# Region-Based Memory Management in Java

Morten V. Christiansen Per Velschow

May 29, 1998

**Abstract**

We present a Java-like language in which objects are explicitly put in regions. The language has constructs for allocating, updating and deallocating regions, as well as region types for objects. For this language we present a static semantics ensuring that well-typed programs use regions safely, and we present a dynamic semantics that is intentional with respect to a region-based store. We formulate and prove a soundness theorem stating that well-typed programs do not go wrong. Finally, we develop a concrete model for implementing regions, and we compare this model to garbage collection for small examples.

# Preface

When we first started working on this thesis, in January 1998, we expected to do a fairly programming intensive project. We expected to do a lot of experimenting with Java, and we had hoped to implement a prototype compiler for Java with region-based memory management.

As we began to get a notion of the difficulty of the task, our questions changed from "How can we do this?" to "Can this be done at all?". We then started an intensive investigation of how to define a subset of Java that uses regions explicitly.

After a couple of months we had designed the semantics of this language, RegJava, and we asked ourselves: "Is the system sound?". This proof of soundness turned out to be very time consuming, but also to be essential for the development of good semantics. We have torn down and rebuilt both the semantics and the proof more times than we care to remember. The semantics and the proof of soundness presented here have not sprung spontaneously into life like Athena from the brow of Zeus, but they are products of a long trial-and-error process. We could likely go on with this process for another six months if we had the time. There are lots of interesting notions and open problems we have not yet explored. All things must come to an end, though, and this is the point where we have had to stop. From the beginning of the project, it has been an important goal to live up to the curriculum at DIKU and finish this thesis within a 6 month period.

Our frequent discussions with our advisor, Fritz Henglein, have contributed much to this process of growth and exploration. He has provided many useful and interesting ideas, and while not all these ideas worked out well, they all helped increase our understanding.

We are also grateful to Henning Niss for his efforts with proofreading this thesis and suggesting improvements and clarifications.

# Contents

# Chapter 1

# Introduction

## 1.1 The Java language

Java is an object-oriented language developed by Sun Microsystems, and it is designed to be small, simple and portable across platforms and operating systems both at the source and at the binary level.

Java has a superficial similarity to C++, but the lack of a pointer type in Java (as well as the automatic memory management, of course) makes a world of difference in programming style.

Internet support is built into the Java language to the degree, that a running program may download runnable parts from the Internet as needed during execution. The two Internet browsers with the largest number of users, Microsoft's Internet Explorer and Netscape's Navigator, both support running Java programs (applets) in limited, secure environments. Most Java applets in use today are small and simple programs creating clever graphics, but the true platform-independence of Java combined with its modern design has convinced many that Java is the programming language of the future.

The strategy used for binary platform independence in Java is interesting. A Java Virtual Machine (JVM) has been designed, and rather than compiling to the underlying platforms, Java code is to be compiled to so-called bytecode. While JVM based solely on interpretation of bytecode do exist, the preferred strategy for executing JVM programs is to use JIT (Just In Time) compilers. JITs generate platform specific code for each method the first time the method is used. This reduces the interpretative overhead, and while Java programs still run at a slower pace than equivalent programs compiled directly to the particular platform, the overhead is not large and is getting smaller all the time.

## 1.2 Garbage collection

Garbage collection is a technique for automatically recovering memory for reuse by a program. A garbage collector is a separate routine in the runtime system that examines the data generated by the program and attempts to identify and reclaim memory used by data that the program can no longer access.

As a programmer it is a great relief not having to worry about memory management. Manual memory management is an extra burden on the programmer and a major source of programming errors, and it tends to make programs much harder to read and understand.

The designers of Smalltalk, Lisp and Prolog did not worry too much about the efficiency of their languages and incorporated garbage collection in them. This made programming easy, but the program execution slow. Sadly, in most real-world applications, efficiency was and is a major consideration.

Only in this decade have the machines become so fast and the garbage collection techniques so efficient, that it is normal to build garbage collection into "practical" languages including those designed in part for low-level systems programming (such as Modula-3 and Oberon). Compared to manual storage management, a modern garbage collector has an overhead of about 10% on high performance machines, according to [Wil94]. While this may still be unacceptable for some very special tasks, in most cases the gains in program quality and development time easily outweigh this cost. It seems possible that programs written in languages without automatic memory management will become, in time, as rare as programs written in assembler code are today.

Garbage collection, today, is an advanced discipline, so deciding what the best type of garbage collection is for a specific language or problem, is not obvious. Mark-and-sweep garbage collectors can recover memory without moving data. Generational garbage collectors can reduce the time spent marking data significantly. Copying garbage collectors give good locality of reference. Incremental collectors can give reasonable guarantees for real-time systems. All garbage collection techniques have two things in common, though:

- they assume that all dynamically reachable data is "live", and so cannot be deallocated.
- They make it hard to predict exactly *when* data is deallocated.

In the Java Language Specification, how to deallocate dead objects and reclaim memory is unspecified. For JVM it is specified that memory should be reclaimed by garbage collection. No further details are given. This leaves the implementers free to pick the garbage collectors suiting the implementation best using state-of-the-art technology. The Java garbage collector used in Sun's JDK **java** is a copying mark-and-sweep garbage collector running in its own thread. It is possible to force a garbage collection, and the garbage collector can be told not to run unless absolutely necessary. Still, for critical real-time programs, Java is not recommended.

## 1.3 Region-based memory management

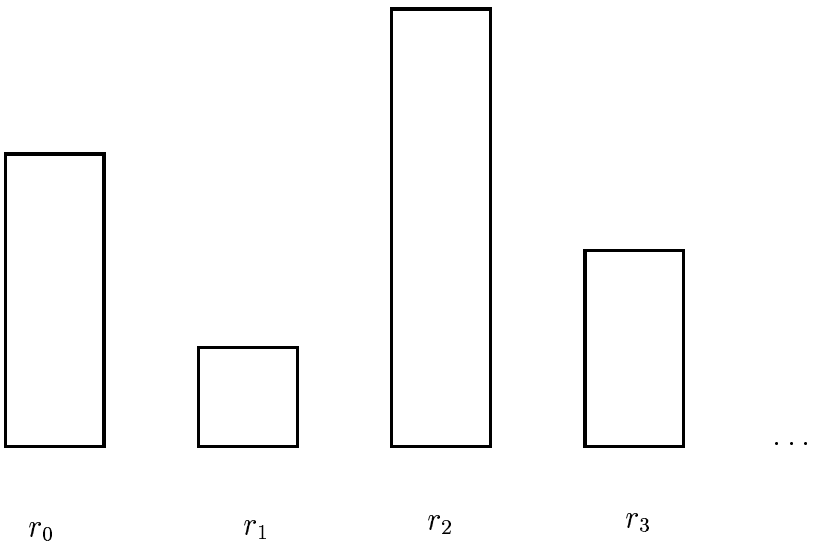
In 1994 Tofte and Talpin suggested a radically different method of memory management in [TT94], and this scheme was subsequently used in an implementation of ML: The ML Kit. This is described in [TT97] and [TBE+97].

The technique is called *region-based memory management*. Unlike garbage collection techniques, it has the major advantage that it does not require a complex and inefficient run-time system in order to reclaim memory. It can deallocate dynamically reachable data when it can be shown by a static analysis of the program that the data will not be used again after a certain point is reached. It has good real-time properties, insuring smooth execution of the program. For many programs, it has good locality of reference. The technique is far from perfect, but new refinements are still being added to it.

The major difference from garbage collection is that it is *statically* determined when data can be reused. There is no garbage collection process examining the generated data. The program itself deallocates data, often in huge chunks, in a single constant time operation.

In the region-based model, the memory is thought of as a stack of regions. Each region is also like an unbounded stack that grows in size until the entire region is popped off the region stack. The ML-Kit also supports resetting regions, i.e. reclaiming all data in a region without popping it off the region stack.

**Figure 1.1** The store is a stack of regions: every region is uniquely identified by a region name and is depicted by a box in the picture.




A region can contain pointers to values in itself or in other regions, both above or below it on the region stack. It is quite valid for pointers to point to regions that no longer exist ("dangling" pointers) if static analysis has shown that these pointers will not be used by the program.

A static analysis then determines at which points in the program it is necessary to allocate regions, what data to place in which regions, and at what point the region can safely be popped off the region stack. This information is then inserted into the program as annotations, and the necessary memory control commands are then inserted by the compiler at their proper place. The concepts is comparable to that of local variables in statically scoped languages, except, of course, that the creation and scope or regions should ideally be determined by an automatic tool rather than a programmer.

Allocating, deallocating and updating regions are all constant time operations, so the resulting program does not "stop" occasionally to handle memory management. This makes region-based memory management well-suited for real-time applications.

In principle, region annotations and region management should be as transparent to the programmer as garbage collection is. Reality is not so perfect. The first automatic region annotations of a program often are less than perfect, and may well have serious space leaks. Then the programmer has to rewrite the program (and perhaps insert manual resettings of regions) to achieve good results. On the other hand, these results then have more predictable behaviour and often use much less memory and run faster than equivalent programs that use garbage collection. Since this is still an emerging technology, it can be hoped that much of the manual program optimisation can be automated at a later point.

Using region-based memory management should also be considered in situations where memory is a limited resource (like smart-cards and household appliances). It is easy to predict exactly how much memory a given program will use (given its input), and there is no memory management overhead such as tagging or copying of data structures.

Eventually, memory management based on static analyses or dynamic garbage collection may not be an either-or proposition. Both methods have advantages and drawbacks, and perhaps it could be possible to combine the advantages of the techniques and minimise weaknesses.

## 1.4 This thesis

Given the current popularity of Java it seems obvious to consider applying region-based memory management to Java rather than garbage collection. This is the purpose of this thesis. We present a subset of Java with region annotations together with a static and a dynamic semantics. We prove a soundness theorem for the two semantics, and we discuss how to translate regular Java programs into the subset language. Last, we compare programs in this region-based model with equivalent programs that use garbage collection.

Using regions, Java might well become more suited for real-time applications. It also seems likely that a region based version of Java would be interesting in environments where memory is a limited resource. There already exist Java smart-cards, and Java has always been intended as a low-level appliance programming language.

We do not try to implement any of the static analysis that give good region annotations, nor do we try to expand our model to the full Java language. What we will try to do in this thesis, is to create the humble foundations upon which such ambitious projects can be built. While the focus of our study is Java, we expect that many ideas in this thesis would be useful for making region-based versions of any object-oriented, imperative language.

We expect the reader of thesis to be familiar with Java (or at least object-oriented languages in general) and at least somewhat acquainted with the region-based model as presented in [TT97, TT94, TBE+97].

# Chapter 2

# Region-Based Model

In this chapter we present our region-based model for implementing Java with regions. We focus on a small language subset of Java and present a rigorous study of this subset with respect to the model.

First, we derive our language subset from Java. This language is then further explored with respect to regions, and we develop a region annotated version called RegJava. A static semantics for RegJava is presented, and we discuss the underlying design choices for this semantics.

Second, we develop an abstract model for evaluating RegJava-programs, and we present an intentional dynamic semantics that uses regions explicitly for storing objects.

Last, we give directions for making a compiler that automatically transforms ordinary Java programs into RegJava programs.

## 2.1 Deriving a subset of Java

The purpose of this section is to derive from Java a language subset that is adequately interesting. The reason for treating a subset of Java instead of the entire language is simplicity. Our aim is to study the basic semantics of the language. Modern languages like Java have many language constructs with essentially equal underlying semantics. Therefore, in the interest of simplicity, it is only necessary to focus on language constructs that have essentially different semantics.

### 2.1.1 Design considerations

When choosing a subset of Java, we are faced with two main objectives. First, we want a language subset that is easy to reason about. We do not want to deal with too many non-orthogonal, syntactic quirks—except when they are essential to the semantics of the language.

Second, as far as possible, we want our derived language to be a *true* subset of Java. By this we mean that any syntactically valid program in our subset language should also be a syntactically valid Java-program. This implies that we do not change or rename any of the existing language constructs, such as the keywords and structure of Java.

These two objectives may contradict each other. We feel that the first objective is the most important one. Consequently, we have made a few syntactic changes in our subset language with the sole purpose of easing the following semantic treatment. We hope that

these differences from Java do not raise doubt about the relevance of our study with respect to Java. With the following discussion we hope to mediate any such doubt.

### 2.1.2 Selecting the major language features

As mentioned above, the subset language must be large enough to include as many essential features of Java as we want to deal with. The major essential features can be summarized as follows:

**Imperative** Java encourages use of strictly imperative languages constructs such as loops and destructive updates. Also, only first-order functions exist, i.e. functions cannot be used as values.

**Object-oriented** Java has object-oriented features such as sub-typing and encapsulation. Java uses dynamic dispatch when calling methods of objects.

**Simple values** Java is not completely object-oriented. In addition to objects, Java uses simple values for numbers, booleans and characters. Correspondingly, fields of objects and local variables can be of simple type.

**Class-based** Java is a class-based, object-oriented language. In the words of [AC96], Java's sub-typing rule is "inheritance-is-sub-typing". Each object is created as an instance of a particular class, and its type is induced from that class. A type is a subtype of another type only if the corresponding classes are subclasses. Java has only single inheritance but uses *interfaces* to implement a restricted kind of structural subtyping.

**Dynamic class loading** Classes are compiled separately and each put into separate class files. When running a Java-program, classes are loaded from these class files only when needed.

**Modularization** A Java-program is organized as a series of *packages* each declaring a series of classes. This kind of modularization is extremely low-level, and its main purpose is security.

**Concurrency** Java has built-in support for concurrent (multi-threaded) execution and for synchronisation.

**Exceptions** Java uses exceptions both explicitly and implicitly as a way of breaking the control flow, for example when an error has occurred.

**Arrays** Java has arrays that are themselves objects. The type of an array is a class type and it is a subtype of the biggest class type `Object`. Java has a peculiar but practical subtyping rule for array: if `A` is a subtype of `B`, then the array type `A[]` is also a subtype of the array type `B[]`. Array objects have a fixed size index range, and every indexing into the array is checked at runtime to assure that it is within the range.

**Automatic memory management** Java has no way of disposing objects. It is the task of the compiler or the run-time system to make sure that unused storage be reclaimed. The language specification[GJS96] hints that this should be done by means of a garbage collector. The JVM specification[LY97], however, demands that a garbage collector be present.

All of these major features are important when developing Java-programs in practice. However, treating all of these is beyond the ambition of this thesis.

First, we believe that the modularization feature has less importance from a semantical point of view. We believe that it is mainly important for security reasons. Also, it enables the programmer to organize his program into logical units. We choose not to include modularization in our language subset.

Second, dynamic class loading and separate compilation complicate the development of a static analysis. Static analyses can generally obtain stronger results when examining the whole program at once. We choose not to include dynamic class loading and separate compilation in our language subset, but we will later discuss how to reintroduce them in the language.

Third, concurrency complicates the application of static analyses even further. Little can be said about the behaviour of a concurrent program. We choose not to include concurrency in our language subset, but again we will later discuss how to reintroduce them in the language.

Fourth, arrays are a very important feature of Java. Because it does affect the notion of subtyping in an unsafe manner, it would not be wise to include arrays in this first treatment.

Last, exception handling is a rather advanced language feature that also complicates static analyses to a great extent. This is because it breaks the flow of control abruptly. We choose not to include exception handling in our language subset.

The remaining major features are all too important to be left out entirely. But a lot of the minor features of these can still be excluded with little or no implications on the generality of the language subset. We discuss this in the following sections.

### 2.1.3 Object-oriented features

Java has many interesting object-oriented features including the following

1. classes and sub-classing
2. abstract and final classes
3. interfaces
4. arrays
5. implicitly constructed strings
6. several levels of encapsulation (`private`, `protected`, `public`, and default)
7. abstract methods
8. class and instance fields and methods
9. constant fields
10. overloading of methods
11. inheritance, overriding and hiding of members
12. constructors
13. subsumption

Many of these features have interesting properties from a memory management point of view. However, it definitely is beyond the ambition level of this thesis to treat them all in detail. The application of the region-based memory model to an object-oriented language is, to our knowledge, new. Therefore, we restrict our attention to the most basic features of object-oriented programming in Java. We discuss how to treat the remaining features in chapter 5.

We include the notion of classes and sub-classing since they are central the type system and to the way objects are constructed in Java. We also study the ability to do subsumption of objects.

Update-able instance fields is also a basic feature we wish to include, but we study neither constant fields nor class fields, and we do not allow for hiding of fields.

Likewise, we study public instance methods but not class methods, abstract methods, private methods, or protected methods. Overriding and inheritance of methods are also included in our subset language, but we do not allow for hiding of methods. Our methods have exactly one formal parameter and always return a value.

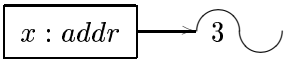
We do not include constructors since they can be easily implemented by defining instance methods instead and calling these after the creation of the object.

## 2.2 Abstract memory model

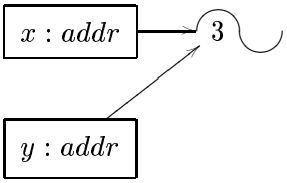
Before designing a region-annotated version of Java, we need to understand just in what ways the memory should be used. Without such understanding, it is impossible to decide in what conditions it is necessary to specify that regions should be used or allocated. Two candidate models have emerged.

### 2.2.1 The reference-based memory model

In the traditional implementations of functional languages like Lisp or Scheme all variables are in principle *references* to value cells in memory. If a value contains the value three, it means that someplace denoted by that variable is an address of a cell containing the value of three.




Values are normally not updated (at least not when the languages are used in a functional style). When a variable is copied to another, it is the address of the value, that is copied.




Expressions create their values somewhere in memory, and return the address of this place.

A "pure" reference-based model is clearly unsuitable for Java because it is essential that all simple types be distinct. Changing one integer variable, for instance, must not affect the values of other integer variables or fields.

A solution is to explicitly dereference simple types, copy them to new memory areas and return those addresses whenever variables of simple type are copied into each other. This is

not aesthetically pleasing, however. It also seems wasteful, that every simple type variable requires two memory values, the address and the value itself.

For objects, on the other hand, the reference model is perfect. An object variable *should* contain the address of an object and a **new** statement should allocate the object in memory and return the address of the object.

### 2.2.2 The variable-based memory model

In imperative languages like Fortran or C, things are different. Variables generally contain the values they refer to, rather than addresses of such values.


When one variable is assigned to another, an actual copying of the value is done.



Expressions evaluate, not to addresses, but to the relevant values themselves.

This is clearly the preferable model for simple types. Not only are the variables copied directly, no explicit value cells are required to be created by expressions generating simple values.

For objects, however, we *need* the indirection. If one object variable is assigned to another, they should refer to the same object, not independent copies. A solution is to let the **new** statement generate object-pointers as well as the objects themselves, and then treat the object variables as pointer variables.

### 2.2.3 Choosing the model

We choose to use the variable-based memory model. This means that only objects are put into regions, and not simple values. The choice seems natural for Java because only objects are created dynamically. If Java is compiled into byte-code and run on a JVM, then simple values and objects references are put on the run-time stack and objects are created on the heap.

## 2.3 RegJava

In this section we present a small Java-like language that use regions explicitly. The types of the language are also discussed.

### 2.3.1 Programs

Since we do not consider packages, programs in RegJava are the basic compilation unit. A program consists of a series of class definitions. The order of the class definitions are not important, i.e. the order cannot affect the semantics of the program.

### 2.3.2 Region variables and the letregion statement

Before going into details about the syntax and semantics of the classes, expressions and statements of RegJava, we introduce the concept of *region variables*. As presented in chapter 1, regions are created and destroyed in a stack-like manner according to the static structure of the program. This is very similar to the way variables on a run-time stack are created and destroyed, and regions are therefore formally represented by special region variables. We normally use $\rho$ to range over region variables.

Region variables, however, differ from ordinary variables, in that they are never assigned a value directly. Informally, the content of a region variable can be regarded as a pointer or a name of a concrete region in the store.

Much like ordinary variables, a new region variable can be introduced in the program by at declarative statement called letregion. For instance, the following statement declares a region variable named $\rho_1$:

$$\text{letregion } \rho_1 \text{ in } \{ \ s \ \}$$

The scope of $\rho_1$ is exactly the statement $s$. The letregion statement not only declares the region variable but also creates a new region and assigns it name to $\rho_1$. This region can then be used in the statement $s$ and is destroyed when execution of $s$ finishes, hereby reclaiming all the storage owned by that region. Thus, the lifetime of the region denoted by $\rho_1$ is corresponds exactly to the scope of $\rho_1$.

Although the letregion statement is the only way to create new regions, there are other ways to introduce new region variables in the program. Similar to the way ordinary parameters of methods are declared, it is possible to declare formal region parameters of both methods and classes. We discuss this as we introduce the rest of RegJava in the following sections.

### 2.3.3 Classes

#### A basic class example

Consider first the following definition of the simple Java-class $A$:

```
class A {
  A a;
  A set(A na) { a=na; return this; }
}
```

Now we want to show a region annotated version of this class. As discussed in section 2.6 there exist many legal annotations of a Java-program. Let us first present a naive annotation and then later refine it:

```
class A[ρ1] at ρ1 extends Object[ρ1] {
  A[ρ1] a;
  A[ρ1] set(A[ρ1] na) { this.a = na; return this }
}
```

In Java the class $A$ is an implicit descendant of the built-in class *Object*. For simplicity, RegJava only has one syntax for defining classes. So we always make the direct superclass

explicit. Notice also, that in RegJava field selection always mentions the object explicitly. In Java, the expression $a$ in the body of the method *set* is actually just syntactic sugar for the expression $\mathsf{this}.a$ in which the object is made explicit.

The region variable $\rho_1$ occurring in brackets after the class name $A$ is called a *formal region of the class*. (There can be more than one formal region parameter of the class. We get back to this below.) When instances of class $A$ is created, e.g. by the expression

$$\mathsf{new}[\rho_{17}]\ A()$$

an actual region denoted by the region variable $\rho_{17}$ is bound to the formal region parameter $\rho_1$, and the object is created in this region.

In RegJava the class *Object* is only used as an abstract ancestor of all classes. That is, the class *Object* has neither fields nor methods. However, instances of class *Object* must include information about in which region in the store the object exist. Therefore we pass the formal region $\rho_1$ of the subclass $A$ to the superclass *Object*.

All types in the class are also annotated with regions. This specific annotation dictates that objects pointed to by the field $a$ be instances of class $A$ and lie also in region $\rho_1$. Also, the method *set* only accepts arguments and only returns values of class $A$ in region $\rho_1$. This way it is ensured that the field update in the method is valid.

Although the class $A$ is extremely simple, there are other legal ways to annotate the above class. The above annotation put objects referenced by the field $a$ in the same region as the enclosing object. This is not really necessary. Instead, we can introduce a new formal region parameter $\rho_2$ and put the field in this region:

$\mathsf{class}\ A[\rho_1, \rho_2]\ \mathsf{at}\ \rho_1\ \mathsf{extends}\ Object[\rho_1]\ \{$
  $A[\rho_2, \rho_2]\ a;$
  $A[\rho_1, \rho_2]\ set(A[\rho_2, \rho_2]\ na)\ \{\ \mathsf{this}.a = na;\ \mathsf{return\ this}\ \}$
$\}$

We say that class $A$ is *spread* over multiple regions. Notice, that this spreading also affects the type of the argument to the method *set* because of the field update in the body of the method.

### A more complex example

Consider the three example Java-classes in figure 2.1 on the following page. The classes represent many of the basic class-based features of Java that we wish to include in RegJava: subclassing, overriding of methods, and simply-typed and object-typed fields. One purpose of this example is to show how the annotation of subclasses may affect the annotation of superclasses.

We start by looking only at class $B$ and $P$. That is, we completely disregard the existence of class $R$. A reasonable way to annotate class $B$ and $P$ is shown in figure 2.2 on the next page. Notice, in particular, the new syntax for declaring methods. The region variable $\rho_2$ occurring in the brackets after the method name *move* is called a *formal region parameter of the method*. The idea is that since the method *move* of class $P$ only uses its argument for reading, the argument can be placed in any region. We say that the method is *region polymorphic* since it can be used with any region bound to $\rho_2$.

The region variable $\rho_2$ is called a parameter because the method gets passed an actual region when invoked:

$$b.move[\rho_{17}](p)$$

**Figure 2.1** Three example classes in Java. $B$ represents abstract geometrical bodies. $P$ implements 2-dimensional points. $R$ implements rectangles by storing two opposite corners.

```
class B {
  B move(P p) { return this; }
}

class P extends B {
  int x;
  int y;
  B move(P p) { x=x+p.x; y=y+p.y; return this; }
}

class R extends B {
  P p1;
  P p2;
  B move(P p) { p1.move(p); p2.move(p); return this; }
}
```

**Figure 2.2** Annotated version of class $B$ and $P$ from figure 2.1.

```
class B[ρ1] at ρ1 extends Object[ρ1] {
  B[ρ1] move[ρ2](P[ρ2] p) { return this }
}

class P[ρ1] at ρ1 extends B[ρ1] {
  int x;
  int y;
  B[ρ1] move[ρ2](P[ρ2] p) {
    this.x = this.x + p.x; this.y = this.y + p.y; return this }
}
```

Here $b$ is a reference to an instance of class $B$ in any region and $p$ is a reference to an instance of class $P$ in the region denoted by $\rho_{17}$. The region variable $\rho_{17}$ is called the *actual region argument* and is passed to the method at run-time. Of course, it is possible to have methods with more than one formal region parameter.

Now, let us consider what happens when we try to annotate also the class $R$. In general, we want to spread classes over many regions. For class $R$ this means that we would like to be able to put the fields $p_1$ and $p_2$ in regions different from the one used for the enclosing class. Unfortunately, the annotation of class $R$ in figure 2.3 is not sound. The reasons for this will

**Figure 2.3** Annotating class $R$ from figure 2.1

```
class R[ρ1, ρ3, ρ4] at ρ1 extends B[ρ1] {
  P[ρ3] p1;
  P[ρ4] p2;
  B[ρ1] move[ρ2](P[ρ2] p) {
    this.p1.move[ρ2](p); this.p2.move[ρ2](p); return this }
}
```

become clear in the following sections. At this point we will just say that the problem arises because of the possibility of subsumption and dynamic dispatch when invoking methods.

