A Type System for Reachability and Acyclicity

Yi Lu and John Potter

Programming Languages and Compilers Group,
School of Computer Science and Engineering,
The University of New South Wales,
Sydney 2052, Australia
{ylu, potter}@cse.unsw.edu.au

Abstract. The desire for compile-time knowledge about the structure of heap contexts is currently increasing in many areas. However, approaches using whole program analysis are too weak in terms of both efficiency and accuracy. This paper presents a novel type system that enforces programmer-defined constraints on reachability via references or pointers, and restricts reference cycles to be within definable parts of the heap. Such constraints can be useful for program understanding and reasoning about effects and invariants, for information flow security, and for run-time optimizations and memory management.

1 Introduction

Pointers and references allow run-time sharing of data structures. In most software, references are unavoidable for pragmatic efficiency reasons, even though they complicate program reasoning and their usage is error-prone. When data structures are mutable, problems are inevitable. Reference cycles, either 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 non-termination. Reference cycles also complicate the task of memory management and cloning; 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.

Object-oriented programming languages use reference semantics for objects which, together with subtyping and a generic coding style, increase the likelihood of unintended object reference cycles. Common object-oriented design patterns, like the decorator, often require an absence of such cycles. 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?

The potential for cycles limits our ability to use class invariants in reasoning 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 run-time assertion checking for class invariants – when should the invariant hold? The problem has been highlighted in recent work on verification of object-oriented programs [3, 21] which suggests enriching the program state to track when object invariants hold; they incorporate a notion of dynamic ownership.

Reference problems are difficult to reason about because in most cases callbacks and infinite loops may be caused by indirect references which a programmer may be unaware of. Shape analysis attempts to characterize the shape of data structures via whole program analysis [26]. However, these approaches suffer from exponential complexity and cannot be accurate especially in the present of cyclic reference structures. They are also hard to scale to large or incomplete programs because of lack of modularity in the analysis techniques.

In this paper, we employ a type system, called *Acyclic Region Type System*, that allows programmers to specify desired reachability relations between objects via regions. We use the name *ARTS* to refer to our type system, the overall model, and underlying language. Regions partition the heap into distinct logical blocks of memory, and every object lives in a fixed region determined from its type. Object reference cycles are only allowed within the same region, that is, regions are partially ordered by the object reachability relation. In this way, programmers are able to express where cycles are allowed, by creating objects in the same region; they can also forbid unwanted cycles by using different regions. ARTS allows modular reasoning on a unbounded number of regions and separate compilation is possible. We also present 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. Besides program reasoning, our type system has potential application in many areas: information flow security, memory management, data copying or cloning, deadlock avoidance, and shape analysis where cycles remain an obstacle.

This paper is organized as follows: an informal overview of ARTS is given in the next section with some program examples. The core language of ARTS is formalized in Section 3, where along we present static semantics. Dynamic semantics and some important properties of the type system are given in Section 4. Discussion and related work are given in Section 5 and Section 6. Section 7 briefly concludes the paper along with some thoughts on future directions.

2 Overview of the Acyclic Region Type System

ARTS uses region-based types to capture the potential of an object to reach other objects, directly or indirectly. In typed languages, the occurrence of a cycle in the run-time object graph implies there must be a cycle in the type dependency graph; in other words, if there is no cycle in the type dependency graph, then there will be none in the run-time object graph. If there are type-level cycles, the type system is powerless to prevent cyclic references, even if they are undesirable.

With our acyclic region type system, we can enforce one-way reachability for objects, that is, one object may reach another via a path that is not part of a cycle in the object graph. *Acyclic reachability* for regions is the key concept in our model: it is a strict partial order which we denote by $\rhd$. Underlying the design of ARTS is the acyclic graph induced by the partition of a directed graph into its strongly connected components (*scc*s). All cycles within the original graph must occur within these components. We have designed our type system so that regions and acyclic reachability provide a static abstraction of the strongly connected components of the dynamic object graph and object reachability between them. The key idea is to formulate static constraints on the object graph by specifying regions within which all reference cycles are trapped. Regions are disjoint sets of objects and 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 if we look at an example illustrating a simple use of ARTS.

```
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 ARTS. Class `A` is parameterized with a region parameter `p`, which represents the region the current object lives in. A region `r` is defined within the class with the constraint `r from p`, which means `p` one-way reaches `r`, or `p` $\rhd$ `r`. This implies there may be a reference sequence starting in `p` that ends in `r` but not vice versa. 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 current object because of the order we force on their types. So any method call made on `f` cannot reenter the current object. Such knowledge allows us to assert properties that are necessarily invariant during the call, such as the reference stored in the field `f`.

ARTS can significantly improve program understanding. Programmers are able to express reference reachability between objects via types, and know that two references cannot be direct aliases if their objects live in different regions. The next example will show how to express complex data structures, such as linked lists with iterators, in ARTS.

2.1 A Linked List Example

Linked lists provide a common example for demonstrating 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 so that a list object can never reach itself via its data objects. In other words, the data objects contained by a list must not reach and thereby alter the list, which could cause iterators on the list to fail, for example. The data objects themselves may well be shared by other parts of the program.

```
class List<list, data from list>
  link from list to data;
  Link<link, data> head;
  Link<link, data> tail;
  void addElement(Data<data> d)
    head = new Link<link, data>(head, d);
    if (tail == null) tail = head;
    tail.next = head;
  Iterator<list, data> getIterator()
    return new Iterator<list, data>(head);

class Link<link, data from link>
  Link<link, data> next;
  Data<data> d;
  Link(Link<link, data> next, Data<data> d)
    this.next = next;
    this.d = d;

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

A list object is implemented by a cycle of link objects. Iterator objects are created inside the list's region, and are used to access the data stored in link objects. They can name the `link` region through a qualification over the type of the list. This removes the naming restriction in instance-based parametric type systems such as ownership types (see Section 6).

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 all 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 or iterator objects; these 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.

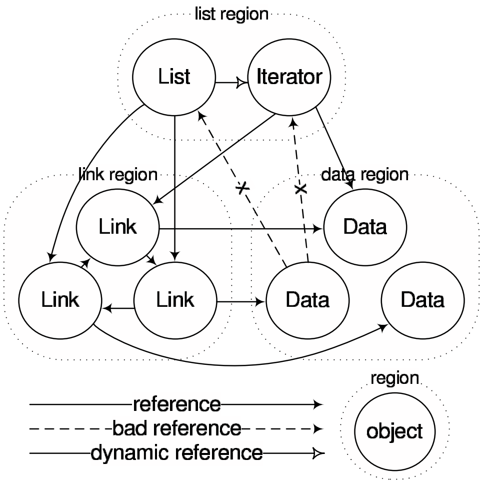

Fig. 1. List Example

3 The Language and Static Semantics of ARTS

In this section we formalize our model, providing an abstract syntax which amounts to a simple Java-like language, and a type system that captures the desired properties. Both the language and the type system are described in some detail, and we illustrate particular language features with small examples.

