On Reachability and Acyclicity

Yi Lu and John Potter
Programming Language and Compiler Group
School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia
{ylu,potter}@cse.unsw.edu.au

UNSW-CSE-TR-0434
15 Oct 2004

Abstract

This paper presents a type system that enforces constraints on reachability via pointers or references, and restricts reference cycles to be within definable regions. Every data object lives in a fixed region, determined from its type.

The motivation for our work is the desire to enforce static constraints on reference structures. Such constraints can be useful for program reasoning, for deadlock avoidance in concurrent contexts, and for runtime optimizations and memory management. For example, data invariants are easily broken by re-entrant code. By restricting cycles to statically enforceable regions, program proof rules can make stronger assumptions about reference structures.

The contributions of this paper are a novel class-based region-parametric type system, with subtyping, that enforces one-way reachability between regions, and a dynamic semantics that allows us to formalize the key structural invariant: object references respect region reachability, so that object cycles occur only within regions.

1 Introduction

1.1 Motivation

References are used in programming to allow sharing of data structures. In most software, references are unavoidable for pragmatic efficiency reasons, even though they complicate program reasoning and their use is error-prone. When the data structures are mutable, problems are inevitable. Reference cycles (direct or indirect) can cause serious programming errors; a sequence of method calls following a reference cycle, may unexpectedly break invariants of local states or cause infinite loops. References also complicate the task of memory management; for instance, safety of explicit memory deallocation may be difficult in the presence of arbitrary cycles and automatic garbage collection cannot rely on reference counting alone in the presence of cycles.

Reference problems are difficult to reason about because in most cases call-backs and infinite loops may be caused by indirect references which a programmer may be unaware of. These problems are especially difficult to address statically because the reference structure is evolving dynamically. As a result, existing type systems are weak in their ability to express where cycles are allowed and where forbidden.

Reference structures are particularly interesting in object-oriented systems, because most object-oriented languages use reference semantics for objects which, together with subtyping and a generic coding style, increase the likelihood of unintended object reference cycles.

Moreover, cycles also cause trouble in common object-oriented design patterns, like the decorator or wrapper. Wrapping an object with self-cycles yields a design problem: should the self-cycles be re-routed via the wrapper or not? When is it safe to wrap an object?

1.2 Acyclic Types

Problems with references and aliasing have been well documented especially in object-oriented programming [22], and various principles espoused for providing alias protection [27]. Object level encapsulation has been a popular approach for alleviating the approach. For example, ownership types [28, 11, 10, 7] clearly define a notion of *inside* and *outside* for objects and maintain a containment invariant. Objects may only access inside via the owner, whereas they may directly access anything outside which is owned by one of its ancestors in the ownership tree. This containment invariant restricts direct inward access, but places no constraints at all on the indirect

reachability of objects: any object may potentially reach any other object. In particular reference cycles may occur anywhere.

This affects our ability to use class invariants to reason about the state of an object before and after method calls. If a method indirectly calls back on an object, via an indirect reference cycle, say, then the call-back may be entering the object in an invalid state, so the call-back code may be working outside of its assumed precondition, and furthermore the original call on the object may not be aware of the indirect effect of the call-back, so its desired postcondition may not be met. With reference cycles we cannot presume that an invariant will hold for all calls on an object. This problem manifests itself in a language like Eiffel which allows runtime assertion checking for class invariants—when should the invariant hold? The problem has been highlighted in recent work on verification of OO programs [3] which suggests enriching the program state to track when object invariants hold; they incorporate a notion of dynamic ownership.

Our approach is to capture the potential of an object to reach other objects, directly or indirectly, in their types. In typed languages, the occurrence of a cycle in the runtime object graph implies there must be a cycle in the type dependency graph; conversely, if there are no cycles in the type dependency graph, then there will be none in the runtime object graph. In any object-oriented programming language, cyclic type dependencies are permitted for user-defined types, because the data values modeled by these types are object references. If there are type-level cycles, the type system is powerless to prevent cyclic references, even if they are undesirable. For example, in the Java Collections library there is a bulk method addAll(Collection) that adds all the elements of the argument to those of the target collection. In one of the implementing classes ArrayList the documentation advises that the behavior is undefined if the list is added to itself, and leaves it up to the programmer to avoid the situation. With our proposal we can insist that such calls do not type-check correctly.

Our goal is to provide an expressive type system that will allow programmers to record their assumptions about acyclicity through constraints on reachability between objects. One-way reachability means that there is a path from one object to another that is not part of a cycle in the object graph. This *acyclic reachability* is the key concept in our model: it is a strict partial order which we denote by $\triangleright$.

The motivation for the design of acyclic types is the well-known factorization of a directed graph into a partially ordered partition of strongly connected components. All cycles within the original graph must occur within these components. We have designed our type system so that regions and

their reachability relation provide a static abstraction of the strongly connected components of the dynamic object graph and their partial order.

The key idea is to formulate static constraints on the object graph by specifying regions within which all reference cycles are trapped. Every object lives in the same region for its lifetime. We impose constraints on region reachability, and guarantee that inter-object references respect these inter-regional reachability constraints.

To use such a model, programmers must be able to decide when they want to ban the possibility of cycles or aliases emanating from particular object fields. This will be clearer when we look at a small language and examples in the next sections. For now we just give an example illustrating a simple use of acyclic types.

```
class A<p>
  r from p;
  B<r> f;
  ...
    // assert a property about this object's fields
    f.m();
    // assert the same property
```

This code shows the simplest case of acyclic types. Class `A` is parameterized with a region parameter. A region `r` is defined with the class with the constraint `r from p`, which means `p` may reach `r`, or `p` $\triangleright$ `r`. The type of the field `f` is `B<r>` which means `f` references an object living in the region `r`. In this case, the object referenced by `f` can never hold a reference to the `this` object because of the order we force on their types. So any method call made on `f` can not re-enter `this` object. Such knowledge allows us to assert properties that are necessarily invariant during the call (for example, the reference stored in the field `f`). More examples to illustrate the expressiveness of the acyclic types will follow.

Acyclic types can significantly improve program understanding. Programmers are able to specify via types that cycles are allowed between two objects by placing them into the same region, or disallowed by placing them in distinct regions. Programmers know that two references are not direct aliases if their objects live in different regions.

1.3 Outline of the Paper

This paper is organized as follows: ATL, a small programming language with acyclic types is presented in the next section. Section 3 gives some program

$c \in$ ClassName; $p \in$ RegionParameter; $r \in$ RegionName;
$x \in$ VarName; $f \in$ FieldName; $m \in$ MethodName

P	$\in$	Program	::=	$\overline{cls}\ e$
cls	$\in$	Class	::=	$\texttt{class}\ c\langle\overline{\rho}\rangle\ \lceil\texttt{extends}\ t\rfloor\ \{\overline{\varsigma}\ \overline{fd}\ \overline{mth}\}$
ρ	$\in$	ClassConstr	::=	$\overline{\kappa} \triangleright p \triangleright \overline{\kappa}$
ς	$\in$	RegionDefn	::=	$\overline{\alpha} \triangleright r \triangleright \overline{\alpha}$
fd	$\in$	Field	::=	$t\ f$
mth	$\in$	Method	::=	$t\ m(\overline{t}\ \overline{x})\{e\}$
e	$\in$	Expression	::=	$x \mid \texttt{new}\ t \mid \texttt{null} \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid$
				$e; e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
α	$\in$	Region	::=	$p \mid r \mid \texttt{base}$
κ	$\in$	RegionExpr	::=	$p \mid t.r \mid \texttt{base}$
t	$\in$	Type	::=	$c\langle\overline{\kappa}\rangle$

Table 1: Abstract Syntax of ATL

examples and Section 4 gives an overview of its static semantics. Section 5 presents a dynamic semantics of ATL and gives proofs for some important properties of the type system. Related work and discussion are given in Section 6 and Section 7 along with some thoughts on future directions. Section 8 concludes the paper.

2 A Programming Language with Acyclic Types: ATL

In this section, we present an informal overview of a class-based programming language featuring acyclic types with subtyping, calling it the *Acyclic Types Language*, or ATL for short. The syntax of the language is given in Table 1; program examples and formal semantics will be given in the following sections.

