \Pi, R, \eta\rangle[e] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[\texttt{nullerr}]}\ \textbf{[D-ERR1]}$$

$$\frac{e\ =\ (lhs = e_1) \mid (e_1\,;e_2) \mid \texttt{ret}(v^*, e_1) \mid \texttt{retr}(a, e_1) \quad \langle \varpi, \Pi, R, \eta\rangle[e_1] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[\texttt{nullerr}]}{\langle \varpi, \Pi, R, \eta\rangle[e] \hookrightarrow \langle \varpi, \Pi, R, \eta\rangle[\texttt{nullerr}]}\ \textbf{[D-ERR2]}$$

We say the run-time environment (ϖ, Π, R_2, η) is *consistent* with the type environment $(\Sigma, \Gamma, R_1, \Psi)$, written $\Sigma, \Gamma, R_1, \Psi \vDash \langle \varpi, \Pi, R_2, \eta\rangle$, if the following judgement holds:

$$\frac{\begin{array}{c} dom(\Gamma) = dom(\Pi) \quad dom(\Sigma) = dom(\varpi) \\ \forall v \in dom(\Pi) \cdot P;\Gamma;R_1;\Psi;\Sigma \vdash \Pi(v) : \Gamma(v) \\ \forall \ell \in dom(\varpi) \cdot P;\Gamma;R_1;\Psi;\Sigma \vdash \varpi(\ell) : \Sigma(\ell) \\ R_1 = R_2 \quad \eta \Rightarrow \Psi \end{array}}{\Sigma,\Gamma,R_1,\Psi \vDash \langle \varpi,\Pi,R_2,\eta\rangle}$$

THEOREM 1 (SUBJECT REDUCTION). *If*

$$\begin{array}{c} P;\Gamma;R;\Psi;\Sigma \vdash e : t, \\ \Sigma,\Gamma,R,\Psi \vDash \langle \varpi,\Pi,R,\eta\rangle \\ \langle \Pi,\varpi,R,\eta\rangle[e] \hookrightarrow \langle \Pi',\varpi',R',\eta'\rangle[e'] \end{array}$$

then for some $\Sigma' \supseteq \Sigma$*, there exist* Γ',Ψ'*, such that*

$$\begin{array}{l} \Gamma' - \mathit{diff}(e',e) = \Gamma - \mathit{diff}(e,e') \\ P;\Gamma';R';\Psi';\Sigma' \vdash e' : t' \\ P;R';\Psi' \vdash t' <: t \\ \Sigma',\Gamma',R',\Psi' \vDash \langle \varpi',\Pi',R',\eta'\rangle. \end{array}$$

Function $\mathit{diff}(e,e')$ returns a list of local variables that appear in e but not e':

$$\begin{array}{lllll} \mathit{diff}(e,e') =_{df} & let & lst & = & local(e) \\ & & lst1 & = & local(e') \\ & & n & = & length(lst) - length(lst1) \\ & in & (take(n,lst) \lhd n \geq 0 \rhd []) & & \\ take(n,lst) =_{df} & ([] \lhd n \leq 0 \rhd [hd(lst)] +\!\!+ take(n-1,tl(lst))) & & & \end{array}$$

Function $local(e)$ returns a list of sets of local variables. It is defined as follows:

$$\begin{array}{lll} local(e) =_{df} \ case\ e\ of & & \\ \quad \texttt{ret}(v^*,e) & \rightarrow & local(e) +\!\!+ [\{v^*\}] \\ \quad \texttt{retr}(r,e) \mid \{(t\ v)^*\ e\} & & \\ \quad \mid v.f = e \mid v = e \mid e\,;e_1 & \rightarrow & local(e) \\ \quad \textbf{letreg}\ r\ \textbf{in}\ eb & \rightarrow & local(eb) \\ \quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & & \\ \quad \mid cn\langle a^*\rangle\langle \delta'^*\rangle \mid \delta \mid v \mid v.f & & \\ \quad \mid \textbf{new}\ cn(v^*) \mid mn(v^*) \mid v.mn(v^*) & \rightarrow & \emptyset \end{array}$$

Note that $\Gamma - [\,] =_{df} \Gamma,\ \Gamma - ([s] +\!\!+ S) =_{df} (\Gamma - s) - S$.

Proof: By structural induction on e.

- $v \mid v.f \mid \delta\,;e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2$

  $R' = R$ follows from the dynamic semantics. Let $\Sigma' = \Sigma, \Gamma' = \Gamma, \Psi' = \Psi$. The consistency relation is trivial as both static environment and runtime environment remain unchanged. The type judgment follows from the static semantics.

- $v = \delta \mid v.f = \delta$

  Note $R' = R$. Let $\Sigma' = \Sigma, \Gamma' = \Gamma, \Psi' = \Psi$. For the first case, the update on Π preserves the consistency relation due to the subtype relation between rhs and lhs. For the second case, the update on ϖ also preserves the consistency relation. The type judgement is trivial as the type remains as **void** before and after the evaluation.

- **new** $cn\langle r_{1..n}\rangle(v_{1..p})$

  Note $R' = R$. Let $\Sigma' = \Sigma + \{\ell : cn\langle r_{1..n}\rangle\}, \Gamma' = \Gamma, , \Psi' = \Psi$. $dom(\varpi') = dom(\varpi) \cup \{\ell\} = dom(\Sigma) \cup \{\ell\} = dom(\Sigma')$, and $\varpi'(\ell)$ is of type $\Sigma'(\ell)$. The consistency relation follows immediately. The type judgement is trivial as $e' = \ell$, which is of type $cn\langle r_{1..n}\rangle$.

- $\texttt{ret}(v^*,\delta)$

  Note $R' = R$. Let $\Sigma' = \Sigma, \Psi' = \Psi$, and $\Gamma' = \Gamma \backslash \{v^*\}$ is in accordance with the update $\Pi' = \Pi \backslash \{v^*\}$. The consistency follows directly. The type judgement is trivial from the static semantics as δ does not refer to v^*.

- $\texttt{retr}(r,\delta)$

  Note that $R' = R - \{r\}$. Let $\Psi' = \Psi \backslash r, \Sigma' = \Sigma, \Gamma' = \Gamma$. As $\eta' = \eta \backslash r$, the consistency is straightforward. The type judgement is trivial.

- $lhs = e_1 \mid e_1\,;e_2 \mid \texttt{ret}(v^*,e_1) \mid \texttt{retr}(r,e_1)$

  From respective static and dynamic rules, $P;\Gamma;R;\Psi;\Sigma \vdash e_1 : t$, for some t, and $\langle \Pi,\varpi,R,\eta\rangle[e_1] \hookrightarrow \langle \Pi',\varpi',R',\eta'\rangle[e_1']$. By induction hypothesis, $\Sigma',\Gamma',R',\Psi' \vDash \langle \varpi',\Pi',R',\eta'\rangle$, $P;\Gamma';R';\Psi';\Sigma' \vdash e_1' : t'$, and $P;R';\Psi' \vdash t' <: t$. The type judgement for e' is concluded by static semantics again.