In figure 2.4 we have shown a sound way to annotate all of the classes from figure 2.1 on the page before. At a first glance, it seems that this annotation is not much difference from

**Figure 2.4** Annotated version of all the classes from figure 2.1.

```
class B[ρ1, ρ3, ρ4] at ρ1 extends Object[ρ1] {
  B[ρ1] move[ρ2](P[ρ2] p) { return this }
}

class P[ρ1] at ρ1 extends B[ρ1, ρ1, ρ1] {
  int x;
  int y;
  B[ρ1] move[ρ2](P[ρ2] p) {
    this.x = this.x + p.x; this.y = this.y + p.y; return this }
}

class R[ρ1, ρ3, ρ4] at ρ1 extends B[ρ1, ρ3, ρ4] {
  P[ρ3] p1;
  P[ρ4] p2;
  B[ρ1] move[ρ2](P[ρ2] p) {
    this.p1.move[ρ2](p); this.p2.move[ρ2](p); return this }
}
```

the earlier annotations. The important difference is in the formal region parameters of class $B$ that have changed due to the annotation of class $R$.

### 2.3.4 Object types

In Java, objects are dynamically generated and put on the heap. The program interacts with objects through *object references*. The type of an object reference is the name of a class declared in the program. Due to subsumption the actual object referenced can be an instance of that class or of any subclass, and the class is the *actual* type of the object.

#### Structural notation

In order to simplify the following semantic treatment, we want to notate the type of an object reference in RegJava purely structural. Often it is only necessary to know that an object has a field or a method of a specific name and type.

In [AC96] the notation of object types is purely structural and is completely independent from class-based features. Our notation is based on that work with some changes and restrictions. In [AC96] there is no distinction between fields and methods; objects have only methods. Since their methods are update-able, fields are just a special kind of method.

However, in Java there is an important distinction between fields and methods. Only fields can be updated. The object types of RegJava should reflect this difference. Furthermore, simple values do not have object types, so we have a distinction between simple types and object types.

Consider an object with an instance field named $x$ of some type $\tau$ and an instance method named $m$ that has method type $\phi$ (also called a method *signature*). The type of such an object is written:

$$[x : \tau \,|\, m : \phi]@\rho$$

Here we have clearly separated the fields and methods of the object by a vertical bar within the brackets. We explain the @-notation in the next section. In general, the type of an object with $n$ fields and $k$ methods is written:

$$[x_i : \tau_i{}^{\,i \in 1 \ldots n} \,|\, m_i : \phi_i{}^{\,i \in 1 \ldots k}]@\rho$$

The order of the fields and methods is not important; two object types are considered equal if they are syntactically identical after a possible reordering of fields and methods.

#### Region annotated types

In order to know when to allocate and deallocate regions, the type of an object reference is refined to include information about the region in which the referenced object exist. This is done using using the @-notation showed in the previous section:

$$[x : \tau \,|\, m : \phi]@\rho$$

When an object reference has this type, it means that it points to an object in the region denoted by the region variable. Given region annotated types for every object reference in the program, it is possible to know when to allocate an deallocate regions at run-time. The basic idea is that the above object type is only legal within the scope of the region variable $\rho$. This scope should therefore correspond to the lifetime of the region bound to $\rho$ at run-time.

### Recursive object types

Consider the following RegJava class:

```
class A[ρ1] at ρ1 extends Object[ρ1] {
  A[ρ1] a;
}
```

With the notation for object types defined so far, it is not possible to notate a type for instances of the class $A$ because the class name $A$ itself occurs within the object. In order to do this we assume a global recursively defined environment $\Pi$ that stores type information for every class in the program. For class $A$ the environment is defined by the following equation:

$$\Pi(A, \rho_1) = [A : \Pi(A, \rho_1)]@\rho_1$$

The environment abstracts over the class names in the class definition. Since we want to be able to store different instances of class $A$ in different regions, $\Pi$ also abstracts over the region variables in the class. So the general form for a class $A$ is

$$\Pi(A, \vec{\rho}) = [x_i : \tau_i \,^{i \in 1 \ldots n} \,|\, m_i : \phi_i \,^{i \in 1 \ldots k}]@\rho$$

where $\vec{\rho} = (\rho_1, \ldots, \rho_p)$ is a vector of free region variables in the object type on the right-hand side (possibly including $\rho$), and $A$ (or any other class name declared in $\Pi$) can occur free in the types $\tau_i$ and signatures $\phi_i$.

Since we construct $\Pi$ as a global environment, there is really no need to repeat $\Pi$ in every recursive occurrence of an object type. Instead, we simply write $A[\vec{\rho}]$ to mean $\Pi(A, \vec{\rho})$.

### Examples of object types

Given region variable $\rho_1$, the instances of class *Object* in the region denoted by $\rho_1$ by default have the empty type $[\,]@\rho_1$. That is, in the global environment $\Pi$ we have

$$Object[\rho] \;=\; [\,]@\rho$$

Consider, again, the classes $B$ and $P$ in figure 2.2 on page 16. Given region variable $\rho_1$, instances of class $B$ in the region denoted by $\rho_1$ has the type $[move : \forall\rho_2.P[\rho_2] \to B[\rho_1]]@\rho_1$, and in the global environment $\Pi$ we have

$$B[\rho_1] \;=\; [move : \forall\rho_2.P[\rho_2] \xrightarrow{\{\rho_1,\rho_2\}} B[\rho_1]]@\rho_1$$

Likewise, we have the following for class $P$:

$$P[\rho_1] \;=\; [x : \mathsf{int}, y : \mathsf{int} \,|\, move : \forall\rho_2.P[\rho_2] \xrightarrow{\{\rho_1,\rho_2\}} B[\rho_1]]@\rho_1$$

We discuss how to interpret the method signatures in section 2.3.5.

#### Equivalence of recursive types

By allowing object types to be expressed using the name of a class, we can write the type of instances of a the class $A$ mentioned above as

$$A[\rho_1]$$

This type is equivalent to the structural notation

$$[a : A[\rho_1]]@\rho_1$$

which in turn is equivalent to

$$[a : [a : A[\rho_1]]@\rho_1]@\rho_1$$

and so forth. Clearly, all of these types should be considered equivalent.

### 2.3.5 Method signatures

In the previous section we mentioned that a method in a class has a method signature $\phi$. In Java a method signature consists of the name of the method and the number and types of formal parameters to the method[GJS96].

#### Region-polymorphic signatures

Method signatures in RegJava essentially has the same functionality as in Java, but they act more like ordinary types in that the name of the method is not included. Since overloading of methods is not part of RegJava, the name of a method uniquely identifies a method within a specific class.

Consider, again, the type $P[\rho_1]$ from section 2.3.4:

$$P[\rho_1] \quad = \quad [x : \mathsf{int}, y : \mathsf{int} \,|\, move : \forall\rho_2.P[\rho_2] \xrightarrow{\{\rho_1,\rho_2\}} B[\rho_1]]@\rho_1$$

Here the method *move* has the signature $\forall\rho_2.P[\rho_2] \xrightarrow{\{\rho_1,\rho_2\}} B[\rho_1]$. We say that method *move* is region polymorphic in region $\rho_2$. The interpretation of this signature is: Given any actual region for $\rho_2$, when the method is invoked with actual argument $p$ of class $P$ in the region denoted by $\rho_2$, it returns a value of class $B$ in the region denoted by $\rho_1$. Furthermore, the regions in the set $\{\rho_1, \rho_2\}$ must be *live* when invoking the method.

A method signature has the general form

$$\forall\vec{\rho}.\tau_1 \xrightarrow{\Delta} \tau_2$$

where $\vec{\rho} = (\rho_1, \ldots, \rho_n)$ is a vector of region variables binding some of the region variables occurring free in $\tau_1$, $\tau_2$, and $\Delta$. $\Delta$ is a set of region variables.

### Examples of method signatures

Consider, again, the class $R$ from figure 2.1 on page 16. We stated that the annotation in figure 2.3 on page 17 together with the annotations of $B$ and $P$ is not sound. Consider the type resulting from the annotation of $R$ in figure 2.3:

$$R[\rho_1, \rho_3, \rho_4] \quad = \quad [p_1 : P[\rho_3], p_2 : P[\rho_4] \,|\, move : \forall \rho_2 . P[\rho_2] \xrightarrow{\{\rho_1, \rho_2, \rho_3, \rho_4\}} B[\rho_1]]@\rho_1$$

This type states that when calling the method *move*, all regions $\rho_1$, $\rho_2$, $\rho_3$ and $\rho_4$ must be alive. However, consider the following program:

```
letregion ρ1 in {
  let b : B[ρ1] = null in {
    letregion ρ3, ρ4 in {
      b = new[ρ1, ρ3, ρ4] R();
      b.p1 = new[ρ3] P();
      b.p2 = new[ρ4] P()
    };
    b.move[ρ2](p)
  }
}
```

The program expects $p$ to be declared somewhere else with type $P[\rho_2]$. Given the above type for instances of class $B$ this program is legal, but it will have unpredictable behaviour because of invoking the method *move* outside the scope of the region variables $\rho_3$ and $\rho_4$. Therefore, we have to change the type for $B$ to demand that those regions be in scope when invoking *move*:

$$B[\rho_1, \rho_3, \rho_4] \quad = \quad [move : \forall \rho_2 . P[\rho_2] \xrightarrow{\{\rho_1, \rho_2, \rho_3, \rho_4\}} B[\rho_1]]@\rho_1$$

With this type the above program is *not* legal. However, by moving the method invocation inside the inner letregion we get a perfectly legal and sound program:

```
letregion ρ1 in {
  let b : B[ρ1] = null in {
    letregion ρ3, ρ4 in {
      b = new[ρ1, ρ3, ρ4] R();
      b.p1 = new[ρ3] P();
      b.p2 = new[ρ4] P();
      b.move[ρ2](p)
    }
  }
}
```

Notice, that the type of $P$ is not affected by the changes in $B$ and $R$. This is because the body of the method *move* in class $P$ uses the same regions as that of class $B$.

### Equivalence of signatures

Two method signatures are considered equivalent if they are syntactically identical after a possible renaming of the region variables bound in the signatures.

### 2.3.6 Simple values and simple types

Java has simple values for numbers, characters and booleans. The type of a simple value is one of the simple types `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`.

#### Simple values

Because we use the variable model (see section 2.2), all simple values are represented unboxed. So, from a memory management point of view, all simple values are treated similarly.

We choose to only deal with integer and boolean values. Integers are needed mainly because many interesting programs require some kind of computation. Logical values are needed since we want to be able to express conditional statements.

This restriction is only done for the sake of making the language syntactically small. The remaining types can easily be added to the subset language without affecting our semantic treatment.

#### Simple types

Since we choose to represent integers and booleans unboxed, they are not put into regions. Therefore, integers and booleans just have the simple types int and boolean.

### 2.3.7 Local variables and the let statement

The Java Language Specification uses the term *variable* for any kind of storage location. This includes fields, formal parameters, array components and local variables. We only use the term variable to mean the local variables in a method or the formal parameters of a method.

#### Variable declaration syntax

Java has a block structured syntax. The scope of a local variable declared in a block is the rest of the block. This scoping rule is central to our treatment since a reference to an object can vanish when exiting from a block.

However, we put a restriction on where variable declarations can occur. In our subset they are only allowed to occur in the beginning of a block. This is only a syntactic restriction since variable declarations in the middle of a block actually imply an invisible block starting from just before the declaration and extending throughout the rest of the block. That is, the following block in Java

```
{
  <s1>
  int i = 0;
  <s2>
}
```

has the same semantics as the following nested blocks

```
{
  <s1>
  {
    int i = 0;
    <s2>
  }
}
```

Moreover, we adopt a quite different syntax for local variable declaration. Our syntax resembles that of let-bindings in functional languages such as ML:

$$\mathsf{let}\ x : \tau = e\ \mathsf{in}\ s$$

where $x$ is an identifier, $\tau$ is a type, $e$ is any expression and $s$ is any statement.

It is important, however, to note that this statement does declare an update-able variable. The innermost block of the above Java example can easily be translated into the corresponding let statement:

$$\mathsf{let}\ i : \mathsf{int} = 0\ \mathsf{in}\ \{\texttt{<s2>}\}$$

### 2.3.8 Blocks and statements

#### Loops

The ability to express iterative algorithms in terms of loops is central to imperative programming languages. Java has three different loop statements: `while`, `do` and `for`. We choose to only include the while statement since this is the most general of the three—the others can easily be implemented by a while statement. Our while statement has the following form

$$\mathsf{while}\ e\ \mathsf{do}\ s$$

where $e$ is any expression and $s$ is any statement.

#### Conditionals

There are two conditional statements in Java: `if-then` and `if-then-else`, and there is one `switch` statement. We choose to only include the `if-then-else` since it can efficiently implement the other two statements. Our conditional statement has the following form

$$\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$$

where $e$ is any expression and $s_1$ and $s_2$ are any statements.

#### Abrupt completion of statements

In Java a statement can be abruptly completed by means of either a sub-statement or sub-expression raising an exception or one of the three statements `break`, `continue` or `return`.

As already mentioned, we do not have exceptions in RegJava. Furthermore, we choose to exclude both the `break` and `continue` statement since they can both be implemented by rewriting the statement in which they occur—although this is not always trivial. From a

programming point of view we do not find them to be very important, but they could cause problems for a static analysis because of the way they break the block structure of a program.

However, we want to treat methods returning a value. We therefore include the `return` statement, but only the one that returns a value. We do not pose any restrictions on where to place the `return` statement. Our return statement has the following form

$$\mathsf{return}\ e$$

where $e$ is any expression.

Since the return value is either a simple value or an object reference there is no need to assume anything about where to put the return value. If the return value is an object reference, the return type of the method is an object type that tells in which region the referenced object lies.

#### Sequences and the empty statement

In Java a semicolon is used to mark the end of a statement. We adopt a slightly different syntax inspired by the language Pascal in which semicolon is used as a delimiter between statements. So a sequence in our subset language has the following form

$$s_1; s_2$$

where $s_1$ and $s_2$ are any statements.

We also include the empty statement. This is necessary since we have only included the `if-then-else` conditional statement. The empty statement is syntactically comprised of the empty string. In the formal grammar and semantic rules the empty string is represented by the meta-symbol $\varepsilon$.

#### Blocks

Java has a block structured syntax, and blocks define the scope of a variable declaration. As mentioned, in RegJava this scoping of variables is enforced by the let statement. However, for purely syntactical reasons we choose to include blocks as in Java. A block has the following form

$$\{s\}$$

where $s$ is any statement.

### 2.3.9 Expressions

Java's expressions can roughly be divided into two categories. First, there are the simple expressions such as assignment and addition. Second, there are a number of expressions related to the object-oriented features of Java discussed in section 2.1.3.

#### Simple expressions

These include expressions for variable access, variable assignment, and arithmetic and logical expressions using one of several unary and binary operators. Many of these are not interesting from a memory management point of view. Therefore, we choose to only include a limited

number of these in our subset. We only include variable access, variable assignment, addition, and equality expressions. These expressions have the following forms

$$
\begin{aligned}
&x \\
&x = e \\
&e_1 + e_2 \\
&e_1 == e_2
\end{aligned}
$$

where $x$ is a variable, and $e_1$ and $e_2$ are any expressions. In addition to these simple expression we include integer constant, the boolean constants false and true, and the special object reference null. The latter has a special type Null that is a kind of "bottom" element in the lattice of object types since it is a subtype of any object type. Thus, null can be assigned to a variable or field of any object type.

#### Object expressions

We include expressions for field selection, field update, method invocation, and object construction. These expressions have the following forms

$$
\begin{aligned}
&e.x \\
&e_1.x = e_2 \\
&e_1.m[\vec{\rho}](e_2) \\
&\mathsf{new}[\vec{\rho}]\ A()
\end{aligned}
$$

where $e_1$ and $e_2$ are any expressions and $x$, $m$ and $A$ are identifiers.

In the preceding discussion of how classes and methods are defined we have already touched on these expressions. The expressions for field selection and field update are exactly as in Java, the only exception being that the object in a field selection is never implicit. The vector of region variables $\vec{\rho}$ is, of course, just a short-hand notation for the actual region arguments of methods and classes.

### 2.3.10 Grammar of RegJava

We conclude this part by presenting the subset of Java that we argued for in the preceding sections. We do this by giving the complete grammar of RegJava in figure 2.5 on the next page. We should make a few notes about the language generated by this grammar.

First, the grammar implicitly declares a number of *keywords*: class, at, extends, let, in, if, then, else, return, letregion, while, do, this, false, true, null, and new. The program is not allowed to use these names for variables, methods and classes.

Second, since all methods in RegJava return a value, it must be assured that the execution of a method body can only exit by means of a return statement. In Java a conservative flow analysis is performed on the method body to ensure this. We accept only RegJava programs that can pass this analysis.

## 2.4 Static semantics

In section 2.3 we presented the core language, RegJava, that we wish to study. We introduced the syntax of the language together with an informal explanation of the way regions and

**Figure 2.5** Grammar of RegJava

| | | | |
|---|---|---|---|
| $\mathcal{P}$ | ::= | $\mathcal{C}_1 \cdots \mathcal{C}_n$ | program |
| $\mathcal{C}$ | ::= | class $A[\vec{\rho}]$ at $\rho$ extends $B[\vec{\rho}]$ $\{b_i{}^{i \in 1 \ldots n}\}$ | class definition |
| $b$ | ::= | $\tau_2\ m[\vec{\rho}](\tau_1\ x)\ \{s\}$ | method definition |
| | \| | $\tau\ x;$ | field definition |
| $s$ | ::= | $\epsilon$ | empty statement |
| | \| | $e$ | statement expression |
| | \| | $\{s\}$ | block |
| | \| | $s_1; s_2$ | sequence |
| | \| | let $x : \tau = e$ in $s$ | variable declaration |
| | \| | if $e$ then $s_1$ else $s_2$ | conditional |
| | \| | return $e$ | return |
| | \| | letregion $\rho$ in $s$ | letregion |
| | \| | while $e$ do $s$ | while loop |
| $e$ | ::= | $x$ | variable |
| | \| | this | self parameter |
| | \| | $i$ | integer constant |
| | \| | false | boolean constant |
| | \| | true | boolean constant |
| | \| | null | null constant |
| | \| | $(e)$ | parenthesize |
| | \| | $e_1 + e_2$ | addition |
| | \| | $e_1 == e_2$ | equality |
| | \| | $e.x$ | field selection |
| | \| | $e_1.m[\vec{\rho}](e_2)$ | method invocation |
| | \| | $e_1.x = e_2$ | field update |
| | \| | $x = e$ | variable assignment |
| | \| | new$[\vec{\rho}]\ A()$ | object construction |

types are used. In this section we elaborate further on this. We present the semantic objects involved in the description, and we present a static semantics by a set of inference rules. The reader should consult with the grammar of RegJava in figure 2.5 on the preceding page.

### 2.4.1 Semantic objects

When describing the static semantics we refer to the semantic objects defined by the equations in figure 2.6. Many of these have already been introduced in section 2.3. However, we adopt

**Figure 2.6** The semantic objects of the static semantics.

$$
\begin{array}{rcl}
x \text{ or } m & \in & \text{Var} \\
i & \in & \text{Integer} \\
 & & \text{Boolean} = \{\mathsf{false}, \mathsf{true}\} \\
\rho \text{ or } p & \in & \text{RegVar} \\
A & \in & \text{ClassName} \\
\tau & \in & \text{Type} = \text{SimpleType} \cup \text{NullType} \cup \text{ObjType} \cup \text{ClassType} \\
 & & \text{SimpleType} = \{\mathsf{int}, \mathsf{boolean}\} \\
 & & \text{NullType} = \{\mathsf{Null}\}
\end{array}
$$

$$
\begin{array}{rcl}
[x_i : \tau_i{}^{i\in 1\ldots k} \mid m_i : \phi{}^{i\in 1\ldots \ell}]@\rho & \in & \text{ObjType} = \\
 & & \cup_{k\geq 0}(\text{Var} \times \text{Type})^k \times \cup_{\ell\geq 0}(\text{Var} \times \text{Signature})^\ell \times \text{RegVar} \\
\phi \text{ or } \forall\vec{\rho}.\tau_1 \xrightarrow{\Delta} \tau_2 & \in & \text{Signature} = \cup_{k\geq 0}\text{RegVar}^k \times \text{Type} \times \text{LiveReg} \times \text{Type}
\end{array}
$$

$$
\begin{array}{rcl}
\Delta & \in & \text{LiveReg} = \text{Fin}(\text{RegVar}) \\
E & \in & \text{TyEnv} = \text{Var} \xrightarrow{\text{fin}} \text{Type} \\
\Pi & \in & \text{ProgType} = \text{ClassName} \xrightarrow{\text{fin}} \text{ClassType} \\
\Gamma & \in & \text{ClassType} = \cup_{k\geq 0}\text{RegVar}^k \xrightarrow{\text{fin}} \text{ObjType}
\end{array}
$$

a couple of shorthand notations for object types. We sometimes leave out the methods as in $[x_i : \tau_i{}^{i\in 1\ldots n}]@\rho$ or the fields as in $[m_i : \phi{}^{i\in 1\ldots n}]@\rho$ when only one of these are important in a given context.

As mentioned in section 2.3.4, recursive object types are introduced using named class types. The class types of a program is described by an environment $\Pi$. We also call $\Pi$ the *type* of the program. The program type $\Pi$ maps class names to an abstraction $\Gamma$:

$$
\begin{array}{rcl}
\Pi(A) & = & \Gamma \\
\Gamma(\rho_1, \ldots, \rho_n) & = & \tau
\end{array}
$$

When $\Pi$ is given from the context, we write $A[\rho_1, \ldots, \rho_n]$ as a short-hand for $\Pi(A)(\rho_1, \ldots, \rho_n)$.

We call $\Gamma$ the type of the class, and it is essentially a map from tuples of region variables $\vec{\rho}$ to object types. Thus, for a class $A$ the class type $\Gamma$ describes the following relationship:

$$A[\rho_1, \ldots, \rho_n] \quad = \quad [x_i : \tau_i{}^{i \in 1 \ldots k} \mid m_i : \phi^{i \in 1 \ldots \ell}]@\rho$$

Object types are considered equivalent if they are syntactically identical after a possible reordering of their members. Further, for recursive object types containing types of the form $A[\vec{\rho}]$ every *unfolding* of these, i.e. replacing $A[\vec{\rho}]$ with the corresponding object type, are considered equivalent.

$\Delta$ is a finite set of region variables. Since it is used in the static semantics to describe what regions are live within a certain scope, we often just refer to $\Delta$ as the live regions.

$E$ is a type environment mapping variables to types. If $x$ is a variable and $\tau$ is a type, then $E' = E + \{x : \tau\}$ is the extended type environment with $\mathrm{Dom}(E') = \mathrm{Dom}(E) \cup \{x\}$, $E'(x) = \tau$, and $E'(y) = E(y)$ for all $y \in \mathrm{Dom}(E)$ with $y \neq x$.

We sometimes need to explicitly substitute the region variables of a type for other region variables. We use $S$ to stand for such a substitution, and we write $\{\rho_1 \mapsto \rho_1', \ldots, \rho_n \mapsto \rho_n'\}$ for the substitution that substitutes $\rho_i$ with $\rho_i'$ for each $i \in 1 \ldots n$.

The region variables of an object type that is not bound in the method signatures are called *free* region variables. We write $\mathrm{frv}(\tau)$ for the set of free region variables in the type $\tau$. This notation extends to type environments: $\mathrm{frv}(E)$. If $\vec{\rho} = (\rho_1, \ldots, \rho_n)$ then we write $\mathrm{frv}(\vec{\rho})$ to stand for the set $\{\rho_1, \ldots, \rho_n\}$.

### 2.4.2 The subtype relation

In this section we define the subtype relation. If $\tau$ and $\tau'$ are types, we write $\tau' <: \tau$ if $\tau$ is a subtype of $\tau'$. Thus, the subtype relation is a subset of the set Type $\times$ Type. It is a *reflexive* and *transitive* relation.

The subtype relation is defined by the set of inference rules in figure 2.7. The rules

**Figure 2.7** The subtype relation.

(RegSub Refl)
$$\frac{}{\vdash \tau <: \tau}$$

(RegSub Trans)
$$\frac{\vdash \tau_1 <: \tau_2 \qquad \vdash \tau_2 <: \tau_3}{\vdash \tau_1 <: \tau_3}$$

(RegSub Null)
$$\frac{}{\vdash \mathsf{Null} <: [x_i : \tau_i{}^{i \in 1 \ldots k} \mid m_i : \phi^{i \in 1 \ldots \ell}]@\rho}$$

(RegSub Object)
$$\frac{\begin{array}{c}\tau = [x_i : \tau_i{}^{i \in 1 \ldots p+r} \mid m_i : \forall \vec{\rho}^{(i)}.\tau_i' \xrightarrow{\Delta_i} \tau_i''{}^{i \in 1 \ldots q+s}]@\rho \\ \tau' = [x_i : \tau_i{}^{i \in 1 \ldots p} \mid m_i : \forall \vec{\rho}^{(i)}.\tau_i' \xrightarrow{\Delta_i'} \tau_i''{}^{i \in 1 \ldots q}]@\rho \\ \Delta_i \subseteq \Delta_i' \quad \forall i \in 1 \ldots q\end{array}}{\vdash \tau <: \tau'}$$

(RegSub Refl) and (RegSub Trans) explicitly define the reflexive and transitive nature of the relation. Also, they are the only rules that match the simple types int and boolean, so we see that the simple types can only be subtypes of themselves, i.e. we only have int <: int and boolean <: boolean. In fact, the rule (RegSub Trans) is not needed because the transitivity of the relation follows directly from rule (RegSub Object).

The rule (RegSub Object) deserves careful study. Informally, it states that an longer object type is a subtype of a shorter object type. This corresponds well with Java's subtyping rule that is induced from explicit subclassing. Java also demands that fields of the same name have identical types and that methods of the same name have identical signatures. We demand that, too.

RegJava has equal demand for fields, but the demand for method signatures is a bit less restrictive. The live regions $\Delta$ above the arrow just have to be *less* than the $\Delta$ of the same method in the superclass. That is, the method signature of the supertype needs to incorporate the live regions in the signature of the subtype. (Recall the example in section 2.3.4 that explained this.)