The key technical contribution of the paper is the way in which new region definitions define a refinement of the acyclic reachability ordering on existing regions. How does this work? Consider instead how subclass definition extends the acyclic inheritance relation on existing classes: the key ensuring that inheritance is acyclic is to ensure that there is an order of definition of classes in which subclasses are defined after superclasses. For region definitions in ARTS, we also rely on a definition order to ensure acyclicity. But extra care is needed. New regions not only extend the reachability, but also refine it by placing new regions between existing ones. The real trick to make the system work, is that, in the case of region refinement, the reachability between the existing regions must already be derivable before the new region is introduced.

3.1 An Abstract Syntax for ARTS

To simplify the abstract syntax presented in Table 1, we use a few abbreviations. The overbar is used for a sequence of constructs; for example, $\overline{\sigma}$ is used for a possibly empty sequence σ_1 ... σ_n, as are $\overline{p}$, $\overline{q}$, $\overline{fd}$, $\overline{mth}$ and $\overline{e}$. Similarly, $\overline{t\ x}$ stands for a possibly empty sequence of pairs $t_1\ x_1$ ... $t_n\ x_n$. In the class production, $[t]_{opt}$ is an optional part of the class. In the type system the equivalence symbol $\equiv$ denotes syntactic equivalence; it is used in defining syntactic lookup and substitution functions in Table 3. Just as in Java, `this` is a distinguished

Table 1. Abstract Syntax

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

$cls \in$ Class	$::= c\ \overline{p}\ [t]_{opt}\ \overline{q}\ \overline{fd}\ \overline{mth}$
$p, q \in$ RegionConstraint	$::= \overline{\sigma} \rhd \sigma \rhd \overline{\sigma}$
$t \in$ Type	$::= c\langle\overline{\sigma}\rangle$
$\sigma \in$ Region	$::= \texttt{base} \mid t.r \mid r$
$fd \in$ Field	$::= t\ f$
$mth \in$ Method	$::= t\ m(\overline{t\ x})\ e$
$e \in$ Expression	$::= x \mid \texttt{new}\ t \mid \texttt{null} \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid e;e \mid \texttt{if}\ e\ e\ e$

variable name used to reference the target object for the current call, that is, the *current object.* In the concrete syntax we use in our examples, we use keywords such as `class` and `extends` for ease of reading.

Classes and Constrained Formal Parameters. Our syntax is close to Java except that classes are parameterized with region parameters and region names are defined as members within classes.

The relational symbol $\rhd$, in region constraints, denotes the *acyclic reachability* relation between regions. In the concrete syntax, the region constraints $q \equiv \overline{\sigma} \rhd r \rhd \overline{\sigma'}$ that defines new region r is written as:

$$r\ \texttt{from}\ \sigma_1, ..., \sigma_m\ \texttt{to}\ \sigma'_1, ..., \sigma'_n \quad \text{where} \quad |\overline{\sigma}| = m \quad \text{and} \quad |\overline{\sigma'}| = n$$

The same applies to the constraints for the formal parameters $p \equiv \overline{\sigma} \rhd r \rhd \overline{\sigma'}$. The formal region parameters of a class are assumed to satisfy the reachability constraints specified in the `from` and `to` clauses of each parameter. They are used within a class to identify regions for objects used by the class. The first formal parameter denotes the region where the current object (`this`) lives. Our syntax for region names r does not distinguish between names of region parameters and locally defined regions. The reason for using two key words `from` and `to` in the concrete syntax instead of one is to make the constraints clear – the parameter or new region to be introduced occurs first.

Types. Classes are type schemas. A type is 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 its class definition. The first region argument is the region where the object of this type resides. 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 ARTS, every object belongs to a fixed region for its entire lifetime. Region expressions are formed from the special global region `base`, from type-qualified regions, and region parameters. To ensure the acyclic property for the region reachability relation, the order of introduction of new regions in definitions is important. A region definition q introduces a fresh name for its region, together with constraints that specify its reachability

Table 2. Static Semantics Given Class Definitions Π

Well-Formed Program and Well-Ordered Class Definitions $\vdash_P e;\ \vdash_c \Gamma;\ \Gamma \vdash_c cls$

[PROGRAM]
$$\frac{\begin{array}{c}\vdash_c \Pi \\ \Pi \vdash_e e : t\end{array}}{\vdash_P \Pi\ e}$$

[CLS−DEF0]
$$\frac{}{\vdash_c \emptyset}$$

[CLS−DEFS]
$$\frac{\begin{array}{c}\vdash_c \Gamma \\ \Gamma \vdash_c cls\end{array}}{\vdash_c \Gamma, cls}$$

[CLS−DEF]
$$\frac{\begin{array}{c}c \notin \Gamma \quad \overline{r} \equiv \mathcal{R}(\overline{p}) \quad E \equiv \Pi, \overline{p}, \texttt{this} : c\langle\overline{r}\rangle \\ \Pi \vdash_r \overline{p} \quad \Gamma, \overline{p} \vdash_r \overline{q} \quad E \vdash_f \overline{fd} \quad E \vdash_m \overline{mth} \\ [t \equiv c'\langle r_1, ...\rangle \ \ E \vdash_t t \ \ c' \in \Gamma]_{opt}\end{array}}{\Gamma \vdash_c c\ \overline{p}\ [t]_{opt}\ \overline{q}\ \overline{fd}\ \overline{mth}}$$

Well-Ordered Region Definitions $E \vdash_r \overline{p};\ \ E \vdash_r p$

[REG−DEF0]
$$\frac{}{E \vdash_r \emptyset}$$

[REG−DEFS]
$$\frac{E \vdash_r \overline{p} \quad E, \overline{p} \vdash_r p}{E \vdash_r \overline{p}, p}$$

[REG−DEF]
$$\frac{r \notin \mathcal{R}(E) \quad E \vdash_\sigma \overline{\sigma}, \overline{\sigma'} \quad E \vdash_\triangleright \overline{\sigma} \triangleright \overline{\sigma'}}{E \vdash_r \overline{\sigma} \triangleright r \triangleright \overline{\sigma'}}$$