- $v_0'.mn\langle a'^*\rangle(v_{1..p}')$

  According to evaluation rule ([**D-IM**]), the actual type of $\ell = \Pi(v_0')$ is $cn\langle a_{1..n}\rangle$, which is a subclass of the expected type of v_0', say $cn'\langle a_{1..m}\rangle$. Furthermore, $\eta \Rightarrow inv.cn\langle a_{1..n}\rangle$. Also, according to static semantics, $\Psi \Rightarrow pre.cn'.mn\langle a_{1..m}, a'^*\rangle$. By the soundness of method overriding, $inv.cn\langle a_{1..n}\rangle \wedge pre.cn'.mn\langle a_{1..m}, a'^*\rangle \Rightarrow [r'^* \mapsto a'^*]\varphi$. Hence $\eta \Rightarrow [r'^* \mapsto a'^*]\varphi$, since $\eta \Rightarrow \Psi$. Note that $R' = R$. Taking Ψ' to be $\Psi \wedge inv.cn\langle a_{1..n}\rangle$, and $\Gamma' = \Gamma + \{this : \Gamma(v_0'), (v : t)_{1..p})\}$ and $\Sigma' = \Sigma$, it follows that e' is well-typed. Furthermore, consistency relation is preserved as the runtime variable environment increases in accordance with the type environment and $\eta' \Rightarrow \Psi'$, while the store and its typing are unchanged. Hence both the type judgement and the consistency relation hold.

- $mn\langle a'^*\rangle(v_{1..p}')$

  Similar to the instance method invocation, except that *this* is not included in Γ', and dynamic dispatch issue need not be considered.

- **letreg** r **in** eb

  After evaluation, $R' = R \cup \{r\}$. Let $\Sigma' = \Sigma, \Gamma' = \Gamma$, and $\Psi' = (\Psi \wedge (R \succ r))$, which coincides with $\eta' = (\eta \wedge (R \succ r))$. The consistency follows directly. The type judgement is obvious from static semantics.

- $\{(t\ v)_{1..p}\ e\}$

  R' remains as R. Let $\Sigma' = \Sigma, \Gamma' = \Gamma \cup \{(v : t)_{1..p}\}, \Psi' = \Psi$. As runtime variable environment changes accordingly, the consistency is preserved. The type judgement is straightforward. □

Before we give the no dangling theorem, we define the concepts of *live location* and *no-dangling*.

DEFINITION 1 (LIVE LOCATION). *A location* ℓ *from* $dom(\varpi)$ *is* live *wrt a set of live regions R, if* $curreg(\ell) \in R$*. Note that* $curreg(\delta) =_{df} a_1$*, if* $\delta = (cn\langle a_{1..n}\rangle)\texttt{null}$ *or* $\varpi(\delta) = cn\langle a_{1..n}\rangle\langle \delta'^*\rangle$.

DEFINITION 2 (NO-DANGLING).

1. *The runtime variable environment* Π *is* no-dangling *wrt R if for all* $v \in dom(\Pi)$*,* $\Pi(v)$ *is either a primitive value, or a live location* ℓ *wrt R.*

2. *The runtime store* ϖ *is* no-dangling *wrt R if for each location* $\ell \in dom(\varpi)$ *that is live wrt R, each field of the object is either a primitive or is live wrt R.*

3. *An expression e is* no-dangling *wrt R if all locations in e (i.e., loc(e)) are live wrt R.*

The following theorem states that no dangling pointers can be generated during program execution. That is, no local references can be dangling, no live objects can contain dangling references, and each runtime code does not hold dangling references.

THEOREM 2 (NO DANGLING). *If*

$$P;\Gamma;R;\Psi;\Sigma \vdash e : t$$
$$\Sigma,\Gamma,R,\Psi \vDash \langle \varpi,\Pi,R,\eta\rangle$$
$$\langle \varpi,\Pi,R,\eta\rangle[e] \hookrightarrow \langle \varpi',\Pi',R',\eta'\rangle[e']$$
$$\Pi,\varpi, e \text{ are no-dangling wrt } R.$$

Then Π',ϖ' *and* e' *are no-dangling wrt* R'.

Proof: By structural induction on e.

- $v \mid v.f \mid \delta\,;e_2 \mid \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2$

  The conclusion is obvious as the variable environment and the store remain unchanged, while no additional locations are introduced in e'.

- $v = \delta \mid v.f = \delta$

  Suppose $P;\Gamma;R;\Psi \vdash lhs : cn\langle a_{1..n}\rangle$. From region subtyping, we have $curreg(\delta) \succeq a_1$. As both Π and ϖ are no-dangling, $a_1 \in R$, thus $curreg(\delta) \in R$. This ensures that the evaluation does not produce a reference from a dead region, thus preserves the no-dangling property for both the variable environment and the store. $e' = ()$ is obviously no-dangling.

- **new** $cn\langle r_{1..n}\rangle(v_{1..p})$

  During the evaluation, the variable environment remains unchanged, existing live locations are still live, therefore, Π' is no-dangling. A new live location is introduced, thus e' is no-dangling. As all the variables $v_{1..p}$ are either primitive or live locations, the newly added object is also live wrt R. It therefore preserves the no-dangling property of the store.

- $\texttt{ret}(v^*,\delta)$

  During the evaluation, some local variables are popped from the variable environment, the residual environment Π' is still no-dangling. The resulting store is no-dangling since there is no update at all on the store. No-dangling property of e' is directly from that of e.

- $\texttt{retr}(r,\delta)$

  Note that R' is decreased from R by r. From the type rule, we know that r does not appear in δ, thus $e' = \delta$ is no-dangling wrt to R'. From the type rule, we also know that r does not escape to Π via the consistency relation with Γ. Also, r is the youngest region. By the invariant property of objects, none of the live locations will reference this region. Hence, both Π' and ϖ' are no-dangling wrt to a smaller R'.

- $lhs = e_1 \mid e_1\,;e_2 \mid \texttt{ret}(v^*,e_1) \mid \texttt{retr}(r,e_1)$

  From respective static and dynamic rules, $P;\Gamma;R;\Psi;\Sigma \vdash e_1 : t$, for some t, and $\langle \Pi,\varpi,R,\eta\rangle[e_1] \hookrightarrow \langle \Pi',\varpi',R',\eta'\rangle[e_1']$. By induction hypothesis, we have ϖ', Π' and e_1' are no-dangling wrt R'. No-dangling property of e' directly follows.

- $v_0'.mn\langle a'^*\rangle(v_{1..p}')$

  The variable environment is increased without violating the no-dangling property since $v_{0..p}'$ from a no-dangling Π point to only live locations which are passed to *this*, $v_{1..p}$ respectively. The store remains unchanged, thus preserves no-dangling property. The resulting expression is also no-dangling as it does not hold any locations directly.

- $mn\langle a'^*\rangle(v_{1..p}')$

  Similar to the instance method invocation.

- **letreg** r **in** eb

  Although R' is enlarged, both the variable environment and the store remain unchanged during the one-step evaluation, thus preserves the no-dangling property. e' preserves the no-dangling property since no location is newly introduced in it.