The rule (RegSub Object) also demands that the two types point to the same region. This is essential since the type of an object reference must tell exactly in which region the referenced object exists.

### 2.4.3 Expressions

In this section we present the typing rules for expressions in RegJava. We do this by presenting a set of inference rules defining a relation of the general form

$$\Pi, \Delta, E \vdash e : \tau$$

This is a syntactic relation on the set

$$\text{ProgType} \times \text{LiveReg} \times \text{TyEnv} \times \text{Exp} \times \text{Type}$$

where Exp is the set of expressions generated by the grammar for expressions in figure 2.5 on page 26. Given a program type $\Pi$, a set of live region variables $\Delta$, and a type environment $E$, the relation states that the expressions $e$ is well-typed and has type $\tau$.

The inference rules that define the relation is given in figure 2.8 on the following page. There is one rule for each production rule in the grammar of expressions in RegJava—except that there is no rule for a parenthesised expression $(e)$ and for the self parameter this. In addition to this, there is one rule for subsumption. The parentheses are purely syntactical and has no effect on the abstract semantics. The self parameter has its own production rule in the grammar because in Java this is a keyword. However, in order to simplify the static semantics we choose to treat it like a variable and put it into the type environment $E$. This is completely safe because keywords cannot be used as variables (see section 2.3.10).

Given the static semantics of Java, many of these rules are pretty straightforward. In the following we discuss what have motivated the design of the rules.

#### Subsumption

The rule (RegExp Sub) is the only rule for expressions where the subtype relation of section 2.4.2 is used explicitly. The rule states that any well-typed expression $e$ with type $\tau$ also has type $\tau'$ if $\tau <: \tau'$, i.e. if $\tau$ is a subtype of $\tau'$.

With this rule for subsumption the rest of the rules can be written without the concern of subtyping. For instance, when writing the rule for assignment, we can safely demand that the type of the variable on the left and the type of the expressions on the right have the same type. The rule (RegExp Sub) then ensures that the type on the right actually just needs to be a subtype of the type on the left.

**Figure 2.8** Inference rules defining the typing relation $\Pi, \Delta, E \vdash e : \tau$ for expressions.

(RegExp Sub)
$$\frac{\Pi, \Delta, E \vdash e : \tau \qquad \vdash \tau <: \tau'}{\Delta, E \vdash e : \tau'}$$

(RegExp $x$)
$$\frac{E(x) = \tau}{\Pi, \Delta, E \vdash x : \tau}$$

(RegExp Int)
$$\frac{}{\Pi, \Delta, E \vdash i : \mathsf{int}}$$

(RegExp Null)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{null} : \mathsf{Null}}$$

(RegExp False)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{false} : \mathsf{boolean}}$$

(RegExp True)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{true} : \mathsf{boolean}}$$

(RegExp Plus)
$$\frac{\Pi, \Delta, E \vdash e_1 : \mathsf{int} \qquad \Pi, \Delta, E \vdash e_2 : \mathsf{int}}{\Pi, \Delta, E \vdash e_1 + e_2 : \mathsf{int}}$$

(RegExp Equal)
$$\frac{\Pi, \Delta, E \vdash e_1 : \tau \qquad \Pi, \Delta, E \vdash e_2 : \tau}{\Pi, \Delta, E \vdash e_1 == e_2 : \mathsf{boolean}}$$

(RegExp Assign)
$$\frac{E(x) = \tau \qquad \Pi, \Delta, E \vdash e : \tau}{\Pi, \Delta, E \vdash x = e : \tau}$$

(RegExp Field)
$$\frac{\Pi, \Delta, E \vdash e : [x : \tau]@\rho \qquad \rho \in \Delta}{\Pi, \Delta, E \vdash e.x : \tau}$$

(RegExp Update)
$$\frac{\Pi, \Delta, E \vdash e_1 : [x : \tau]@\rho \qquad \Pi, \Delta, E \vdash e_2 : \tau \qquad \rho \in \Delta}{\Pi, \Delta, E \vdash e_1.x = e_2 : \tau}$$

(RegExp New)
$$\frac{\Pi(A, \vec{\rho}) = \tau = [\cdots]@\rho \qquad \mathrm{frv}(\vec{\rho}) \subseteq \Delta \qquad \rho \in \Delta}{\Pi, \Delta, E \vdash \mathsf{new}[\vec{\rho}]\ A() : \tau}$$

(RegExp Method)
$$\frac{\begin{array}{c}\Pi, \Delta, E \vdash e_0 : [m : \forall \vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2]@\rho \\ \Pi, \Delta, E \vdash e_1 : S(\tau_1) \\ \rho \in \Delta \qquad S(\Delta') \subseteq \Delta \qquad S(\mathrm{frv}(\vec{\rho})) \subseteq \Delta\end{array}}{\Pi, \Delta, E \vdash e_0.m[S(\vec{\rho})](e_1) : S(\tau_2)}$$

### Simple expressions

The rules (RegExp $x$), (RegExp Int), (RegExp Null), (RegExp True), (RegExp False) and (RegExp Plus) do not deserve any further comments. The rule (RegExp Equal) covers both equality of simple values and equality of object references. Because of subsumption, two object references can be compared if and only if the type of one them is a subtype of the type of the other.

As explained above, the rule (RegExp Assign) allows for assignment with an expression if its type is a subtype of the type of the variable.

### Object expressions

The rules for simple expressions and subsumption do not explicitly involve the region annotations in the program and in the types. The rules for object expressions take care of this. All of the rules share one premise: $\rho \in \Delta$, where $\rho$ denotes the region in which the object exists and $\Delta$ is the region variables that are live at that point in the program. This premise ensures that the region—and therefore the object—exists at run-time when the operation on the object is performed.

For instance, consider the rule (RegExp Field) for field selection. Given the expression $e.x$, this rule demands that $e$ have the object type $[x : \tau]@\rho$ and that $\rho$ be live in $\Delta$. This ensures that, at run-time, the $e$ evaluate to a reference to an object in the region denoted by $\rho$, that this object have a field named $x$ of the correct type $\tau$, and that the region is alive.

**Object construction** Consider the rule (RegExp New) for object construction. The expression $\mathsf{new}[\vec{\rho}]\ A()$ constructs an instance of the class named $A$. The object is put in the region denoted by the *instantiated* object type $\tau$ for $A$ in the program type $\Pi$.

In the program type $\Pi$ we have the following information for $A$:

$$A[\rho'_1, \ldots, \rho'_n] \quad = \quad [x_i : \tau_i \,^{i \in 1 \ldots k} \mid m_i : \phi \,^{i \in 1 \ldots \ell}]@\rho$$

At this point, we do not actually care about the structure of the type $\tau$. We only need to assert that it has the right number of formal region parameters, and we need to know where to put the constructed object.

Furthermore, with the condition $\mathrm{frv}(\vec{\rho}) \subseteq \Delta$ we make sure that all the actual regions are alive when the object is constructed. However, it is perfectly legal to use the same region variable more than once in the vector $\vec{\rho}$, and different region variables may also denote the same actual region at run-time. Although the formal type for $A$ allows instances to be spread over $n$ regions, it is sometimes needed that less regions be used for a particular instance. For instance, consider the following program

$$\begin{array}{l} \mathsf{letregion}\ \rho'_1\ \mathsf{in}\ \{ \\ \quad x.m[\rho'_1](\mathsf{new}[\rho'_1, \rho'_1]\ A()) \\ \} \end{array}$$

where $x$ has the type $[m : \forall \rho_1.A[\rho_1, \rho_1] \xrightarrow{\{\rho_1\}} \mathsf{int}]@\rho'_2$.

**Method invocation** Rule (RegExp Method) deserves careful study. Recall the interpretation of a method signature $\forall\rho_1 \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2$ from section 2.3.5: Given actual regions for $\rho_1, \ldots, \rho_n$, when the method is invoked with an actual argument of type $\tau_1$, it returns a value of type $\tau_2$. Furthermore, the regions in the set $\Delta'$ above the arrow must be *live* when invoking the method.

The substitution $S$ in the rule maps the formal region variables of the method $m$ to actual region variables, i.e. $S = \{\rho_1 \mapsto \rho'_1, \ldots, \rho_n \mapsto \rho'_n\}$. Thus, the expression is $e_0.m[\rho'_1, \ldots, \rho'_n](e_1)$, and $S(\tau_1)$ and $S(\tau_2)$ are the argument type and return type of the method with actual regions substituted for the formal regions.

It may not be entirely obvious that we need the premise $\rho \in \Delta$ in this rule. However, it is needed because RegJava—like Java—uses dynamic dispatch when invoking instance methods. It is not possible at compile-time to determine which method should actually be invoked since $m$ can be an overridden method. Therefore, information about what method should be invoked is stored along with the object, and this information is read at run-time in order to invoke the appropriate method. Exactly how this works is explained in section 2.5. The premise makes sure that reading from the object in the store is legal.

The premises $S(\Delta') \subseteq \Delta$ and $S(\mathrm{frv}(\vec{\rho})) \subseteq \Delta$ ensure that the regions potentially used by the method are alive. It is important to notice that the set $\Delta'$ can include other region variables than $\rho_1, \ldots, \rho_n$. For instance, consider the following class definition:

```
class A[ρ1, ρ2] at ρ1 extends Object[ρ1] {
  int x;
  A[ρ2, ρ2] a;
  int m(int y) { this.a.x = y; return y }
}
```

An instance of the class $A$ with actual regions $\rho'_1$ and $\rho'_2$ has type

$$[x : \mathsf{int} \mid m : \forall().\mathsf{int} \xrightarrow{\{\rho'_1,\rho'_2\}} \mathsf{int}]@\rho'_1$$

Region $\rho'_2$ is included in set above the arrow because the method $m$ updates this region through an update of the field $a.x$.

### 2.4.4 Statements

In this section we present the typing rules for statements in RegJava. We do this by presenting a set of inference rules defining a relation of the general form

$$\Pi, \Delta, E \vdash s : \tau$$

This is a syntactic relation on the set

$$\mathrm{ProgType} \times \mathrm{LiveReg} \times \mathrm{TyEnv} \times \mathrm{Stm} \times \mathrm{Type}$$

where Stm is the set of statements generated by the grammar for statements in figure 2.5 on page 26. Given a program type $\Pi$, a set of live region variables $\Delta$, and a type environment $E$, the relation states that the expressions $s$ is well-typed and if it returns a value, this value has type $\tau$.

**Figure 2.9** Inference rules defining the typing relation $\Pi, \Delta, E \vdash s : \tau$ for statements.

(RegStm Empty)

$$\frac{}{\Pi, \Delta, E \vdash \epsilon : \tau}$$

(RegStm Exp)

$$\frac{\Pi, \Delta, E \vdash e : \tau'}{\Pi, \Delta, E \vdash e : \tau}$$

(RegStm Sequence)

$$\frac{\Pi, \Delta, E \vdash s_1 : \tau \quad \Pi, \Delta, E \vdash s_2 : \tau}{\Pi, \Delta, E \vdash s_1; s_2 : \tau}$$

(RegStm Return)

$$\frac{\Pi, \Delta, E \vdash e : \tau}{\Pi, \Delta, E \vdash \mathsf{return}\ e : \tau}$$

(RegStm Cond)

$$\frac{\Pi, \Delta, E \vdash e : \mathsf{boolean} \quad \Pi, \Delta, E \vdash s_1 : \tau \quad \Pi, \Delta, E \vdash s_2 : \tau}{\Pi, \Delta, E \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \tau}$$

(RegStm Declaration)

$$\frac{\Pi, \Delta, E \vdash e : \tau' \quad \Pi, \Delta, E + \{x : \tau'\} \vdash s : \tau \quad \mathrm{frv}(\tau') \subseteq \Delta}{\Pi, \Delta, E \vdash \mathsf{let}\ x : \tau' = e\ \mathsf{in}\ s : \tau}$$

(RegStm Letregion)

$$\frac{\Pi, \Delta \cup \{\rho\}, E \vdash s : \tau \quad \rho \notin \Delta}{\Pi, \Delta, E \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s : \tau}$$

(RegStm While)

$$\frac{\Pi, \Delta, E \vdash e : \mathsf{boolean} \quad \Pi, \Delta, E \vdash s : \tau}{\Pi, \Delta, E \vdash \mathsf{while}\ e\ \mathsf{do}\ s : \tau}$$

The inference rules that define the relation is given in figure 2.9 on the preceding page. There is one rule for each production rule in the grammar of statements in RegJava—except that there is no rule for a block $\{s\}$. Just like the parenthesised expression a block is purely syntactical (see also section 2.3.8).

Again, the design of most of the statement rules follows directly from the static semantics of Java. Only the rules for variable declaration and letregion deserves further comments. Also, the type $\tau$ for a statement is explained.

### Return type

It may seem strange that we associate a type with a statement. After all, statements do not produce a value. The reason is the way we treat the return type of a method. The type of statements containing a return statement must match the return type of the method in which it occurs. Statements that do not contain a return statement can have any type. For instance, the conditional statement

$$\mathsf{if\ true\ then\ return\ 42\ else\ false}$$

has type int because of the following derivation:

$$\frac{\Pi, \Delta, E \vdash \mathsf{true} : \mathsf{boolean} \qquad \dfrac{\Pi, \Delta, E \vdash 42 : \mathsf{int}}{\Pi, \Delta, E \vdash \mathsf{return}\ 42 : \mathsf{int}} \qquad \dfrac{\Pi, \Delta, E \vdash \mathsf{false} : \mathsf{boolean}}{\Pi, \Delta, E \vdash \mathsf{false} : \mathsf{int}}}{\Pi, \Delta, E \vdash \mathsf{if\ true\ then\ return\ 42\ else\ false} : \mathsf{int}}$$

Here (RegStm Exp) if used to infer that the statement expression false has type int although the expression false has type boolean. But the conditional statement

$$\mathsf{if\ true\ then\ return\ 42\ else\ return\ false}$$

cannot be typed because the two expressions in the conditional branches cannot be given matching types.

Notice that the relation $\Pi, \Delta, E \vdash e : \tau$ occurring above the line in every rule in figure 2.9 is the expression relation from figure 2.8 on page 30 and *not* the statement expression.[1]

### Variable declaration

Rule (RegStm Declaration) is straightforward except for the premise $\mathrm{frv}(\tau') \subseteq \Delta$. It states that the free region variables of the type $\tau'$ must be alive. The need for this is explained further in chapter 3 when we prove soundness. Here we just say that the premise is included in order to ensure that all region variables in the type environment $E$ are alive, i.e. we want to make sure that the condition $\mathrm{frv}(E) \subseteq \Delta$ always holds.

#### letregion

Rule (RegStm Letregion) is the only rule for statements and expressions that allows introduction of new region variables. The premise $\rho \notin \Delta$ ensures that the $\rho$ is in fact a new region variable. To see why this is necessary, consider the following program:

[1] We initially felt that it was elegant to have a similar looking relation for expressions and statements. In retrospect, this may have been a poor choice.

```
letregion ρ1 in {
  let a : [x : int]@ρ1 = null in {
    letregion ρ1 in {
      a = new[ρ1] A()   // Assume that A[ρ] = [x : int]@ρ
    }
    a.x = a.x + 1
  }
}
```

This program will fail when executing the expression $a.x = a.x + 1$ because the object referenced by $a$ was created in the region denoted by the innermost $\rho_1$, and this region has vanished at this point.

### 2.4.5 Programs and classes

In this section we present the typing rules for program and class definitions in RegJava. We do this by presenting inference rules defining four different syntactic relations. There is one each for program definition, class definition, field definition, and method definition corresponding to the remaining production rules of the grammar in figure 2.5 on page 26. The rules defining

**Figure 2.10** Inference rules defining the typing relations for programs, classes, fields and methods.

(Reg Program)

$$\frac{\Pi = \{A_1 \mapsto \Gamma_1, \ldots, A_n \mapsto \Gamma_n\} \quad \Pi, A_1, \Delta \vdash \mathcal{C}_1 : \Gamma_1 \quad \cdots \quad \Pi, A_n, \Delta \vdash \mathcal{C}_n : \Gamma_n}{\Delta \vdash \mathcal{C}_1 \cdots \mathcal{C}_n : \Pi}$$

(Reg Class)

$$\frac{\begin{array}{rl} \tau = [x_i : \tau_i{}^{\,i \in 1 \ldots k} \mid m_i : \phi_i{}^{\,i \in k+1 \ldots n}]@\rho & \\ \Pi(B, \vec{\rho}) = \tau' & \vdash \tau <: \tau' \\ \Delta \cap \mathrm{frv}(\vec{\rho}) = \emptyset & \rho \in \Delta \cup \mathrm{frv}(\vec{\rho}) \\ \Delta \cup \mathrm{frv}(\vec{\rho}) \vdash b_i : \tau_i & \forall i \in 1 \ldots k \\ \Pi, \Delta \cup \mathrm{frv}(\vec{\rho}), \tau \vdash b_i : \phi_i & \forall i \in k+1 \ldots n \end{array}}{\Pi, A, \Delta \vdash \mathsf{class}\ A[\vec{\rho}]\ \mathsf{at}\ \rho\ \mathsf{extends}\ B[\vec{p}]\ \{b_i{}^{\,i \in 1 \ldots n}\} : \lambda \vec{\rho}.\tau}$$

(Reg Field)

$$\frac{\mathrm{frv}(\tau) \subseteq \Delta}{\Delta \vdash \tau\ x : \tau}$$

(Reg Method)

$$\frac{\Pi, \Delta', \{\mathsf{this} : \tau\} + \{x : \tau_1\} \vdash s : \tau_2 \quad \mathrm{frv}(\tau_1) \cup \mathrm{frv}(\tau_2) \cup \mathrm{frv}(\tau) \subseteq \Delta' \quad \Delta' \subseteq \Delta \cup \mathrm{frv}(\vec{\rho})}{\Pi, \Delta, \tau \vdash \tau_2\ m[\vec{\rho}](\tau_1\ x)\ \{s\} : \forall \vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2}$$

these four relations is given in figure 2.10.

### Programs

A program consists of a series of class definitions. The typing relation for programs has the form

$$\Delta \vdash \mathcal{P} : \Pi$$

This is a syntactic relation on the set

$$\text{LiveReg} \times \text{Prog} \times \text{ProgType}$$

where Prog is the set of programs generated by the grammar for programs in figure 2.5 on page 26. Recall from section 2.4.1 that a program type $\Pi$ is syntactically defined as an environment mapping class names to class types. In the rule (Reg Program) we assume the existence of the program type in order to type the classes.

The set of region variables $\Delta$ consists of any *global* region variables that are initially alive. We do not have a way of explicitly declaring such global regions in RegJava. However, we choose to include them to make in our treatment more general.

### Classes

The type relation for classes has the form

$$\Pi, A, \Delta \vdash \mathcal{C} : \Gamma$$

This is a syntactic relation on the set

$$\text{ProgType} \times \text{ClassName} \times \text{LiveReg} \times \text{Class} \times \text{ClassType}$$

where Class is the set of class definitions generated by the grammar for classes in figure 2.5 on page 26.

There is a lot of premises of the rule (Reg Class). The premises $\Pi(B, \vec{p})$ and $\vdash \tau <: \tau'$ assert that the class $B$ is defined in the program and that the type of $B$ is a supertype of the type $\tau$. The premises $\Delta \cap \text{frv}(\vec{\rho}) = \emptyset$ and $\rho \in \Delta \cup \text{frv}(\vec{\rho})$ state that the formal region variables of the class are fresh variables and that $\rho$ is either one of these or one of the global region variables.

The rest of the premises has to do with the field and method definitions of the class. The rule assumes that the first $k$ member definitions of the class are fields, and the last $n - k$ are methods. This should not be considered a restriction since the order of fields and methods does not affect the semantics of the program.[2]

Both field and method definitions are typed within the set of region variables $\Delta \cup \text{frv}(\vec{\rho})$ consisting of the global region variables and the formal region variables of the class. This way it is ensured that the type $\tau$ contain no "unknown" region variables.

### Fields

The rule (Reg Field) is extremely simple. It only checks that the free region variables are in $\Delta$.

[2] Of course, this fact relies on the choice to exclude initialisation code for fields.

#### Methods

The purpose of the rule (Reg Method) is to check that the signature of the method is consistent with the definition of the method. The rule ensures that the body of the method is well-typed in a small type environment consisting only of the self parameter this and the argument $x$. In this environment this is given the type $\tau$ which is the type of the class in which the method definition occurs.

Furthermore, the set of region variables, $\Delta'$, that have to be alive when calling the method are the ones that have to be alive when typing the body of the method. Also, the free region variables are checked to be in this set.

### 2.4.6 Comparisons with the ML-Kit

The type system and the static semantics of RegJava are inspired by the "ML-Kit with Regions" in [TBE+97] and the region-based model in [TT97] for a smaller language. However, because of the nature of the language, RegJava, we have made some things quite different.

#### Region annotations

Because we have adopted the variable model for the memory (see section 2.2), variables in RegJava are not stored in regions. Thus, we do not need the special type-and-place syntax for variables as is the case for update-able references in the ML-Kit.

Simple values in RegJava are not stored regions, so there are no region annotations on those, either. In the ML-Kit all values are put in regions and, as an optimisation, values that are represented unboxed are put into a special, global region. This region does not really act as a region at run-time since those values are just stored in registers.

#### Polymorphic types and effects

ML has explicit polymorphic types using *type schemes*. RegJava, like Java, has another kind of polymorphism called subtype-polymorphism. This eliminates the need for *effects* and *effect variables* as they are used in the ML-Kit.

Instead, we use the set of live region variables $\Delta$ that closely resemble effects. The difference is that $\Delta$ is not strictly constructed while typing the program. In the ML-Kit, there is one and only one effect of an expression. The static semantics for RegJava only states restrictions that defines the minimally needed $\Delta$ for every expression and statement in the program. For method signatures this means that the $\Delta$ on the arrow can reflect the effect of overridden methods as well as the method itself.

In the ML-Kit effects contain more information than $\Delta$ does; every region variable has associated attributes that are called *get* and *put*. These attributes that tell whether a region is being used only for reading, writing or for both. These attributes are used in an analysis called *multiplicity inference*. Since we do not do this analysis, we do not need these attributes. However, $\Delta$ can easily be extended with the same information on each region variable exactly the way it is done in the effects.

#### Region-polymorphic types

The region polymorphism in RegJava is twofold. Both object types and method signatures are region-polymorphic This means that the formal region variables in the type of methods get

instantiated in two steps; first by construction of the enclosing object, second when invoking the method.

In the ML-Kit, only functions and constructors for recursive datatypes are region-polymorphic. Function types are instantiated only once; when a function closure is constructed.

## 2.5 Dynamic Semantics

In this section we present and discuss the dynamic semantics of RegJava. This semantics should correspond closely to that of Java as given in Java Language Specification [GJS96]. This means that given a legal RegJava program, removing all the region annotation and transforming the few syntactic differences should produce a legal Java program with the same semantics.

Unfortunately, to the best of our knowledge, the semantics of Java has not yet been formalised in terms of inference rules, for instance. Instead, we formulate a dynamic semantics for RegJava and argue, as far as possible, that it corresponds well with the intended semantics of Java.

### 2.5.1 Intentional semantics

Our aim is to formulate the dynamic semantics of RegJava in terms of inference rules. This can be done in a purely abstract way with no reference to any underlying memory model. However, we intend to do more than that. We also want to show how the memory is used with respect to regions. We want to show when and how region operations such as creation and destruction occurs. Therefore, we formulate an intentional semantics in which objects are stored in a store that is divided into regions.

### 2.5.2 Semantic objects

When describing the dynamic semantics we refer to the semantic objects defined by the equations in figure 2.11 on the following page in addition to some of the semantic object from figure 2.6 on page 27.

Regions are identified by a region name. We assume a denumerably infinite set of region names, RegName. We use $r$ to range over region names. A region is a finite map that maps offsets to values. We assume a denumerably infinite set of offsets, OffSet. We use $o$ to range over offsets. In a concrete implementation both region names and addresses would probably just be simple pointers—actually this is the way we do it in section 4.1.4.

The region variables of the static semantics is bound to region names in the region environment $R$. The region environment can only be extended with new region variables, never updated.

The store $\sigma$ is a finite mapping from region names to regions. So, an *address* $a$ of a value in the store is a pair $(r, o)$ consisting of a region name and an offset. We sometimes write $\sigma(a)$ or $\sigma(r, o)$ when we really mean $(\sigma(r))(o)$. Likewise, the expression $a \in \mathrm{Dom}(\sigma)$ simply means that $r \in \mathrm{Dom}(\sigma)$ and $o \in \mathrm{Dom}(\sigma(r))$.

The set of values that can be assigned to variables and fields is denoted by Value. Such a value is either a simple value or it is an address into the store. Since the store is only used for storing object values, an address corresponds to what we call an object reference in the static semantics.

**Figure 2.11** The semantic objects of the dynamic semantics.

$$
\begin{array}{rcl}
r & \in & \text{RegName} \\
o & \in & \text{OffSet} \\
a \text{ or } (r,o) & \in & \text{Addr} = \text{RegName} \times \text{OffSet} \\
v & \in & \text{TargetVal} = \text{Value} \cup \{\mathsf{error}, \mathsf{noval}\} \\
 & & \text{Value} = \text{Const} \cup \text{Addr} \\
c & \in & \text{Const} = \text{Int} \cup \{\mathsf{true}, \mathsf{false}, \mathsf{null}\} \\
<A, V, \vec{r}> & \in & \text{ObjVal} = \text{ClassName} \times \text{VarEnv} \times (\text{RegName})^m \\
V & \in & \text{VarEnv} = \text{Var} \xrightarrow{\text{fin}} \text{Value} \\
R & \in & \text{RegEnv} = \text{RegVar} \xrightarrow{\text{fin}} \text{RegName} \\
\sigma & \in & \text{Store} = \text{RegName} \xrightarrow{\text{fin}} \text{Region} \\
 & & \text{Region} = \text{OffSet} \xrightarrow{\text{fin}} \text{ObjVal}
\end{array}
$$

An object value consists of a class name $A$, a variable environment $V$ mapping field names to values, and a vector $\vec{r}$ of region names. $A$ is the name of the class that the object is an instance of. It is used for implementing dynamic dispatch when invoking methods. $V$ is not really an environment since it can only be updated, never extended; an update of field $x$ with value $v$ is written $V + \{x \mapsto v\}$ which implicitly assumes that $x \in \text{Dom}(V)$. The region names are the ones that are bound to the formal region parameters of the class $A$.