Well-Defined Field and Method $E \vdash_f fd;\ \ E \vdash_m mth$

[FIELD]
$$\frac{\begin{array}{c}t_o \equiv c_o\langle r, ...\rangle \quad t \equiv c\langle\sigma, ...\rangle \\ E \vdash_e \texttt{this} : t_o \quad E \vdash_t t \quad E \vdash_\triangleright r \trianglerighteq \sigma\end{array}}{E \vdash_f t\ f}$$

[METHOD]
$$\frac{\begin{array}{c}E \vdash_t t, \overline{t} \\ E, \overline{x : t} \vdash_e e : t\end{array}}{E \vdash_m t\ m(\overline{t\ x})\ e}$$

Well-Formed Region and Type $E \vdash_\sigma \sigma;\ \ E \vdash_t t$

[REG−BASE]
$$\frac{}{E \vdash_\sigma \texttt{base}}$$

[REG−NAME]
$$\frac{r \in \mathcal{R}(E)}{E \vdash_\sigma r}$$

[REG−QUAL]
$$\frac{E \vdash_t t \quad t.r \in \mathcal{R}(t)}{E \vdash_\sigma t.r}$$

[TYPE]
$$\frac{\begin{array}{c}t \equiv c\langle\overline{\sigma}\rangle \quad \mathcal{C}(t) \equiv c\ \overline{p}\ ... \\ c \in E \quad E \vdash_\sigma \overline{\sigma} \quad E \vdash_\triangleright \overline{p}\end{array}}{E \vdash_t t}$$

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

[SUBTYPE−REFL]
$$\frac{}{\vdash_{<:} t <: t}$$

[SUBTYPE−EXTEND]
$$\frac{\mathcal{C}(t) \equiv ...\ t'\ ...}{\vdash_{<:} t <: t'}$$

[SUBTYPE−TRANS]
$$\frac{\vdash_{<:} t <: t' \quad \vdash_{<:} t' <: t''}{\vdash_{<:} t <: t''}$$

Region Reachability $E \vdash_\triangleright \sigma \triangleright \sigma';\ \ E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma'';\ \ E \vdash_\triangleright \sigma \trianglerighteq \sigma'$

[REACH−ENV]
$$\frac{\sigma \triangleright \sigma' \in E}{E \vdash_\triangleright \sigma \triangleright \sigma'}$$

[REACH−DEF−TO]
$$\frac{t.r \triangleright \sigma \in \mathcal{Q}(t)}{E \vdash_\triangleright t.r \triangleright \sigma}$$

[REACH−DEF−FROM]
$$\frac{\sigma \triangleright t.r \in \mathcal{Q}(t)}{E \vdash_\triangleright \sigma \triangleright t.r}$$

[REACH−TRANS]
$$\frac{E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma''}{E \vdash_\triangleright \sigma \triangleright \sigma''}$$

[REACH−COMB]
$$\frac{E \vdash_\triangleright \sigma \triangleright \sigma' \quad E \vdash_\triangleright \sigma' \triangleright \sigma''}{E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma''}$$

[REACH−REFL]
$$\frac{}{E \vdash_\triangleright \sigma \trianglerighteq \sigma}$$

[REACH−EXT]
$$\frac{E \vdash_\triangleright \sigma \triangleright \sigma'}{E \vdash_\triangleright \sigma \trianglerighteq \sigma'}$$

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

[EXPR−VAR]
$$\frac{}{E \vdash_e x : E(x)}$$

[EXPR−NEW]
$$\frac{E \vdash_t t}{E \vdash_e \texttt{new}\ t : t}$$

[EXPR−NULL]
$$\frac{E \vdash_t t}{E \vdash_e \texttt{null} : t}$$

[EXPR−FIELD]
$$\frac{\begin{array}{c}(t\ f) \in \mathcal{F}(t_o) \\ E \vdash_e e : t_o\end{array}}{E \vdash_e e.f : t}$$

[EXPR−ASSIGN]
$$\frac{\begin{array}{c}(t\ f) \in \mathcal{F}(t_o) \\ E \vdash_e e : t_o \quad E \vdash_e e' : t\end{array}}{E \vdash_e e.f = e' : t}$$

[EXPR−CALL]
$$\frac{\begin{array}{c}\mathcal{M}(t_o, m) \equiv (\overline{t}, \_, t, \_) \\ E \vdash_e e : t_o \quad E \vdash_e \overline{e : t}\end{array}}{E \vdash_e e.m(\overline{e}) : t}$$

[EXPR−SEQ]
$$\frac{\begin{array}{c}E \vdash_e e : t \\ E \vdash_e e' : t'\end{array}}{E \vdash_e e; e' : t'}$$

[EXPR−IF]
$$\frac{\begin{array}{c}E \vdash_e e : t \\ E \vdash_e e' : t' \quad E \vdash_e e'' : t'\end{array}}{E \vdash_e \texttt{if}\ e\ e'\ e'' : t'}$$

[EXPR−SUBSUM]
$$\frac{\begin{array}{c}E \vdash_e e : t \\ \vdash_{<:} t <: t'\end{array}}{E \vdash_e e : t'}$$

Table 3. Auxiliary Lookup Functions for Static Semantics

$$\begin{array}{c}\text{[LOOKUP−CLASS]}\\ \dfrac{t \equiv c\langle\overline{\sigma}\rangle \quad \overline{r} \equiv \mathcal{R}(\overline{p}) \quad \overline{r'} \equiv \mathcal{R}(\overline{q}) \quad \Pi(c) \equiv cls \equiv c\ \overline{p}\ ...\overline{q}\ ...}{\mathcal{C}(t) \equiv cls[\overline{\sigma/r}, \overline{t.r'/r'}]}\end{array} \qquad \begin{array}{c}\text{[LOOKUP−DEF]}\\ \dfrac{\mathcal{C}(t) \equiv ...\ \overline{q}\ ...}{\mathcal{Q}(t) \equiv \overline{q}}\end{array}$$

$$\begin{array}{c}\text{[LOOKUP−FIELD]}\\ \dfrac{\mathcal{C}(t_o) \equiv ...\ [t']_{opt}\ ...\ \overline{t\ f}}{\mathcal{F}(t_o) \equiv [\mathcal{F}(t')]_{opt}, \overline{t\ f}}\end{array} \qquad \begin{array}{c}\text{[LOOKUP−METHOD]}\\ \dfrac{\mathcal{C}(t_o) \equiv ...\ t\ m(\overline{t\ x})\ e\ ...}{\mathcal{M}(t_o, m) \equiv (\overline{t}, \overline{x}, t, e)}\end{array} \qquad \begin{array}{c}\text{[LOOKUP−METHOD}'\text{]}\\ \dfrac{\begin{array}{c}\mathcal{C}(t_o) \equiv ...\ t\ ...\ \overline{mth}\ ...\\ m \notin \overline{mth}\end{array}}{\mathcal{M}(t_o, m) \equiv \mathcal{M}(t, m)}\end{array}$$