- $\{(t\,v)_{1..p}\ e\}$

  At the declaration site, each variable is assigned an initial value according to its type, the no-dangling property is thus preserved. The store remains unchanged. The no-dangling property for e' follows immediately from that for e. □

Before stating the progress theorem, we define the transitive closure of our small step reduction as follows:

$$\hookrightarrow^0 = Id \qquad \hookrightarrow^{n+1} = \hookrightarrow^n \circ \hookrightarrow \qquad \hookrightarrow^* = \bigcup_{n\geq 0} \hookrightarrow^n$$

THEOREM 3 (PROGRESS). *1. If*

$$P;\Gamma;R;\Psi;\Sigma \vdash e : t$$
$$\Sigma,\Gamma,R,\Psi \vDash \langle \varpi,\Pi,R,\eta\rangle$$

where ϖ,Π,e *are no-dangling wrt R, then either e is a value, or*

$$\langle \varpi,\Pi,R,\eta\rangle[e] \hookrightarrow \langle \varpi,\Pi,R,\eta\rangle[\texttt{nullerr}]$$

or there exist ϖ',Π',R',η',e', *such that*

$$\langle \varpi,\Pi,R,\eta\rangle[e] \hookrightarrow \langle \varpi',\Pi',R',\eta'\rangle[e'].$$

2. Given a well-formed program $P = def^*\ meth^*\ main()\{eb\}$, *where eb is a closed term without free regions (except heap) nor free variables. Starting from the initial runtime environment* $\langle\{\},\{\},\{heap\},true\rangle$, *either there exist* ϖ,Π,R,η, *such that*

$$\langle\{\},\{\},\{heap\},true\rangle[eb] \hookrightarrow^* \langle \varpi,\Pi,R,\eta\rangle[\texttt{nullerr}]$$

or there exist ϖ,Π,R,η, *and a value* δ, *such that*

$$\langle\{\},\{\},\{heap\},true\rangle[eb] \hookrightarrow^* \langle \varpi,\Pi,R,\eta\rangle[\delta]$$

or for all ϖ,Π,R,η,e, *that*

$$\langle\{\},\{\},\{heap\},true\rangle[eb] \hookrightarrow^* \langle \varpi,\Pi,R,\eta\rangle[e]$$

there exist ϖ',Π',R',η',e', *such that*

$$\langle \varpi,\Pi,R,\eta\rangle[e] \hookrightarrow \langle \varpi',\Pi',R',\eta'\rangle[e'].$$

Proof: (1): By structural induction on e.

- $\ell \mid k \mid (ca)\texttt{null}$

  e is a value.

- $v \mid (v = \delta) \mid \textbf{new } cn\langle r_{1..n}\rangle(v_{1..p}) \mid \texttt{ret}(v^*,\delta)$ $\mid \texttt{retr}(r,\delta) \mid \delta\,;e_2$

  By dynamic semantics, e yields e' through one step evaluation.

- $v.f \mid v.f = \delta$

  Notice that v cannot be dangling. If $\Pi(v) = (ca)\texttt{null}$, the evaluation of e reports `nullerr`. Otherwise, the former yields a value δ, while the latter yields a constant () in one step.

- $lhs = e_1 \mid e_1; e_2 \mid \texttt{ret}(v^*, e_1) \mid \texttt{retr}(r, e_1)$

  By hypothesis, the evaluation of e_1 either reports `nullerr`, or yields e_1' by one step. In the former case, the evaluation of e reports `nullerr`, while in the latter case, it yields a corresponding expression in one step as given in the dynamic semantics.

- $v_0'.mn\langle a'^* \rangle (v_{1..p}')$

  Note that v_0' cannot be dangling. The evaluation of e reports `nullerr` if $\Pi(v_0) = (ca)\texttt{null}$; otherwise, it yields $\texttt{ret}((this, v_{1..p}), [r'^* \mapsto a'^*]eb)$ in one step as indicated in the dynamic rule.

- $mn\langle a'^* \rangle (v_{1..p}') \mid \textbf{letreg}\ r\ \textbf{in}\ eb \mid \{(t\,v)_{1..p}\ \ e\}$
  $\mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2$

  By dynamic semantics, each e yields its corresponding e' in one step.

(2): As the initial store, stack, and *eb* are no-dangling wrt $\{heap\}$, the proof follows from an inductive application of (1). □

The above theorems state the safety properties of our region type system, which have been proved by induction. In previous effect-based region type system, Tofte and Talpin [37] and Christiansen and Velschow [17] made use of co-induction to prove the soundness. Later, Calcagno [11] used stratified operational semantics to avoid co-induction in the proof of safety properties of a simple region calculus, while Helsen [30] introduced a special constant for defunct regions in their big-step semantics which made the soundness proof simpler. These two new proofs looked more elegant and simpler than the ones using co-induction. However, they only focused on simple functional languages.

5. 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 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.

5.1 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 [15]) by erasing all region-related notations. That is, if $\vdash P'$, then $\vdash_N \textbf{erase}(P')$. 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.

5.1.1 Class Declarations

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

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

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 $split(fds, cn_2)$ 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*.

5.1.2 Method Declarations

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

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

This rule can be used for both instance and static methods, with $\Gamma = \{this : cn\langle r^+ \rangle\}$ 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, $\tau_0\langle r_0^* \rangle$.

5.1.3 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 highlight 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**.

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

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

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

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

$$\frac{}{\Gamma \vdash k \Rrightarrow k : \mathit{prim}_k\langle\rangle, \mathit{true}}\ [\textbf{EC2}] \qquad \frac{\Gamma \vdash e_i \Rrightarrow e_i' : t_i, \varphi_i \quad i = 1,2}{\Gamma \vdash e_1\,;e_2 \Rrightarrow e_1'\,;e_2' : t_2, \varphi_1 \wedge \varphi_2}\ [\textbf{ES}]$$

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

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

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

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

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

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

$$\frac{def_2 = \textbf{class}\ cn_2\ \textbf{extends}\ cn_1\ \{fd^*\ (meth_i)_{1..q}\} \quad \mathit{split}(fd^*, cn_2) = ([(\tau_i\ f_i)]_{i=1}^{p}, [(cn_2\ f_i)]_{i=p+1}^{n}) \quad \tau_i{\neq}cn_2 \quad i{=}1..p \quad \vdash \tau_i \Rrightarrow \tau_i\langle r_i^*\rangle, \varphi_i \quad i{=}1..p \quad \varphi_0 = (r \succeq r_1 \lhd p{<}n \rhd \mathit{true}) \quad ca_2 = (cn_2\langle r_{1..l}, r_{1..p}^*, r\rangle \lhd p{<}n \rhd cn_2\langle r_{1..l}, r_{1..p}^*\rangle) \quad \Gamma = \{\mathit{this} : ca_2\} \quad \vdash cn_1 \Rrightarrow cn_1\langle r_{1..l}\rangle, \varphi_1 \quad \Gamma \vdash meth_i \Rrightarrow meth_i'; Q_i,\ i{=}1..q \quad Q' = \{\mathit{inv}.ca_2 = (\varphi_1 \wedge \bigwedge_{i=1}^{p} \varphi_i \wedge \bigwedge_{i=1}^{p} (r_i^* \succeq r_1) \wedge \varphi_0)\} \quad def_2' = \textbf{class}\ ca_2\ \textbf{extends}\ cn_1\langle r_{1..l}\rangle\ \textbf{where}\ \varphi\ \{(\tau_i\langle r_i^*\rangle f_i)_{1..p},\ (cn_2\langle r, r_{2..l}, r_{1..p}^*, r\rangle f_i)_{p+1..n},\ (meth_i')_{1..q}\}}{\vdash def_2 \Rrightarrow def_2'; Q' \cup \bigcup_{i=1}^{q} Q_i}\ [\textbf{CD}]$$