$V$ is a mapping from program variables to values. A value is either a constant, or a reference to an object in the store. Expression and statement evaluation update the variable environment, and so return and updated variable environment $V'$.

In addition to the values in Value, there are two values, which can be generated by expressions or statements, but cannot be assigned to variables. These are error and noval. The error value is not further used in this section, but it will be used in chapter 3 when we add error rules to the dynamic semantics. The noval value is used as a mechanism for breaking the evaluation sequence when a return statement is evaluated.

When constructing an object, fields are assigned initial values according to their type. For this purpose we use a simple function **init** mapping types to values. The definition of this function follows the conventions of initial values in Java:

$$
\mathbf{init}(\tau) = \begin{cases} 0 & \text{if } \tau = \mathsf{int}, \\ \mathsf{false} & \text{if } \tau = \mathsf{boolean}, \\ \mathsf{null} & \text{if } \tau \text{ is an object type.} \end{cases}
$$

### 2.5.3 Expressions

The evaluation rules for expressions are given in figure 2.12 on page 41. They define the evaluation relation for expressions that has the general form

$$\sigma, V, R \vdash e \to v, \sigma', V'$$

which is a syntactic relation on the set

$$\text{Store} \times \text{VarEnv} \times \text{RegEnv} \times \text{Exp} \times \text{TargetValStore} \times \text{VarEnv}$$

The relation is read as follows: In store $\sigma$, variable environment $V$, and region environment $R$, the evaluation of the expression $e$ terminates and produces the value $v$, the updated store $\sigma'$, and the updated variable environment $V'$.

#### Simple expressions

The rules for simple expressions follow directly from the semantics of Java since they do not use either the store or the region environment.

The rule (DynExp Equal True) and (DynExp Equal False) imply that two objects are equal only if their addresses are the same. This is the normal convention in Java. Due to the scoping rules for the regions, as defined by the static semantic, it is actually not possible to compare *dangling pointers*, in spite of the fact that the regions are actually not referenced in the comparison.

#### Method invocation

The rule (DynExp Method) for invoking methods is rather complex. First, the subexpression $e_0$ evaluates to an address $a$ of an object value in the updated store $\sigma'$. The class name $A$ in the object value is the actual class. Next, the subexpression $e_1$ evaluates in the updated store $\sigma'$ to the parameter value $v$.

Then, a new variable environment is created with only two values: the self parameter this bound to the address $a$, and the formal parameter $x$ bound to the actual parameter value $v$.

A new region environment $\tilde{R}$ is constructed containing two sets of region bindings. One set is the bindings of the formal region parameters $\rho_1, \ldots, \rho_k$ of class $A$ to the region names in the object value. The other set is the binding of formal region parameters $\rho_{k+1}, \ldots, \rho_n$ of the method $m$ to the actual region arguments given by $\rho'_{k+1}, \ldots, \rho'_n$. The formal names of the region variables are all looked up in the program (using the program type $\Pi$) in which the evaluation occurs.

The method body $s$ is fetched from the program text and is evaluated using the above environments. This produces a return value $v'$, an updated store $\sigma'''$, and an updated variable environment $V'''$. The latter is simply thrown away since there are no other variables in it than the self parameter and the formal argument.

#### Object construction

The rule (DynExp New) for object construction is pretty simple. It creates a new object value with initial values for the fields according the type of the class $A$ in the program. Furthermore, the object value is placed at a new offset in the region $r$ bound to the region variable $\rho$ from the type of $A$.

### 2.5.4 Statements

The evaluation rules for statements are given in figure 2.13 on page 43. They define the evaluation relation for statements that has the general form

$$\sigma, V, R \vdash s \to v, \sigma', V'$$

**Figure 2.12** Evaluation rules for expressions: $\sigma, V, R \vdash e \to v, \sigma', V'$

(DynExp $x$)
$$\frac{V(x) = v}{\sigma, V, R \vdash x \to v, \sigma, V}$$

(DynExp Const)
$$\frac{}{\sigma, V, R \vdash c \to c, \sigma, V}$$

(DynExp Plus)
$$\frac{\sigma, V, R \vdash e_1 \to i_1, \sigma', V' \quad \sigma', V', R \vdash e_2 \to i_2, \sigma'', V''}{\sigma, V, R \vdash e_1 + e_2 \to i_1 + i_2, \sigma'', V''}$$

(DynExp Equal True)
$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \quad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \quad v_1 = v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{true}, \sigma'', V''}$$

(DynExp Equal False)
$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \quad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \quad v_1 \neq v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{false}, \sigma'', V''}$$

(DynExp Assign)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash x = e \to v, \sigma', V' + \{x \mapsto v\}}$$

(DynExp Field)
$$\frac{\sigma, V, R \vdash e \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, \vec{r}> \quad \tilde{V}(x) = v}{\sigma, V, R \vdash e.x \to v, \sigma', V'}$$

(DynExp Update)
$$\frac{\sigma, V, R \vdash e_1 \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, \vec{r}> \quad \sigma', V', R \vdash e_2 \to v, \sigma'', V''}{\sigma, V, R \vdash e_1.x = e_2 \to v, \sigma'' + \{a \mapsto <A, \tilde{V} + \{x \to v\}, \vec{r}>\}, V''}$$

(DynExp Method) ($x$ and $s$ is the formal parameter and body of method $m$ in class $A$)
$$\frac{\begin{array}{c}\Pi(A) = \lambda\rho_1 \cdots \rho_k.[\cdots, m : \forall\rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta} \tau 2, \cdots]@\rho \\ \sigma, V, R \vdash e_0 \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \quad \sigma', V', R \vdash e_1 \to v, \sigma'', V'' \\ \tilde{R} = \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k, \rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\} \\ \sigma'', \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}, R + \tilde{R}, \vdash s \to v', \sigma''', V'''\end{array}}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to v', \sigma''', V''}$$

(DynExp New)
$$\frac{\begin{array}{c}\Pi(A, \rho_1, \ldots, \rho_k) = [x_i : \tau_i \,^{i \in 1 \ldots n}]@\rho \\ \tilde{V} = \{x_1 \mapsto \mathbf{init}(\tau_1), \ldots, x_n \mapsto \mathbf{init}(\tau_n)\} \qquad R(\rho) = r \qquad o \notin \mathrm{Dom}(\sigma(r))\end{array}}{\sigma, V, R \vdash \mathsf{new}\ A[\rho_1, \ldots, \rho_k]() \to (r, o), \sigma + \{(r, o) \mapsto <A, \tilde{V}, R(\rho_1), \ldots, R(\rho_k)>\}, V}$$

which is a syntactic relation on the set

$$\text{Store} \times \text{VarEnv} \times \text{RegEnv} \times \text{Stm} \times \text{TargetValStore} \times \text{VarEnv}$$

The relation is read exactly as the relation for expressions.

#### Using the noval value

As mentioned in section 2.5.2, we have introduced the pseudo-value noval in order to cope with abruptly completing the evaluation of a method when a return statement is evaluated. The idea is that if a statement returns noval, evaluation should continue. If a statement returns a value other than noval, evaluation should stop in the sense that the value should be propagated unaltered to the outermost statement of the method without evaluating any other statements. The two rules for sequences take care of this.

The rule (DynStm Exp) states that when an expression statement is evaluated, the generated value is not used and it therefore just returns noval. The rule (DynStm Return) is the only statement rule that actually produces a value other than noval.

#### Variable declaration

Local variables are initialised in the declaration. The variable environment is extended with the variable $x$ bound to the initial value. This binding is then removed when exiting from the block. This should be interpreted correctly, however. If the initial environment $V$ already has a binding of a variable named $x$, this should be reinserted in the resulting environment.[3]

#### The letregion statement

In rule (DynStm Letregion) a new, empty region is created in the store for the evaluation of the statement $s$. Afterwards, the region is destroyed again. It is, however, possible that the final store still will contain references into the dead region. In a well-typed program such pointers can never be used, so this is completely harmless.

## 2.6 Annotating programs

In the preceding sections we have introduced the language RegJava that explicitly uses regions. However, RegJava should only be considered as an *intermediate* language in a compiler for Java. RegJava plays the same role as the intermediate language *RegionExp* in the ML-Kit[TBE+97].

### 2.6.1 Region inference

A region-based Java compiler should do a static analysis on the source program, written in Java, and return a region annotated program, written in RegJava, with the same semantics as the original program. This static analysis, called *region inference*, has been implemented for a subset of ML in the ML-Kit and is described in [TB98].

[3] It would probably have been better to make this explicit using a variable environment defined as a list of bindings instead of as a finite map.

**Figure 2.13** Evaluation rules for statements: $\sigma, V, R \vdash s \to v, \sigma', V'$

(DynStm Exp)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash e \to \mathsf{noval}, \sigma', V'}$$

(DynStm Return)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{return}\ e \to v, \sigma', V'}$$

(DynStm Decl)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V' \qquad \sigma', V' + \{x \mapsto v\}, R \vdash s \to v', \sigma'', V''}{\sigma, V, R \vdash \mathsf{let}\ x = e\ \mathsf{in}\ s \to v', \sigma'', V'' \setminus\setminus \{x\}}$$

(DynStm Letregion)
$$\frac{r \notin \mathrm{Dom}(\sigma) \qquad \sigma + \{r \mapsto \emptyset\}, V, R + \{\rho \mapsto r\} \vdash s \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s \to v, \sigma' \setminus\setminus \{r\}, V'}$$

(DynStm Cond False)
$$\frac{\begin{array}{c}\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V' \\ \sigma', V', R \vdash s_2 \to v, \sigma'', V''\end{array}}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Cond True)
$$\frac{\begin{array}{c}\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \\ \sigma', V', R \vdash s_1 \to v, \sigma'', V''\end{array}}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Sequence Return)
$$\frac{\sigma, V, R \vdash s_1 \to v, \sigma', V' \qquad v \neq \mathsf{noval}}{\sigma, V, R \vdash s_1; s_2 \to v, \sigma', V'}$$

(DynStm Sequence Cont)
$$\frac{\begin{array}{c}\sigma, V, R \vdash s_1 \to \mathsf{noval}, \sigma', V' \\ \sigma', V', R \vdash s_2 \to v', \sigma'', V''\end{array}}{\sigma, V, R \vdash s_1; s_2 \to v', \sigma'', V''}$$

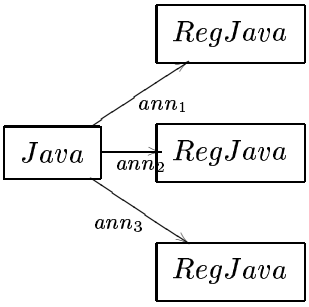
(DynStm While False)
$$\frac{\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V'}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{noval}, \sigma', V'}$$

(DynStm While True)
$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \qquad \sigma', V', R \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}$$

In this thesis, we do not develop a region inference algorithm for Java. In this section, however, we discuss the possibilities of such an algorithm. That is, given a Java program we give directions as to how it could be translated into a legal and efficient RegJava program. This translation is also called an *annotation* of the Java program.

Of course, we only consider a subset of Java corresponding to RegJava. For every legal program in this subset of Java there exist one or more well-typed RegJava programs that are annotations of this Java program. In other words, for each Java program, there exists a number of annotations giving RegJava programs as depicted in figure 2.14. These RegJava

**Figure 2.14** To every Java program there exists several different legal annotations in RegJava.




programs, while all legal, are not equally desirable. In principle, all Java programs can be annotated so that it only uses one region in which all objects are created. Such an annotation leads to no memory reclamation whatsoever. Clearly, this is not desirable for most programs because it would lead to serious space leaks. The best annotation is usually the one, that leads to the most aggressive memory reclamation possible.

In the following sections, we discuss the rules and guidelines to use when annotating a Java program.

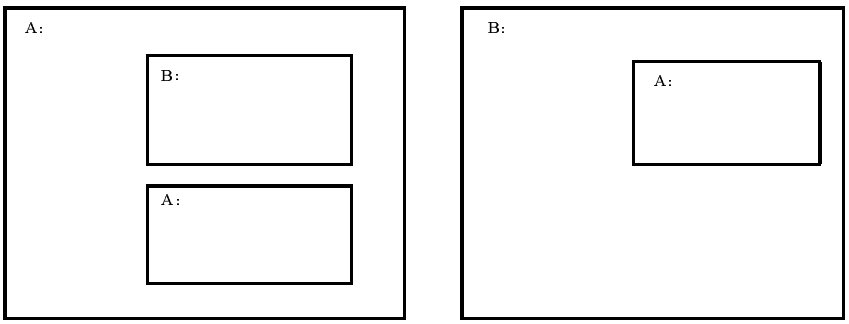
### 2.6.2 Class types

All classes are annotated with a set of formal region parameters. These can be used to spread an object over more than one region. For instance, if a "local" object has a field that references a "global" object, it would be a serious limitation if these two object were forced into the same global region. Instead, the type of the local object can be spread over both a local and a global region.

Another use of spreading the object type could be to optimise locality of reference in programs, that access mainly specific parts of objects. Keeping the frequently referenced parts from a number of objects together, only these parts of the objects are placed in the memory caches, and not the full objects.

Assume that objects of class A contain an object of class B, as well as an object of Class A, as well as some data fields of simple types.

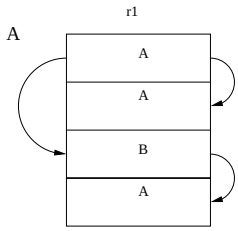
Class B contains an object of class A, and some data-fields of simple types.




The simple data-field are all placed in the primary regions of the objects, and are not drawn here.

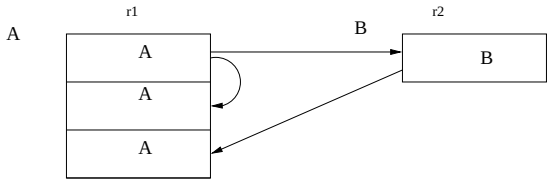
The simplest way of making region types, would be to place everything in one region per object.

$$A[\rho] = [b : B[\rho], a : A[\rho]]@\rho$$
$$B[\rho] = [a : A[\rho]]@\rho$$




Another way might be to place all objects of type A in one region and all objects of type B in another:

$$A[\rho_1, \rho_2] = [b : B[\rho_2, \rho_1], a : A[\rho_1, \rho_2]]@\rho_1$$
$$B[\rho_1, \rho_2] = [a : A[\rho_2, \rho_1]]@\rho_1$$




Other annotations are also possible, such as making special regions for the first few object levels, or placing the A-objects defined inside B-objects in a different region from those defined inside A-objects themselves. With such an annotation, an A-object could not be shared between a B-object, and another A-object. For practical purposes, however, we believe that the two variations above , and combinations of them, should suffice.

The list of regions parameters in a class must be finite, and consequently it is not possible to spread an object over and infinite number of regions, even if such a design might be practical in some situations.

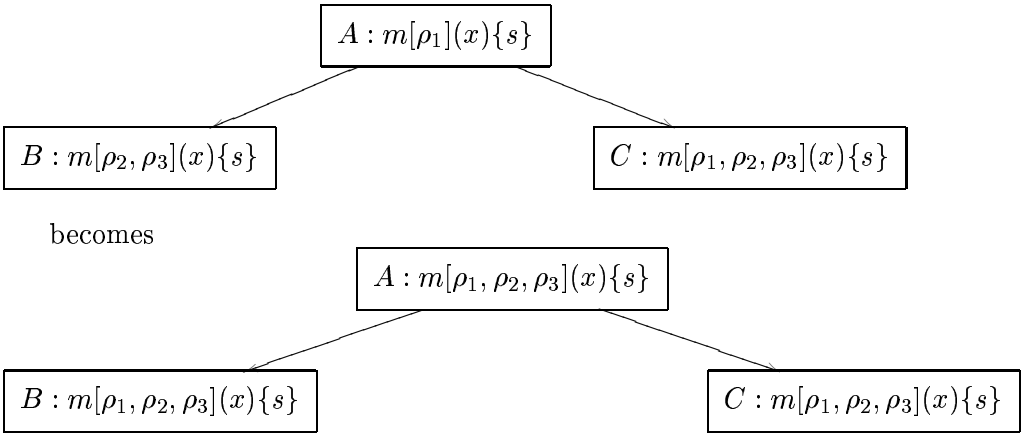
If an extra region is introduced in a class A, all classes containing free occurrences of the type A must also either grow an extra region, or assign a region already in use to also be used in their objects of class A.

### 2.6.3 Method types

Each method in a class must have a list of formal parameters. These parameters should specify which regions are used by the method, aside from the object regions. This list can be determined by finding the minimum $\Delta$ required by the method body in the static semantic, and removing the region variables that are formal object region variable. What is left, will be a subset of the regions of the parameter value, the regions of the returned value as well as possibly some "dummy" regions, i.e. regions that temporary objects can be placed in. Such temporary objects will just take up space, since they cannot be accessed between the end of the method invocation and the deallocation of the region.

A program with good annotations, would have no "dummy" region parameters of this kind, but would rather create local regions as needed inside the body of the method. Finally, the region parameter list may be forced to contain region variables that are not used at all in the method body, because of a super type has a method using such a region parameter, that the current method definition overloads, or a subtype uses the region parameter in a method, that overload the current method.

In other words, all method definition that overload each other, must have the same set of region parameters. This set is the least common unifier of the region parameters, that could be determined for each of the methods independent of each other (see section 2.4).




Here, the region annotations for a method in class B becomes dependent on the regions used in a method in C, just because they both overload the same method in A.

### 2.6.4 Program annotations

In general, regions should be as small and as short-lived as possible. Even if two objects are created at the same time and also referenced the last time at the same place in the program,

it is still a good idea to keep them in different regions in order to maximise the locality of reference.

Ideally, the regions should always be created immediately before the first object is created, and should be destroyed immediately after the last reference to an object in the region.

The block structure of the RegJava imposes some limitations on this, however. Regions are effectively created on a stack, and must be deallocated so that the latest region is deallocated first. Regions cannot be deallocated until the last local variable declared within the scope of the region has been used for the last time. It does not matter if this last variable actually use the region or not.

The block structure of RegJava imposes another inefficiency on region allocation. A region must be declared in the outermost block it is used, regardless of where it is first and last used. Consider this example:

```
letregion ρ1 in {
  let a : A[ρ1] = null in {
    ...
    a = new[ρ1] A() // First use of a
    ...
    ... a ... // Last use of a
    ...
  }
}
```

The distance between the region allocation and the first use of $a$ is not too much of a problem, since an empty region does not take space on the heap. The distance between the last use of $a$ and the region deallocation is another story. $a$ may be quite large, and may be kept alive for some time unnecessarily. Sadly, nothing can be done in this case.

In general, it is better to create regions inside loops, than outside them. That is

```
letregion ρ1 in {
  while b ≠ 5 do {
    ...
    ... new[ρ1] A() ...
    ...
  }
}
```

generally does not perform as well as

```
while b ≠ 5 do {
  letregion ρ1 in {
    ...
    ... new[ρ1] A() ...
    ...
  }
}
```

Of course, this only works, if it is not necessary for the created objects to survive from iteration to iteration. While creating and deallocating regions does take some time, the aggressive memory reclamation usually makes this worthwhile.

Similarly, as many regions as possible should be local to methods, rather than externally declared and passed to the method as parameters. This is particularly important, since adding an extra region parameter to a method affects the type of all sub- and super types in which the method is defined.

Given these guidelines, we have found that making good annotations by hand is relatively straightforward. The trickiest part is determining the correct region annotations of the recursive class types and the types of the methods.

# Chapter 3

# Soundness

In this chapter we set out to show that the static semantics of section 2.4 are consistent with the dynamic semantics of section 2.5. First, we give a definition of soundness and formulate a soundness theorem showing that well-typed RegJava-programs do not go wrong. Next, we formulate and prove a number of lemmas. Last, we give a proof of out soundness theorem.

## 3.1 Definition of soundness

A *well-typed* program is a program typed according to the rules in the static semantics. This typing should be *sound* in some sense of the word. Informally we would like to know that the evaluation of a well-typed program does not go wrong.

First, we define the meaning of "go wrong" by adding error rules to the dynamic semantics. Next, we define a consistency relation for typing a value in a store. Last, we define soundness in terms of a semantic interpretation relating the static semantics with the dynamic semantics.

### 3.1.1 Error rules

What does it mean that a program goes wrong? Consider the statement $s$, and assume that in store $\sigma$, value environment $V$, and region environment $R$ it evaluates to the value $v$, i.e. $\sigma, V, R \vdash s \rightarrow v, \sigma', V'$. Actually, just by making this assumption we have stated that the evaluation does not go wrong. This is because the rules of the dynamic semantics are not exhaustive. Specifically, there are no rules that handle error situations.

For example, in order to use the rule (DynExp $x$) on the expression $x$ the prerequisite $V(x) = v$ must be true. But if this is not the case, i.e. if $x$ is not in the domain of $V$, there are no other rules to apply.

Hence, in order to say that an evaluation goes wrong, we need an error rule like the following:

$$\frac{x \notin \mathrm{Dom}(V)}{\sigma, V, R \vdash x \rightarrow \mathsf{error}, \sigma, V}$$

The evaluation is made to return the pseudo-value error to indicate that the evaluation has gone wrong and should not continue any further. We briefly introduced the value error in section 2.5.2 when defining the dynamic semantics.[1]

[1] Actually, there is no need for the store and the variable environment once the evaluation has gone wrong. We could have defined the relation by using an error *state* to the right of the arrow instead of an error *value*.

For every prerequisite above the line in every rule there should exist another rule containing the same prerequisite negated. Most of these rules just take care of propagating error values. For instance, corresponding to the rule (DynExp Plus) we have the following two error rules:

$$\frac{\sigma, V, R \vdash e_1 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e_1 + e_2 \to \mathsf{error}, \sigma', V'}$$

$$\frac{\sigma, V, R \vdash e_1 \to i, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash e_1 + e_2 \to \mathsf{error}, \sigma'', V''}$$

In the following sections we aim at proving that none these error rules will ever be applied when evaluating a well-typed program. However, there are a few additional error situations that do not fit into this category. If an expression of an object type evaluates to null, it is not allowed to, for instance, select a field from this expression. Trying to do so should make the program stop and return some error value different from error. This kind of error does not indicate flaws in our system. Rather, it indicates flaws in the design of the program.

For simplicity, we have chosen just to ignore the possibility of dereferencing null. Therefore, when we assume that a statement evaluates to a value, we also implicitly assume that it does not try to dereference null.

### 3.1.2 Value consistency

Consider a statement $s$ that has type $\tau$:

$$\Pi, \Delta, E \vdash s : \tau$$

and an evaluation of that statement:

$$\sigma, V, R \vdash s \to v, \sigma', V'$$

We would then like to show that the value $v$ is not error and that it has the type $\tau$ in the resulting store. We can state in the form of a consistency relation:

$$\Pi, \sigma', R, \Delta \models v : \tau$$

This is a relation on the set

$$C = \text{ProgType} \times \text{Store} \times \text{RegEnv} \times \text{LiveReg} \times \text{TargetVal} \times \text{Type}$$

of program types, stores, region environments, value environments and type environments.

The purpose of this section is to define this relation. However, since the store can contain cyclic references, such a relation cannot be defined using induction. Instead, we state a set of inference rules in figure 3.1 on the next page that any relation $\Pi, \sigma, R, \Delta \vdash v : \tau$ on the set $C$ must satisfy in order to be a consistency relation.

Let $\mathcal{F}$ be a functional version of this inference system. That is, $\mathcal{F} : \mathcal{P}(C) \to \mathcal{P}(C)$ is the operator on the powerset of $C$ such that for any set $A \subseteq C$ and tuple $c \in C$, we have $c \in \mathcal{F}(A)$ if and only if $c$ is a conclusion of one of the above inference rules using only tuples in $A$ as premises.

Clearly, $\mathcal{F}$ is monotonic: $A \subseteq B$ implies $\mathcal{F}(A) \subseteq \mathcal{F}(B)$. Thus, by Tarski's fixed point theorem (see [MT91]), there exists a greatest fixed point for $\mathcal{F}$ and this greatest fixed point is also the greatest set $A$ satisfying $A \subseteq \mathcal{F}(A)$. We therefore make the following definition:

**Figure 3.1** Rules defining a consistency relation $\Pi, \sigma, R, \Delta \vdash v : \tau$

(Con Int)
$$\frac{}{\Pi, \sigma, R, \Delta \vdash i : \mathsf{int}}$$

(Con Null)
$$\frac{}{\Pi, \sigma, R, \Delta \vdash \mathsf{null} : \mathsf{Null}}$$

(Con Noval)
$$\frac{}{\Pi, \sigma, R, \Delta \vdash \mathsf{noval} : \tau}$$

(Con False)
$$\frac{}{\Pi, \sigma, R, \Delta \vdash, \mathsf{false} : \mathsf{boolean}}$$

(Con True)
$$\frac{}{\Pi, \sigma, R, \Delta \vdash \mathsf{true} : \mathsf{boolean}}$$

(Con Sub)
$$\frac{\Pi, \sigma, R, \Delta \vdash v : \tau \qquad \tau <: \tau'}{\Pi, \sigma, R, \Delta \vdash v : \tau'}$$

(Con Dangle)
$$\frac{\rho \notin \Delta}{\Pi, \sigma, R, \Delta \vdash (r, o) : [x_i : \tau_i \,^{i \in 1 \ldots n} \mid \ldots]@\rho}$$

(Con Object) $(\tau = [x_i : \tau_i \,^{i \in 1 \ldots n} \mid m_i : \phi_i \,^{i \in 1 \ldots k}]@\rho)$
$$\frac{\begin{array}{cc} r \in \mathrm{Dom}(\sigma) & o \in \mathrm{Dom}(\sigma(r)) \\ \rho \in \Delta & R(\rho) = r \\ \sigma(r, o) = <A, \{x_1 = v_1, \ldots, x_n = v_n\}, \vec{r}> & \\ \Pi, \sigma, R, \Delta \vdash v_i : \tau_i \text{ for all } 1 \leq i \leq n & \\ A \in \mathrm{Dom}(\Pi) & \exists \vec{\rho} : \Pi(A, \vec{\rho}) = \tau \end{array}}{\Pi, \sigma, R, \Delta \vdash (r, o) : \tau}$$

**Definition 1 (Value consistency).** Let the relation $\Pi, \sigma, R, \Delta \models v : \tau$ be greatest fixed point for the monotonic operator $\mathcal{F}$.