$$\begin{array}{c}\text{[LOOKUP−REGION−ENV]}\\ \dfrac{}{\mathcal{R}(\emptyset) \equiv \emptyset \quad \mathcal{R}(E, cls) \equiv \mathcal{R}(E) \quad \mathcal{R}(E, p) \equiv \mathcal{R}(E), \mathcal{R}(p) \quad \mathcal{R}(E, x : t) \equiv \mathcal{R}(E)}\end{array}$$

$$\begin{array}{c}\text{[LOOKUP−REGION−DEF]}\\ \dfrac{}{\mathcal{R}(\overline{\sigma} \triangleright \sigma'' \triangleright \overline{\sigma'}) \equiv \sigma'' \quad \mathcal{R}(\overline{q}, q) \equiv \mathcal{R}(\overline{q}), \mathcal{R}(q) \quad \mathcal{R}(t) \equiv \mathcal{R}(\mathcal{Q}(t))}\end{array}$$

properties in terms of previously defined regions. Furthermore these constraints must not impose any further requirement 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 3.2.

Compared with constraints on formal parameters, which just impose requirements on the actual region arguments for a class, region definitions actually determine the region structure for the system. Every class defines a type schema, with its own locally defined regions. We allow types (not just class names) to qualify region names – every such qualified region $t.r$ uniquely determines a particular region in the system. To guarantee this uniqueness, we do not allow regions to be inherited by subclasses; otherwise the same region could be identified through a subtype which inherits it. In fact it is straightforward to allow region inheritance, but adds no extra expressivity, and slightly complicates the type system.

Interestingly, regions and types are recursively defined: regions are defined within classes and named via type qualification; types are formed by binding class parameters with actual regions. Because of this recursive structure there are an unbounded number of regions in the system, all globally accessible via region path expressions. Fortunately the programmer does not have 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.

The primary goal of ARTS is to allow programmers to define a static preorder on the run-time object structure which induces a partial order on the scc's of the objects, thereby restricting where cycles can occur. By making class-defined region names publicly accessible we keep flexibility in that every ground region has a unique global name. Ground regions are those with no free region parameter: either `base`, or a region $t.r$ qualified by a ground type t; a ground type is one with no free region parameter. 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 it determines the region an object lives in).

The special region `base` is pre-defined and is the only unqualified ground region in the system. An important note here is that regions do *not* have to be reachable from `base` or vice-versa; also any root object (objects created in the main method) does *not* have to be placed in `base`. `base` is just the base region used to name any other ground region. 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. In fact, we could allow the expression defining the main method of a program to define its own regions, making the use of global regions superfluous.

All region constraints are statically verified by the type system to ensure global acyclicity. Any strongly connected component in the object heap during the execution of an ARTS program must occur entirely within a region. Regions need not exist at run-time 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 ARTS, 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. The 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 arguments to access objects which are not accessible to the current object, that is, the regions of method parameters and results do not have to be reachable from the region of the `this` 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. These dynamic references can be considered safe because they emanate from the calling stack rather than the heap, which is not referenceable from the heap and deallocated at the exit of the method.

It is worth acknowledging at this stage that our choice of restrictions for field and method access is somewhat arbitrary. It is easy to modify the type rules to enforce different invariants; for example, if we wish to block the possibility of cycles with dynamic references as well, then we could impose the restriction that all method arguments and results are accessible from the region of `this` just like we do to fields. The key point of the type system is that we have the ability to invent such type rules, because we have a mechanism for specifying and statically checking acyclic reachability of regions.

3.2 Static Semantics

We present an overview of the static semantics of ARTS, along with detailed description of some important typing rules. The complete type system can be found in Table 2. In addition to the type system, we define auxiliary functions to lookup and bind classes, types, regions, region definitions, fields and methods in Table 3. We intentionally move all occurrences of substitution into the auxiliary functions to simplify the typing rules, for example, the class lookup for a given type substitutes the actual region arguments into the class definition, and qualifies all of the local region definitions with the type.

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; region names, including formal parameters, cannot be declared more than once within the same class. In fact, we explicitly check this in the case of class names and region names, because we need to check that region definition is well-ordered in the rules.

We use some syntactic shortcuts. By default $E \vdash_\sigma \overline{\sigma}$ represents a possibly empty sequence of judgements $E \vdash_\sigma \sigma_1$ ... $E \vdash_\sigma \sigma_n$. The same abbreviation is used for judgements involving $\overline{fd}$, $\overline{mth}$ and $\overline{t}$. Similarly, $E \vdash_e \overline{e : t}$ stands for $E \vdash_e e_1 : t_1$ ... $E \vdash_e e_n : t_n$, and $[\overline{\sigma/r}]$ abbreviates a sequence of substitutions $[\sigma_1/r_1]$ ... $[\sigma_n/r_n]$.

A program P is a pair consisting a fixed sequence of class definitions $\mathit{\Pi}$ and an expression e. $\mathit{\Gamma}$ denotes any sequence of class definitions. An environment E may extend a sequence of class definitions with constraints on region names, or the types of variables. The order of the elements in the environment is significant.

$$P ::= \mathit{\Pi}\ e \qquad \mathit{\Pi} ::= \mathit{\Gamma} \qquad \mathit{\Gamma} ::= \emptyset \mid \mathit{\Gamma}, cls \qquad E ::= \mathit{\Gamma} \mid E, x : t \mid E, \overline{\sigma} \rhd r \rhd \overline{\sigma}$$

Because the number of recursively defined regions and types is 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, as checked by the $\mathit{\Gamma}, \overline{p} \vdash_r \overline{q}$ judgement in the [CLS-DEF] rule.

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 $\sigma \rhd \sigma'$ is already defined, then a new region r can be defined via $\sigma \rhd r \rhd \sigma'$, but not $\sigma' \rhd r \rhd \sigma$ because this would introduce cycles into the region structure. Similarly, if there is no existing reachability between σ and σ' can be derived, then $\sigma \rhd r \rhd \sigma'$ is invalid too because it implies $\sigma \rhd \sigma'$ which is unspecified originally. This constraint is enforced by the [REG-DEF] rule which guarantees that the new region variable and the relations on it do not violate the consistency of the region ordering.