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

$$\frac{P = def_{1..n}\ meth_{1..m} \quad \vdash_{def} def_i \Rrightarrow def_i'; Q_i\ \ i \in 1..n \quad \{\} \vdash meth_i \Rrightarrow meth_i'; Q_i'\ \ i \in 1..m \quad Q'' = \bigcup_{i=1}^{n} Q_i \cup \bigcup_{i=1}^{m} Q_i' \quad P' = def_{1..n}'\ meth_{1..m}'\ Q''}{\vdash P \Rrightarrow P'}\ [\textbf{PROG}]$$

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

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

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

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

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

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

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

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

Figure 3: Region Inference Rules

5.2 Examples

We next illustrate our region inference rules via some examples.

5.2.1 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.

The result type of this block is `Pair⟨r2,r2a,r2b⟩`, with the constraints `r4a`$\succeq$`r4`$\wedge$`r4b`$\succeq$`r4`, `r3a`$\succeq$`r3`$\wedge$`r3b`$\succeq$`r3`$\wedge$`r4`$\succeq$`r3a`, `r2a`$\succeq$`r2`$\wedge$`r2b`$\succeq$`r2`$\wedge$`r4`$\succeq$`r2b` , `r1a`$\succeq$`r1`$\wedge$`r1b`$\succeq$`r1`$\wedge$`r2`$\succeq$`r1a`, and `r3`$\succeq$`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).

5.2.2 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`$\succeq$`r1b`$\wedge$`r1b`$\succeq$`r1`$\wedge$`r1`$\succeq$`r2a`$\wedge$ `r2a`$\succeq$`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`$\succeq$`r1`$\wedge$`r1a`$\succeq$`r1`. As all declared regions escape from this expression block, the [EB] rule does not introduce any localised regions here.

5.2.3 Recursive Methods

We shall illustrate how region annotations and constraints are inferred 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.

5.3 Correctness

This subsection is devoted to the correctness of our region inference, which is given in terms of soundness and completeness.

We first give two lemmas that are useful in the proof of the correctness theorem.

LEMMA 4. *Suppose* $P;\Gamma;R;\Phi \vdash e : t$.

(a). If $\Phi' \Rightarrow \Phi$, *then* $P;\Gamma;R;\Phi' \vdash e : t$.

(b). If $R' \supseteq R$, *then* $P;\Gamma;R';\Phi \vdash e : t$.

(c). If $\rho = [(r_i \mapsto a_i)_{1..p}]$, *and for all* $i = 1,..,p$, *either* $a_i \notin R$ *or* $\Psi \Rightarrow (r_i = a_i)$, *then* $P;\rho\Gamma;\rho R;\rho\Phi \vdash \rho e : \rho t$.

Proof: By structural induction on e. □

LEMMA 5. *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

$$\Gamma \vdash meth \Rrightarrow meth';Q$$

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

$$\vdash def \Rrightarrow def';Q$$

and all classes and static methods that def' *depends are well-formed in* P', *then*

$$\vdash_{def} def'$$

Proof: (a) By structural induction on e.

- $k \mid (ca)\texttt{null} \mid v \mid v.f$

  Straightforward.

- $lhs = e_1$

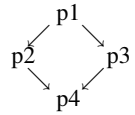
  Note that $e' = (lhs = e_1')$. From inference rule [EA], and induction hypothesis, there exist $R_1 \supseteq reg(\Gamma) \cup reg(t')$ and $R_2 \supseteq reg(\Gamma) \cup reg(t)$, such that $P';\Gamma;R_1;\varphi' \vdash e_1' : t'$, and $P';\Gamma;R_2;true \vdash lhs : t$. Let $R = R_1 \cup R_2$, then $P';R;\varphi \vdash t' <: t$. It concludes directly from lemma 4 and the assignment type rule.

- **new** $cn(v^*)$

  By inference rule [EN], we know $e' =$ **new** $cn\langle x^+\rangle(v^*)$, $t = cn\langle x^+\rangle$, $\varphi = \wedge_{i:0..p}\varphi_i$. The judgement follows directly from the type checking rule for object creation.

- $eb = \{(\tau_j\ v_j)_{j:1..p}\ \ e\}$

  From inference rule [EB], we have
  $\Gamma \cup \{v_j : \tau_j\langle x_j^*\rangle\}_{j:1..p} \vdash e \Rrightarrow e' : \tau\langle r^*\rangle, \varphi$.
  By induction hypothesis, we have
  $P';\Gamma \cup \{v_j : \tau_j\langle x_j^*\rangle\}_{j:1..p};R_1;\varphi \vdash e' : \tau\langle r^*\rangle$
  where $R_1 \supseteq reg(\Gamma) \cup \{r^*\} \cup \cup_{i:1..p}\{x_i^*\}$.
  From the type checking rule for expression block,
  $P';\Gamma;R_1;\varphi \vdash \{(\tau_j\langle x_j^*\rangle\ v_j)_{j:1..p}\ e'\} : \tau\langle r^*\rangle$.
  By lemma 4(c), we have



(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

$P'; \Gamma; \rho'\rho R_1; \rho'\rho\varphi \vdash \rho'\rho\{(\tau_j\langle x_j^*\rangle\, v_j)_{j:1..p}\, e'\} : \rho\tau\langle r^*\rangle$.
Notice that $\rho'\rho\Gamma = \Gamma$, $\rho'\rho\tau\langle r^*\rangle = \rho\tau\langle r^*\rangle$. By region checking rule for region declaration, we have
$P'; \Gamma; R; \varphi' \vdash$ **letreg** r **in** $\rho'\rho\{(\tau_j\langle x_j^*\rangle\, v_j)_{j:1..p}\, e'\} : \rho\tau\langle r^*\rangle$.
Note that $R = \rho'\rho R_1 - \{r\}$, $\rho'(\rho\varphi\backslash rs) = (\rho'\rho\varphi)\backslash\{r\}$. It then follows directly from the type checking rule for expression block and lemma 4(a).

- $v_1'.mn(v_{2..p}')$

  It follows directly from inference rule [EMI], lemma 4(c) and type checking rule for instance method invocation.

  The case of static method invocation $mn(v_{1..p}')$ is similar.

- e_1 **;** e_2 | **if** v **then** e_1 **else** e_2

  By induction hypothesis, $P'; \Gamma; R_i; \varphi_i \vdash e_i' : t_i$, for $i = 1, 2$. Let $R = R_1 \cup R_2$, it follows directly from lemma 4(a) and (b).