To simplify the abstract syntax, we use a few abbreviations. The overbar is used for a sequence of constructs; for example, $\overline{cls}$ is used for a possibly empty sequence cls_1 ... cls_n, as are $\overline{\rho}$, $\overline{\varsigma}$, $\overline{fd}$, $\overline{mth}$, $\overline{\kappa}$ and $\overline{e}$. Similarly, $\overline{t}\ \overline{x}$ stands for a possibly empty sequence of pairs $t_1\ x_1$ ... $t_n\ x_n$. In the class production, $\lceil\texttt{extends}\ t\rfloor$ is an optional part of the class. Just as in Java, `this` is a variable name used to reference the target object for the current

call.

In the concrete syntax we use for our examples, the region definition $\overline{\alpha} \triangleright r \triangleright \overline{\alpha'}$ is written as:

$$r \texttt{ from } \alpha_1, ..., \alpha_m \texttt{ to } \alpha'_1, ..., \alpha'_n$$

where $|\overline{\alpha}| = m$ and $|\overline{\alpha'}| = n$. The same applies to the class constraint $\overline{\kappa} \triangleright r \triangleright \overline{\kappa'}$. The relational symbol $\triangleright$ denotes the *acyclic reachability* relation between regions in the abstract syntax.

Classes and Class Constraints. Our syntax is close to Java except that classes are parameterized with region parameters and regions are defined within classes like fields. Formal region parameters can be used within a class to identify regions for objects used by the class. The first formal parameter of a class is bound to the actual region where `this` object lives.

Class constraints restrict the object reference structure. The region parameters of a class are required to satisfy the reachability constraints specified in the `from` and `to` clause of each parameter.

To ensure the acyclic property for the region reachability relation, the order of introduction of new region parameters is important. A class constraint ρ introduces a fresh name for its region parameter, together with constraints that specify the required reachability properties of any actual region bound to this parameter. These constraints are expressed in terms of previously defined regions κ, including region expressions formed from earlier parameters, from the special global region `base` and from type-qualified regions. Furthermore these constraints must not impose any further requirements on the reachability relation for the previously defined regions; in this way we are able to inhibit cycles in the region reachability relation. This is checked and enforced by the type system as formalized in Section 4.

Types. Classes are type schemas. Types are formed by binding the region parameters to actual regions in the environment where the type is formed. A type consists of a class name and the region arguments required by the class definition. The first region argument is the region where the object of `this` type reside. Unsurprisingly, for a type to be valid, the actual region arguments must satisfy the class constraints defined on the formal parameters.

Regions and Region Definitions. In ATL programs, every object belongs to a fixed region for its entire lifetime. All regions except the special region `base` are defined within a class. Just as for class constraints, a region definition ς introduces a fresh region name into the scope of the current class with constraints to locate the new region strictly between existing regions.

Unlike the region parameters of a class constraint, these region names define actual regions. Acyclicity is maintained by region definitions, in the same way as by class constraints—the location bounds for the new region cannot introduce any further restrictions on the existing regions.

Compared with class constraints, which just impose requirements on the actual region arguments for a class, region definitions actually determine the region reachability relation for the system. Every class defines a type schema, with its own locally defined regions; we allow types to qualify region names—every such qualified region $t.r$ determines a particular region. If t and t' have a common ancestor whose class introduces r, then $t.r$ denotes the same region as $t'.r$; otherwise the two are distinct regions. Classes inherit all the regions of their ancestors, and may not further constrain the inherited reachability relations; however classes may extend their inherited reachability relations with new region definitions. In order to guarantee that newly defined regions can never form cycles, we exclude qualified regions $t.r$ from the bounds α of the region definition ς. The reason for this will be illustrated by an example in the next section.

The primary goal of ATL is to allow programmers to define a static preorder on the runtime object structure, thereby restricting where cycles can occur. By making class-defined region names publicly accessible we keep flexibility in that every ground region has a global name. Ground regions are those with no free region parameters: either `base`, or a ground type qualified region $\tau.r$. A ground type is one with no free region parameters. Objects of the same types (*not just the same class*) share the same region; objects of different type may occupy the same region, but their types must share the same first region argument (because this determines the region the object lives in).

Interestingly, regions and types are recursively defined: regions are defined within classes and named via type qualification; types are formed by binding classes with actual regions. Because of this recursive structure there are an unbounded number of ground regions, all globally accessible via region path expressions. Fortunately the programmer does not need to deal with global region names, because the region parameters of a class localize the expression of the regions relevant for a class, so normally region expressions do not need to be nested through more than one level of type qualification.

It may be noticed that the syntax of region expressions requires that all regions including locally defined regions to be referred by a qualified type (this certainly does not apply to the formal parameters and `base`). This would help the type system reason about region identities in the presence of subtypes. However for the convenience reason, in the actual language we

allow programmers to write local regions unqualified as they are named in their definitions, because the type checker can easily annotate these local regions with their qualifying type, which is the type of the object `this`.

The special region `base` is the only unqualified ground region. An important note here is that regions do *not* have to be reachable from `base` or vice-versa; also any root objects do *not* have to be placed in `base`. `base` is just the base region used to name any other regions. The ability to name a region is different from the ability to access a region, which is determined by the `to` and `from` clauses of the region definitions.

All region constraints are statically verified by the type system to ensure global acyclicity. Any strongly connected components in the object heap during the execution of an ATL program must occur entirely within a region. Regions need not exist at runtime for ATL because the expression language only uses types for object allocation, and regions can be erased from that with no change in behaviour. Regions are only used in type checking to help organize and reason about cyclic references and object reachability.

Fields and Methods. In our language, the structural invariant is maintained by imposing a stronger restriction on object fields than on other reference-valued entities, namely method arguments and results. Object fields are singled out for special attention because they can form unwanted hard-to-detect cycles in object graphs. ATL's structural invariant states that if an object contains a reference to another object then both objects must be in the same region or the region of the former object must be able to reach the region of the latter object with no return reference possible.

However, we do allow method parameters to access objects which are not accessible to `this` object. So within the scope of a method, `this` is not necessarily the root object. This implies that we still allow inter-region callbacks, but that they occur within method scope, and the method's type signature indicates whether any such call-back is allowed or not.

Default Types for Legacy Code. Regions can be circumvented when they are not needed. ATL is compatible with existing programming patterns; it is easy to integrate existing library classes that do not use region parameters. This can be achieved using default types. A compiler can automatically annotate classes with a single region parameter and types declared in them with that region parameter. In the main routine, `base` (or any other ground region) can be used as the default region for types. Region-based code that needs access to the library classes simply needs to declare that the `base` region is reachable.

3 Examples

In this section, we first give some toy examples to illustrate the use of regions, then a recursive linked list data structure with an iterator, and finally show how regions can capture fixed application-specific ordering properties.

3.1 Toy Examples

We illustrate class constraints and region definitions.

```
class A<p1, p2 from p1, p3>
  a1 from p1 to p2;    // OK
  a2 from p2 to p1;    // BAD cycle p1 to p2 to p1
  a3 from p1 to p3;    // BAD require p1 to p3
  a4 from base;        // OK

class B<p from base>
  b1 from p;  b2 from b1;  b3 to p;
  A<p, b2, p> f1;      // OK
  A<b1, b2, p> f2;     // OK
  A<p, b3, b2> f3;     // BAD require b3 from p
  A<base, p, b1> f4;   // BAD require p to base
  A<p, B<p>.b1, B<b1>.b3> f5; // OK
  A<p, B<p>.b3, p> f6; // BAD require B<p>.b3 from p

class C<
  p1 from base,        // OK
  p2 from B<p1>.b1,    // OK
  p3 from B<p3>.b1,    // BAD p3 undefined
  p3 to B<base>.b3,    // OK
  p4 from p1 to B<base>.b3
>                      // BAD require p1 to B<base>.b3
```

Class `A` shows how regions are defined. The key point in region definitions is that any newly introduced region can not change the relations of any previous defined region. Class `B` shows that a valid type needs to satisfy its class constraints. Class `C` shows various class constraints on region parameters, some with qualified regions.