Next, we simply lift the relation to environments by point-wise extension by the following definition:

**Definition 2 (Environment consistency).** $\Pi, \sigma, R, \Delta \models V : E$ if all of the following conditions hold:

1. $\mathrm{Dom}(V) = \mathrm{Dom}(E)$
2. $\Pi, \sigma, R, \Delta \models V(x) : E(x)$ for all $x \in \mathrm{Dom}(V)$

This definition of value consistency has a few unusual points that should be understood. (Con Dangle) makes any dangling pointer consistent. Since we know that region based memory management makes it entirely valid for a data-structure to have dangling pointers, we do not wish to make such values appear inconsistent. Since the static typing ensures that no dangling pointers are ever used, however, it would be possible to define a soundness relation without (Con Dangle). This would make the proof a bit harder, since Lemmas 6, 7,and 13 would no longer hold. The effect would mainly be that it would be possible for an address of an object to be consistent with a supertype of the objects actual type, but not with the actual type itself.

There is no notion of "a consistent store" in the general sense. $\sigma$ is consistent with respect to given live regions, values and types only. In the extreme cases, any $\sigma$ is consistent if the value and type match a simple type, a noval, a dangling pointer or null.

### 3.1.3 Semantic interpretation

The static semantics is expressed as a system of inference rules. These rules are purely syntactic since they only give a relationship between the syntactic categories: programs and types. In this section we give a semantic interpretation of this syntactic relationship between programs and types.

In the static semantics we have five different kinds of judgements:[2]

| | |
|---|---|
| $\Delta \vdash \mathcal{P} : \Pi$ | program judgement |
| $\Pi, A, \Delta \vdash \mathcal{C} : \Gamma$ | class judgement |
| $\Pi, \Delta, \tau \vdash b : \phi$ | method judgement |
| $\Pi, \Delta, E \vdash s : \tau$ | statement judgement |
| $\Pi, \Delta, E \vdash e : \tau$ | expression judgement |

To each of these static judgements we will give a semantic interpretation. Since these interpretations give properties about exactly the syntactic elements mentioned in the above judgements, the semantic interpretations naturally are expressed as very similar looking relations:

| | |
|---|---|
| $\Delta \models \mathcal{P} : \Pi$ | program interpretation |
| $\Pi, A, \Delta \models \mathcal{C} : \Gamma$ | class interpretation |
| $\Pi, \Delta, \tau \models b : \phi$ | method interpretation |
| $\Pi, \Delta, E \models s : \tau$ | statement interpretation |
| $\Pi, \Delta, E \models e : \tau$ | expression interpretation |

Once we have defined the above relations, the soundness property that we are looking for can be expressed very elegantly: *given a static judgement, the corresponding semantic relation holds.* The following definitions define the semantic relations. It is no coincidence that we use $\models$ both for value and environmental consistency, and for the semantic interpretations. We say, for instance, "$\mathcal{P}$ is consistent in $\Pi$ given $\Delta$" if $\Delta \models \mathcal{P} : \Pi$, even though program consistency and value consistency do not otherwise have much to do with each other.

**Definition 3 (Program interpretation).** Given a set of live region variables $\Delta$, a program $\mathcal{P} = \mathcal{C}_1 \cdots \mathcal{C}_n$, and a program type $\Pi = \{A_1 \mapsto \Gamma_1, \ldots, A_n \mapsto \Gamma_n\}$, the semantic relation

$$\Delta \models \mathcal{P} : \Pi$$

holds if the following holds:

$$\Pi, A_i, \Delta \models \mathcal{C}_i : \Gamma_i$$

for each $i \in 1 \ldots n$.

[2] Actually, there are six kinds of judgements because there is also one for typing the fields of a class. However, since the type of fields are completely defined by the object type, we do not need a semantic interpretation for fields.

**Definition 4 (Class interpretation).** Given a set of live region variables $\Delta$, a program type $\Pi$, a class name $A$, a class $\mathcal{C}$, and a class type $\Gamma$ with

$$\mathcal{C} = \mathsf{class}\ A[\vec{\rho}]\ \mathsf{at}\ \rho \mathsf{extends}\ B[\vec{p}]\ \{b_i\ ^{i\in 1\ldots n}\}$$
$$\Gamma = \lambda\vec{\rho}.[x_i : \tau_i\ ^{i\in 1\ldots k} \mid m_i : \phi_i\ ^{i\in k+1\ldots n}]@\rho$$

the semantic relation

$$\Pi, A, \Delta \models \mathcal{C} : \Gamma$$

holds if the following holds with $\tau = \Pi(A, \vec{\rho})$:

$$\Pi, \Delta \cup \mathrm{frv}(\vec{\rho}), \tau \models b_i : \phi_i$$

for each $i \in k+1 \ldots n$.

**Definition 5 (Method interpretation).** Given a set of live region variables $\tilde{\Delta}$, a program type $\Pi$, a type $\tau$, a method declaration $b$ and signature $\phi$ with

$$b = m[\rho_{k+1}, \ldots, \rho_n](x)\ \{s\}$$
$$\phi = \forall\rho_{k+1}\cdots\rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2$$

the semantic relation

$$\Pi, \tilde{\Delta}, \tau \models b : \phi$$

holds if the following implication hold: For any program $\mathcal{P}$, region environments $R$, live regions $\Delta$, stores $\sigma$ and $\sigma'$, value environment $V'$, address $a$, values $v$ and $v_0$, and type $\tau_0$, if the following conditions hold

(i) $\tilde{\Delta} \models \mathcal{P} : \Pi$

(ii) $\Pi, \sigma, R, \Delta \models a : \tau$

(iii) $\Pi, \sigma, R, \Delta \models v : \tau_1$

(iv) $\Pi, \sigma, R, \Delta \models v_0 : \tau_0$

(v) $\Delta \subseteq \mathrm{Dom}(R)$

(vi) $\Delta' \subseteq \Delta$

(vii) $\sigma, \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}, R \vdash s \to v', \sigma', V'$

then it holds that

$$\Pi, \sigma', R, \Delta \models v', \tau_2$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0$$

**Definition 6 (Statement interpretation).** Given a program type $\Pi$, live regions $\Delta$, type environment $E$, statement $s$, and type $\tau$, the semantic relation

$$\Pi, \Delta, E \models s : \tau$$

holds if the following implication hold: For any program $\mathcal{P}$, live regions $\tilde{\Delta}$, program type $\Pi$, stores $\sigma$ and $\sigma'$, value environments $V$ and $V'$, values $v$ and $v_0$, and type $\tau_0$, if the following conditions hold

(i) $\tilde{\Delta} \models \mathcal{P} : \Pi$

(ii) $\Pi, \sigma, R, \Delta \models V : E$

(iii) $\Pi, \sigma, R, \Delta \models v_0 : \tau_0$

(iv) $\Delta \subseteq \mathrm{Dom}(R)$

(v) $\mathrm{frv}(E) \subseteq \Delta$

(vi) $\sigma, V, R \vdash s \to v, \sigma', V'$

then it holds that

$$\Pi, \sigma', R, \Delta \models v : \tau$$
$$\Pi, \sigma', R, \Delta \models V', E$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0$$

**Definition 7 (Expression interpretation).** Given a program type $\Pi$, live regions $\Delta$, type environment $E$, expression $e$, and type $\tau$, the semantic relation

$$\Pi, \Delta, E \models e : \tau$$

holds if the same implication as in Definition 6 holds with any occurrence of $s$ in the implication replaced with $e$.

We are now ready to formulate our theorem about soundness:

**Theorem 1 (Soundness).** *For any program $\mathcal{P}$, program type $\Pi$, class name $A$, class $\mathcal{C}$, class type $\Gamma$, region variables $\vec{\rho}$, method declaration $b$ and signature $\phi$, live region variables $\Delta$, type environment $E$, statement $s$, expression $e$, and type $\tau$, the following implications hold:*

$$\begin{aligned}
\Delta \vdash \mathcal{P} : \Pi \quad &\Longrightarrow \quad \Delta \models \mathcal{P} : \Pi \\
\Pi, A, \Delta \vdash \mathcal{C} : \Gamma \quad &\Longrightarrow \quad \Pi, A, \Delta \models \mathcal{C} : \Gamma \\
\Pi, \Delta, \tau \vdash b : \phi \quad &\Longrightarrow \quad \Pi, \Delta, \tau \models b : \phi \\
\Pi, \Delta, E \vdash s : \tau \quad &\Longrightarrow \quad \Pi, \Delta, E \models s : \tau \\
\Pi, \Delta, E \vdash e : \tau \quad &\Longrightarrow \quad \Pi, \Delta, E \models e : \tau
\end{aligned}$$

## 3.2 Useful lemmas

Before proving Theorem 1 in the next section, we need to prove a number of lemmas.

**Lemma 1 (Update).** *If the following conditions hold:*

*(i)* $\Pi, \sigma, R, \Delta \models a : [x : \tau \mid \ldots]@\rho$

*(ii)* $\sigma(a) = <A, [\ldots, x \mapsto v', \ldots], R>$

*(iii)* $\Pi, \sigma, R, \Delta \models v : \tau$

*(iv)* $\Pi, \sigma, R, \Delta \models v_0 : \tau_0$

*(v)* $\Pi, \sigma, R, \Delta \models V : E$

*(vi)* $\sigma' = \sigma + \{a \mapsto <A, [\ldots, x \mapsto v, \ldots], R>\}$

*Then*

$$\Pi, \sigma', R, \Delta \models a : [x : \tau]@\rho$$
$$\Pi, \sigma', R, \Delta \models V : E$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0$$

**Proof** Lemma 1 follows from the following Lemma 3, 4 and 5.

We start by proving a much simpler lemma:

**Lemma 2.** *If the following hold:*

*(i)* $\Pi, \sigma, R, \Delta \models a : [x : \tau \mid \ldots]@\rho$

*(ii)* $\Pi, \sigma, R, \Delta \models v : \tau$

*Then* $\Pi, \sigma + \{a \mapsto <A, \{x \mapsto v\}, \vec{r}>\}, R, \Delta \models a : [x : \tau \mid \ldots]@\rho$

**Proof** In other words, any consistent address can be updated with a consistent value of the proper type, and the address in the new store is still consistent. This is not trivial, since the update may introduce a circular reference.

To show this lemma we use co-induction. We try to use the notational style of [MT91] and let $C$ denote the set of tuples $(\Pi, \sigma, R, \Delta, v, \tau)$, let $\mathcal{F}$ denote a functional version of our consistency relation, and $S$ is the maximum fix-point of $\mathcal{F}$. For any $Q \subseteq U$, in order to prove $Q \subseteq S$, it is sufficient to prove that $Q$ is $\mathcal{F}$-consistent, i.e. that $Q \subseteq \mathcal{F}(Q)$.

Given $\Pi, \sigma, R, \Delta, a, v_0, \tau$ such that $\Pi, \sigma, R, \Delta \models a : [x_0 : \tau \mid \ldots]@\rho$ and $\Pi, \sigma, R, \Delta \models v_0 : \tau$ we define a set $Q$ in this manner: $Q = \{(\Pi, \sigma', R, \Delta, v', \tau') \in C \mid \sigma' = \sigma + \{a \mapsto \{x_0 \mapsto v_0\}\} \wedge \Pi, \sigma, R, \Delta \models v' : \tau'\}$

It is apparent that $(\Pi, \sigma', R, \Delta, v_0, \tau) \in Q$ and $(\Pi, \sigma', R, \Delta, a, [x_0 : \tau \mid \ldots]@\rho) \in Q$ by the preconditions.

Intuitively, $v', \tau'$ in $Q$ spans all the values that are consistent in $\sigma$. $\sigma'$ is the store we wish to show that the new value in $a$ has not introduced any inconsistencies in.

Now we wish to show that $Q$ is consistent, i.e. that $Q \subseteq S$, and we do this using the proof technique of *co-induction* by showing that $Q \subseteq \mathcal{F}(Q)$ (see [MT91]).

Here is the case analysis of Q:

**$\tau'$ is a simple type**

For all elements in $Q$ that have $\tau'$s different from object types, it is trivial that they belong to $\mathcal{F}(Q)$, since they belong to $\mathcal{F}(\emptyset)$.

**$v'$ is a dangling pointer in $\sigma$**

For $(\Pi, \sigma', R, \Delta, v', [x_i : \tau_i \mid \ldots]@\rho) \in Q$ and $\rho \notin \Delta$, all such values belong in $\mathcal{F}(Q)$ by definition, since $\Pi, \sigma', R, \Delta \models v' : [x_i : \tau_i \mid \ldots]@\rho$ by (Con Dangle) (and perhaps (Con Sub)), and so can be concluded by $\mathcal{F}(\emptyset)$.

**$v'$ is** $\mathsf{null}$

For $(\Pi, \sigma', R, \Delta, \mathsf{null}, [x_i : \tau_i \mid \ldots]@\rho) \in Q$, all such values also belong in $\mathcal{F}(Q)$ by definition, since they belong in $\mathcal{F}(\emptyset)$ by (Con Null) and (Con Sub). If $\tau'$ is $\mathsf{Null}$, then they belong in $\mathcal{F}(Q)$ directly by (Con Null).

**$v'$ is** $\mathsf{noval}$

For $(\Pi, \sigma', R, \Delta, \mathsf{noval}, \tau') \in Q$ and $\rho \in \Delta$, all such values also belong in $\mathcal{F}(Q)$ by definition, since they belong in $\mathcal{F}(\emptyset)$ by (Con Noval).

**The address $a$ itself**

For $(\sigma', R, \Delta, a, [x_0 : \tau \mid \ldots]@\rho) \in Q$ and $\rho \in \Delta$, such values belong in $\mathcal{F}(Q)$ by (Con Object), since $v_0, \tau$ is consistent in $\sigma$, and so we know that $(\Pi, \sigma', R, \Delta, v_0, \tau) \in Q$

**Other addresses $a'$**

For $(\Pi, \sigma', R, \Delta, a', [x_i : \tau_i \mid \ldots]@\rho) \in Q$ and $\rho \in \Delta$ and $a' \neq a$, such values belong in $\mathcal{F}(Q)$, because we know that all $(\sigma', R, \Delta, v_i, \tau_i)$, where $\{x_i \mapsto v_i\}$ in $a'$, is in $Q$, since we know that the condition for $a'$ being in $Q$ is that all $\Pi, \sigma, R, \Delta \models v_i, \tau_i$, and the contents of $a'$ has not changed in $\sigma$ to $\sigma'$ for $a' \neq a$. (If $v_i$ should happen to be $a$ itself we also know that $(\Pi, \sigma', R, \Delta, a, [x_0 : \tau \mid \ldots]@\rho) \in Q$ )

These cases cover all potential members of Q.

We now extend Lemma 1 to work if $a$ is of a type with more than one value:

**Lemma 3.** *If the following holds:*

*(i)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models a : [x_i : \tau_i \mid \ldots]@\rho$

*(ii)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v : \tau_j$

*Then* $\Pi, \sigma + \{a \mapsto <A, \tilde{V} + \{x_j \mapsto v\}, \vec{r}>\}, R, \Delta \models a : [x_i : \tau_i \mid \ldots]@\rho$

**Proof** Since all $v_i \neq v$ bound in $\tilde{V}$ are consistent in $\sigma$, $(\Pi, \sigma', R, \Delta, v_i, \tau_i)$ becomes elements of $Q$, and so it is a trivial extension to the previous proof to show, that $(\Pi, \sigma', R, \Delta, a, [x_0 : \tau_0 \mid \ldots]@\rho_0) \in Q$ still belong in $\mathcal{F}(Q)$. For all other values, the proof is identical.

We wish to generalise the lemma to the variable environment $V$ like this:

**Lemma 4.** *If the following holds:*

*(i)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models V : E$

*(ii)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v : \tau_j$

*Then* $\Pi, \sigma + \{a \mapsto <A, \tilde{V} + \{x_j \mapsto v\}, \vec{r}>\}, R, \Delta \models V : E$

**Proof** It follows from the precondition, that all values $(\Pi, \sigma', R, \Delta, v_i, \tau_i) \in Q$ for all $\{x_i \mapsto v_i\} \in V$. Since no $v_i$ changes in $V$, this proves that $\Pi, \sigma', R, \Delta \models V : E$.

Finally, we wish to generalise the lemma to any value consistent before the update:

**Lemma 5.** *If the following holds:*

*(i)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v_0 : \tau_0$

*(ii)* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v : \tau_j$

*Then* $\Pi, \sigma + \{a \mapsto <A, \tilde{V} + \{x_j \mapsto v\}, \vec{r}>\}, R, \Delta \models v_0 : \tau_0$

**Proof** It follows from the precondition, that $(\Pi, \sigma', R, \Delta, v_0, \tau_0) \in Q$. Since we know that $Q \subseteq S$ this proves the case.

For any environments $E$ and $E'$ we define the relation $E \sqsubseteq E'$ to mean that $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(E')$ and $E(x) = E'(x)$ for all $x \in \mathrm{Dom}(E)$. There are a number of changes to the environments in the consistency relation that does not destroy consistency. We list a number of these in the following lemma.

The following lemma states that we can safely remove dead region variables from $\Delta$.

**Lemma 6 (Removal of dead regions).** *If the following holds:*

*(i)* $\Pi, \sigma, R, \Delta \models V, E$

*(ii)* $\Delta' \subseteq \Delta$

*Then* $\Pi, \sigma, R, \Delta' \models V : E$

**Proof** $\Delta$ can be reduced, since this just means that all values in $\rho$ previously consistent by by (Con Object) now become consistent by (Con Dangle).

The following lemma states, that what $R$ contains aside from the entries matched in $\Delta$ does not matter.

**Lemma 7 (Change R outside $\Delta$).** *If the following holds:*

*(i)* $\Pi, \sigma, R, \Delta \models V, E$

*(ii)* $R' \sqsubseteq R$

*(iii)* $\Delta \subseteq \mathrm{Dom}(R')$

*(iv)* $R' \sqsubseteq R''$

*Then* $\Pi, \sigma, R', \Delta \models V : E$ *and* $\Pi, \sigma, R'', \Delta \models V : E$

**Proof** By definition of the consistency relation, all dangling pointers (with $\rho \notin \Delta$) are consistent.

**Lemma 8 (New address).** *If the following holds:*

*(i)* $\Pi, \sigma, R, \Delta \models v : \tau$

*(ii)* $a \notin \mathrm{Dom}(\sigma)$

*Then* $\Pi, \sigma + \{a \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v : \tau$

**Proof** If $a = (r, o)$, $R(\rho) = r$ and $\rho \in \Delta$ then $\Pi, \sigma, R, \Delta \models a : \tau'$ must have been false for any $\tau'$, and so we know that $v \neq a$ and also that $a$ is in no substructure of $v$.

If $\rho \notin \Delta$ or there is no binding in $R$, then any references to $a$ remain as consistent as they were before.

**Lemma 9 (New region).** *If the following conditions hold:*

*(i)* $\Pi, \sigma, R, \Delta \models v : \tau$

*(ii)* $\rho \notin \text{frv}(\tau)$

*(iii)* $r \notin \{\forall \rho' \in \Delta \mid R(\rho')\}$

*Then* $\Pi, \sigma + \{r \mapsto \emptyset\}, R + \{\rho \mapsto r\}, \Delta \cup \rho \models v : \tau$

**Proof** This lemma just states, that it is legal to create entirely new regions. Since $v$ cannot be typed to be in the new region, this does not affect consistency.

**Lemma 10 (Consistency subsumption).** *If the following conditions hold:*

*(i)* $\Pi, \sigma, R, \Delta \models v : \tau$

*(ii)* $\tau <: \tau'$

*Then* $\Pi, \sigma, R, \Delta \models v : \tau'$.

**Proof** Just apply (Con Sub) to the prerequisites.

**Lemma 11 (Substitution).** *Let $S$ be a substitution of region variables such that $R \circ S = R + \tilde{R}$. Then* $\Pi, \sigma, R, S(\Delta) \models v : S(\tau) \iff \Pi, \sigma, R + \tilde{R}, \Delta \models v : \tau$

**Proof** By induction on the depth of the consistency derivation, using

$$S(\text{frv}(\tau)) \subseteq S(\Delta) \iff \text{frv}(\tau) \subseteq \Delta$$

**Lemma 12 (R expansion).** *Given*

*(i)* $R(\rho') = r$

*(ii)* $\rho' \in \Delta$

*(iii)* $\rho \notin \Delta$

*(iv)* $\rho \notin \text{frv}(\tau)$

*Then* $\Pi, \sigma, R, \Delta \models v : \tau \iff \Pi, \sigma, R + \{\rho \mapsto r\}, \Delta + \{\rho\} \models v : \tau$

**Proof** Since the set of live regions is not expanded, no uses of (Con Dangle) can be transformed to uses of (Con Object). Since $\rho \notin \mathrm{frv}(\tau)$, all RegVars in $\tau$ have unchanged status with respect to $\Delta$, and so all derivations are unchanged by the expansion of R.

**Lemma 13 (Object Subtyping).** *If the following conditions hold:*

*(i)* $\Pi, \sigma, R, \Delta \models a : [x_i : \tau_i{}^{i\in 1\ldots n} \mid m_i : \phi_i{}^{i\in 1\ldots k}]@\rho$

*(ii)* $\rho \in \mathrm{Dom}(R)$

*Then*

$$r \in \mathrm{Dom}(\sigma)$$
$$o \in \mathrm{Dom}(\sigma(r))$$
$$\sigma(r, o) = <A, \tilde{V}, \vec{r}>$$
$$\exists \vec{\rho} : \Pi(A, \vec{\rho}) = \tau$$
$$\tau <: [x_i : \tau_i{}^{i\in 1\ldots n} \mid m_i : \phi_i{}^{i\in 1\ldots k}]@\rho$$
$$\Pi, \sigma, R, \Delta \models a : \tau$$

**Proof** Since $\rho \in \mathrm{Dom}(R)$, the final steps in the derivation of $\Pi, \sigma, R, \Delta \models a : [x_i : \tau_i{}^{i\in 1\ldots n} \mid m_i : \phi_i{}^{i\in 1\ldots k}]@\rho$ must be by the rule (Con Object), followed by zero or more (Con Sub) rules. The conclusion is just the prerequisites in the (Con Object).

## 3.3 Proof of soundness

This is the proof of Theorem 1. I is done by rule-based induction on the static semantics. Thus, there is a case for each rule in static semantics. The induction principle states the following. We are given a derivation tree leading to the judgement $\mathcal{J}$ which can take any of the five forms described in section 3.1.2. The rule directly leading to judgement $\mathcal{J}$ in the bottom of the derivation tree has the general form:

$$\frac{\mathcal{J}_1 \cdots \mathcal{J}_n}{\mathcal{J}}$$

having $n$ sub-judgements above the line (and possible also some additional side conditions). If we can prove that the theorem holds for the judgement $\mathcal{J}$ assuming only that theorem holds for all of the $n$ sub-judgements $\mathcal{J}_1, \ldots, \mathcal{J}_n$, then the theorem holds for any judgement.

Each of the implications of the theorem matches one of the five different kinds of judgements. Correspondingly, the proof is naturally divided into five parts.

### 3.3.1 Programs

For programs we assume the judgement of the form

$$\Delta \vdash \mathcal{P} : \Pi$$

and we aim at proving the semantic relation

$$\Delta \models \mathcal{P} : \Pi$$

as defined in Definition 3. There is only one rule that matches the above judgement.

**(Reg Program)** The conclusion of this rule is

$$\Delta \vdash \mathcal{C}_1 \cdots \mathcal{C}_n : \Pi \tag{1}$$

and the premises are

$$\Pi = \{A_1 \mapsto \Gamma_1, \ldots, A_n \mapsto \Gamma_n\} \tag{2}$$
$$\Pi, A_i, \Delta \vdash \mathcal{C}_i : \Gamma_i \text{ for all } i \in 1 \ldots n \tag{3}$$

By induction on each of the $n$ sub-judgements in 3, we get

$$\Pi, A_i, \Delta \models \mathcal{C}_i : \Gamma_i \text{ for all } i \in 1 \ldots n \tag{4}$$

and from Definition 3 we therefore have

$$\Delta \models \mathcal{C}_1 \cdots \mathcal{C}_n : \Pi \tag{5}$$

which proves the case.

### 3.3.2 Classes

For classes we assume the judgement of the form

$$\Pi, A, \Delta \vdash \mathcal{C} : \Gamma$$

and we aim at proving the semantic relation

$$\Pi, A, \Delta \models \mathcal{C} : \Gamma$$

as defined in Definition 4. There is only one rule that matches the above judgement.

**(Reg Class)** The conclusion of this rule is

$$\Pi, A, \Delta \vdash \mathsf{class}\ A[\vec{\rho}]\ \mathsf{at}\ \rho\ \mathsf{extends}\ B[\vec{p}]\ \{b_i\ ^{i \in 1 \ldots n}\} : \lambda \vec{\rho}.\tau \tag{6}$$

and the premises are

$$\Pi(B, \vec{p}) = \tau' \tag{7}$$
$$\vdash \tau <: \tau' \tag{8}$$
$$\tau = [x_i : \tau_i\ ^{i \in 1 \ldots k} \mid m_i : \phi_i\ ^{i \in k+1 \ldots n}]@\rho \tag{9}$$
$$\Delta \cup \mathrm{frv}(\vec{\rho}) \vdash b_i : \tau_i \text{ for all } i \in 1 \ldots k \tag{10}$$
$$\Pi, \Delta \cup \mathrm{frv}(\vec{\rho}), \tau \vdash b_i : \phi_i \text{ for all } i \in k+1 \ldots n \tag{11}$$
$$\Delta \cap \mathrm{frv}(\vec{\rho}) = \emptyset \tag{12}$$
$$\rho \in \Delta \cup \mathrm{frv}(\vec{\rho}) \tag{13}$$

By induction on each of the sub-judgements in (11), we get

$$\Pi, \Delta \cup \mathrm{frv}(\vec{\rho}), \tau \models b_i : \phi_i \text{ for all } i \in k+1 \ldots n \tag{14}$$

and from Definition 4 we therefore have

$$\Pi, A, \Delta \models \mathsf{class}\ A[\vec{\rho}]\ \mathsf{at}\ \rho\ \mathsf{extends}\ B[\vec{p}]\ \{b_i\ ^{i \in 1 \ldots n}\} : \Gamma \tag{15}$$

which proves the case.