Class definition ordering is established by [CLS-DEFS]. Class well-formedness is checked in [CLS-DEF]. Each class defines its own environment E formed from all given class definition, its formal parameter constraints and the type of the current object. Note that we do not put region definitions into E, because we want local regions to be qualified when used in region expressions and types. We recall that the constraints on formal region parameters, $\overline{p}$, are requirements of the class. The well-formedness of the parameter constraints is checked by the judgement $\Pi \vdash_r \overline{p}$; this assumes a whole program context, so that the constraints need not rely just on previously defined regions. Actually we could omit this judgement altogether and rely on checking that the actual region arguments satisfy the constraints whenever the class is used to form a type; including the check on parameter constraints implies that we will reject unusable classes even if they are not used. A region definition q is checked in a context restricted to previously defined classes in Γ, the formal parameter constraints $\overline{p}$ of the current class, and, by the unwinding of [REG-DEFS] earlier local definitions $\overline{q}$, thus guaranteeing the partial ordering of regions, as indicated above. If the class is extended from a supertype then the supertype needs to be valid in the class environment E, both subtype and supertype must live in the same region (their first parameter r_1) to ensure that the region of the current object is not lost through subtyping, and the superclass must be pre-defined in Γ. Finally for a well-formed class, all fields and methods must be well-formed in E.

For a type to be valid by [TYPE], we require the class to be defined in the environment. Its actual region arguments must satisfy the defined constraints by substituting the actual regions into the class definition. If two formal parameters are unconstrained, then they may be bound to any valid region; they may even both be bound to the same region.

Once a well-formed environment is established, it is used to infer region reachability for all valid regions. Region reachability is defined to be transitive; irreflexivity and antisymmetry are consequences of our system. The first three [REACH] rules check the region relations as directly defined by programmers, all other relations are inferred through transitivity rule [REACH-TRANS]. The [REACH-ENV] rule checks defined relations between formal region parameters while the pair of rules [REACH-DEF-FROM/TO] check the defined reachability relations in region definitions.

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 is same to or can reach the regions of the other objects. This is an important invariant of our programs.

In the current setting of our type system, we allow method arguments and result types to freely reference objects in any region. 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. As discussed earlier, stronger constraints could easily be written into this rule to prohibit inter-region dynamic cycles.

3.3 Examples

We give some toy examples to illustrate the use of regions, then show how regions can be used as security levels to capture the ordering of information flow.

Toy Examples

```
class A<p1, p2 from p1, p3>
  a1 from p1 to p2;              // OK
  a2 from p2 to p1;              // BAD require 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 by transitivity
  A<b1, b2, p> f2;               // OK
  A<p, b3, b2> f3;               // BAD require b3 from p
  A<p, base, b1> f4;             // BAD require base from p
  A<p, B<b3>.b2, p> f5;          // BAD require B<b3>.b2 from p

class C<
  p1 from base,                  // OK
  p2 from B<p1>.b1,              // OK p1 already introduced
  p3 from B<p3>.b1,              // BAD p3 undefined yet
  p4 to B<base>.b3,              // BAD require base from base
  p5 from p1 to B<p2>.b2         // OK by transitivity
  >
```

Class A shows how regions are defined. The key point in region definitions is that any newly introduced region cannot change the relation 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 formal region parameters, some with qualified regions.

```
class M<p>
  m to N<p>.n;                   // BAD require ordering of classes

class M'<p>
  m to p;                        // OK

class M"<p1, p2 to N<p1>.n>      // OK do not require ordering of classes
  m to p2;

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

The examples above show the importance of class ordering. Class ordering is not only important for correct inheritance, but also important in keeping the ordering of region definitions. In classes `M` and `N`, there is a cycle between `M<p>.m` and `N<p>.n` whenever their class parameters are bound to the same region. To solve this problem, we forbid regions in earlier classes to be defined to relate to regions defined in later classes to enforce the order of region definitions, that is, regions definitions in earlier classes are considered to be earlier than those in later classes. Since class `M` is defined earlier than class `N`, class `M` is not valid and must be rewritten to class `M'` while later class `N` is always valid. Alternatively, class `M` can be rewritten to class `M"` where the region `m` can be defined to reach `N<p1>.n` through the formal parameter `p2` since class ordering does not apply to the constraints on formal region parameters.

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

As we discussed earlier our current version of ARTS allows dynamic references (method arguments and results) to be treated differently to static/field references. There is no restriction on what regions can be used for dynamic references. Unrestricted dynamic references add flexibility which can be seen in the factory example. The `SubjectFactory` class models a factory in region `factory` for creating `Subject` objects in region `subject`. Because there is no known reachability between `factory` and `subject`, the type system prevents the `SubjectFactory` class from holding field references to objects in `subject`. However, since dynamic references are not bound by this restriction, new objects can be created in `subject` and returned through the `makeSubject` method for clients to use.

Type Enforced Security Levels. Besides using regions to prevent reference cycles and to reason about aliasing, acyclic regions can also be used as security levels to control access and secure information flow in a multi-level security system (see Section 6). Programmers can express desired access control and information flow policies in the region structure of a program. Because these security policies are now expressed in types, they can be enforced statically by the type system.

```
class Machine<floor>
  display from floor;
  Display<display> disp;
  Machine()
    disp = new Display<display>();
  adjust()                      // modify the display
    ...
class Operator<skill, floor from skill>
  Machine<floor> mach;
  Display<Machine<floor>.display> disp;
  set(Machine<floor> mach)
```