A qualified region has nested regions and types. The qualified name of a region deep in the reachability relation may be long; fortunately, the long name will only be required in those rare circumstances where direct access

to a deeply buried object is needed. To reduce complexity programmers can choose to work with a flattened region structure and/or restrict access to local regions.

```
class SubjectFactory<here, there>
  Subject<here> s1;                // OK
  Subject<there> s2;               // BAD
  Subject<there> makeSubject()     // OK
    return new Subject<there>();   // OK
```

Dynamic references (method arguments and results) are treated differently to field references. There is no restriction on what regions can be used for dynamic references. Dynamic references are safe because they can only be used in a limited scope (within a method) and at runtime they emanate from the calling stack rather than the heap, which is deallocated at the exit of the method. Unrestricted dynamic references add flexibility. The `SubjectFactory` class models a factory in region `here` for creating `Subject` objects in region `there`. The type system prevents the `SubjectFactory` class from holding field references to objects in `there`, but because dynamic references are not bound by this restriction, new objects can be created in `there` and returned through the `makeSubject` method for clients to use.

```
class M<p>      class N<p>
  m to N<p>.n;    n to M<p>.m;  // BAD syntax

class M'<p1, p2 to N<p1>.n>
  m to p2;      // OK

class K<p>      class J<p1, p2 from p1> extends K<p1>
  k;                j to k;

class L<p>
  l from p;
  K<p> f1;
  J<p, l> f2;
  A<K<p>.k> f3;
  A<J<p>.k> f4;
  ...
    f1 = f2;    // OK subtype
    f3 = f4;    // OK same type
```

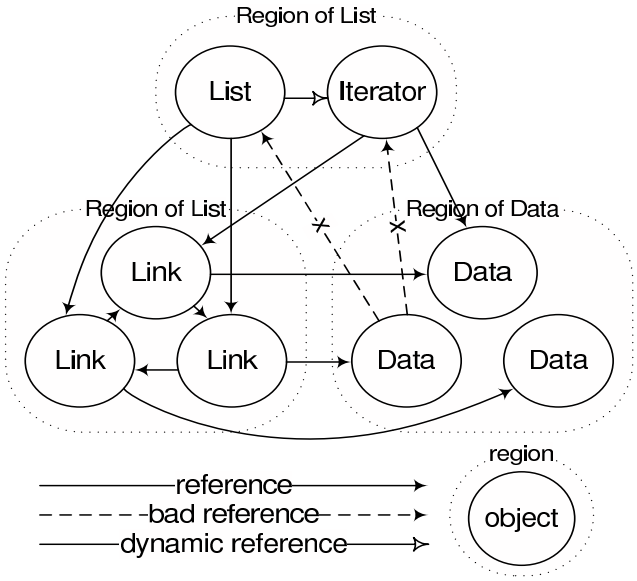

Figure 1: List Example

Classes `M` and `N` show a potential problem with region definitions using qualified regions. In the example, there is a cycle between `M<p>.m` and `N<p>.n` whenever their parameters are bound to the same region.

To solve this problem we forbid qualified regions in region definitions. This does not limit the expressiveness of ATL as class `M'` shows. In `M'` the region `m` is defined to reach `N<p1>.n` through the formal parameter `p2`. So now that `M'<r, r'>.m` acyclic reaches `N<r>.n` can be proved to be safe, because for the type of `M'<r, r'>` to be valid `r'` must be proved to acyclic reach `N<r>.n` by the class constraints of `M'`, which implies that `N<r>.n` must be a valid region defined before `M'<r, r'>`. In this way, the order between `m` in the type `M'<r, r'>` and `N<r>.n` are kept.

Classes `K`, `J`, `L` show the issues with subtyping. In ATL, we allow regions and region definitions to be inherited from ancestor classes. Also regions can be named through types. This means the same region may have different qualified names because of subtyping. Our type system copes with the same region having different qualified names.

3.2 A Linked List Example

Linked lists provide a common example used to demonstrate expressiveness of language features dealing with references. Our list example will show

how regions work and how acyclic properties are expressed in a program. In particular, this example shows how the list data structure is handled in such a way that a list object can never reach itself via its data objects. In other words, the data objects contained by a list should not be able to reach and thereby alter the list, which may cause iterators on the list to fail. The data objects themselves may well be shared by other parts of the program.

```
class Link<here, dr from here>
  Link<here, dr> next;
  Data<dr> data;
  Link(Link<here, dr> next, Data<dr> data)
    this.next = next;
    this.data = data;

class List<here, dr from here>
  kr from here to dr;
  Link<kr, dr> head;
  Link<kr, dr> tail;
  void addElement(Data<dr> data)
    head = new Link<kr, dr>(head, data);
    if (tail == null) tail = head;
    tail.next = head;
  Iterator<here, dr> getIterator()
    return new Iterator<here, dr>(head);

class Iterator<here, dr from here>
  Link<List<here, dr>.kr, dr> current;
  Iterator(Link<List<here, dr>.kr, dr> link)
    current = link;
  void next()
    current = current.next;
  Data<dr> element()
    return current.data;
```

The `List` object is implemented by a cyclic sequence of `Link` objects. `Iterator`s are created inside the `List` region, and are used to access the data stored in `Link` objects. The relations between the regions in the example are shown in Figure 1. All `Link` objects live in the same region defined in the `List` class, and `List`, `Link` and `Iterator` objects have access to the data objects in another region given by a parameter of their classes.

The type system enforces a lack of cycles between regions. `Data` objects can never reference the `List` object and `Iterator` objects, which are shown in the graph as 'bad references'. As expected, cyclic references are allowed within a region. In the example, `Link` objects form a cycle within their region.

3.3 An Application Specific Ordering

Programmers have control over how to use regions, they can use regions in various situations. Besides using regions to prevent reference cycles, we can encode some application specific ordering in the region structure of a program. The type system then enforces the ordering.

```
Machine<m>
  d from m;
  Display<d> disp;
  init()
    disp = new Display<d>;
  adjust()   // modify the display
    ...

Operator<o, m from o>
  Machine<m> mach;
  Display<Machine<m>.d> disp;
  set(Machine<m> ma)
    mach = ma;
    disp = ma.disp;
  operate()  // do the job, such as adjust the machine
    ...

Factory<f>
  skill1 from f;
  skill2 from f to skill1;
  floor1 from skill1;
  floor2 from skill2;
  Machine<floor1> mach1;
  Machine<floor2> mach2;
  Operator<skill1, floor1> op1;
  Operator<skill2, floor2> op2;
  Operator<skill2, floor1> op3;
```

```
Operator<skill1, floor2> op4; // BAD
```

A factory has a number of operators and machines. Different machines require different level of skills to operate. In this example, two machines are placed in two regions (floors) and four operators occupy two other regions (skill levels).

The region structure of `Operator` requires an operator to have enough skill to work on a machine. The relation between different skills and floors are defined in the `Factory` class—operators with `skill2` can work on machine on any floor while operators with `skill1` can only operate the machines on `floor1`.

4 The Type System

We present an overview of the static semantics of ATL, along with detailed descriptions of some important typing rules. The complete type system can be found in Table 2.

In addition to the type system, we formally define some auxiliary functions to lookup and bind classes, types, region definitions, fields, methods and relations in Table 3. We intentionally move all occurrences of substitution into the auxiliary functions to simplify the typing rules. Region, field and method lookup need to recursively include all inherited constructs in any ancestor classes. The region definition lookup function is left-recursive because the order of region definitions is important for a well-formed environment. Regions introduced in a subclass can be defined in terms of inherited regions; hence region definitions inherited from ancestor classes must be included before the regions of a subclass.

Several other simple functions extract variable names (var), region names ($regvar$) respectively from an environment, or extract the first region argument from a type (reg), which is the actual region objects of this type live in. We also assume that, for a program to be valid, no identifier can be declared more than once within the same scope. That is, no class name can be declared more than once; no field and method name can be declared more than once within the same class; no region name including the name of region parameters can be declared more than once within the same class.