### 3.3.3 Methods

For methods we assume the judgement of the form

$$\Pi, \Delta, \tau \vdash b : \phi$$

and we aim at proving the semantic relation

$$\Pi, \Delta, \tau \models b : \phi$$

as defined in Definition 5. There is only one rule that matches the above judgement.

**(Reg Method)** The conclusion of this rule is

$$\Pi, \Delta, \tau \vdash \tau_2\ m[\vec{\rho}](x)\ \{s\} : \forall \vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2 \tag{16}$$

and the premises are

$$\Pi, \Delta', E \vdash s : \tau_2 \tag{17}$$
$$\mathrm{frv}(\tau_1) \cup \mathrm{frv}(\tau_2) \cup \mathrm{frv}(\tau) \subseteq \Delta' \tag{18}$$
$$\Delta' \subseteq \Delta \cup \mathrm{frv}(\vec{\rho}) \tag{19}$$

where $E = \{x : \tau_1\} + \{\mathsf{this} : \tau\}$.

We want to prove that

$$\Pi, \Delta, \tau \models \tau_2\ m[\vec{\rho}](x)\ \{s\} : \forall \vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2 \tag{20}$$

as defined in Definition 5. Therefore, we start by assuming that we are given program $\mathcal{P}$, region environments $R$, live regions $\Delta$, stores $\sigma$ and $\sigma'$, value environment $V'$, address $a$, values $v$ and $v_0$, and type $\tau_0$.

We further assume that the following conditions hold

$$\tilde{\Delta} \models \mathcal{P} : \Pi \tag{21}$$
$$\Pi, \sigma, R, \Delta \models a : \tau \tag{22}$$
$$\Pi, \sigma, R, \Delta \models v : \tau_1 \tag{23}$$
$$\Pi, \sigma, R, \Delta \models v_0 : \tau_0 \tag{24}$$
$$\Delta \subseteq \mathrm{Dom}(R) \tag{25}$$
$$\Delta' \subseteq \Delta \tag{26}$$
$$\sigma, V, R \vdash s \to v', \sigma', V' \tag{27}$$

where $V = \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}$.

We will then try to prove that

$$\Pi, \sigma', R, \Delta \models v', \tau_2 \tag{28}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{29}$$

Using Lemma 6 on (22), (23) and (24) we get

$$\Pi, \sigma, R, \Delta' \models a : \tau \tag{30}$$
$$\Pi, \sigma, R, \Delta' \models v : \tau_1 \tag{31}$$
$$\Pi, \sigma, R, \Delta' \models v_0 : \tau_0 \tag{32}$$

and because of (25) and (26) we also have

$$\Delta' \subseteq \mathrm{Dom}(R) \tag{33}$$

By induction on (17) we get

$$\Pi, \Delta', E \models s : \tau_2 \tag{34}$$

Because of (30) and (31) we get by the definition of $V$ in (19)that

$$\Pi, \sigma, R, \Delta' \models V, E \tag{35}$$

From (35) and Definition 6 we get that

$$\Pi, \sigma', R, \Delta' \models v', \tau_2 \tag{36}$$
$$\Pi, \sigma', R, \Delta' \models V' : E \tag{37}$$
$$\Pi, \sigma', R, \Delta' \models v_0 : \tau_0 \tag{38}$$

because the assumptions in the definition are matched by (21), (35), (32), (33), (18) and (27) respectively.

We have

$$\Pi, \sigma', R, \Delta \models v', \tau_2 \tag{39}$$

since we can expand $\Delta'$ to $\Delta$ since $\tau_2$ can not be typed in the new regions (from (18)).

Now consider the derivation of (24). For all sub-derivations of the form $\Pi, \sigma, R, \Delta \models a : [\cdots]@\rho$ and $R(\rho) \in \{R(\rho') \mid \rho' \in \Delta'\}$ we know that $\Pi, \sigma', R, \Delta' \models a : [\cdots]@\rho'$, where $\rho' \in \Delta'$ and $R(\rho') = R(\rho)$ by the induction on $s$ (let $v_0$ be $a$). Some derivations now may be by (Con Dangle) instead of the original (Con Object). Since there is no way to create *new* dangling pointers to regions outside the local $\Delta$, and no way to "reawaken" a dead region outside its scope, we can conclude $\Pi, \sigma', R, \Delta \models a : [\cdots]@\rho'$ since the regions outside $\{R(\rho') \mid \rho' \in \Delta'\}$ cannot have changed from $\sigma$ to $\sigma'$. Effectively, the "lost links" in the chain are guaranteed to be restored correctly.

Given this, we can now conclude

$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{40}$$

by the same argument.

(39) and (40) prove the case.

### 3.3.4 Expressions

For expressions we assume the judgement of the form

$$\Pi, \Delta, E \vdash e : \tau$$

and we aim at proving the semantic relation

$$\Pi, \Delta, E \models e : \tau$$

as defined in Definition 7. That is, we assume the following:

$$\tilde{\Delta} \models \mathcal{P} : \Pi \tag{E1}$$
$$\Pi, \sigma, R, \Delta \models V : E \tag{E2}$$
$$\Delta = \mathrm{Dom}(R) \tag{E3}$$
$$\sigma, V, R \vdash e \to v, \sigma', V' \tag{E4}$$
$$\mathrm{frv}(E) \subseteq \Delta \tag{E5}$$
$$\Pi, \sigma, R, \Delta \models v_0 : \tau_0 \tag{E6}$$

Then we want to prove that the following hold:

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{E7}$$
$$\Pi, \sigma', R, \Delta \models V', E \tag{E8}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{E9}$$

There are several rules to check; one for each syntactic form of the expression $e$ and one for subsumption. We make a case for each rule.

When applying the induction hypothesis on each of the sub-judgements of a rule, one generally has to justify that all the assumptions in Definition 7 hold. However, in many cases these assumptions hold trivially because they are the same as the six basic assumptions (E1)–(E6). This is especially true for assumptions (E1), (E5) and (E3). We only argue for assumption (E3) explicitly if $\Delta$ or $R$ changes. We only argue for (E5) when $E$ change (never in expressions). Assumption (E1) obviously hold through the derivation, since neither $\mathcal{P}$ nor $\Pi$ changes anywhere in the derivation of expression.

Likewise, in many cases conclusion (E9) is implied trivially by the similar conclusion in the last inductive step of the case. However, in the first case, for the sake of clarity, we argue explicitly for conclusion (E9).

**(RegExp Plus)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e_1 + e_2 : \mathsf{int} \tag{41}$$

and the premises are

$$\Pi, \Delta, E \vdash e_1 : \mathsf{int} \tag{42}$$
$$\Pi, \Delta, E \vdash e_2 : \mathsf{int} \tag{43}$$

We will now examine which of the rules in the dynamic semantics that can be applied in the evaluation of $e_1 + e_2$. There is only one explicit rules the matches this expression. However, as mentioned in section 3.1.1 there are also two implicit error rules that match. We first consider the error rules.

(DynExp Plus)
$$\frac{\sigma, V, R \vdash e_1 \to i_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to i_2, \sigma'', V''}{\sigma, V, R \vdash e_1 + e_2 \to i_1 + i_2, \sigma'', V''}$$

(DynExp Plus Err1)
$$\frac{\sigma, V, R \vdash e_1 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e_1 + e_2 \to \mathsf{error}, \sigma', V'}$$

(DynExp Plus Err2)
$$\frac{\sigma, V, R \vdash e_1 \to i_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash e_1 + e_2 \to \mathsf{error}, \sigma'', V''}$$

Consider the evaluation $\sigma, V, R \vdash e_1 \to v_1, \sigma', V'$. Because (42) is a premise of rule (RegExp Plus), we can apply the induction hypothesis to obtain

$$\Pi, \sigma', R, \Delta \models v_1 : \mathsf{int} \tag{44}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{45}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{46}$$

From (44) we can draw the important conclusion that $v_1$ must be an integer. Specifically $v_1$ cannot be error. And from that conclusion we can further conclude that the error rule (DynExp Plus Err1) cannot be applied.

Next, consider the evaluation $\sigma', V', R \vdash e_2 \to v_2, \sigma'', V''$. Because (43) is a premise of rule (RegExp Plus) and because of (45), we can apply the induction hypothesis again to obtain

$$\Pi, \sigma'', R, \Delta \models v_2 : \mathsf{int} \tag{47}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{48}$$
$$\Pi, \sigma'', R, \Delta \models v_0 : \tau_0 \tag{49}$$

From (47) we can now conclude that $v_2$ is an integer and not error. We therefore conclude that rule (DynExp Plus Err2) cannot be applied.

Consequently, the only rule (DynExp Plus) can be applied. Furthermore, we have that $v = v_1 + v_2$ is an integer, so we can write

$$\Pi, \sigma'', R, \Delta \models v : \mathsf{int} \tag{50}$$

From (48),(49) and (50) we have now proven the case. In the further parts of the proof, we will usually not give the conclusions corresponding to (49), if they are as trivial as in this case.

**(RegExp Sub)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e : \tau' \tag{51}$$

and the premises are

$$\Pi, \Delta, E \vdash e : \tau \tag{52}$$
$$\vdash \tau <: \tau' \tag{53}$$

Any of the expression rules in the dynamic semantics can match $e$. We therefore just consider the evaluation $\sigma, V, R \vdash e \to v, \sigma', V'$. By induction on (52) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{54}$$
$$\Pi, \sigma', R, \Delta \models V : E \tag{55}$$

Because of (53) and (54) we can use Lemma 10 to further get

$$\Pi, \sigma', R, \Delta \models v : \tau' \tag{56}$$

From (55) and (56) we have now proven the case.

**(RegExp $x$)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash x : \tau \tag{57}$$

and the premise is

$$E(x) = \tau \tag{58}$$

There is one explicit rule of the dynamic semantics that match, and there is one implicit error rule:

$$\frac{V(x) = v}{\sigma, V, R \vdash x \to v, \sigma, V} \text{ (DynExp } x\text{)} \qquad \frac{x \notin \mathrm{Dom}(V)}{\sigma, V, R \vdash x \to \mathsf{error}, \sigma, V} \text{ (DynExp } x \text{ Err)}$$

Precondition E2 implies that $\mathrm{Dom}(V) = \mathrm{Dom}(E)$ and (58) implicitly states that $x \in \mathrm{Dom}(E)$, so we see that $x \in \mathrm{Dom}(V)$. Therefore, the rule (DynExp $x$ Err) cannot be applied.

Consequently, only rule (DynExp $x$) can be applied. Because of (58), precondition E2 gives us what we need:

$$\Pi, \sigma, R, \Delta \models V : E \tag{59}$$
$$\Pi, \sigma, R, \Delta \models v : \tau \tag{60}$$

**(RegExp Int)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash i : \mathsf{int} \tag{61}$$

and there are no premises.

Only one rule in the dynamic semantics applies:

(DynExp Const)

$$\frac{}{\sigma, V, R \vdash c \to c, \sigma, V}$$

Since neither $\sigma$ nor $V$ is changed during evaluation, it is trivial that

$$\Pi, \sigma, R, \Delta \models V : E \tag{62}$$
$$\Pi, \sigma, R, \Delta \models i : \mathsf{int} \tag{63}$$

Here (63) holds by definition.

**(RegExp False)** The proof of this case is completely similar to the case for (RegExp Int).

**(RegExp True)** The proof of this case is completely similar to the case for (RegExp Int).

**(RegExp Null)** The proof of this case is completely similar to the case for (RegExp Int).

**(RegExp Equal)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e_1 == e_2 : \mathsf{boolean} \tag{64}$$

and the premises are

$$\Pi, \Delta, E \vdash e_1 : \tau \tag{65}$$
$$\Pi, \Delta, E \vdash e_2 : \tau \tag{66}$$

There are two explicit rules in the dynamic semantics that match the expression, and there are two implicit error rules:

(DynExp Equal True)

$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \qquad v_1 = v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{true}, \sigma'', V''}$$

(DynExp Equal False)

$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \qquad v_1 \neq v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{false}, \sigma'', V''}$$

(DynExp Equal Err1)

$$\frac{\sigma, V, R \vdash e_1 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{error}, \sigma', V'}$$

(DynExp Equal Err2)
$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{error}, \sigma'', V''}$$

Consider the evaluation $\sigma, V, R \vdash e_1 \to v_1, \sigma', V'$. By induction on the premise (65) we get that

$$\Pi, \sigma', R, \Delta \models v_1 : \tau \tag{67}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{68}$$

Here (67) tells us that $v_1$ is not error, so rule (DynExp Equal Err1) cannot be applied.

Next, we consider the evaluation $\sigma', V', R \vdash e_2 \to v_2, \sigma'', V''$. Because of (68) we can use induction on the premise (66), and we get that

$$\Pi, \sigma'', R, \Delta \models v_2 : \tau \tag{69}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{70}$$

Here (69) tells us that $v_2$ is not error, so rule (DynExp Equal Err2) cannot be applied either.

Now we only need to consider the two explicit rules. So the resulting value is either true or false. By definition we have

$$\Pi, \sigma'', R, \Delta \models \mathsf{true} : \mathsf{boolean} \tag{71}$$
$$\Pi, \sigma'', R, \Delta \models \mathsf{false} : \mathsf{boolean} \tag{72}$$

and since we also have (70), the case is proven.

**(RegExp Assign)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash x = e : \tau \tag{73}$$

and the premises are

$$E(x) = \tau \tag{74}$$
$$\Pi, \Delta, E \vdash e : \tau \tag{75}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there is one implicit error rule:

(DynExp Assign)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash x = e \to v, \sigma', V' + \{x \mapsto v\}}$$

(DynExp Assign Err)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash x = e \to \mathsf{error}, \sigma', V'}$$

Consider the evaluation $\sigma, V, R \vdash e \to v, \sigma', V'$. By induction on the premise (75) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{76}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{77}$$

From (76) we then see that $v \neq \mathsf{error}$. Therefore, (DynExp Assign Err) cannot be applied.

Precondition E2 implies that $\mathrm{Dom}(V) = \mathrm{Dom}(E)$ and (74) implicitly states that $x \in \mathrm{Dom}(E)$, so we see that $x \in \mathrm{Dom}(V)$. That is, in (DynExp Assign) we only *update* the environment $V'$.

From (77) we have that

$$\mathrm{Dom}(V') = \mathrm{Dom}(E) \tag{78}$$

Putting $V'' = V + \{x \mapsto v\}$ we have

$$\mathrm{Dom}(V'') = \mathrm{Dom}(E) \tag{79}$$

Furthermore, from (74), (76) and (77) we also have

$$\Pi, \sigma', V'', R, \Delta \models V'', E \tag{80}$$

This together with (76) proves the case.

**(RegExp Field)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e.x : \tau \tag{81}$$

and the premises are

$$\Pi, \Delta, E \vdash e : [x : \tau \mid \ldots]@\rho \tag{82}$$
$$\rho \in \Delta \tag{83}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there are four implicit error rules:

(DynExp Field)
$$\frac{\sigma, V, R \vdash e \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, \vec{r}> \qquad \tilde{V}(x) = v}{\sigma, V, R \vdash e.x \to v, \sigma', V'}$$

(DynExp Field Err1)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e.x \to \mathsf{error}, \sigma', V'}$$

(DynExp Field Err2)
$$\frac{\sigma, V, R \vdash e \to (r, o), \sigma', V' \qquad r \notin \mathrm{Dom}(\sigma)}{\sigma, V, R \vdash e.x \to \mathsf{error}, \sigma', V'}$$

(DynExp Field Err3)
$$\frac{\sigma, V, R \vdash e \to (r, o), \sigma', V' \quad r \in \mathrm{Dom}(\sigma) \quad o \notin \mathrm{Dom}(\sigma(r))}{\sigma, V, R \vdash e.x \to \mathsf{error}, \sigma', V'}$$

(DynExp Field Err4)
$$\frac{\sigma, V, R \vdash e \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, \vec{r}> \quad x \notin \mathrm{Dom}(\tilde{V})}{\sigma, V, R \vdash e.x \to \mathsf{error}, \sigma', V'}$$

Consider the evaluation $\sigma, V, R \vdash e \to a, \sigma', V'$. By induction on (82) we get

$$\Pi, \sigma', R, \Delta \models a : [x : \tau \mid \ldots]@\rho \tag{84}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{85}$$

From (84) we then see that $a \neq \mathsf{error}$. Therefore, (DynExp Field Err1) cannot be applied.

From (83) (and by $\Delta = \mathrm{Dom}(R)$ from our induction hypothesis ) we see that (Con Object) was applied to obtain (84). Therefore, the premises of (Con Object) hold:

$$r \in \mathrm{Dom}(\sigma) \tag{86}$$
$$o \in \mathrm{Dom}(\sigma(r)) \tag{87}$$
$$\tilde{V}(x) = v \tag{88}$$
$$\Pi, \sigma', R, \Delta \models v : \tau \tag{89}$$

Because of (86) and (87) we see that neither (DynExp Field Err2) nor (DynExp Field Err3) can be applied. Last, because of (88) we see that (DynExp Field Err4) cannot be applied.

From (85) and (89) we have proven the case.

**(RegExp Update)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e_1.x = e_2 : \tau \tag{90}$$

and the premises are

$$\Pi, \Delta, E \vdash e_1 : [x : \tau \mid \ldots]@\rho \tag{91}$$
$$\Pi, \Delta, E \vdash e_2 : \tau \tag{92}$$
$$\rho \in \Delta \tag{93}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there are four implicit error rules:

(DynExp Update)
$$\frac{\sigma, V, R \vdash e_1 \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, \vec{r}> \quad \sigma', V', R \vdash e_2 \to v, \sigma'', V''}{\sigma, V, R \vdash e_1.x = e_2 \to v, \sigma'' + \{a \mapsto <A, \tilde{V} + \{x \to v\}, \vec{r}>\}, V''}$$

(DynExp Update Err1)
$$\frac{\sigma, V, R \vdash e_1 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e_1.x = e_2 \to \mathsf{error}, \sigma', V'}$$

(DynExp Update Err2)
$$\frac{\sigma, V, R \vdash e_1 \to (r, o), \sigma', V' \qquad r \notin \mathrm{Dom}(\sigma)}{\sigma, V, R \vdash e_1.x = e_2 \to \mathsf{error}, \sigma', V'}$$

(DynExp Update Err3)
$$\frac{\sigma, V, R \vdash e_1 \to (r, o), \sigma', V' \qquad r \in \mathrm{Dom}(\sigma) \qquad o \notin \mathrm{Dom}(\sigma(r))}{\sigma, V, R \vdash e_1.x = e_2 \to \mathsf{error}, \sigma', V'}$$

(DynExp Update Err4)
$$\frac{\sigma, V, R \vdash e_1 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, \vec{r}> \qquad \sigma', V', R \vdash e_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash e_1.x = e_2 \to \mathsf{error}, \sigma'', V''}$$

Consider $\sigma, V, R \vdash e_1 \to a, \sigma', V'$. By induction on (91) we get

$$\Pi, \sigma', R, \Delta \models a : [x : \tau \mid \ldots]@\rho \tag{94}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{95}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{96}$$

From (94) we then see that $a \neq \mathsf{error}$. Therefore, (DynExp Update Err1) cannot be applied.

From (93) and by $\Delta = \mathrm{Dom}(R)$ from our induction hypothesis we see that (Con Object) was applied to obtain (94). Therefore, the following premises of (Con Object) hold with $a = (r, o)$:

$$r \in \mathrm{Dom}(\sigma) \tag{97}$$
$$o \in \mathrm{Dom}(\sigma(r)) \tag{98}$$

Because of (97) and (98) we see that neither (DynExp Update Err2) nor (DynExp Update Err3) can be applied.

Next, consider $\sigma', V', R \vdash e_2 \to v, \sigma'', V''$. Because of (95), by induction on (92) we get

$$\Pi, \sigma'', R, \Delta \models v : \tau \tag{99}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{100}$$
$$\Pi, \sigma'', R, \Delta \models v_0 : \tau_0 \tag{101}$$

From (99) we then see that $v \neq \mathsf{error}$. Therefore, (DynExp Update Err4) cannot be applied. Consequently, only (DynExp Update) can be applied.

In the second inductive step we can also use that a value remains consistent during evaluation ($v'$ in our induction hypothesis). We use this on the value $a$ in (94) to obtain

$$\Pi, \sigma'', R, \Delta \models a : [x : \tau \mid \ldots]@\rho \tag{102}$$

Again, from (93) and $\Delta = \mathrm{Dom}(R)$ we see that (Con Object) was applied to obtain (102). Therefore, the following premise of (Con Object) holds:

$$\sigma''(a) =< A, [\ldots, x \mapsto v', \ldots], R > \tag{103}$$

Using this together with (99) and (100) and by defining $\sigma''' = \sigma'' + \{a \mapsto < A, [\ldots, x \mapsto v, \ldots], R >$ we obtain from Lemma 1 that

$$\Pi, \sigma''', R, \Delta \models a : [x : \tau \mid \ldots]@\rho \tag{104}$$
$$\Pi, \sigma''', R, \Delta \models V'' : E \tag{105}$$

Moreover, from (93) and $\Delta = \mathrm{Dom}(R)$ we see that (Con Object) was applied to obtain (104). From the premises of (Con Object) we therefore also have

$$\Pi, \sigma''', R, \Delta \models v : \tau \tag{106}$$

By Lemma 5,(99) and (101) we have

$$\Pi, \sigma''', R, \Delta \models v_0 : \tau_0 \tag{107}$$

From (104), (106) and (107) we have proven the case.

**(RegExp New)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{new}[\rho_1, \ldots, \rho_k]\ A() : \tau@\rho \tag{108}$$

and the premises are

$$A[\rho_1, \ldots, \rho_k] = [x_i : \tau_i{}^{\,i \in 1 \ldots n} \mid \ldots]@\rho \tag{109}$$
$$\{\rho_1, \ldots, \rho_k\} \subseteq \Delta \tag{110}$$
$$\rho \in \Delta \tag{111}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there is one implicit error rule:

(DynExp New)

$$\frac{\tilde{V} = \{x_1 \mapsto \mathbf{init}(\tau_1), \ldots, x_n \mapsto \mathbf{init}(\tau_n)\} \qquad \vec{r} = \{R(\rho_1), \ldots, R(\rho_k)\} \quad A[\rho_1, \ldots, \rho_k] = [x_i : \tau_i{}^{\,i \in 1 \ldots n} \mid \ldots]@\rho \qquad R(\rho) = r \qquad o \notin \mathrm{Dom}(\sigma(r))}{\sigma, V, R \vdash \mathsf{new}\ A[\rho_1, \ldots, \rho_k]() \to (r, o), \sigma + \{(r, o) \mapsto < A, \tilde{V}, \vec{r} >\}, V}$$

(DynExp New Err)

$$\frac{\{\rho, \rho_1, \ldots, \rho_k\} \not\subseteq \mathrm{Dom}(R)}{\sigma, V, R \vdash \mathsf{new}\ A[\rho_1, \ldots, \rho_k]() \to \mathsf{error}, \sigma, V}$$

In this case, the only thing that can result in an error is the explicit lookup in the region environment $R$. Actually, (DynExp New Err) may be seen as $k + 1$ separate error rules—one for each region variable. We choose to treat them together because the arguments are the same for each of them. There are no other error rules since the remaining conditions are just abbreviations or express that the address $(r, o)$ is a fresh address in $\sigma$.

Because of (110), (111) and prerequisite E3, we easily see that

$$\{\rho, \rho_1, \ldots, \rho_k\} \subseteq \mathrm{Dom}(R) \tag{112}$$

which means that (DynExp New Err) cannot be applied. Consequently, only (DynExp New) can be applied.

By definition we have

$$\Pi, \sigma, R, \Delta \models \mathbf{init}(\tau) : \tau \tag{113}$$

Using this it is easy to see that extending the store with the newly constructed object value $<A, \tilde{V}, \vec{r}>$ preserves consistency:

$$\Pi, \sigma + \{(r,o) \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models (r,o) : [x_i : \tau_i \,^{i \in 1 \ldots n} \mid \ldots]@\rho \tag{114}$$

Also, since $(r,o)$ is a fresh address in $\sigma$, we can use Lemma 8 on every value in $V$ to obtain

$$\Pi, \sigma + \{(r,o) \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models V : E \tag{115}$$

From Lemma 8 we also have

$$\Pi, \sigma + \{(r,o) \mapsto <A, \tilde{V}, \vec{r}>\}, R, \Delta \models v_0 : \tau_0 \tag{116}$$

From (114),(115) and (116) we have proven the case.

**(RegExp Method)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e_0.m[S(\rho_{k+1}), \ldots, S(\rho_n)](e_1) : S(\tau_2) \tag{117}$$

and the premises are

$$\Pi, \Delta, E \vdash e_0 : [m : \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2]@\rho \tag{118}$$
$$\Pi, \Delta, E \vdash e_1 : S(\tau_1) \tag{119}$$
$$\rho \in \Delta \tag{120}$$
$$S(\Delta') \subseteq \Delta \tag{121}$$
$$\{S(\rho_{k+1}), \ldots, S(\rho_n)\} \subseteq \Delta \tag{122}$$

Where $S$ is the substitution $\{\rho_{k+1} \mapsto \rho'_{k+1}, \ldots, \rho_n \mapsto \rho'_n\}$.

There is one explicit rule in the dynamic semantics that matches the expression, and there are seven implicit error rules.