(b) From inference rule [MD],
$\Gamma,\ (v_j : \tau_j\langle r_j^*\rangle)_{j:2..p} \vdash eb \Rrightarrow eb' : \tau_0'\langle x_0^*\rangle,\ \varphi$.
Then from the above (1),
$P'; \Gamma \cup \{(v_j : \tau_j\langle r_j^*\rangle)_{j:2..p}\}; R; \Psi \vdash eb' : \tau_0'\langle x_0^*\rangle$.
From the assumption, all the parameter types and result types are well-formed. The judgement follows directly from the type checking rule for well-formedness of methods.

(c) From inference rule [CD], $\Gamma \vdash m_j \Rrightarrow m_j'; Q_j$, j=1..q. By the above (b), all these instance methods are well-formed. By assumption, all fields are well-formed. Then the judgement follows directly from type checking rule for well-formedness of class definitions. □

THEOREM 6 (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 P' is well-formed, i.e., $\vdash P'$.

Proof sketch: (a). Applying inference rules according to the order given by the global dependency graph, we can always obtain a region-annotated program P' (with region constraints) for any (well-normal-typed) source program P. Moreover, final region constraints in P' are always satisfiable due to the following facts: (1) we only use outlives relation($\succeq$). The only case in which strictly outlives relation ($\succ$) is used is the local region introduction, where our inference rule makes sure that the localised region does not escape. (2) During the inference process, we may encounter unsatisfiable constraints due to method overriding, but they are amended to be satisfiable by the override checks resolution.

(b). Based on the order in which region inference is proceeded, we can re-organise the declarations in $P = def^*\ meth^*$ as the following:

$$\{def_1^*,\ meth_1^*\},\ ...,\ \{def_p^*,\ meth_p^*\}$$

Each $\{def_i^*,\ meth_i^*\}$ forms a strongly connected component (SCC) in the global dependency graph. By applying theorem 5 to these SCCs in accordance with the above order, we get the well-formedness for all the definitions and well-typedness of the body of the program. Mutual dependency does not pose a problem as we can provide type signature (including constraint abstraction) ahead of time before the inference is applied simultaneously to each class and method in the SCC. Then we conclude from the type checking rule for the whole program. Notice that our override conflict resolution can only strengthen the constraint, which preserves the well-typedness due to lemma 4(a). □

6. 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.

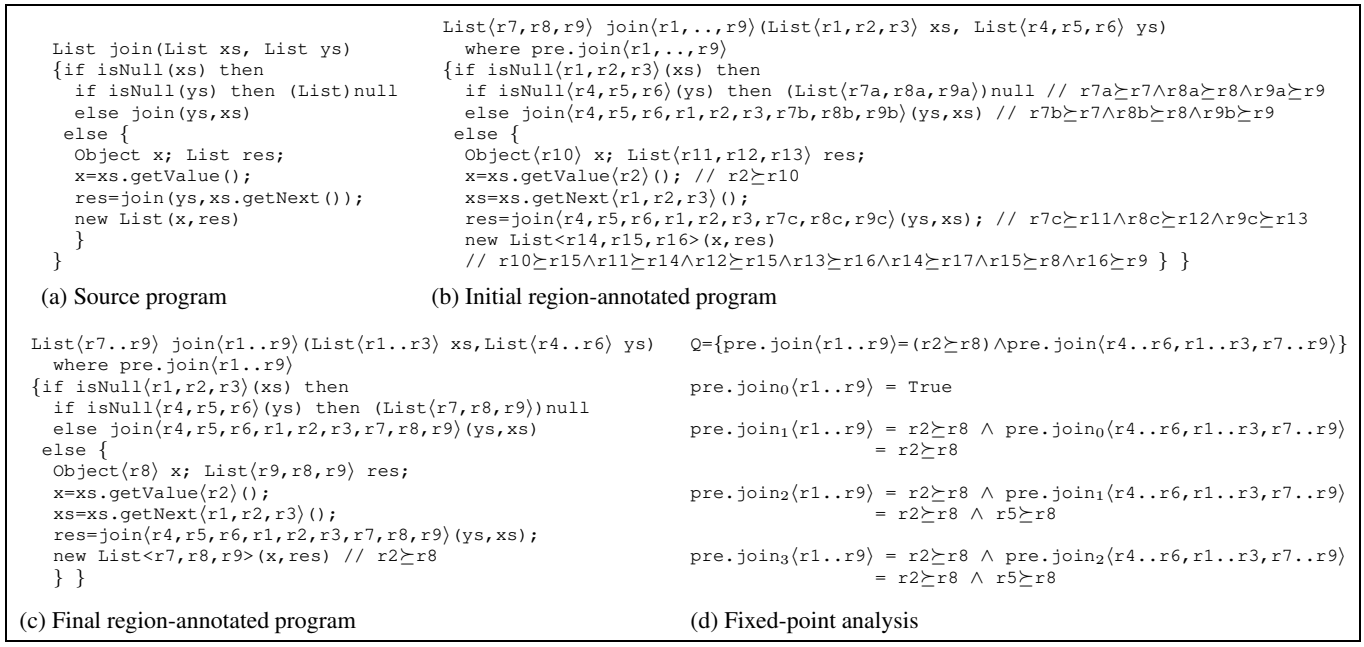


Figure 6: Region Inference for Recursive Method

For this purpose, we propose a new 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, while the last two dependencies are induced by method overriding check of the form:

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

These dependencies can be systematically gathered in the following rule:

$$\frac{P = def_{1..n}\ meth_{1..m}\ eb \qquad \vdash Overrides(P),\ O \\ \{\} \vdash meth_i \rightsquigarrow M_i,\ i = 1..m \qquad \vdash def_i \rightsquigarrow D_i,\ i = 1..n}{\vdash P \rightsquigarrow D_{1..n} \cup M_{1..m} \cup conv(O)}$$

Note that $\vdash Overrides(P),\ O$ gathers a set of method override checks. This relation is defined below. The function $conv(O)$ converts each override check from O into its two dependencies, namely $\{cn'.mn \rightarrow cn.mn,\ cn \rightarrow cn.mn\}$.

6.1 Inferring Override Checks

We propose to gather all method override checks by traversing each class declaration in search of every pair of methods that override.

Given a program P, we can gather all such checks using

$$\vdash Overrides(P), M$$

where M is a mapping from each class to its set of method override checks. The inference rules for this task are given below.

$$\frac{P = def_{1..n}\ meth^*\ eb \\ P \vdash Overrides(def_i), m_i,\ i = 1,..,n}{\vdash Overrides(P), \bigcup_{i:1..n} m_i}$$

$$\frac{def = \textbf{class}\ cn\ extends\ cn'...\{field^*\ meth_{1..p}\} \\ P, cn, cn' \vdash Overrides(meth_i),\ C_i,\ i = 1,..,p}{P \vdash Overrides(def), \{cn \mapsto \bigcup_{i:1..p} C_i\}}$$

$$\frac{meth = \tau_0\ mn((\tau_i\ v_i)_{i:1..q})... \\ meth' = \tau_0\ mn((\tau_i\ v_i)_{i:1..q})... \\ C = if\ (P \vdash meth' \in cn')\ then\ \{(inv.cn, pre.cn'.mn, pre.cn.mn)\}\ else\ \{\}}{P, cn, cn' \vdash Overrides(meth), C}$$

Notice that for each class, we check each method to see if it overrides a corresponding method in one of its super-classes. If so, we gather the method override check. We search each class declaration in P, until all such method override checks are collected.