We use a number of syntactical abbreviations. $P; E \vdash_\kappa \overline{\kappa}$ represents a possibly empty sequence of judgements $P; E \vdash_\kappa \kappa_1 \ ... \ P; E \vdash_\kappa \kappa_n$. The same abbreviation is used for judgements involving $\overline{cls}$, $\overline{fd}$, $\overline{mth}$ and $\overline{t}$. Similarly, $P \vdash_\simeq \overline{\kappa} \simeq \overline{\kappa'}$ stands for $P \vdash_\simeq \kappa_1 \simeq \kappa'_1 \ ... \ P \vdash_\simeq \kappa_n \simeq \kappa'_n$ while $P; E \vdash_e \overline{e} : \overline{t}$

stands for $P; E \vdash_e e_1 : t_1$ ... $P; E \vdash_e e_n : t_n$. In the auxiliary definitions we use $[\overline{\kappa}/\overline{r}]$ to abbreviate a sequence of substitutions $[\kappa_1/r_1]$ ... $[\kappa_n/r_n]$.

In our model, each object may only contain references to objects in the same region or regions that its region can reach. The type system guarantees this by superimposing a static region structure on the object graph. It ensures that the region reachability relation forms a partial order on regions, that objects live in one region for their life-time, and that object references respect the region reachability relation. It follows that any reference cycle must occur within a region.

The environment may contain the types of variables, class constraints and region definitions. The order of the elements in the environment is significant.

$$E ::= \epsilon \mid E, x : t \mid E, \overline{\kappa} \rhd \alpha \rhd \overline{\kappa}$$

Because in our language, types and regions are defined recursively, the number of regions and types are infinite. To achieve a strict partial order among all possible regions, we identify two requirements that must be satisfied by the type system. Firstly any reachability defined between two regions needs to be one-way only. Secondly, any extension to the reachability relation should not introduce any more reachability between existing regions, but rather introduce new regions together with their reachability relative to existing ones.

Our type system satisfies these two requirements. To ensure one-way reachability, the reachability relation between any two regions can be defined *once* only. This can be achieved by ordering the region definitions so that the later definitions are expressed in terms of earlier ones. Our region definition syntax $\overline{\alpha} \rhd r \rhd \overline{\alpha}$ requires that a region is defined in terms of previously defined regions, formal parameters and `base`. The definition order is kept because formal parameters have to be bound before the region can be defined in the class, and the `base` region is pre-existing.

To satisfy the second requirement, the type system guarantees that if a new region lies between existing regions, then those regions were already related. For example, if $r \rhd r'$ is already defined, then a new region r'' can be defined via $r \rhd r'' \rhd r'$, but not $r' \rhd r'' \rhd r$ because this would introduce cycles into the region structure. Similarly, if there is no existing reachability between r and r', then $r \rhd r'' \rhd r'$ is invalid too because it implies $r \rhd r'$ which is unspecified originally. This constraint is enforced by the environment rule [ENV-REGION] which guarantees that the new region variable and the relations on it do not violate the consistency of the environment.

Class well-formedness is checked in the [CLASS] rule. Each class defines its

Well-Formed Environment $P \vdash_E E$

$$\frac{}{P \vdash_E \epsilon}\ [\text{ENV-EMPTY}] \qquad \frac{x \notin var(E) \quad P \vdash_E E \quad P; E \vdash_t t}{P \vdash_E E, x : t}\ [\text{ENV-VAR}]$$

$$\frac{\alpha \notin regvar(E) \cup \texttt{base} \quad P \vdash_E E \quad P; E \vdash_\kappa \overline{\kappa} \quad P; E \vdash_\kappa \overline{\kappa'} \quad P; E \vdash_\kappa \kappa \triangleright \kappa' \quad \forall \kappa \in \overline{\kappa}\ \forall \kappa' \in \overline{\kappa'}}{P \vdash_E E, \overline{\kappa} \triangleright \alpha \triangleright \overline{\kappa'}}\ [\text{ENV-REGION}]$$

Well-Formed Program and Class $\vdash_P P, P \vdash_c cls$

$$\frac{P \equiv \overline{cls}\ e \quad P \vdash_c \overline{cls} \quad P; \epsilon \vdash_e e : t}{\vdash_P P}\ [\text{PROGRAM}]$$

$$\frac{P \vdash_E \overline{\rho}, \lceil \mathcal{D}(P,t) \rfloor, \overline{\varsigma} \quad \lceil regvar(\rho_1) = reg(t) \rfloor \quad E \equiv \overline{\rho}, \texttt{this} : c\langle regvar(\overline{\rho}) \rangle \quad \lceil P; E \vdash_t t \rfloor \quad P; E \vdash_f \overline{fd} \quad P; E \vdash_m \overline{mth}}{P \vdash_c \texttt{class}\ c\langle \overline{\rho} \rangle\ \lceil \texttt{extends}\ t \rfloor\ \{\overline{\varsigma}\ \overline{fd}\ \overline{mth}\}}\ [\text{CLASS}]$$

Well-Formed Type $P; E \vdash_t t$

$$\frac{\mathcal{C}(P,t) = \texttt{class}\ c\langle \overline{\rho} \rangle\ ... \quad P; E \vdash_\kappa \overline{\rho}}{P; E \vdash_t t}\ [\text{TYPE}]$$

Subtype Relation $P \vdash_{<:} t <: t'$

$$\frac{P \vdash_\simeq \overline{\kappa} \simeq \overline{\kappa'}}{P \vdash_{<:} c\langle \overline{\kappa} \rangle <: c\langle \overline{\kappa'} \rangle}\ [\text{SUBTYPE-REFL}] \qquad \frac{P \vdash_{<:} t <: t'' \quad P \vdash_{<:} t'' <: t'}{P \vdash_{<:} t <: t'}\ [\text{SUBTYPE-TRANS}]$$

$$\frac{\mathcal{C}(P,t) = ...\ \texttt{extends}\ t''\ ... \quad P \vdash_{<:} t'' <: t'}{P \vdash_{<:} t <: t'}\ [\text{SUBTYPE-EXTEND}]$$

Well-Formed Region $P; E \vdash_\kappa \kappa$

$$\frac{\kappa \in regvar(E)}{P; E \vdash_\kappa \kappa}\ [\text{REGION-ENV}] \qquad \frac{}{P; E \vdash_\kappa \texttt{base}}\ [\text{REGION-BASE}]$$

$$\frac{P; E \vdash_t t \quad t.r \in regvar(\mathcal{R}(P,t))}{P; E \vdash_\kappa t.r}\ [\text{REGION-TYPE}]$$

Region Equivalence $P \vdash_{\simeq} \kappa \simeq \kappa'$

$$\frac{}{P \vdash_{\simeq} \kappa \simeq \kappa}\ [\text{REGEQ-REFL}] \qquad \frac{P \vdash_{<:} t <: t'' \quad P \vdash_{<:} t' <: t'' \quad r \in regvar(\mathcal{D}(P, t''))}{P \vdash_{\simeq} t.r \simeq t'.r}\ [\text{REGEQ-SUBTYPE}]$$

Region Relation $P; E \vdash_{\kappa} \kappa \rhd \kappa'$

$$\frac{\kappa \rhd \kappa' \in E}{P; E \vdash_{\kappa} \kappa \rhd \kappa'}\ [\text{RELATION-ENV}] \qquad \frac{(\kappa \neq \kappa') \quad P; E \vdash_{\kappa} \kappa \rhd \kappa'' \quad P; E \vdash_{\kappa} \kappa'' \rhd \kappa'}{P; E \vdash_{\kappa} \kappa \rhd \kappa'}\ [\text{RELATION-TRANS}]$$

$$\frac{(\kappa \neq \kappa') \quad P; E \vdash_{\kappa} \kappa'' \rhd \kappa''' \quad P \vdash_{\simeq} \kappa \simeq \kappa'' \quad P \vdash_{\simeq} \kappa''' \simeq \kappa'}{P; E \vdash_{\kappa} \kappa \rhd \kappa'}\ [\text{RELATION-REGEQ}]$$

$$\frac{P; E \vdash_t t \quad t.r \rhd \kappa \in \mathcal{R}(P, t)}{P; E \vdash_{\kappa} t.r \rhd \kappa}\ [\text{RELATION-TYPE1}] \qquad \frac{P; E \vdash_t t \quad \kappa \rhd t.r \in \mathcal{R}(P, t)}{P; E \vdash_{\kappa} \kappa \rhd t.r}\ [\text{RELATION-TYPE2}]$$