```
    this.mach = mach;
    this.disp = mach.disp;
  operate()                     // do a job, such as adjust the machine
    ...
class Factory<factory>
  skill1 from factory;
  skill2 from factory 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, not a valid type
  op1.set(mach1);               // OK
  op1.set(mach2);               // BAD, op1 is on wrong floor
```

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 different floor regions and four operators occupy two skill level regions. The region structure of `Operator` requires an operator to have enough skill to work on a machine. The relations between different skills and floors are defined in the `Factory` class - operators with `skill2` can work on the machines on any floor while operators with `skill1` can only operate the machines on `floor1`. Moreover, operators with `skill1` can never obtain a reference to machines on `floor2`, which implies information stored in region `floor2` can never flow to objects in region `floor1`.

4 Some Properties and a Dynamic Semantics

In this section, 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 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. Finally we state a standard subject reduction theorem, that, amongst other things, states that heap goodness is preserved through reductions.

First we capture the idea of one region being defined earlier than another. A class definition sequence Γ, that is well-formed, $\vdash_c \Gamma$, determines a sequence of region definitions. A qualified region $t.r$ that is well-formed, $\Gamma \vdash_\sigma t.r$, is associated with a unique index in the definition sequence, namely that which defines the region name of the qualified region. We call this the *rank* of the region. We also define the rank of `base` to be 0. A region σ with a smaller rank than another σ' is defined earlier; we will also write this as $\sigma \prec \sigma'$. If a region appears as a bound in a definition for another, then the regions must have different ranks; essentially this is because of the [REG-DEF] rule.

Table 4. Dynamic Features

$\iota, \iota_t \in$ TypedLocation		
$e \in$ Expression	$::=$ ... $\mid \iota$	
$v \in$ Value	$::= \iota \mid$ `null`	
$obj \in$ Object	$=$ FieldName $\longrightarrow$ Value	
$H \in$ Heap	$=$ TypedLocation $\longrightarrow$ Object	

We are now in a position to state a fundamental property of reachability proofs: any reachability between two regions can be defined through a sequence of region definitions, where the successive ranks are strictly decreasing until a minimum rank is reached, after which the ranks are strictly increasing.

Lemma 1 (Reachability via Earlier Regions). *Given* $\vdash_c \Gamma$ *and* $\Gamma \vdash_\sigma \sigma, \sigma'$*:* *If* $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ *then* $\exists \sigma_1 ... \sigma_n$ *for* $n > 1$ *such that:*

1. $\sigma \equiv \sigma_1 \triangleright ... \triangleright \sigma_n \equiv \sigma'$, *and*
2. $\sigma_1 \succ \sigma_2 \succ ... \succ \sigma_m \prec ... \prec \sigma_n$ *for some* $m \in 1..n$ *where* $(\sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_i)$ *for* $1 \leq i < m$ *(where* $\sigma_i \equiv t_i.r$*), and* $(\sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_{i+1})$ *for* $m \leq i < n$.

Proof Outline. Any proof of reachability must construct a sequence of one or more applications of a definition via [REACH-DEF-FROM/TO]. So the first part of the lemma follows, with successive pairs belonging to some definition. Suppose the second part does not hold. Then we can find $(\sigma_{i-1} \triangleright \sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_i)$ with $(\sigma_{i-1} \prec \sigma_i \succ \sigma_{i+1})$. But by the requirement of [REG-DEF] that any reachability of constraints can be derivable from earlier definitions, we see that we can omit σ_i from our proof. The result follows by an induction on the maximum region definition rank in Γ.

Theorem 1 (Acyclicity of Regions). *Given* $\vdash_c \Gamma$ *and* $\Gamma \vdash_\sigma \sigma, \sigma'$*:* *if* $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ *then* $\Gamma \nvdash_\triangleright \sigma' \trianglerighteq \sigma$.

Proof Outline. Suppose, by contradiction that $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ and $\Gamma \vdash_\triangleright \sigma' \trianglerighteq \sigma$. Without loss of generality, assume that $\sigma \prec \sigma'$. By transitivity of $\triangleright$, we find that $\Gamma \vdash_\triangleright \sigma \triangleright \sigma$. From Lemma 1 we can see that we can find two (possibly same) earlier regions σ_1 and σ_2, such that $\sigma \triangleright \sigma_1 ... \sigma_2 \triangleright \sigma$ and $\sigma_1, \sigma_2 \prec \sigma$. But again, we must have $(\sigma_2 \triangleright \sigma \triangleright \sigma_1) \in \mathcal{Q}(t)$, so it follows that $\sigma_2 \triangleright \sigma_1$. The result follows by an induction on the minimum rank of the two regions under consideration.

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

We incorporate full type information (with regions) with the locations of the heap, rather than in the objects. These help to simplify the semantics and the proof of dynamic properties. Note that none of the reduction behavior depends on this type information; the `new` t reduction only uses the field names of the class; method dispatch only depends on the class and not on the region bindings

Table 5. Dynamic Semantics

$$\text{[RED-NEW]}\quad \frac{\iota_t \notin dom(H) \quad \mathcal{F}(P,t) = \overline{\_\ f} \quad H' \equiv H, \iota_t \mapsto \overline{f \mapsto \mathtt{null}}}{H, \mathtt{new}\ t \Downarrow \iota_t, H'}$$

$$\text{[RED-FIELD]}\quad \frac{H, e \Downarrow \iota, H'}{H, e.f \Downarrow H(\iota)(f), H'}$$

$$\text{[RED-ASSIGN]}\quad \frac{H, e \Downarrow \iota, H' \quad H', e' \Downarrow v, H'' \quad H''' \equiv H''[\iota \mapsto H''(\iota)[f \mapsto v]]}{H, e.f = e' \Downarrow v, H'''}$$

$$\text{[RED-CALL]}\quad \frac{H, e \Downarrow \iota_t, H' \quad H', \overline{e} \Downarrow \overline{v}, H'' \quad \mathcal{M}(P,t,m) = (\_, \overline{x}, \_, e') \quad H'', e'[\iota_t/\mathtt{this}, \overline{v/x}] \Downarrow v, H'''}{H, e.m(\overline{e}) \Downarrow v, H'''}$$

$$\text{[RED-SEQ]}\quad \frac{H, e \Downarrow \_, H' \quad H', e' \Downarrow v, H''}{H, e; e' \Downarrow v, H''}$$

$$\text{[RED-IF-LOCATION]}\quad \frac{H, e \Downarrow \iota, H' \quad H', e' \Downarrow v, H''}{H, \mathtt{if}\ e\ e'\ e'' \Downarrow v, H''}$$

$$\text{[RED-IF-NULL]}\quad \frac{H, e \Downarrow \mathtt{null}, H' \quad H', e'' \Downarrow v, H''}{H, \mathtt{if}\ e\ e'\ e'' \Downarrow v, H''}$$

Table 6. Auxiliary Rules for Dynamic Semantics

$$\text{[HEAP-WELLFORMED]}\quad \frac{\forall \iota_t \in dom(H) \cdot \quad \Gamma \vdash_t t \quad H(\iota_t) = \overline{f \mapsto v} \quad \mathcal{F}(t) = \overline{t\ f} \quad \Gamma \vdash_e \overline{v : t}}{\Gamma \vdash_H H}$$