6.2 Inferring Dependencies

The following rules gather the dependencies for each class and method declaration.

$$\frac{def = \textbf{class}\ cn\ \textbf{extends}\ cn'\ \{(\tau_i\ f_i)_{i:1..n}\ meth_{1..m}\} \\ \{this : cn\} \vdash meth_i \rightsquigarrow M_i,\ i = 1..m \\ C = \{(cn \rightarrow \tau_i) \mid i \in \{1..n\} \wedge isClassType(\tau_i)\} \\ C' = \{(cn.mn \rightarrow cn) \mid i \in \{1..n\} \wedge mn = name(meth_i)\}}{\vdash def,\ M_{1..m} \cup \{cn \rightarrow cn'\} \cup C \cup C'}$$

$$\frac{m = if\ (this : cn) \in \Gamma\ then\ cn.mn\ else\ mn \\ \Gamma + \{(v_i : \tau_i)_{i:1..n}\} \vdash eb \rightsquigarrow C, M \\ C' = C \cup \{\tau_i \mid i \in \{0..n\} \wedge isClassType(\tau_i)\}}{\Gamma \vdash \tau_0\ mn((\tau_i\ v_i)_{i:1..n})\ eb \rightsquigarrow \{m \rightarrow p \mid p \in C' \cup M\}}$$

For expressions, we attempt to gather all types and methods that are being used. Given an expression e and a type environment Γ, we gather a set of types used C, and a set of methods invoked M, as follows:

$$\Gamma \vdash e \rightsquigarrow C, M$$

The syntax-directed inference rules for expressions are detailed below.

$$\frac{se = k \mid v \mid v.f}{\Gamma \vdash se \rightsquigarrow \{\}, \{\}} \qquad \frac{lhs = v \mid v.f \qquad \Gamma \vdash e \rightsquigarrow C, M}{\Gamma \vdash lhs = e \rightsquigarrow C, M}$$

$$\frac{se = (cn)\ \mathbf{null} \mid \mathbf{new}\ cn(v^*)}{\Gamma \vdash se \rightsquigarrow \{cn\}, \{\}}$$

$$\frac{\Gamma + \{(v_i : \tau_i)_{1..n}\} \vdash e \rightsquigarrow C, M \qquad C' = C \cup \{\tau_i \mid i \in \{1..n\} \wedge isClassType(\tau_i)\}}{\Gamma \vdash \{(\tau_i\ v_i)_{i:1..n}\ e\} \rightsquigarrow C', M}$$

$$\frac{(v_0 : cn) \in \Gamma \qquad (cn', mn)) \in cn}{\Gamma \vdash v_0.mn(v_1, .., v_n) \rightsquigarrow \{\}, \{cn'.mn\}} \qquad \Gamma \vdash mn(v_1, ..v_n) \rightsquigarrow \{\}, \{mn\}$$

$$\frac{\Gamma \vdash e_i \rightsquigarrow C_i, M_i,\ i = 1, 2}{\Gamma \vdash e_1\ ;\ e_2 \rightsquigarrow C_1 \cup C_2, M_1 \cup M_2}$$

$$\frac{\Gamma \vdash e_i \rightsquigarrow C_i, M_i,\ i = 1, 2}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow C_1 \cup C_2, M_1 \cup M_2}$$

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.

7. 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:

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

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:

(a). 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.

(b). 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 $c \equiv r_1 \geq r_2 \mid r_1 = r_2$. 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.

8. HANDLING DOWNCAST

One important feature that is missing from our Core-Java is the *downcast* operation. In general, this operation may be type unsafe if the object in question is not the subtype that was expected. 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 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. 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`.

```
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

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 = 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 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 subsequently. 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. For each such object, we pad its region-type with a sufficient number of extra regions to support downcast operations later.

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 the following.