Well-Formed Field and Method $P; E \vdash_f fd, P; E \vdash_m mth$

$$\frac{E(\texttt{this}) = c\langle \kappa, .. \rangle \quad t \equiv c'\langle \kappa, .. \rangle \quad P; E \vdash_t t}{P; E \vdash_f t\ f}\ [\text{FIELD}] \qquad \frac{E(\texttt{this}) = c\langle \kappa, .. \rangle \quad t \equiv c'\langle \kappa', .. \rangle \quad P; E \vdash_t t \quad P; E \vdash_{\kappa} \kappa \rhd \kappa'}{P; E \vdash_f t\ f}\ [\text{FIELD-REACHABLE}]$$

$$\frac{P; E \vdash_t \overline{t'} \quad P; E, \overline{x} : \overline{t'} \vdash_e e : t}{P; E \vdash_m t\ m(\overline{t'}\ \overline{x})\{\overline{e}\}}\ [\text{METHOD}]$$

Well-Formed Expression $P; E \vdash_e e : t$

$$\frac{P; E \vdash_e e : t' \quad P \vdash_{<:} t' <: t}{P; E \vdash_e e : t}\ [\text{EXPR-SUBSUM}] \qquad \frac{E(x) = t}{P; E \vdash_e x : t}\ [\text{EXPR-VAR}] \qquad \frac{P; E \vdash_t t}{P; E \vdash_e \texttt{new}\ t : t}\ [\text{EXPR-NEW}] \qquad \frac{P; E \vdash_t t}{P; E \vdash_e \texttt{null} : t}\ [\text{EXPR-NULL}]$$

$$\frac{P; E \vdash_e e : t_o \quad (t\ f) \in \mathcal{F}(P, t_o)}{P; E \vdash_e e.f : t}\ [\text{EXPR-FIELD}] \qquad \frac{P; E \vdash_e e : t_o \quad (t\ f) \in \mathcal{F}(P, t_o) \quad P; E \vdash_e e' : t}{P; E \vdash_e e.f = e' : t_o}\ [\text{EXPR-ASSIGN}]$$

$$\frac{P, E \vdash e : t_o \quad \mathcal{M}(P, t_o, m) = (\overline{t_p}\ \_, t_r, \_) \quad P, E \vdash \overline{e'} : \overline{t_p}}{P; E \vdash_e e.m(\overline{e'}) : t_r}\ [\text{EXPR-CALL}]$$

$$\frac{P; E \vdash_e e' : t' \quad P; E \vdash_e e : t}{P; E \vdash_e e'; e : t}\ [\text{EXPR}-\text{SEQ}] \qquad \frac{P; E \vdash_e e : t' \quad P; E \vdash_e e' : t \quad P; E \vdash_e e'' : t}{P; E \vdash_e \texttt{if } e \texttt{ then } e' \texttt{ else } e'' : t}\ [\text{EXPR}-\text{IF}]$$

Table 2: Static Semantics

$$\frac{P(c) = \texttt{class } c\langle\overline{\rho}\rangle \ ... \ \{\overline{\varsigma} \ ...\} \quad t \equiv c\langle\overline{\kappa}\rangle \quad \overline{r} = regvar(\overline{\rho}) \quad \overline{r'} = regvar(\overline{\varsigma})}{\mathcal{C}(P,t) = P(c)[\overline{\kappa}/\overline{r}][t.\overline{r'}/\overline{r'}]}\ [\text{LOOKUP}-\text{CLASS}]$$

$$\frac{P(c) = \texttt{class } c\langle\overline{\rho}\rangle \ ... \ \{\overline{\varsigma} \ ...\} \quad t \equiv c\langle\overline{\kappa}\rangle \quad \overline{r} = regvar(\overline{\rho}) \quad \overline{r'} = regvar(\overline{\varsigma})}{\mathcal{T}(P,t,t') = t'[\overline{\kappa}/\overline{r}][t.\overline{r'}/\overline{r'}]}\ [\text{LOOKUP}-\text{TYPE}]$$

$$\frac{\overline{r} = regvar(\mathcal{D}(P,t))}{\mathcal{R}(P,t) = \mathcal{D}(P,t)[t.\overline{r}/\overline{r}]}\ [\text{LOOKUP}-\text{RELATION}]$$

$$\frac{P(c) = \texttt{class } c\langle\overline{\rho}\rangle \ \lceil\texttt{extends } t'\rfloor\{\overline{\varsigma} \ ...\} \quad t \equiv c\langle\overline{\kappa}\rangle \quad \overline{r} = regvar(\overline{\rho})}{\mathcal{D}(P,t) = \lceil\mathcal{D}(P,t')\rfloor, \ \overline{\varsigma}[\overline{\kappa}/\overline{r}]}\ [\text{LOOKUP}-\text{DEFINITION}]$$

$$\frac{\mathcal{C}(P,t) = ... \ \lceil\texttt{extends } t'\rfloor \ \{... \ \overline{t} \ \overline{f} \ ...\}}{\mathcal{F}(P,t) = \lceil\mathcal{F}(P,t')\rfloor, \ \overline{t} \ \overline{f}}\ [\text{LOOKUP}-\text{FIELD}]$$

$$\frac{\mathcal{C}(P,t) = ... \ \{... \ t_r \ m(\overline{t_p} \ \overline{x})\{e\} \ ...\}}{\mathcal{M}(P,t,m) = (\overline{t_p} \ \overline{x}, t_r, e)}\ [\text{LOOKUP}-\text{METHOD1}]$$

$$\frac{\mathcal{C}(P,t) = ... \ \texttt{extends } t' \ \{... \ \overline{mth} \ ...\} \quad m \ not \ defined \ in \ \overline{mth}}{\mathcal{M}(P,t,m) = \mathcal{M}(P,t',m)}\ [\text{LOOKUP}-\text{METHOD2}]$$

Table 3: Auxiliary Lookup Functions

own environment formed from its class constraints and the type of `this` object. Note that we do not put region definitions into the class environment, because we want local regions to be qualified when used as region expressions. However, because programmers do make mistakes, some constraints may violate others. We need to check the class constraints and region definitions together to ensure that no later relations can violate relations defined earlier, by applying the rule [ENV-REGION] on all class constraints and all region definitions including those defined in ancestor classes if any (by using the auxiliary function $\mathcal{D}$). If the class is extended from a supertype then the supertype needs to be valid in the environment formed from the class constraints. Furthermore, all fields and methods need to be checked for well-formedness.

Once a well-formed environment is established, it is used to infer region reachability for all valid regions. Region definitions define the reachable relation between regions. Reachability is defined to be transitive; irreflexivity and antisymmetry are consequences of our system. The pair of rules [RELATION-TYPE1/2] check the reachability relation between a region κ and a region $t.r$ in the relations defined in type t. The auxiliary function $\mathcal{R}(P, t)$ is used to lookup region relations that are just region definitions qualified by the type of `this` object. Both rules have an implicit condition that we do not show - κ has to be `base` or one of the region arguments of t. Recall that this condition is enforced by the syntax and guarantees the order of region definitions within different types. In any other case, regions are related through transitivity [RELATION-TRANS], or by region equivalence [RELATION-REGEQ] discussed next.

The same region can have multiple names because it can be qualified by many types, either based on the regions defining class or any of its subclasses. To identify a region we need also to identify its qualifying type with [REGEQ] rules. Moreover, because types are recursively defined with regions, we need to identify all the regions to the types before we can decide the subtype relations in rule [SUBTYPE-REFL].