$$\text{[EXPR-LOCATION]}\quad \frac{\Gamma \vdash_t t}{\Gamma \vdash_e \iota_t : t}$$

of the target object. Again, for simplicity, we do not use local variables in our model, so we use substitution of method arguments into method bodies, thus avoiding the extra machinery of a stack frame. Note that the `if` 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. Consequently, reference cycles must occur within regions.

Lemma 2 (Direct Referenceability). *Given* $\vdash_c \Gamma$ *and* $\Gamma \vdash_H H$*:*
if $\iota_{c\langle\sigma...\rangle} \mapsto [...\ \_ \mapsto \iota'_{c'\langle\sigma'...\rangle}\ ...] \in H$*, then* $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$*.*

Proof. By the [HEAP-WELLFORMED] rule, $c\langle\sigma...\rangle$ must be well-formed and $c'\langle\sigma'...\rangle$ is the type of one of class c's fields. By the [FIELD] rule, $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$.

Theorem 2 (Reachability and Cycles). *Given* $\vdash_c \Gamma$ *and* $\Gamma \vdash_H H$*:*
if $\iota_{c\langle\sigma...\rangle}$ *can reach* $\iota'_{c'\langle\sigma'...\rangle}$ *through a path of direct references in the heap* H*, then* $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$*. Furthermore, if there is a path in* H *in the reverse direction, then* $\sigma = \sigma'$*.*

Proof. By Lemma 2 for each direct reference in the path, and the transitivity of $\triangleright$, the first part follows. When there is a reverse path, the second part follows by the acyclicity of $\triangleright$, as in Theorem 1.

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 3 (Preservation). *Given $\vdash_c \Gamma$, and $\Gamma \vdash_H H$:*

$$\textit{if} \begin{cases} \Gamma \vdash_e e : t \\ H, e \Downarrow v, H' \end{cases} \textit{then} \begin{cases} \Gamma \vdash_e v : t \\ \Gamma \vdash_H H'. \end{cases}$$

Proof Outline. The proof for type preservation is completely standard by structural induction on the form of expressions over reduction rules. We do not show a subtype of t for v, because it is covered by subsumption.

5 Discussion

5.1 Expressiveness and Limitation

ARTS provides a powerful framework for allowing a succinct description of many classes of reachability relations. It employs an intuitive notion of region to capture the concept of strongly connected components in graph theory, which appears to be natural and flexible enough to express various data structures in programs. The type checking is simple yet efficient and powerful; it locally checks programmer-defined reachability relations to guarantee global acyclicity so that separate compilation can be allowed. ARTS allows the programmer to name any possible region even though the number of regions may be unbounded. The ability to multiply instantiate region definitions through type qualification gives programmers enough choice to identify as many distinct regions as they need.

ARTS can significantly improve program understanding. Programmers are able to specify via types whether cycles are allowed or disallowed. ARTS can also be used to reason about aliasing because regions are disjoint and objects live in a single region for their lifetime. Moreover, ARTS can express some information flow policy (see Section 6) and even encapsulate objects (see later this Section). ARTS has 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 [4]. However, in this paper we only consider the fundamental issues in reachability and acyclicity in the object graph, and do not cover the issues with multi-threading.

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, that is, all objects live in the same region. In Section 5.2 we will discuss 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 do not 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.

ARTS 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. ARTS allows the programmer to name any possible region even though the number of regions is unbounded. This flexibility complicates the task of type checking on the reachability relations. In the worst case, to check the relation between two regions, the type checker may have to look into all classes in the recursive type qualification steps for both regions and for each class the type checker may have to look into all region definitions of the class. However, in practice both the number of recursive steps to make sensible use of a region and the number of region definitions in a class are very small, so that the runtime for type checking should not be significant. Moreover, any reachability relation is decidable because each region can only have a finite number of recursive steps and each class can only have a finite number of region definitions.

5.2 Extensions to the Core Language

Object Encapsulation with Owned Regions. Our type system can provide region existential polymorphism at almost no cost. Existential regions can provide the same level of object encapsulation as ownership types do (see Section 6). In our extended language, regions can be hidden by using an *owned* declaration; owned regions cannot be named via a type qualification and hence local to the scope of the class. To enable type checking on owned regions, we simply need to add an optional key word `owned` in front of the region name in the syntax when a region is introduced, and disable type qualification over owned regions in the region rules.

It is important that owned regions are instance level and are encapsulated by their defining objects because no one can name them from outside. This is similar to the internal context `this` in ownership types. An owned region can only be named within the defining object or propagated to its encapsulation via region parameters. Unowned regions remain static to types, but may not be named globally if their types are parameterized by an owned region. This results in ownership-like structures on some parts of the DAG structure of our object model. Programmers may have better understanding and fine-grained control over the reference structure.

For example, owned regions can be used to express ownership-like linked lists without suffering the long-recognized problem of expressing iterators in ownership types. In the next example, every list object now has its own implementation encapsulated by the owned region `link`. Because owned regions are instance-based, they are not shared by different list objects of the same type (like unowned regions are). The link objects are owned by the list because no object from outside of the list can name the owned region `link`. An iterator can access the internal data of the list, because it is created inside the list, but can still be used from outside because it lives in the same region as the list object. We use subtyping to hide the name of the `link` region in the type of the iterator, so the client can give a type for the iterator without knowing the right name for the `link` region.