```
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
```

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.

8.1 Downcast Flow Analysis

We introduce a backward flow analysis which attempts to analyse downcast operations that may be subsequently performed for the objects of a given program. In the interest of simplicity, our analysis is flow-insensitive and context-insensitive. Moreover it is a global analysis for which each program point is represented uniquely.

The parameters `v*` of each static method `mn` will be represented by `mn.v*`, while its result is represented by `mn` itself. For each instance method of class `cn`, we represent its parameters using `cn.mn.v*` and `cn.mn.this`, while the result is denoted by `cn.mn`.

Every `new` statement will be labelled with a unique program point using `l:new cn(..)`, and similarly for each block `l:{(t v)* e}`. The purpose of labelling the former is to identify its source location, while the label for block is to allow local variables to be uniquely renamed. Specifically, we rename `v*` by `l.v*` in `l:{(t v)* e}` to make all variable declarations unique.

We shall now introduce a new inference rule to gather a set of backward flows. The purpose of this flow analysis is to identify the set of classes to which the object at a program point may be downcasted later. The inference rule is expressed as the relation:

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

Note that x is a *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. For convenience, we shall omit Γ in the rules and assume that it has been properly maintained with the set of live variables and their types at each scope.

Each backward flow is represented using either $v_1 \dashrightarrow v_2$ or $v_1 -\!\!\!-_D\!\!\!\rightarrow v_2$, 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, namely v_2, may be subjected to a downcast-to-D operation.

The rule for downcast operation itself is defined as follows:

$$x \vdash (D)\, v, \{x -\!\!\!-_D\!\!\!\rightarrow v\}$$

The value of v may flow into its outer receiver x and be subjected to a downcast operation. This is captured by the backward flow $x -\!\!\!-_D\!\!\!\rightarrow v$.

The next few rules show the backward flow of variables, object fields and primitives.

$$\frac{C = \mathit{if\ isPrimType}(v)\ \mathit{then}\ \{\}\ \mathit{else}\ \{x \dashrightarrow v\}}{x \vdash v, C} \qquad x \vdash (cn)\texttt{null}, \{\}$$

$$\frac{C = \mathit{if\ isPrimType(v.f)\ then\ \{\}\ else\ \{x \dashrightarrow v.f\}}}{x \vdash v.f, C} \qquad x \vdash k, \{\}$$

Take note that we only capture the flows of object values and ignore those from primitive values, including `null`, as no regions are involved for the latter.

The next few rules are due to various expressions. Note how both branches of a conditional flow into its outer receiver. For expression block, we make the local variables unique with the help of label *l*.

$$\frac{\mathit{lhs} = v \mid v.f \quad \mathit{lhs} \vdash e, C}{x \vdash \mathit{lhs} = e, C} \qquad \frac{x \vdash e_1, C_1 \quad x \vdash e_2, C_2}{x \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2, C_1 \cup C_2}$$

$$\frac{\mathit{fresh}\ w \quad w \vdash e_1, C_1 \quad x \vdash e_2, C_2}{x \vdash e_1; e_2, C_1 \cup C_2} \qquad \frac{x \vdash [v \mapsto l.v]^* e, C}{x \vdash l : \{(\tau\ v)^*\ e\}, C}$$

In the case of `new c(v1..p)` constructor, we denote its created object using *l*. The object itself flows into *x*, while $v_{1..p}$ will flow into the respective fields of the *l* object.

$$\frac{C = \{l.f_i \dashrightarrow v_i \mid i \in 1..p \ \wedge\ \neg\ \mathit{isPrimType}(v_i)\} \cup \{x \dashrightarrow l\}}{x \vdash l : \textbf{new}\ c(v^*), C}$$

In the case of static method invocation, the result of the method will flow into outer receiver *x*, while the arguments w^* may flow in either directions (covariant or contravariant), depending on how the parameters $m.v^*$ are used. Value flows also occur within each of the methods. To gather this set of backward flows, we introduce the relation: $\vdash\ \mathit{meth}, R; W; C$, where *R* and *W* denote the sets of parameters are being used as source (covariant via read) and destination (contravariant via write) respectively. Some parameters are used as both read and write, and they will appear twice. Others may not appear in either set, if they are of primitive types or have not been used in the method body.

$$\frac{\vdash \tau_0\ mn((\tau\ v)^*)\ eb; rset; wset; _}{x \vdash mn(w_{1..n}), link(w_{1..n}, rset, wset) \cup \{x \dashrightarrow mn\}}$$

$$\frac{mn \vdash [v \mapsto mn.v]^*\ eb, C}{\vdash \tau_0\ mn((\tau\ v)^*)\ eb, readSet(C, mn.v^*); writeSet(C, mn.v^*); C}$$

Two auxiliary functions, that are used to identify the directions of parameter flow, are defined as below:

$$readSet(C, v_{1..n}) = \{(v_i, i) \mid i \in 1..n \wedge (w \dashrightarrow v_i | v_i.f) \in C\}$$

$$writeSet(C, v_{1..n}) = \{(v_i, i) \mid i \in 1..n \wedge (v_i.f \dashrightarrow w) \in C\}$$

Another function, called *link*, attempts to connect up the arguments $w_{1..n}$ with the two parameter sets, R and W, for reading and writing.

$$link(w_{1..n}, R, W) = \{v \dashrightarrow w_i \mid (v, i) \in R\} \cup \{w_i \dashrightarrow v \mid (v, i) \in W\}$$

For instance method, the parameter list also includes the current object, named *cn.mn.this*. The two rules for instance method invocation and method declaration are given below. Due to method overriding and inheritance, each method invocation may have to deal with a set of methods at compile-time. To handle this, we provide bidirectional flows for each of the non-primitive parameters, as shown in the first rule.

$$\frac{(w_1 : cn) \in \Gamma \qquad C = \{cn.mn.v_i \dashrightarrow w_i, w_i \dashrightarrow cn.mn.v_i, \mid i \in 1..p \wedge \neg \mathit{isPrimType}(w_i)\}}{x \vdash w_1.mn(w_{2..n}), C \cup \{cn.mn.this \dashrightarrow w_1, w_1 \dashrightarrow cn.mn.this, x \dashrightarrow cn.mn\}}$$

$$\frac{cn.mn \vdash [(v \mapsto cn.mn.v)^*,\ this \mapsto cn.mn.this]\ eb, C \quad V = [cn.mn.this, cn.mn.v^*]}{cn \vdash \tau_0\ mn((\tau\ v)^*)\ eb, readSet(C, V); writeSet(C, V); C}$$

The backward flows for the entire program is captured by the next two rules.

$$\frac{\vdash meth_i, C_i \quad i = 1, .., n \quad \vdash def_i, C'_i \quad i = 1, .., m \qquad \mathit{fresh}\ w \quad w \vdash eb, C \quad C'' = \bigcup_{1..n} C_i \cup \bigcup_{1..m} C'_i \cup C}{\vdash def_{1..m}\ meth_{1..n}\ eb,\ C''}$$

$$\begin{array}{r} I = \{(meth, cn'') \mid cn'' \in \mathit{supclass}(cn) \wedge meth \in_D cn'' \wedge meth \in cn\} \\ CI = \{\mathit{link}(cn.mn.v^*, R, W) \cup \{cn.mn \dashrightarrow cn''.mn\} \mid \\ \qquad (meth,\ cn'') \in I \wedge cn'' \vdash meth, R, W, _\} \\ O = \{(meth_i, cn'') \mid i \in 1..m \wedge cn'' \in \mathit{subclass}(cn) \wedge \\ \qquad \exists meth'_i \in_D cn'' \cdot \textbf{name}(meth_i) = \textbf{name}(meth'_i)\} \\ CO = \{\mathit{link}(cn.mn.v^*, R, W) \cup \{cn.mn \dashrightarrow cn''.mn\} \mid \\ \qquad (meth,\ cn'') \in O \wedge cn'' \vdash meth, R, W, _\} \\ cn \vdash meth_i, _, _, C_i, \quad i = 1, .., m \end{array}$$
$$\vdash \textbf{class}\ cn\ \textbf{extends}\ cn'\ \{(\tau\ f)^*,\ meth_{1..m}\}, \bigcup_{1..m} C_i \ \cup\ CI \ \cup\ CO$$

The flow analysis for classes is complicated by both method inheritance and method overriding. The set *CI* captures the flows that are induced by inheritance for each method that is declared in a superclass and inherited in the present class. Note that $\in_D$ captures direct membership within a given class, while $\in$ denotes membership with inheritance. The flow set *CO* captures the flows that might occur for all overridden methods that exist in the subclasses of *cn*. The linking is based on the read/write flow set for each instance method. Notice that $\mathit{supclass}(cn)$ returns all superclasses of *cn*, while $\mathit{subclass}(cn)$ returns all subclasses of *cn*.

8.2 Transitive Closure of Downcast Flows

After the entire set of backward flows have been generated, we can proceed to perform a transitive closure to gather all program points 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 program point (usually variable or object field).

For each variable, *v*, or object field, *v.f*, we associate a set of casts, *D*, as follows: *v*[*D*] or *v.f*[*D*]. Take note that labels *l* are also permitted in place of *v*. These sets are initially empty, with the first elements obtained by converting each arrow with downcast, as follows:

$$v \text{—}D\text{→} w \ \wedge\ w[S] \ \Rightarrow\ v \dashrightarrow w \ \wedge\ w[S \cup \{D\}]$$

After that, we perform a *closure of the backward flow* with the following propagation rules:

$$v \dashrightarrow w \ \wedge\ w \dashrightarrow x \ \Rightarrow\ v \dashrightarrow x$$
$$v \dashrightarrow w.f \ \wedge\ w \dashrightarrow x \ \wedge\ \mathit{var}(x) \ \Rightarrow\ v \dashrightarrow x.f$$

In the second rule, $\mathit{var}(x)$ states that *x* must not be an object field. This is to avoid accumulating fields of the form $v.f_1.\cdots.f_n$ in our set of program points. Once this closure is completed, we can proceed to perform the *closure of downcast set analysis* using the following rule.

$$v[S_1] \ \wedge\ w[S_2] \ \wedge\ v \dashrightarrow w \ \wedge\ S_1 \not\subseteq S_2 \ \Rightarrow\ w[S_1 \cup S_2]$$

This rule is repeatedly applied until a fix-point is reached. Both these closure operations are expected to terminate. Backward flow

analysis is expected to terminate since there is a finite number of variables and their fields. Similarly, closure of downcast is expected to terminate as there is a finite number of classes for each program.

Consider the earlier program fragment in Fig. 7. The gathered set of flows is:

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

This set is initially converted to:

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

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 us tell when 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 needed when downcast succeeds.

9. 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[34]; 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 [17] 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 [17]. 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 our program to run on a region-based custom allocator, called Titanium[32], 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 region subtyping.

To check the scalability of our region inference, we manually converted a set of ten programs from the Olden benckmark[12] to Core-Java, and measured their inference times, as shown in Fig 9.

10. RELATED WORK

Tofte and Talpin [36, 37] 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 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 [17]. 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 its class hierarchy, including those from its subclasses. 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[25, 26, 31, 38]. 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 [28].

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 [36, 37, 20, 28, 17] and ownership types [19, 18, 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. Moreover, we infer region types automatically across procedures, whilst they have limited support through intra-procedure inference and the use of defaults for region types.

Deters and Cytron [22] 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.

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

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

In research performed concurrently with ours, Cherem and Rugina have developed a region inference algorithm for Java [14]. 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.

11. 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. [24]), 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[21] 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.

12. 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] C. Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *ACM POPL*, London, January 2001.

[12] 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.

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

[14] S. Cherem and R. Rugina. Region Analysis and Transformation for Java Programs. Technical report, Computer Science Department Cornell University, April 2004.

[15] 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⟩.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

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

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

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

[27] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Sondergaard. Boolean constraints for binding-time analysis. In *Programs as Data Objects II*, pages 39–62, Aarhus, Denmark, May 2001.

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

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

[30] S. Helsen. *Region-Based Program Specialization*. PhD thesis, Universität Freiburg, 2002.

[31] 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.

[32] 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.

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

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

[35] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

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

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

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

Appendix

13. PHANTOM REGIONS

Phantom regions pose a number of problems for region inference. For example, if we have to patch up the `Object` class with the two extra regions `r2,r3` from `Pair`, we have several more issues to consider, including:

- Is the outlive relation, namely $\texttt{r2} \succeq \texttt{r1} \wedge \texttt{r3} \succeq \texttt{r1}$, required on the phantom regions for the `Object` class?
- What specific regions should be used for the `Object` fields that are found in other classes, such as the `Pair` class itself?
- How should the extra regions from the other sub-classes of `Object` be handled? Must the phantom regions be propagated mutually across the sub-classes, via their common super-class, as done in [17]?

The last requirement greatly increases the number of regions needed. In addition, the total required regions for each class is only known after all the classes have been defined, requiring a close-world assumption for region compilation.

Apart from these issues, phantom regions may also cause a loss in lifetime precision. This may sound surprising but a closer look at an example will reveal why. Assume we have to add two phantom regions (from the `Pair` sub-class) to `Object`, as follows:

```
class Object⟨r1,r2,r3⟩ where ...
class Pair⟨r1,r2,r3⟩ extends Object⟨r1,r2,r3⟩
   where r2≥r1 ∧ r3≥r1 {
 Object⟨r2,r2,r3⟩ fst
 Object⟨r3,r2,r3⟩ snd
 ...
}
```

This inclusion of phantom regions forces all connected `Pair` objects to have an identical region for their `fst` field, and another identical region for `snd`. In the case of the example in Fig 4, it forces objects `p2,p4` to be in the same region, and similarly for objects `p3,p4`. A consequence is that `p2,p3,p4` are now in one region, while `p1` is in another. This is undesirable as the `p3` object cannot be freed now, as it is in the same region as `p2` and `p4`, even though `p3` is already dead after this code fragment. Thus, phantom regions can add *extra* region constraints that cause loss in lifetime precision.

14. OTHER JAVA FEATURES

We shall now look at how some other features of Java may be handled in our approach to region inference.

14.1 Statements

Java supports both statements and expressions, though some overlap is present in the two syntactic categories. Our philosophy has been to keep Core-Java minimalist without loss of expressiveness. We shall show that statements can be dealt as syntactic sugar. The syntax of statement is typically given as:

$$s ::= e \mid s_1 \,;\, s_2 \mid \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \mid \textbf{return}\, e \mid \{(\tau\ v)^*\ s\}$$

Due to the optional return of values, statements can be regarded as expressions of the *Maybe* type that either returns a value via $\mathbf{OK}(e)$, or nothing via **None**. The following rules show that all statements can be translated into expressions of Core-Java with the help of an option type.

$$\vdash e \twoheadrightarrow e\,;\mathit{None} \qquad \vdash \textbf{return}\, e \twoheadrightarrow \mathit{OK}\, e$$

$$\frac{\mathit{fresh}\ v \quad \vdash s_1 \twoheadrightarrow e_1 \quad \vdash s_2 \twoheadrightarrow e_2}{\vdash s_1\,;s_2 \twoheadrightarrow \{\tau\ v\,;v = e_1\,;\textbf{if}\ \mathit{isNone}(v)\ \textbf{then}\, e_2\ \textbf{else}\ v\}}$$

$$\frac{\vdash s_1 \twoheadrightarrow e_1 \quad \vdash s_2 \twoheadrightarrow e_2}{\vdash \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \twoheadrightarrow \textbf{if}\, e\, \textbf{then}\, e_1\, \textbf{else}\, e_2} \qquad \frac{\vdash s \twoheadrightarrow e}{\vdash \{(\tau\ v)^*\ s\} \twoheadrightarrow \{(\tau\ v)^*\ e\}}$$

14.2 Static Fields

Static class fields may also be added to Core-Java. As they must persist throughout the entire program execution, objects created here must be placed into either the heap or immortal memory.

14.3 Exceptions

Exceptions may alter the flow of computation, and may also escape non-lexically through run-time call stack before they are caught by exception handlers. As a result, exception themselves are harder to place into regions. One simple solution is to place them in the heap or immortal memory. Another solution is to use explicit throw-catch mechanism for each method declaration (via a pre-processing phase), in order to track the flow of exception objects. Due to flow insensitivity of region inference, the rest of the objects are unaffected by the exception mechanism.

14.4 Multithreading

Multi-threading in Java is used to support concurrency. Each thread may have its own execution lifetime that is synchronized by access to shared objects. Correspondingly, regions may also be shared amongst a group of threads. The lifetime of regions must outlive the last use within the respective threads. This can be guaranteed via a reference counting mechanism on the shared region, but stack-like behaviour is lost.

14.5 Genericity

At present, each class in Core-Java has monomorphic base type but is region-polymorphic. There have been several recent proposals [10] to add generic types to Java. Such extensions will help minimise on downcast operations, and could help improve the lifetimes of polymorphic regions. To support genericity, we have to expand region types to include polymoprhic region variables that can be instantiated, similar to techniques proposed in [27]