The [TYPE] rule says that for a type to be valid, its region arguments must satisfy its class constraints. We simply place the class constraints into the test after substituting all the regions parameters with the arguments. Because ρ is a syntactical short hand for a region with a set of region relations. We feel that defining a new judgement and typing rules for this kind of syntactical short hand is trivial so that we assume the type checker will break down these relations into pairs before they are checked against the relation rules. If two region parameters are unrelated in a class constraint, then they may be bound to any valid regions; they may even both be bound

$$\frac{H = \emptyset \quad S = \emptyset \quad P \equiv \overline{cls}\ e \qquad H; e \Downarrow_{P,S} v; H'}{H; P \Downarrow_{P,S} v; H'}\ [\text{RED}-\text{PROGRAM}] \qquad \frac{}{H; x \Downarrow_{P,S} S(x); H}\ [\text{RED}-\text{VAR}]$$

$$\frac{\begin{array}{c}\iota \notin dom(H) \quad \mathcal{F}(P,t) = (\_\ \overline{f}) \\ \texttt{this} \notin S \Longrightarrow H' \equiv H, \iota \mapsto \{t, \overline{f} \mapsto \texttt{null}\} \\ H(S(\texttt{this})) = \{\tau, ...\} \Longrightarrow H' \equiv H, \iota \mapsto \{\mathcal{T}(P,\tau,t), \overline{f} \mapsto \texttt{null}\}\end{array}}{H; \texttt{new}\ t \Downarrow_{P,S} \iota; H'}\ [\text{RED}-\text{NEW}]$$

$$\frac{H; e \Downarrow_{P,S} \iota; H'}{H; e.f \Downarrow_{P,S} H(\iota)(f); H'}\ [\text{RED}-\text{FIELD}] \qquad \frac{H; e \Downarrow_{P,S} \iota; H' \quad H'; e' \Downarrow_{P,S} v; H'' \quad H''' \equiv H''[\iota \mapsto H''(\iota)[f \mapsto v]]}{H; e.f = e' \Downarrow_{P,S} \iota; H'''}\ [\text{RED}-\text{ASSIGN}]$$

$$\frac{}{H; \texttt{null} \Downarrow_{P,S} null; H}\ [\text{RED}-\text{NULL}] \qquad \frac{\begin{array}{c}H; e \Downarrow_{P,S} \iota; H' \quad H'; \overline{e'} \Downarrow_{P,S} \overline{v}; H'' \\ H''(\iota) = \{\tau,\ ...\} \quad \mathcal{M}(P,\tau,m) = (\_\ \overline{x}, \_, e_m) \\ S' \equiv \texttt{this} \mapsto \iota, \overline{x} \mapsto \overline{v} \quad H''; e_m \Downarrow_{P,S'} v_m; H'''\end{array}}{H; e.m(\overline{e'}) \Downarrow_{P,S} v_m; H'''}\ [\text{RED}-\text{CALL}]$$

$$\frac{H; e \Downarrow_{P,S} \_; H' \quad H'; e' \Downarrow_{P,S} v; H''}{H; e; e' \Downarrow_{P,S} v; H''}\ [\text{RED}-\text{SEQ}] \qquad \frac{\begin{array}{c}H; e \Downarrow_{P,S} \texttt{null}; H' \Longrightarrow H'; e'' \Downarrow_{P,S} v; H'' \\ H; e \Downarrow_{P,S} \iota; H' \Longrightarrow H'; e' \Downarrow_{P,S} v; H''\end{array}}{H; \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \Downarrow_{P,S} v; H''}\ [\text{RED}-\text{IF}]$$

Table 4: Dynamic Semantics

to the same region.

Another key rule of the type system is the [FIELD] rule where global reachability properties are preserved by placing local restrictions on field references. Fields are static (that is, heap-based) references, so that the field rule needs to ensure that `this` can reference other objects if and only if its region can reach the regions of the other objects. This is an important invariant of our programs. The [CLASS] rule ensures that the first region parameter of a class is not lost through subtyping.

We allow method arguments and return types to freely reference objects in any regions. The [METHOD] rule merely checks if the types of arguments and the method body are correct. Dynamic references are considered safe in the sense that they will not form cycles in the object graph as they are local to a method stack.

ι	$\in$	Location		
ω	$\in$	GroundRegion	::=	$\tau.r \mid \texttt{base}$
τ	$\in$	GroundType	::=	$c\langle\overline{\omega}\rangle$
v	$\in$	Value	::=	$\iota \mid \texttt{null}$
obj	$\in$	Object	=	$\{\tau, \overline{f} \mapsto \overline{v}\}$
H	$\in$	Heap	=	Location $\longrightarrow$ Object
S	$\in$	StackFrame	=	VarName $\longrightarrow$ Value

Table 5: Dynamic Features of ATL

5 Dynamic Semantics and Invariants

First we formalize some static properties about regions for a well-formed program: namely that the region reachability relation is acyclic. Then, after briefly introducing a formal big-step semantics, we state a subject reduction theorem, that, amongst other things, states that heap goodness is preserved through reductions. Finally we characterize the invariants for object references on good heaps for well-formed programs: inter-object references either occur within regions or respect the region reachability relation.

Lemma 1 (Irreflexivity) *If* $\vdash_P P$, $P \vdash_E E$ *and* $P; E \vdash_\kappa \kappa \triangleright \kappa'$ *then* $P \nvdash_\simeq \kappa \simeq \kappa'$.

Proof Outline. Assume $P; E \vdash_\kappa \kappa \triangleright \kappa'$. There is no type rule which infers $P; E \vdash_\kappa \kappa \triangleright \kappa$, so we $\kappa \neq \kappa'$. Assume also $P \vdash_\simeq \kappa \simeq \kappa'$. We need to show a contradiction.

By [REGEQ] rules, κ and κ' must have the form $t.r$ and $t'.r$ respectively, where t and t' have a common supertype that defines r. Any reachability derivation for $t.r$ or $t'.r$ can be replaced by one for $t''.r$. Now later definitions do not let us derive any further reachability properties involving just r or it's qualified variants. So we can prove the contradiction by arguing inductively on the order of definition of the bounds for regions.

Lemma 2 (Asymmetry) *If* $\vdash_P P$, $P \vdash_E E$ *and* $P; E \vdash_\kappa \kappa \triangleright \kappa'$ *then* $P; E \nvdash_\kappa \kappa' \triangleright \kappa$.

Proof. Assume $P; E \vdash_\kappa \kappa \triangleright \kappa'$ and $P; E \vdash_\kappa \kappa' \triangleright \kappa$. By transitivity, we see that $P; E \vdash_\kappa \kappa \triangleright \kappa$ which contradicts Lemma 1.

Theorem 1 (Acyclicity) *If* $\vdash_P P$, $P \vdash_E E$ *and* $P; E \vdash_\kappa \kappa \rhd \kappa'$ *then* $P; E \nvdash_\kappa \kappa' \rhd \kappa$ *and* $P \nvdash_\simeq \kappa \simeq \kappa'$.

Proof. Immediate from Lemmas 1 and 2.

Let us now consider the dynamic semantics. Table 5 formulates some dynamic features of ATL and the dynamic semantics is given in Table 4. Table 6 shows the rules for well-formedness of heap, stack-frame and expressions in the dynamic model.

We incorporate full type information (with regions) in the heap, to make the proof of soundness properties relatively straightforward. Note that none of the reduction behavior depends on this type information; the `new` t reduction only uses the field names of the class; the region bindings are only used to store the type.

Some of the reduction rules are written with a choice within the rule; this simply avoids repeating rules with similar structure. Note that the `if-then-else` test branches on `null` test value.

Now a well-formed heap ensures that any field of an object in the heap stores the value whose actual type respects the declared type of the field in the type of the object.

This leads directly to the following property for good heaps, whose proof follows directly from the static properties of regions. This states that object references respect region reachability.

Theorem 2 (Reachability) *If* $\vdash_P P$, $P \vdash_H H$, $H(\iota) = \{c\langle\omega, ...\rangle,\ ...\ f \mapsto \iota'\ ...\}$ *and* $H(\iota') = \{c'\langle\omega', ...\rangle,\ ...\}$, *then* $P; \_ \vdash_\kappa \omega \rhd \omega'$ *or* $P \vdash_\simeq \omega \simeq \omega'$.

Consequently, reference cycles must occur within regions.