(DynExp Method) ($x$ and $s$ is the formal parameter and body of method $m$ in class $A$)

$$\frac{\begin{array}{c}\Pi(A) = \lambda\rho_1 \cdots \rho_k.[\cdots, m : \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta} \tau 2, \cdots]@\rho \\ \sigma, V, R \vdash e_0 \to a, \sigma', V' \quad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \quad \sigma', V', R \vdash e_1 \to v, \sigma'', V'' \\ \tilde{R} = \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k, \rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\} \\ \sigma'', \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}, R + \tilde{R}, \vdash s \to v', \sigma''', V'''\end{array}}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to v', \sigma''', V''}$$

(DynExp Method Err1)

$$\frac{\sigma, V, R \vdash e_0 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma', V'}$$

(DynExp Method Err2)

$$\frac{\sigma, V, R \vdash e_0 \to (r, o), \sigma', V' \qquad r \notin \mathrm{Dom}(\sigma)}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma', V'}$$

(DynExp Method Err3)

$$\frac{\sigma, V, R \vdash e_0 \to (r, o), \sigma', V' \qquad r \in \mathrm{Dom}(\sigma) \qquad o \notin \mathrm{Dom}(\sigma(r))}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma', V'}$$

(DynExp Method Err4)

$$\frac{\sigma, V, R \vdash e_0 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \qquad m \text{ does not exists in class } A}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma''', V''}$$

(DynExp Method Err5) ($x$ and $s$ is the formal parameter and body of method $m$ in class $A$)

$$\frac{\sigma, V, R \vdash e_0 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \qquad \sigma', V', R \vdash e_1 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma'', V''}$$

(DynExp Method Err6)

$$\frac{\{\rho'_{k+1}, \ldots, \rho'_n\} \not\subseteq \mathrm{Dom}(R)}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma, V}$$

(DynExp Method Err7) ($x$ and $s$ is the formal parameter and body of method $m$ in class $A$)

$$\frac{\begin{array}{c}\sigma, V, R \vdash e_0 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \\ \sigma', V', R \vdash e_1 \to v, \sigma'', V'' \\ \tilde{R} = \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k, \rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\} \\ \sigma'', \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}, R + \tilde{R} \vdash s \to \mathsf{error}, \sigma''', V'''\end{array}}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to \mathsf{error}, \sigma''', V''}$$

Consider the evaluation $\sigma, V, R \vdash e_0 \to a, \sigma', V'$. By induction on (118) we get

$$\Pi, \sigma', R, \Delta \models a : [m : \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2]@\rho \tag{123}$$

$$\Pi, \sigma', R, \Delta \models V' : E \tag{124}$$

$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{125}$$

$$\tag{126}$$

From (123) we then see that $a \neq \mathsf{error}$. Therefore, (DynExp Method Err1) cannot be applied.

From (120) (and $\Delta = \mathrm{Dom}(R)$ from the induction hypothesis) we see that (Con Object) was applied to obtain (123). Therefore, the following premises of (Con Object) hold with $a = (r, o)$:

$$r \in \mathrm{Dom}(\sigma) \tag{127}$$
$$o \in \mathrm{Dom}(\sigma(r)) \tag{128}$$

Because of (127) and (128) we see that we cannot apply neither (DynExp Method Err2) nor (DynExp Method Err3).

From (Con Object) we have

$$\Pi(A, \rho'_1, \ldots, \rho_n, m) = \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2 \tag{129}$$

By our induction hypothesis we have $\tilde{\Delta}\mathcal{P} : \Pi$, so $m$ must exists in $A$ in $\mathcal{P}$, and (DynExp Method Err4) cannot be used.

Next, consider the evaluation $\sigma', V', R \vdash e_1 \to v, \sigma'', V''$. Because of (124), we find by induction on (119):

$$\Pi, \sigma'', R, \Delta \models v : S(\tau_1) \tag{130}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{131}$$
$$\Pi, \sigma'', R, \Delta \models v_0 : \tau_0 \tag{132}$$
$$\tag{133}$$

From (130) we then see that $v \neq \mathsf{error}$. Therefore, (DynExp Method Err5) cannot be applied. So we are left with (DynExp Method),(DynExp Method Err6) and (DynExp Method Err7). We want to prove that only the former can be applied.

From (122) we see that (DynExp Method Err6) cannot be applied.

using $a$ for $v'$ in the induction hypothesis also gives us

$$\Pi, \sigma'', R, \Delta \models a : [m : \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2]@\rho \tag{134}$$

By Lemma 13 using (120) we get

$$\exists \rho'_1, \ldots, \rho'_k : \Pi, \sigma'', R, \Delta \models a : \Pi(A, \rho'_1, \ldots, \rho'_k) \tag{135}$$

To avoid name capture, all occurrences of $\rho_1 \ldots \rho_k$ are substituted the new Region variables in $R$,$\Delta$,$\tau_0$ and $\mathrm{Rng}(S)$, and $\rho'_1, \ldots, \rho'_n$. This does not affect any consistency results up to this point, since the consistency relation is safe for such global renaming .

We define $\Delta''$ such that

$$S(\Delta'') = \Delta \tag{136}$$

We now insert the formal parameter bindings in $R$. We get

$$\Pi, \sigma'', R + \{\rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\}, \Delta'' \models a : \Pi(A, \rho'_1, \ldots, \rho'_k) \quad (137)$$

$$\Pi, \sigma'', R + \{\rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\}, \Delta'' \models v_0 : \tau_0 \quad (138)$$

from (135) and (132) by using lemma (12) for $\rho_i \notin \mathrm{Dom}(R)$. For $\rho_i \notin \mathrm{Dom}(R)$ the substitution maps R to itself. To achieve a substitution in $\Delta$ rather than an expansion, we use lemma 6.

We get

$$\Pi, \sigma'', R + \{\rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\}, \Delta'' \models v : \tau_1 \quad (139)$$

from (130) using lemma 11 for S. We now insert the object bindings in $R$.

We define a substitution S', such that

$$S' = \{\rho_1 \mapsto \rho'_1, \ldots, \rho_k \mapsto \rho'_k\} \quad (140)$$

We define $\Delta'''$ such that

$$S'(\Delta''') = \Delta'' \quad (141)$$

We get

$$\Pi, \sigma'', R + \tilde{R}, \Delta''' \models a : \Pi(A, \rho_1, \ldots, \rho_k) \quad (142)$$

by using lemma 11 (and lemma 7 for the $\rho'$ not bound in $R$).

We get

$$\Pi, \sigma'', R + \tilde{R}, \Delta''' \models v : \tau'_1 \quad (143)$$

where $\tau'_1$ occurs in $\Pi(A, \rho_1, \ldots, \rho_k)$ where $\tau_1$ occurs in $\Pi(A, \rho'_1, \ldots, \rho'_k)$. from (139) by lemma 12 for the values $\rho_1 \ldots \rho_k$ that are not in $\mathrm{frv}(\tau_1)$ and have $r_i \in \mathrm{Rng}(R)$, and by lemma 11 for the others.

We get

$$\Pi, \sigma'', R + \tilde{R}, \Delta''' \models v_0 : \tau_0 \quad (144)$$

by lemma 12, since we insured by our renaming, that $\rho_1 \ldots \rho_k$ does not occur free in $\tau_0$.

From our induction hypothesis we have $\tilde{\Delta} \models \mathcal{P} : \Pi$, which together with (142) implies

$$\Pi, \tilde{\Delta}, \Pi(A, (\rho_1, \ldots, \rho_k)), \models m[\rho_{k+1}, \ldots, \rho_n](x)\ \{s\} : \forall \rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta'} \tau_2 \quad (145)$$

The seven assumptions from definition 5 are fulfilled by

(i) $\tilde{\Delta} \models \mathcal{P} : \Pi$ from the induction hypothesis

(ii) (142)

(iii) (143)

(iv) (144)

(v) $\Delta' \subseteq \{\rho_1, \ldots, \rho_n\}$, and this is in the domain of $R + \tilde{R}$.

(vi) $\Delta' \subseteq \Delta'''$ by (121) and definition of $\Delta'''$

(vii) from the dynamic semantic.

Now we can use induction to get

$$\Pi, \sigma''', R + \tilde{R}, \Delta''' \models v' : \tau_2' \tag{146}$$

$$\Pi, \sigma''', R + \tilde{R}, \Delta''' \models v_0 : \tau_0 \tag{147}$$

where $\tau_2'$ is defined similarly to $\tau_1'$.

Because of (146) we know that (DynExp Method Err7) cannot be applied.

Using lemmas 11 and 12 in the other direction as we did to get (143) and (144) we get

$$\Pi, \sigma''', R + \{\rho_{k+1} \mapsto R(\rho_1'), \ldots, \rho_n \mapsto R(\rho_n')\}, \Delta'' \models v' : \tau_2 \tag{148}$$

$$\Pi, \sigma''', R + \{\rho_{k+1} \mapsto R(\rho_1'), \ldots, \rho_n \mapsto R(\rho_n')\}, \Delta'' \models v_0 : \tau_0 \tag{149}$$

Using lemmas 11 and 12 in the other direction as we did to get (138) and (139) we get

$$\Pi, \sigma''', R, \Delta \models v' : S(\tau_2) \tag{150}$$

$$\Pi, \sigma''', R, \Delta \models v_0 : \tau_0 \tag{151}$$

At this point, we can undo the renaming we did to avoid name-capture in $R$,$\Delta$,$\tau_0$ and $\mathrm{Rng}(S)$.

By letting $v_0$ assume the values of all the values in $V''$ (allowable because of (131)), we get

$$\Pi, \sigma''', R, \Delta \models V'' : E \tag{152}$$

(150),(151), and (152) proves the case.

### 3.3.5 Statements

For statements we assume the judgement of the form

$$\Pi, \Delta, E \vdash s : \tau$$

and we aim at proving the semantic relation

$$\Pi, \Delta, E, \Delta \models s : \tau$$

as defined in Definition 6. That is, we assume the following:

$$\tilde{\Delta} \models \mathcal{P} : \Pi \tag{S1}$$

$$\Pi, \sigma, R, \Delta \models V : E \tag{S2}$$

$$\Delta = \mathrm{Dom}(R) \tag{S3}$$

$$\sigma, V, R \vdash s \to v, \sigma', V' \tag{S4}$$

$$\mathrm{frv}(E) \subseteq \Delta \tag{S5}$$

$$\Pi, \sigma, R, \Delta \models v_0 : \tau_0 \tag{S6}$$

Then we want to prove that the following hold:

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{S7}$$
$$\Pi, \sigma', R, \Delta \models V', E \tag{S8}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{S9}$$

There are several rules to check; one for each syntactic form of the statement $s$. We make a case for each rule. The comments about the proof cases for expressions apply also to the cases for statements. We argue for (S5) only in (RegStm Decl) since that is the only place where $E$ changes.

**(RegStm Exp)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash e : \tau \tag{153}$$

and the premise is

$$\Pi, \Delta, E \vdash e : \tau \tag{154}$$

Note, the conclusion and premise are not actually identical, they just look identical! In the conclusion $e$ is in fact a statement, whereas in the premise $e$ is an expression. So the premise is the conclusion of one of the expression rules.

There is one explicit rule in the dynamic semantics that matches the expression, and there is one implicit error rule.

(DynStm Exp)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash e \to \mathsf{noval}, \sigma', V'}$$

(DynStm Exp Err)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}$$

Consider the evaluation $\sigma, V, R \vdash e \to v, \sigma', V'$. By induction on (154) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{155}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{156}$$

From (155) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Exp Err) cannot be applied. Consequently, only (DynStm Exp) can be applied.

From (155) and (156) we have proven the case.

**(RegStm Sequence)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash s_1; s_2 : \tau \tag{157}$$

and the premises are

$$\Pi, \Delta, E \vdash s_1 : \tau \tag{158}$$
$$\Pi, \Delta, E \vdash s_2 : \tau \tag{159}$$

There are two explicit rules in the dynamic semantics that match the expression, and there are two implicit error rules.

(DynStm Sequence Return)
$$\frac{\sigma, V, R \vdash s_1 \to v, \sigma', V' \qquad v \neq \mathsf{noval}}{\sigma, V, R \vdash s_1; s_2 \to v, \sigma', V'}$$

(DynStm Sequence Cont)
$$\frac{\sigma, V, R \vdash s_1 \to \mathsf{noval}, \sigma', V' \quad \sigma', V', R \vdash s_2 \to v', \sigma'', V''}{\sigma, V, R \vdash s_1; s_2 \to v', \sigma'', V''}$$

(DynStm Sequence Err1)
$$\frac{\sigma, V, R \vdash s_1 \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash s_1; s_2 \to \mathsf{error}, \sigma', V'}$$

(DynStm Sequence Err2)
$$\frac{\sigma, V, R \vdash s_1 \to v, \sigma', V' \qquad v \neq \mathsf{noval} \quad \sigma', V', R \vdash s_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash s_1; s_2 \to \mathsf{error}, \sigma'', V''}$$

Consider the evaluation $\sigma, V, R \vdash s_1 \to v, \sigma', V'$. By induction on (158) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{160}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{161}$$

From (160) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Sequence Err1) cannot be applied.

Next, consider the evaluation $\sigma', V', R \vdash s_2 \to v', \sigma'', V''$. Because of (161), by induction on (159) we get

$$\Pi, \sigma'', R, \Delta \models v' : \tau \tag{162}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{163}$$

From (162) we then see that $v' \neq \mathsf{error}$. Therefore, (DynStm Sequence Err2) cannot be applied.

Consequently, only (DynStm Sequence Return) or (DynStm Sequence Cont) can be applied. If $v \neq \mathsf{noval}$, the former is applied and the case follows from (160) and (161). Otherwise, if $v = \mathsf{noval}$, the latter is applied and the case follows from (162) and (163).

**(RegStm Letregion)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s : \tau \tag{164}$$

and the premises are

$$\Pi, \Delta \cup \{\rho\}, E \vdash s : \tau \tag{165}$$
$$\rho \notin \Delta \tag{166}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there is one implicit error rule.

(DynStm Letregion)
$$\frac{r \notin \mathrm{Dom}(\sigma) \qquad \sigma + \{r \mapsto \emptyset\}, V, R + \{\rho \mapsto r\} \vdash s \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s \to v, \sigma' \setminus\setminus \{r\}, V'}$$

(DynStm Letregion Err)
$$\frac{r \notin \mathrm{Dom}(\sigma) \qquad \sigma + \{r \mapsto \emptyset\}, V, R + \{\rho \mapsto r\} \vdash s \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s \to \mathsf{error}, \sigma' \setminus\setminus \{r\}, V'}$$

Assume $\Pi, \sigma, R, \Delta \models v_0 : \tau_0$.

Consider the evaluation $\sigma + \{r \mapsto \emptyset\}, V, R + \{\rho \mapsto r\} \vdash s \to v, \sigma', V'$. Because of (166) and prerequisite E3, we easily see that

$$\Delta \cup \{\rho\} \subseteq \mathrm{Dom}(R + \{\rho \mapsto r\}) \tag{167}$$

Since we have (166) and (S5), from Lemma 9 we then have

$$\Pi, \sigma + \{r \mapsto \emptyset\}, R + \{\rho \mapsto r\}, \Delta \cup \{\rho\} \models V : E \tag{168}$$
$$\Pi, \sigma + \{r \mapsto \emptyset\}, R + \{\rho \mapsto r\}, \Delta \cup \{\rho\} \models v_0 : \tau_0 \tag{169}$$

Using this, by induction on (165) we get

$$\Pi, \sigma', R + \{\rho \mapsto r\}, \Delta \cup \{\rho\} \models v : \tau \tag{170}$$
$$\Pi, \sigma', R + \{\rho \mapsto r\}, \Delta \cup \{\rho\} \models V' : E \tag{171}$$
$$\Pi, \sigma', R + \{\rho \mapsto r\}, \Delta \cup \{\rho\} \models v_0 : \tau_0 \tag{172}$$

From (170) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Letregion Err) cannot be applied.

Using Lemma 6 and lemma 7 to reduce $\Delta$ and $R$ with $\rho$ we can now conclude the proof of the case:

$$\Pi, \sigma', R, \Delta \models v, \tau \tag{173}$$
$$\Pi, \sigma', R, \Delta \models V', E \tag{174}$$
$$\Pi, \sigma', R, \Delta \models v_0 : \tau_0 \tag{175}$$

**(RegStm Decl)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{let}\ x : \tau = e\ \mathsf{in}\ s : \tau' \tag{176}$$

and the premises are

$$\Pi, \Delta, E \vdash e : \tau \tag{177}$$
$$\Pi, \Delta, E + \{x \mapsto \tau\} \vdash s : \tau' \tag{178}$$
$$\mathrm{frv}(\tau) \subseteq \Delta \tag{179}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there are two implicit error rules.

(DynStm Decl)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V' \qquad \sigma', V' + \{x \mapsto v\}, R \vdash s \to v', \sigma'', V''}{\sigma, V, R \vdash \mathsf{let}\ x = e\ \mathsf{in}\ s \to v', \sigma'', V'' \setminus\setminus \{x\}}$$

(DynStm Decl Err1)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash \mathsf{let}\ x = e\ \mathsf{in}\ s \to \mathsf{error}, \sigma', V'}$$

(DynStm Decl Err2)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V' \qquad \sigma', V' + \{x \mapsto v\}, R \vdash s \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash \mathsf{let}\ x = e\ \mathsf{in}\ s \to \mathsf{error}, \sigma'', V'' \setminus\setminus \{x\}}$$

Consider the evaluation $\sigma, V, R \vdash e \to v, \sigma', V'$. By induction on (177) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{180}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{181}$$

From (180) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Decl Err1) cannot be applied.

Combining (180) and (181) we also have

$$\Pi, \sigma', R, \Delta \models V' + \{x \mapsto v\} : E + \{x \mapsto \tau\} \tag{182}$$

From (179) and $\mathrm{frv}(E) \subseteq \Delta$ from the induction hypothesis we have

$$\mathrm{frv}(E + \{x \mapsto \tau\}) \subseteq \Delta \tag{183}$$

Using this we consider the evaluation $\sigma', V' + \{x \mapsto v\}, R \vdash s \to v', \sigma'', V''$ and get by induction on (178)

$$\Pi, \sigma'', R, \Delta \models v' : \tau' \tag{184}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E + \{x \mapsto \tau\} \tag{185}$$

From (184) we then see that $v' \neq \mathsf{error}$. Therefore, (DynStm Decl Err2) cannot be applied. Consequently, only (DynStm Decl) can be applied.

Since we can by definition reduce the set of values we reason about, we have

$$\Pi, \sigma'', R, \Delta \models V'' \setminus\setminus \{x\} : E \tag{186}$$

If there was an original binding of x, it, too, is consistent in $\sigma''$, since $v_0$ could be set to this value.

From (184) and (186) we have proven the case.

**(RegStm Return)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{return}\ e : \tau \tag{187}$$

and the premise is

$$\Pi, \Delta, E \vdash e : \tau \tag{188}$$

There is one explicit rule in the dynamic semantics that matches the expression, and there is one implicit error rule.

(DynStm Return)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{return}\ e \to v, \sigma', V'}$$

(DynStm Return Err)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash \mathsf{return}\ e \to \mathsf{error}, \sigma', V'}$$

Consider the evaluation $\sigma, V, R \vdash e \to v, \sigma', V'$. By induction on (188) we get

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{189}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{190}$$

From (189) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Return Err) cannot be applied. Consequently, only (DynStm Return) can be applied.

From (189) and (190) we have proven the case.

**(RegStm Cond)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \tau \tag{191}$$

and the premise is

$$\Pi, \Delta, E \vdash e : \mathsf{boolean} \tag{192}$$
$$\Pi, \Delta, E \vdash s_1 : \tau \tag{193}$$
$$\Pi, \Delta, E \vdash s_2 : \tau \tag{194}$$

There are two explicit rules in the dynamic semantics that match the expression, and there are three implicit error rules.

(DynStm Cond True)
$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \quad \sigma', V', R \vdash s_1 \to v, \sigma'', V''}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Cond False)
$$\frac{\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V' \quad \sigma', V', R \vdash s_2 \to v, \sigma'', V''}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Cond Err1)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to \mathsf{error}, \sigma', V'}$$

(DynStm Cond Err2)
$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \quad \sigma', V', R \vdash s_1 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to \mathsf{error}, \sigma'', V''}$$

(DynStm Cond Err3)
$$\frac{\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V' \quad \sigma', V', R \vdash s_2 \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to \mathsf{error}, \sigma'', V''}$$

Consider the evaluation $\sigma, V, R \vdash e \to b, \sigma', V'$. By induction on (192) we get

$$\Pi, \sigma', R, \Delta \models b : \mathsf{boolean} \tag{195}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{196}$$

From (195) we then see that $b \neq \mathsf{error}$. Therefore, (DynStm Cond Err1) cannot be applied.

Next, if $b = \mathsf{true}$ we consider the evaluation $\sigma', V', R \vdash s_1 \to v, \sigma'', V''$. Because of (195) we can use induction on (193) to obtain

$$\Pi, \sigma'', R, \Delta \models v : \tau \tag{197}$$
$$\Pi, \sigma'', R, \Delta \models V'' : E \tag{198}$$

From (197) we then see that $v \neq \mathsf{error}$. Therefore, (DynStm Cond Err2) cannot be applied.

Likewise, if $b = \mathsf{false}$ we consider the evaluation $\sigma', V', R \vdash s_2 \to v, \sigma'', V''$ and use the same set of arguments to obtain by induction on (194) that (DynStm Cond Err3) cannot be applied.

Consequently, only (DynStm Cond True) or (DynStm Cond False) can be applied. In both situations (197) and (198) prove the case.

**(RegStm While)** The conclusion of this rule is

$$\Pi, \Delta, E \vdash \mathsf{while}\ e\ \mathsf{do}\ s : \tau \tag{199}$$

and the premise is

$$\Pi, \Delta, E \vdash e : \mathsf{boolean} \tag{200}$$
$$\Pi, \Delta, E \vdash s : \tau \tag{201}$$

There are two explicit rules in the dynamic semantics that match the expression, and there are two implicit error rules.

(DynStm While False)
$$\frac{\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V'}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{noval}, \sigma', V'}$$

(DynStm While True)
$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \qquad \sigma', V', R \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}$$

(DynStm While Err1)
$$\frac{\sigma, V, R \vdash e \to \mathsf{error}, \sigma', V'}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{error}, \sigma', V'}$$

(DynStm While Err2)
$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \qquad \sigma', V', R \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{error}, \sigma'', V''}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{error}, \sigma'', V''}$$

This case is special because the statement itself occurs in the premises of rules (DynStm While True) and (DynStm While Err2). Therefore, in order to prove that

$$\Pi, \Delta, E \models \mathsf{while}\ e\ \mathsf{do}\ s : \tau \tag{202}$$

we do an rule-based inductive proof on the derivation tree in the dynamic semantics leading to the judgement

$$\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{noval}, \sigma', V' \tag{203}$$

We call this proof on the dynamic semantics the *inner* inductive proof, and the proof on the static semantics the *outer* inductive proof.

First, let us state clearly what the induction hypotheses of the *outer* inductive proof are:

$$\Pi, \Delta, E \models e : \mathsf{boolean} \tag{204}$$
$$\Pi, \Delta, E \models s : \tau \tag{205}$$

Second, we consider the evaluation $\sigma, V, R \vdash e \to b, \sigma', V'$. Because of (204) we have

$$\Pi, \sigma', R, \Delta \models b : \mathsf{boolean} \tag{206}$$
$$\Pi, \sigma', R, \Delta \models V' : E \tag{207}$$

From (206) we see that $b \neq \mathsf{error}$. Therefore, rule (DynStm While Err1) cannot be applied.

Next, consider the evaluation

$$\sigma', V', R \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s \rightarrow v, \sigma'', V'' \tag{208}$$

Using (207) we get by *inner* induction on (208)

$$\Pi, \sigma', R, \Delta \models V'' : E \tag{209}$$

$$\Pi, \sigma', R, \Delta \models v : \tau \tag{210}$$

From (210) we see that $v \neq \mathsf{error}$. Therefore, rule (DynStm While Err2) cannot be applied.

Now, if rule (DynStm While False) is applied, the case is proven because we know that $\Pi, \sigma', R, \Delta \models \mathsf{noval} : \tau$. If rule (DynStm While True) is applied, the case is proven because of (209) and (210).

This concludes the proof of Theorem 1. □

## 3.4 Conclusion

The soundness property demonstrated in this proof implies that errors due to type or memory faults will not occur in RegJava if the program terminates. If a program does not terminate due to an endless loop, this is not our problem. There is another possibility of non-termination, though. A program may not terminate, because no usable rule exist in the dynamic semantics. We have tried to complete the dynamic semantics with error rules, but the possibility exists that we have overlooked an obscure situation somewhere. We believe that the dynamic semantics of RegJava is complete, but have not tried proving it.

There are parts of the proof, that we feel less than perfectly content with. The reasoning in the last part of section 3.3.3 relies on properties of RegJava that are obviously true, but unproven (no new dangling pointers to old regions, no local "re-awakening" of regions that are in $\sigma$ but not accessible through $\Delta$ and $R$). This is far from an ideal situation.

The proof is large and complex, and relies on co-induction and reasoning about substitutions. We would much prefer a smaller and simpler proof without co-induction and without substitutions. Henning Niss, DIKU, is currently working on a different proof scheme that looks very promising. Instead of using a co-inductive consistency relation, he uses a relation with *store typing*. This eliminates the need for co-induction.

# Chapter 4

# Implementing Regions

In this chapter we consider possible ways of implementing regions. We discuss how to fit the region-annotated language of chapter 2 with the flat memory space of the conventional von Neumann machine model.

We are mainly concerned with the problems of memory management. Thus we will not deal with general implementation issues of object-oriented languages that are not important to memory management.

We also state a number of hypotheses about our region-based model for Java, and we compare the model with garbage collection for a few small program examples.

## 4.1 Concrete memory model

### 4.1.1 Minimal requirements

In the abstract execution model of section 2.5, the store $\sigma$ is only used for storing objects. Local variables in methods are managed by the variable environment $V$. Similarly, region variables are managed by the region environment $R$. In addition to these environments, there is an implicit call stack for controlling method invocations and return values.

Normally, one would probably use a single runtime stack to implement both the control stack and the two environments $V$ and $R$. A traditional store of objects would probably be managed by a heap—presumably a garbage-collected heap.