```
class List<list, data from list>
  owned link from list to data;
  Link<link, data> head;
  Link<link, data> tail;
  Iterator<list, data> getIterator()
    return new ListIterator<list, data, link>(head);
  ...
class Iterator<list, data from list>
  ...
class ListIterator<list, data from list, link from list to data>
      extends Iterator<list, data>
  Link<link, data> current;
  Iterator(Link<link, data> current)
    this.current = current;
  ...
```

Default Regions and Interoperability with Legacy Code. The burden of region annotations and the interoperability with legacy code may affect the ease of use for the language. In practice, regions are only used when needed. We expect programmers to use regions in some critical sections, for example, to protect crucial class invariants from unexpected method reentrant calls. In many cases regions can be circumvented when they are not needed and programmers need not even be aware of regions. We design a few region defaults to help reduce the number of region annotations and integrate non-region code such as legacy library classes. The compiler may automatically annotate regions with the default policy.

For classes that do not have formal region parameters, the key word `here` is used to refer the region the current object lives in. This is similar to `this`, `self` or `me` used in many object-oriented languages to refer the current object. For bindings of regions in types, if a type is declared without any region, i.e. only a class name, then a default region is bound to all formal parameters of the class. In the main routine, `base` can be used as the default region for types while in classes the first formal parameter of the class (or `here` if there is none) is the default region. For legacy code all formal region parameters and type declarations are defaulted.

Flexible Class Constraints. Our language can be easily extended with more possible relations other than acyclic reachability. Of course, these possible relations can only be used to constrain formal region parameters of a class, i.e., not to the region definitions. They are just class constraints for valid region parameter bindings, they should not be confused with the reachability relation defined between actual regions. The simplest is to allow reachability from one region parameter to another to be declared without forbidding backwards reachability, denoted as $\trianglerighteq$ in the type system which is a reflexive closure of the acyclic reachability $\triangleright$. When declared, 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 for more flexible programming.

6 Related Work

To our knowledge, this 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 similarities with other type systems, such as parametric polymorphism and existential polymorphism on regions. Our type qualified regions and region parameterized types provide a novel way to support global naming and static reasoning on an unbounded number of regions.

Ownership Type Systems. Many type systems focus on alias management and attempt to restrict references into a limited scope. Early work like Islands [16] and Balloons [2] enforced full encapsulation on objects which prevented referencing across the encapsulation. They are generally too restrictive. Universes [19] improved the expressiveness of full encapsulation by introducing read-only references to cross the boundary of encapsulation. Ownership types [7, 6, 5] 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 use 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 cannot 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 the `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 instance-based and enforced through ownership, which is a local property of an object whereas acyclicity is global.

A variant of ownership types has recently been proposed [1] to allow programmers to specify aliasing policy between ownership domains (which partition an owner's context) rather than ownership types' owner-as-dominator property. Their domains and link polices are defined within classes in a similar fashion to our regions and region constraints. Their aliasing policies define referenceability, the ability to directly reference an object, between domains. They are concerned with neither reachability – the link policies are not transitive, nor acyclicity – it is possible to link domains cyclically. In fact, our regions can also be used to reason about aliasing in a similar way but with a different policy – we can enforce a deep reachability policy instead of a shallow direct referenceability policy.

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 ownership domains and our regions, their lock levels and ordering are defined within classes. 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.

In contrast, our system supports region parametric polymorphism for code reuse and region existential polymorphism for object level encapsulation. We also provide a novel type-based naming mechanism to allow an unbounded number of regions to be named throughout the system. Type qualified regions allow completely static reasoning on any possible region and their constraints, excluding owned regions which only can be reasoned about locally. 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 and Shape Analysis. Pointer analysis attempts to acquire knowledge about run-time pointers via whole program analysis and uses this information to help program understanding and optimization [15]. Shape analysis is built on the top of pointer analysis to identify the shapes of data structures [10, 26]. Hackett and Rugina have recently proposed a shape analysis algorithm that breaks down global analysis about entire heap into local analysis about smaller memory abstractions which they also call regions [12]. Similar to our regions, theirs are disjoint sets of memory locations, which allow the subsequent shape analysis to safely conclude that an update in one region will not change the values of locations in other regions. Different from ours, their memory regions are not acyclic and they simply identify direct *points-to* relations between regions rather than transitive reachability relations.

Compared to type systems, pointer and shape analysis require 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 and shape analysis not only require exponential time for verification, but are undecidable, so in practice, may be of relatively low precision. They are also hard to scale to large or incomplete programs. Moreover, most work in this area has been done for C-like languages, and less for object-oriented languages.

In order to improve accuracy of program analysis, 'define for analyzability' approaches, such as ADDS and ASAP [14, 17], ask the programmer to explicitly describe some properties of data structures. They use the concept of dimension which is related to the depth in a linked list or a tree. ARTS's region concept is different, but the knowledge of regions may also be useful to allow more accurate program analysis and better performance, especially for cyclic data structures.

Type-Based Information Flow Security. ARTS is primarily designed to reason about reference reachability and confine reference cycles. But it turns out to have direct application in the area of information flow security. This is because the partially ordered acyclic regions are conceptually same to the classic lattice model of information flow control by Denning and Denning [8, 9].

Type systems have already been used to secure information flow within programs in a multi-level security system based on the lattice model. The general idea is that every type is associated with a particular security class [9, 24, 25]. Security classes, sometimes also called security groups, roles, users or principals, form a lattice which is a presumed finite set partially ordered by the level of security (high or low). Information flow operations, such as assignments, must respect the order of the security classes attached to types.

Some type systems, instead of associating types with a single security class, they allow multiple security classes to form a security property or label for each type [13, 20]. These access control list like security labels allow more complex security control and dynamic manipulation. The labels are pre-ordered rather than partially ordered. The correctness of information flow still relies on the order of predefined security classes.

ARTS is similar to these type systems in the way types are formed with regions corresponding security classes. However, the soundness of these systems depends on the predefined finite security classes which are assumed to be partially ordered, i.e., they do not actually check the correctness of security classes and their ordering. The type checking only ensures that programs respect the security policies embedded in the types. Besides the similar treatment of types, ARTS allows the programmer to specify desired security classes and the security level between them. These programmer-defined security classes are infinite and guaranteed to be partially ordered via type checking. This is generally a harder task for modular type checkers because global acyclic invariant is ensured by checking local reachability relations. We prove our type system is sound.

Most work on information flow security has been done in procedural languages, which mainly deal with primitive data types. They allow data in a low security class to be assigned to a variable in a high security class, but not vice versa. In the simple object-oriented language we present in this paper, we only have object types, and security classes are bound to class parameters. Because objects have fields, the security class has be to invariant for assignments. Otherwise, we could employ some simple covariant mechanisms such as declared covariant on class parameters like JFlow does [20] or more powerful mechanisms such as variant parametric types [18].

Region-Based Memory Management. Our notion of region is similar to that used in region-based memory management [22, 23, 11] 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 is based on lifetimes, that is, on which regions may outlive others. Instead our regions represent a static abstraction of strongly connected components in object graphs and focus on reference cycles. Although the region structures are different, 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.

7 Conclusion and Future Work

The major result of this paper is a class-based region-parametric 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 intractable, 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. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2004.
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. D. Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
6. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, 2001.
7. 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.
8. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
10. 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.
11. 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.
12. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
13. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.
14. L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 249–260, New York, NY, 1992. ACM Press.
15. 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.
16. 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.
17. J. Hummel, L. J. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *8th International Parallel Processing Symposium*, pages 208–216, Cancun, Mexico, 1994.

18. A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469. Springer-Verlag, 2002.
19. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *Programming Languages and Fundamentals of Programming*, 1999.
20. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
21. K. Rustan, M. Leino, and P. Müller. Object invariants in dynamic contexts. In *European Conference for Object-Oriented Programming (ECOOP)*, 2004.
22. 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.
23. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.
24. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
25. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
26. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.