Theorem 3 (Cycles) *If* $\vdash_P P$, $P \vdash_H H$, $H(\iota) = \{c\langle\omega, ...\rangle,\ ...\ f \mapsto \iota'\ ...\}$ *and* $H(\iota') = \{c'\langle\omega', ...\rangle,\ ...\ f' \mapsto \iota\ ...\}$, *then* $P \vdash_\simeq \omega \simeq \omega'$.

Proof. By using Theorem 2 twice, we get ($P; \_ \vdash_\kappa \omega \rhd \omega'$ or $P \vdash_\simeq \omega \simeq \omega'$) and ($P; \_ \vdash_\kappa \omega' \rhd \omega$ or $P \vdash_\simeq \omega \simeq \omega'$). By Theorem 1, only $P \vdash_\simeq \omega \simeq \omega'$ can hold.

Finally we present a standard subject reduction result, together with a statement that goodness of a heap is invariant through expression reductions. This implies that the heap invariants are maintained through program execution.

Theorem 4 (Preservation) *Given* $\vdash_P P$, $P \vdash_E E$, $P \vdash_H H$ *and* $P; E; H \vdash_S S$, *if*

$$\frac{H(\iota) = \{\tau, \overline{f} \mapsto \overline{v}\} \quad \forall \iota \in dom(H) \quad \mathcal{C}(P,\tau) = \texttt{class}\ c\ ...\{...\ \overline{\tau}\ \overline{f}\ ...\} \quad P; \_ \vdash_t \tau \quad P; \_; H; \_ \vdash_\tau \overline{v} : \overline{\tau}}{P \vdash_H H}\ [\text{GOOD-HEAP}]$$

$$\frac{H(\iota) = \{\tau,\ ...\} \quad P; \_ \vdash_t \tau}{P; E; H; S \vdash_\tau \iota : \tau}\ [\text{GRDTYPE-LOCATION}]$$

$$\frac{\texttt{this} \in dom(S) \quad H(S(\texttt{this})) = \{\tau,\ ...\} \quad P; \_ \vdash_t \tau \quad \overline{x} \in dom(S) \quad P; E \vdash_e \overline{x} : \overline{t} \quad P; \_; H; \_ \vdash_\tau S(\overline{x}) : \mathcal{T}(P,\tau,\overline{t})}{P; E; H \vdash_S S}\ [\text{GOOD-STACK}]$$

$$\frac{P; \_ \vdash_t \tau}{P; E; H; S \vdash_\tau \texttt{null} : \tau}\ [\text{GRDTYPE-NULL}]$$

$$\frac{P; E; H; S \vdash_\tau e : \tau' \quad P \vdash_{<:} \tau' <: \tau}{P; E; H; S \vdash_\tau e : \tau}\ [\text{GRDTYPE-EXPR-SUBSUM}]$$

$$\frac{\texttt{this} \notin dom(S) \quad P; E \vdash_e e : \tau}{P; E; H; S \vdash_\tau e : \tau}\ [\text{GRDTYPE-EXPR-MAIN}]$$

$$\frac{\texttt{this} \in dom(S) \quad H(S(\texttt{this})) = \{\tau,\ ...\} \quad P; E \vdash_e e : t}{P; E; H; S \vdash_\tau e : \mathcal{T}(P,\tau,t)}\ [\text{GRDTYPE-EXPR-METHOD}]$$

Table 6: Auxiliary Rules for Dynamic Features

- $P; E; H; S \vdash_\tau e : \tau$, *and*
- $H; e \Downarrow_{P,S} v; H'$

then

- $P; E; H'; S \vdash_\tau v : \tau'$
  where $P \vdash_{<:} \tau' <: \tau$, *and*
- $P \vdash_H H'$

Proof Outline. By structural induction on the form of expressions. If v is `null`, then τ' is arbitrary.

6 Related Work

To our knowledge, our object model is the first attempt to reason about cycles and sharing in programs based on a type system imposing reachability

constraints on the object graph. Our type system does have some similarity with others. Moreover, some of the work in pointer and shape analysis gave us some insight relevant properties of reference structures.

Ownership Types and Alias Protection. Uniqueness type systems [25] ensure no aliasing and allow programmers to declare references as unaliased. Linear types [32] guarantee uniqueness by tracking objects linearly. Linear types have been used in a number of applications, such as tracking resource usage [12, 18] and ensuring safety of explicit region-based memory deallocation [14, 33].

Other type systems focus on alias management and attempt to restrict references into a limited scope. Early work like Islands [21] and Balloons [2] enforced full encapsulation on objects which prevented referencing across the encapsulation. They are generally considered as too restrictive. Universes [26] improved the expressiveness of full encapsulation by introducing read-only references to cross the boundary of encapsulation. Confined types [31, 19] used a different approach by restricting references within a specified package scope.

Ownership types [11, 10, 7] improved the previous work on object level encapsulation by allowing unrestricted outgoing references from an encapsulation while still preventing incoming referencing into an encapsulation. Ownership types and their combination with uniqueness have been used in many applications, such as program and effects reasoning, [9], external uniqueness [8] and data race/deadlock elimination [5, 4]. Both the work on confined types and the work on ownership types emanated from some general principles for flexible alias protection [27].

Ownership types used parameterized type systems to pass the names of objects via class parameters. In order to declare a type for a reference, one must be able to name the 'owner' that encapsulates this object. Encapsulation is protected from incoming referencing because the owners of objects inside an encapsulation can not be named from the outside. However, objects inside an encapsulation are able to name the objects living outside through the owner names passed in as class parameters.

Our type system is close to ownership types that are parameterized classes in a similar way, and the first parameter identifies the owner/region of `this` object. However, the invariant of our system is about acyclicity rather than encapsulation. The major difference between these two properties is that encapsulation is a local property of an object while acyclicity is rather a global property and should not be restricted to local constraints.

Other Region-Based Systems. Our notion of region is shared with that used in region-based memory management [29, 30] because they both

refer to a partition of data objects. Region-based memory management focuses on the safety and efficiency of explicit memory allocation and deallocation on the basis of regions. The ordering relation on regions are based on lifetimes, that is, on which regions may outlive others. Early region languages were lexically scoped to ensure safe deallocation in a LIFO discipline [29, 30]. There have been a number of extensions to the original region system, such as an analysis to free some regions early [1], making use of linear types or key sets to statically track living regions at each control flow point [13, 34, 14, 18] and counting references for individual regions at runtime to check if deallocation of regions is safe - no pointer should point a deallocated region [15, 16].

Our concept of region is different. Our regions represent a static abstraction of *scc*s in object graphs. However, the structure of our model is naturally compatible with the structure of object lifetimes, because objects with cyclic references are more likely to share the same lifetime. We take this idea as one of our future directions.

Lock Levels. One way to prevent deadlocks in multi-threaded programs with locks is to place a partial order on the locks to ensure they will be grabbed and released in the same order by all threads. SCJ [4] introduced a concept of lock level to help order locks statically, extending the idea of using ownership types to identify those objects which are not shared by threads (so require no locks). In their language, lock levels are partially ordered and all locks are partitioned into lock levels and therefore ordered according to their lock levels. Similar to our regions, their lock levels and ordering are defined within a class. However their lock levels are static to classes which means the number of all lock levels is limited by the number of classes. Moreover, their published type system does not appear to check the partial ordering of lock levels; however they do claim to check this in an interprocedural flow analysis.

In contrast, our regions and types are recursively defined, so the number of regions is unlimited. This gives a richer model for our region structure, and means that a programmer is able to be more discriminating in choosing an appropriate region structure. We have proved that our type system guarantees acyclicity amongst an unbounded number of regions.

Pointer Analysis. Pointer analysis has been an active research area for the past few years. It attempts to acquire the knowledge of pointer behavior at runtime via whole program analysis and uses this information to help program understanding and optimization [20]. In particular, pointer analysis can be used to compute sharing of data storage, read/write effects [6, 24] and shape of the heap [17].

Compared to type systems, pointer analysis requires little or no language annotation. Proponents of this approach often consider type systems are too restrictive and may rule out some good programs unnecessarily. However, exact pointer analysis is undecidable, so in practice it may be of relatively low precision. It is hard to scale to large or incomplete programs. Most work has been done for C-like languages, and less for object-oriented languages.

7 Discussion

7.1 Expressiveness and Limitations

Of course, as with any type system, there is a price to pay for the improved safety offered by strong type checking. First, there is the extra syntactic weight associated with more expressiveness; the syntax burden is not too taxing, amounting to the cost of parameterized types. For our purposes, it is essential to distinguish between type (schema) definition and the use of a type (instance). Without this distinction, our proposal would provide little more than the name-based access restrictions offered by module or package-based approaches. Second and more importantly, what are the expressive limitations of our approach? In some sense, none, because the type system proposed in this paper allows programmers to code with no structural constraint whatsoever. In Section 2 we discussed the ability to integrate with region-free code. Realistically though, in order to benefit from the ability of our type system to inhibit cycles and/or sharing, it is necessary to make inhibiting design decisions. Our type system will insist that programmers decide which object fields may form part of a cycle, and which may not. It is relatively simple then to record types which will cause the design decision to be enforced. Again as with any type system, there is a trade-off between extra safety offered by strong type checking, and the loss of flexibility in the programming model, or at least annoyance at being made to impose restrictions early on in a design. In practice our system will not be too annoying, because when programmers don't care about cycles, they can effectively allow them to occur anywhere, and the appropriate types are the least complex to express, corresponding exactly to the marked up legacy code.

Acyclic types can significantly improve program understanding. Programmers are able to specify via types whether cycles are allowed between two objects by placing them into same region, or disallowed by placing them into acyclic reached or disjoint regions.

Acyclic types have direct application in multi-threaded programs. Multi-

threading will not affect the structure of the object graph, but knowledge of the region structure allows, for example, ordered locking strategies to be imposed (c.f. [4]). However, in this paper we only consider the fundamental issues in reachability and acyclicity in object graph, and do not cover the issues with multi-threading.

Acyclic types can express recursive data structures. However, such a structure needs to have fields with the same type as the self type. Because they have the same type, all the structural objects forming the recursive data structure must all live in the same region. As a result, all the data objects will also live in the same region as each other.

7.2 Future Work

Future work might follow several directions.

The Theoretical Model. The theoretical foundations for reachability relations are interesting. Our formal model can be extended with more possible relations other than acyclic reachability. The simplest is to allow reachability from one region to another to be declared, without forbidding backwards reachability, as acyclic reachability does. When declared between region parameters of a class, it means that we can allow references in the direction of the declared reachability, but we can allow both parameters to be bound to the same actual region. This is a somewhat trivial but useful extension.

More significantly, we hope to formalize disjointness constraints that rely on reachability to ensure (deep) non-sharing between distinct regions, thereby permitting stronger assertions about alias-free parts of a system.

There are strong dependencies between reachability and disjointness, so the rules for environment extension with these constraints will be complicated. In fact, for consistency, they appear to require an ability to express other kinds of constraints, in particular, the *possible sharing* and *unreachability* relations. We leave a more detailed analysis for future work.

The Language and Type System. We briefly indicate some possible type system extensions. First it would be nice to allow the entries in a container to belong in different regions, rather than force them to belong to the same region as our current system generally requires. One promising approach is to use bounded existential regions, hiding the actual region where an entry lives, but still allow from and to relations based on the stated bounds.

Because of the similarities with ownership type systems, the idea of combining the reachability constraints of acyclic types, with the containment

properties of ownership types is attractive. We have made some progress along these lines already.

If we want to have an acyclic tree data structure, we will put all the node objects into distinct regions. If a node lives in a deep branch in the tree structure, we have to name all the regions and types along the branch to be able to name the region where the object lives. This is not very reasonable, so a method to escape the naming restriction may lead to a more flexible, powerful and expressive language.

Knowledge of the acyclic structure should provide help with some common operations that suffered from unintended cycles, such as deep copy and cloning, and object marshalling for distributed applications and object persistence.

Access Control and Concurrency Control. It would be natural to control access and formulate ordered locking disciplines based on the region ordering given by acyclic types. This may provide a natural extension of the lock levels of [4].

Assertion Languages. It would be interesting to see if dynamic reachability relations could be reasoned about in some program logic. Separation logic [23] uses assertions to reason about aliasing based on the different parts of heap, so may provide a starting point for further work.

Formal Specification and Effects. Unexpected aliasing between components may cause serious safety and security problems in systems. Component and subsystem isolation has become an important safety issue in software engineering. Good program understanding on sharing and effects may be helpful in addressing these issues. Moreover, our model of reachability may also help in the formal specification and refinement of component-based systems and object-oriented programs [3].

Memory Management. Although our region structures are different to those used in region-based memory management, it would be interesting to see if objects that live in our regions with cyclic references share the same lifetime. Moreover the outlive relationship between regions of memory needs to be acyclic as well, where our concept of regions may just fit in.

For garbage collection [35], we might choose just to implement a counting collector for inter-regional references, and a more sophisticated collector for the intra-regional references.

Implementation. We have a prototype implementation of a type checker for a Java-based variant of ATL. The type checking is modular. We plan to use this as a basis for further implementation. In particular our next job is to implement a region-erasing translator into standard Java; after that, we will consider a run-time model with explicit regions, and explore language

extensions involving dynamic region manipulation. Eventually we hope to blend this work with our parallel work on ownership types.

8 Conclusion

In this paper we have presented a type system that allows programmers to specify where cycles are allowed, and where they are forbidden. Our approach has been illustrated with a small object-oriented language having class-based types parameterized over regions. Object types are formed by binding the type parameters to actual regions; the first region parameter of a type defines the region where objects live for their lifetime. As well as having region parameters, each class defines its own region members. These members denote the primitive regions in our system, and are unique to their defining types.

Programmers can specify reachability constraints for regions, so that an object's type determines what other regions, and hence objects, it may reach. Our type system guarantees that reachability forms a partial order on regions. All object field references are restricted to being within the same region, or to other regions reachable from the source object's region. The major result of this paper then is a type system that allows programmers to specify regions which trap all object reference cycles, and to otherwise control the acyclic reachability for all objects.

This provides a novel contribution to ongoing work investigating the use of type systems, and other formalisms, for taming arbitrary object reference structures. There are fruitful avenues opened up for ongoing research. For us the most promising direction is to investigate incorporating more kinds of constraints, such as possible sharing, non-sharing, and ownership-like containment properties. It is still unclear to us whether attempting to combine a number of such kinds of constraints will be untractable, both in terms of the syntactic load, and the semantic complexity brought about by the interactions between various kinds of constraints. We remain hopeful that by pursuing these ideas from a graph theoretic viewpoint, more fruitful and expressive approaches will surface.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In

SIGPLAN Conference on Programming Language Design and Implementation, pages 174–185, 1995.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. In S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs (FTfJP)*, July 2003. Published as Technical Report 408 from ETH Zurich.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1993.

[7] D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.

[8] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *In European Conference for Object-Oriented Programming (ECOOP)*, July 2003.

[9] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[10] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. *Lecture Notes in Computer Science*, 2072:53–76, 2001.

[11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[12] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[13] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[14] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001.

[15] D. Gay and A. Aiken. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[16] D. Gay and A. Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[17] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM Press, 1996.

[18] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293. ACM Press, 2002.

[19] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 241–255. ACM Press, 2001.

[20] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[21] J. Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285. ACM Press, 1991.

[22] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[23] S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2001.

[24] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 56–67. ACM Press, 1993.

[25] N. H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 189–209. Springer-Verlag, 1996.

[26] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *Programming Languages and Fundamentals of Programming*, 1999.

[27] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. *Lecture Notes in Computer Science*, 1445:158–185, 1998.

[28] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *In Australian Software Engineering Conference*. IEEE Press, 1998.

[29] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[30] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[31] J. Vitek and B. Bokowski. Confined types. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96. ACM Press, 1999.

[32] P. Wadler. Linear types can change the world. *Programming Concepts and Methods*, April 1990.

[33] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[34] D. Walker and K. Watkins. On regions and linear types. In *International Conference on Functional Programming*, pages 181–192, 2001.

[35] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.