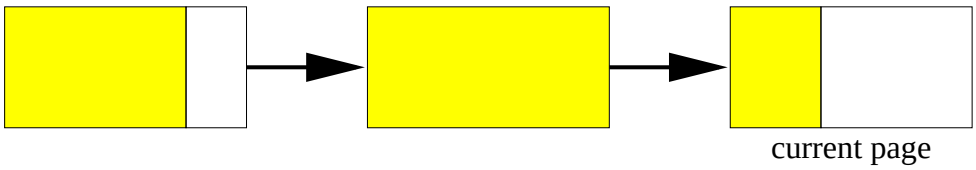
However, the store in our model is divided into dynamically created regions and should allow for regions to be explicitly created and destroyed at runtime in a stack-like manner—i.e. if region $r_1$ is created *before* $r_2$, then $r_1$ is destroyed *after* $r_2$. Furthermore, when new storage is claimed for an object, the allocation operation should explicitly request storage from a specific region.

The size of a region can grow at runtime and there are no restrictions on how they are laid out in the memory, e.g. they do not need to occupy contiguous memory. However, in order to make the execution predictable, all region operations should take constant time. At least, it should be possible to compute, at compile-time, an upper limit on the time of a region operation.

### 4.1.2 Lists of pages

Considering the minimal requirements in the previous section, our first attempt is to devise a model in which a region is implemented as a list of memory pages as shown in figure 4.1. The idea is to have one *active* or *current* page in each region from which new storage is allocated. When all the storage of the current page is used, a new page is fetched, inserted into the list, and made the new current page.

**Figure 4.1** Memory pages in a region. The individual pages may be only partially filled. Storage is allocated from the *current* page.




In our scheme, regions can store objects of completely different types and sizes. It would therefore seem wise to use pages with a size big enough to hold several of such objects. On the other hand, if pages are too big, regions containing only few objects would waste too much space.

If possible, pages should have a predetermined, fixed size. This ensures that fragmentation of the store is not a problem. Also, it means that pages can be treated equally, so when a page is needed the system does not have to spend time on choosing a page, any page will do.

If the compiler can infer an upper-limit on the size of objects of any type, determining the appropriate page size can be done per program before execution. However, it might not be possible to determine such an upper-limit. For instance, if arrays with dynamically determined bounds are a part of the language, then it would not be possible. We will discuss the size of pages further in section 4.2.

As depicted in figure 4.2 on the next page, the system keeps an infinite list of free pages. Every time a region needs to be extended, a page is removed from the free list and inserted into the regions list. When the region is destroyed, its page list is reinserted into the free list.

### 4.1.3 Abstract data structures

Using the right data structures and algorithms for lists, it is possible to ensure that creation, extension and destruction of regions can be done in constant time.

The data structures needed for this implementation can be expressed as quite simple abstract data structures (ADT). We present them formally here. Later, we will show how they can be implemented efficiently using C++.

First, a memory page can be regarded simply as an array of bytes from which storage is allocated. The page keeps a pointer to the next available storage. The Page ADT is shown in figure 4.3 on the following page.

Second, pages are put into lists of pages. It is important that each list operation take constant time. In particular, it should be possible to append two lists in constant time. This can be achieved using a doubly-linked list implementation. The page list is used for managing

**Figure 4.2** Regions consist of a list of memory pages allocated from a global free list of pages.

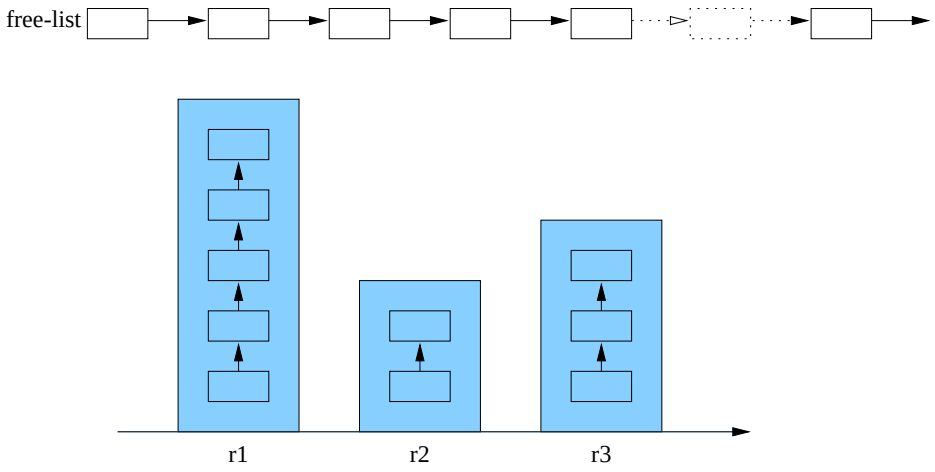



**Figure 4.3** The Page ADT

DATA

**data** : Byte[0..*SIZE*]
An array of bytes from which storage is allocated.

**top** : Integer
The next free index in **data**.

OPERATIONS

**create** : Page
Creates a new memory page with *SIZE* bytes and **top** initialized to zero.

**allocate**(*size* : Integer) : Pointer
Returns a pointer to *size* bytes of fresh storage from the page. This is done by returning a pointer to **data**[**top**] and then adding *size* to **top**.

each region and for managing the systems list of free pages. The PageList ADT is shown in figure 4.4.

**Figure 4.4** The PageList ADT

OPERATIONS

**create** : PageList
Creates a new empty list of memory pages.

**insert**(*page* : Page)
Inserts a new page in the list.

**pop** : *page* : Page
Deletes and returns a page from the list.

**append**(*pages* : PageList)
Appends another list of pages to this list.

Third, the system manages free pages. It keeps a list of such available pages. When a region needs a new page, the system pops one of its pages from the list. For now, we will not consider what happens if the list gets empty.

**Figure 4.5** The System ADT

DATA

**freeList** : PageList
A list containing free memory pages.

OPERATIONS

**fetch** : *page* : Page
Pops and return a fresh page from **freeList**.

**release**(*pages* : PageList)
Appends **pages** to **freeList** making them available for other regions to fetch.

Last, a region consists of a list of memory pages. One page is the current page. Storage is only allocated from the current page. When more storage is claimed than is available in the current frame, a new one is fetched from the system. This page is inserted into the list and made the new current page, and the storage is then allocated from the page. The Region ADT is shown in figure 4.6 on the following page.

### 4.1.4 A region framework in C++

We would like to do some testing and demonstration of our region model. Since we have not developed a compiler for RegJava, we need some other mechanism that enables us to run and test small programming examples using regions. Because we have to code the examples by hand, it should be a relatively simple framework. At the same time, it should ne adequately close to the Java language model, and it should use regions as the only mechanism for storing objects.

**Figure 4.6** The Region ADT

DATA

**pages** : PageList
A list containing the memory pages from which storage is allocated.

**current** : Page
Points to the currently active page.

OPERATIONS

**create** : Region
Creates an new empty region, i.e. **pages** is initialized to the empty list.

**delete**
Destroys the region freeing all occupied space, i.e. reinserting **pages** into the systems free list.

**allocate**(*size* : Integer) : Pointer
Returns a pointer to *size* bytes of fresh storage. The storage is obtained from the current page in **pages**. If there is not enough free storage in the current page, a new page is allocated and is made the new current page.

We believe that C++ will serve both purposes. First, the Java syntax and semantics come close to being af true subset of that of C++ [1]. Second, C++ allows the programmer to control where and when dynamic data is allocated. We can utilize this to introduce regions in an elegant manner. Last, we shall see that the simple data structures of section 4.1.3 can easily be implemented in C++.

The code of the framework is presented in appendix D. In the rest of this section we describe the framework and how to use it.

### The datatypes of the framework

All the ADT's in section 4.1.3 can be implemented elegantly in C++. For the purpose of extensibility we have chosen to make a completely abstract base class called `Region`. This class is then extended by the class `URegion` that implements the functionality of a region of unbounded size as described in figure 4.6. The two classes are shown in figure 4.7 on the following page.

We can now take advantage of the block structure of C++. Regions are defined just like ordinary local variables in the beginning of a block. When the program exits from this block, the destructor for the region is then implicitly called to free all pages occupied by that region. The following example shows the idea:

```
{ URegion rho1; // Here the region is created
  ...
  ... // Some code that may use the region rho1
  ...
} // Here the region gets deallocated
```

[1] This is not suprising since the designers of Java[AG96] have adopted much of the syntax and semantics directly from C++.

**Figure 4.7** The Region and URegion classes in C++. The globale variable `free_pages` is the system's list of free pages.

```
class Region {
public:
  Region() {}
  virtual ~Region() {}
  virtual void* alloc(size_t) = 0;
};

class URegion : public Region {
protected:
  PageList pages;
  size_t free;
public:
  URegion() {
    free = 0;
  }
  virtual ~URegion() {
    free_pages.free(pages);
  }
  virtual void* alloc(size_t s) {
    if (pages.empty() || (s > free)) {
      pages.add(free_pages.alloc());
      free = REGION_PAGE_SIZE;
    }
    size_t top = REGION_PAGE_SIZE - free;
    free -= s;
    return &(pages.top()->data[top]);
  }
};
```

### Overloading the operator new

Our use of C++ relies heavily on the versatile **new**-operator that can be elegantly customized in several different ways. As explained in [Str94, r.5.3.3], it can be overloaded in several ways. We have focused on the global `::operator new()` with the special *placement* syntax as shown in figure 4.8. The **new**-operator gets passed the size of memory needed and a reference to the

**Figure 4.8** Overloading the operator **new** in C++.

```
inline void* operator new(size_t size, Region& rho)
{
  return rho.alloc(size);
}
```

region from which the storage should be allocated. The body of the operator takes care of the actual memory allocation. Here it simply tells the region `rho` to allocate and return the specified amount of memory. The following code snippet illustrates how the operator is used in practice:

```
A a = new(rho1) A();
```

A new instance of class `A` is created in the region denoted by the variable `rho1`. First, the body of the operator specified in figure 4.8 is executed to obtain a chunk of storage from the specified region. Next, the appropriate constructor for class `A` is called in order to any initialization of fields.

### Translating Java into C++

We can justify the use of C++ since most of the features of Java that we are interested in, can be translated into C++ quite easily:

1. All objects in Java are dynamically allocated using the built-in operator **new**. Consequently, every Java object should be translated into a dynamic C++ object created using the overloaded operator **new**.

2. In Java, variables of class type are actually references to objects. This just means that every such variable should be translated into a pointer to the relevant class type. Furthermore, the variable must be dereferenced every time before using it. For instance, a field selection `a.x` is translated into `a->x`.

3. Java assigns a default initial value to fields of a class if one is not specified. This must be assured manually by the programmer in C++ by explicit assignments in every constructor for that class.

4. A Java compiler must demand that each local variable be *definitely assigned*. Again, this must be assured manually by the programmer in C++ by explicit assignments.

Having made these justifications, we will write our programs entirely in C++ from now on. We will take care not to use any features of C++ that cannot trivially be translated back into Java.[2]

[2]This remark applies only to the example programs. The basic region framework is, of course, allowed to use any feature of C++.

### Translating RegJava into C++

The previous section showed how to translate basic Java features into C++. This section shows how to translate the additional features of RegJava. For the discussion we focus on the RegJava program in figure 4.9.

**Figure 4.9** Example RegJava-program

```
class List[ρ1] at ρ1 extends Object[ρ1] {
  abstract int hd();
  abstract List[ρ1] tl();
  abstract boolean isEmpty();
  abstract List[ρ2] concat[ρ2](List[ρ2] list);
}

class Nil[ρ1] at ρ1 extends List[ρi] {
  boolean isEmpty() { return true; }
  List[ρ2] concat[ρ2] (List[ρ2] list) {
    return list;
  }
}

class Cons[ρ1] at ρ1 extends List {
  int head;
  List[ρ1] tail;

  Cons(int hd, List[ρ1] tl) {
    this.head = hd;
    this.tail = tl;
  }

  int hd() { return this.head; }
  List[ρ1] tl() { return this.tail; }
  boolean isEmpty() { return false; }
  List[ρ2] concat[ρ2] (List[ρ2] list) {
    let concat_tail : List[ρ2] = this.tail.concat[ρ2](list) in {
    return new[ρ2] Cons(this.head, concat_tail);
  }
}
```

For the sake of beauty, we have taken the liberty of using in the example some language features not directly included in RegJava; there are a few methods that are declared *abstract*, and there is also an explicit constructor. As mentioned in section 2.3, constructors are easily added to the language since they are just implicit methods. The abstract methods are also harmless since they could simply be implemented with a body returning some dummy value.

Before discussing the translation to C++, we rush to present the translated version of the example (see figure 4.10 on the next page). Except for a few trivial syntactical translations

**Figure 4.10** The example program in figure 4.9 translated to C++.

```
class List {
public:
  virtual int hd() = 0;
  virtual List* tl() = 0;
  virtual boolean isEmpty() = 0;
  virtual List* concat(Region& rho2, List* list) = 0;
}

class Nil : public List {
public:
  virtual boolean isEmpty() { return true; }
  virtual List* concat(Region& rho2, List* list) {
    return list;
  }
}

class Cons : public List {
public:
  int head;
  List* tail;

  Cons(int hd, List* tl) {
    this->head = hd;
    this->tail = tl;
  }
  virtual int hd() { return this->head; }
  virtual List* tl() { return this->tail; }
  virtual boolean isEmpty() { return false; }
  virtual List* concat(Region& rho2, List* list) {
    List* concat_tail = this->tail->concat(rho2, list);
    return new(rho2) Cons(this->head, concat_tail);
  }
}
```

(e.g. the keyword `extends` is translated to a colon), we make the following important comments about the translation:

1. Each class type is translated to a pointer type.
2. Region annotations is removed from types.
3. Each method is made explicitly virtual.
4. The formal regions of methods are made explicit arguments of type `Region&`.
5. The formal regions of the classes are removed completely since they are only used when creating instances. If they were needed explicitly by a method—e.g. in order to create an object in that region—they would have appeared as a field of type `Region&` and would have been passed to and initialized by the constructor.

The example in figure 4.9 does not show the important letregion-statement. The following code snippet shows how to create an object in a newly declared region:

```
letregion rho1 in {
   return new[rho1] A();
}
```

This code translates into the following C++ program:

```
{
  URegion rho1;
  return new(rho1) A();
}
```

**Using the framework for stand-alone memory management**

The region framework in C++ described above is designed with the sole purpose of testing RegJava programs in the absence of a compiler for RegJava. Nevertheless, we believe that it has some very interesting properties in its own right.

Programmers in C++ are sometimes faced with task of having to implement a program solving an extremely time critical problem that needs to run in narrow space. Often the programmer is scared away from using the built-in heap management because of possible time and memory overhead. Instead he invents his own memory management.

This kind of ad-hoc techniques are bound to fail from time to time and give rise to bugs. Because our framework is both general, simple and efficient, we believe that it could be used as a substitute for such ad-hoc techniques. With the current implementation of the framework it does use the built-in heap, but only for allocating fairly large pieces of memory. It should also be possible to entirely avoid using the built-in heap.

## 4.2 Hypotheses and assumptions about the model

In this section we discuss what behaviour to expect from a region-based memory-model. We examine possible advantages and disadvantages of the model. When discussing this, we normally compare our model with that of garbage collection, but in this section we also investigate how our model differs from the way regions are implemented in the ML-Kit.

### 4.2.1 Overview

Our hypotheses about the behaviour of the region-based model are motivated by the very reasons for choosing the model:

**Smoothness** Because of less overhead in memory management and because deallocation is directed by the programs static structure, we expect that RegJava-programs run in a more smooth manner than with garbage collection.

**Efficiency** Because of less overhead and also because of better reclamation of storage, we expect that RegJava-programs use both time and memory resources more sparingly than equivalent programs with garbage collection.

**Predictability** Since every memory management operation is explicit in the program, one has much better possibilities of predicting the execution behaviour of a RegJava-program than equivalent programs with garbage collection.

**Success in ML** As shown by Mads Tofte and others[TBE+97], the region-based model seems to work pretty well in ML. Although Java is quite different from ML, we hope to benefit from the successful results of using regions in the ML-Kit.

In order to make these hypotheses more concrete and precise we divide the following discussion into three parts. First, we discuss the execution speed of RegJava-programs. Next, we turn to a discussion about the memory usage. Last, we examine the expected differences from and similarities with the ML-Kit.

It is important to notice that the discussion about speed and memory are closely related. This is because, as a rule of thumb, one can say that an increase in the amount of memory used implies a similar increase in the execution speed.

Based on these considerations, we now try to formulate and refine some hypothesis about the behaviour of Java with regions.

### 4.2.2 Speed

Because of the simplicity of the model, we expect less execution overhead than for a model using garbage collection. Moreover, one can reason about the overhead in execution speed, and therefore region-based programs run in a more *predictable* manner.

#### RegJava has speed that can be reasoned about

This is a important claim for real-time programming as well as for predicting running times of algorithms. It follows directly from the claim that any region operation take constant time (see section 4.1). In practice, however, the validity of this claim is based on the assumption that the surrounding environment, in which a RegJava-program executes, is also predictable.

For instance, if the program executes under the control of a multi-tasked operating system, the mechanism for allocating memory from the system may not have a constant worst-time behaviour. This is because the operating system may have to do some degree of bookkeeping.

### RegJava ensures "smooth" execution

This is an important property for real-time applications where even small delays in the execution can be fatal. It is also of importance for programs with an interactive user interface; it can be extremely annoying if the user often has to wait for responses from the program.

The definition of *smoothness* can therefore be both objective and subjective depending on the nature of the application. Wilson characterises the requirement as follows[Wil94]:

> ... a more realistic requirement for real time performance is that the application always be able to use the CPU for a given fraction of the time *at a time-scale relevant to the application.* (Naturally, the relevant fraction will depend on both the application and the speed of the processor.)

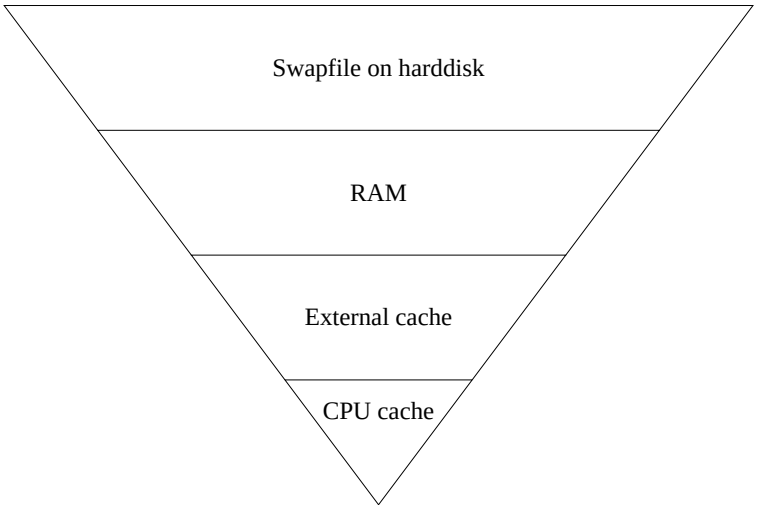
The property is strongly related to the predictability property mentioned above. If the programmer can predict the execution time of every single operation in the program, then he should also be able to make the program run smoothly.

### RegJava has good locality of reference

Because physical memory (RAM) is expensive, most modern computers use a *virtual* memory scheme that enables applications to address and use much more memory than is physically available in the machine. Furthermore, since fast memory is even more expensive, a caching scheme is also used, so that only frequently accessed data lie in fast memory.

Both schemes are made possible by a multi-layered memory layout as depicted on figure 4.11. The memory is organized in a series of smaller and smaller layers of faster and faster

**Figure 4.11** Multi-layered memory




storage using different media. On the top is the largest and slowest storage. For this layer a swap-file on the hard disk is often used. Next is the ordinary physical memory, and then one or more levels of cache using specialised, extra fast memory chips. The fastest cache is often inside the CPU itself.

Due to this multi-layered caching schemes, the way programs utilise the memory can be of huge importance. Significant speed gains can be attained if the program can often access a faster memory layer at the bottom, rather than a slower memory layer at the top. Programs have good *locality of reference* if references to memory that are close in time also tend to be close in memory.

With garbage collection there is no way to reason about where on the heap a particular object exists, so a method accessing only a few data structures may need data from all over the heap. This would imply less than good locality of reference. Some garbage collection schemes, notably copying garbage collectors, have other methods to insure good locality of reference, though[Wil94].

However, using regions it is often possible to reason about when two references are close. If they both reference memory in the same region, they are more likely to be close than if they reference memory in different regions. Just how much more likely depends on the size of the region compared to the entire size of the memory in use by that program.

We expect that region annotations often make it possible to reason about how good the potential locality of reference is for a particular RegJava-program. For instance, if parts of the program, due to the block structure, only access few regions, one can expect good locality of reference for this part of the program. This analysis is particularly relevant for methods since they can only access regions mentioned in the formal region parameter list and some of the formal regions of the class together with any locally defined regions.

### 4.2.3 Memory Usage

We hope that a region-based program will usually use less memory than a garbage-collected program. Again, because of the simplicity of the model, we believe that the memory is use in a "better" way than with garbage collection. This should also imply that the memory usage of programs can be reasoned about.

#### RegJava uses less memory

This claim is highly dependent on the individual program. Some programs are well-suited for regions and result in a high degree of early reclamation of dead data as compared with garbage collection. Unfortunately, some programs keep dead data far longer than with garbage collection—and even some never reclaims data at all.

However, because deallocation take constant time regardless of the amount of storage, suspending deallocation for a while is not necessarily bad. For example, consider a program in which one region grows for a long time before being deallocated. If the program accesses data only from the "end" of a region, it does not matter much that the region keeps growing. The old, dead data from the "beginning" of the region will eventually be swapped out of physical memory and will never need to be swapped in—not even when the entire region is finally deallocated.

The possible advantage in the example is that as long as there is no deallocation, there is also no overhead in memory management. A garbage-collected implementation might start the collector several times during the computation. This saves overall memory but also slows down the execution of the program.

If a region grows too big—e.g. if it never gets deallocated—we say that the program has a *memory leak*. This should be avoided whenever possible. Sometimes this can be achieved

by a manual rewriting of the program. When this is not enough, we have to say that garbage collection is superior to the region-based model.

### RegJava has predictable memory usage

Since both region allocation and deallocation are explicit, the memory behaviour of a region-based program is predictable. This is in sharp contrast to the case with most garbage collection techniques.

As with the predictable behaviour in terms of speed, this is an important property for real-time programming. For instance, it enables the programmer to guarantee that the program will run within a confined space. It is also an advantage in situations where it is important that programs not be "memory hogs".

### RegJava handles container classes poorly

This is one of the worst examples of memory leaks mentioned above. Instances of container classes must demand that all its element essentially be in the same region. Every object that is ever put into a container will live at least as long as the container object itself. This means that long-lived structures such as stacks, queues and dictionaries will be unable to reuse the memory of dead elements—even after they are no longer in the container. The containers regions just keeps growing with every element added.

With the current model there is no way around this sad fact. The only advice to programmers is not to use long-lived container classes. For some programs it may be possible to rewrite in a manner such that container classes are either made short-lived or are eliminated entirely. It is obvious, however, that there are situations where such rewriting is not practically feasible.

### RegJava handles "trapped" dangling pointers well

Consider the following class definitions in RegJava:

```
class A[ρ1] at ρ1 extends Object[ρ1] {
  int x;
}

class B[ρ1, ρ2] at ρ1 extends A[ρ1] {
  A[ρ2] y;
}
```

and the following code snippet using these two classes:

```
letregion ρ3 in {
  let a : A[ρ3] = null in {
    letregion ρ4 in {
      let b : B[ρ3, ρ4] = new[ρ3, ρ4] B() in {
        b.y = new[ρ4] A();
        a = b;
      }
    } // destroy ρ4
```

```
    a.x = 42; // a is alive and has a harmless dangling pointer into ρ4
  }
} // destroy ρ3
```

In this examples the object referenced by the field $y$ in the object referenced by $a$ is dead after exiting from the letregion that defines $\rho_4$. This is legal as long as the program does not try to access this field (and this cannot be done without an explicit typecast). To our knowledge, no garbage collector can reclaim memory that is actually pointed to by live data structures. Therefore, the memory for that object will be reclaimed no sooner than at the end of the entire block in which $a$ is declared.

### 4.2.4 Regions for Java as compared to ML

ML and Java are very different languages, so it may be hard to generalise results from using regions with ML to Java. ML is a functional language that uses high-order function, tuples and lists. Moreover, ML uses a call-by-value strategy for passing arguments to functions. This result in extensive creation of small temporary, i.e. short-lived, values on the heap.

Java is an imperative, object-oriented language that relies heavily on its run-time stack. Java does not support high-order functions, and it has no built-in tuple or list datatypes. Java has two strategies for passing of arguments to methods. For simple values it uses call-by-value, and for objects it uses call-by-reference. Since the stack is used for passing arguments, there is no implicitly generated temporaries on the heap.

These differences has made our region model for Java differ in several respects from the one used in the ML-Kit. We do not use regions to store simple values, we put them on an ordinary run-time stack. Only objects are stored in regions.

#### Regions tend to live longer in RegJava

An important difference between the two languages is that Java-programs uses destructive updates much more than ML-programs. We assume that the object contained in regions will be destructively updated all the time. This may mean that regions will survive longer, than they would in corresponding situations in the ML-Kit.

A "natural" concatenation in ML would typically create a new list—potentially in a new region—and leave the originals untouched—potentially to be deallocated. A "natural" concatenation in Java would just modify the one list to point to the other.

#### Fewer regions are used in RegJava

Again, this is an effect of the lack of implicit copying operations in imperative languages. It is also caused by our design choice of only putting objects in regions, not simple variables. This corresponds somewhat to the choice made in the ML-Kit to put simple variable in a common "special" region (r2).

#### Most regions in RegJava have more than one element

This is a wild guess. The effects mentioned above may mean, that regions generally contain several elements. In the ML-Kit most regions contain only one element.

To summarise the above, we can shortly state that compared with the ML-Kit we expect the the regions in RegJava are fewer, larger, and live longer.

## 4.3 Measurements

In this section we examine some of the hypotheses from last section, using our region-based C++ framework and a garbage collection package for C++ [AFI97].

Here are the hypotheses again

1. RegJava has speed that can be reasoned about.
2. RegJava ensures smooth operation.
3. RegJava has good locality of reference.
4. RegJava use less memory than garbage collection.
5. RegJava has predictable memory usage.
6. RegJava handles container classes poorly.
7. RegJava handles "trapped" dangling pointers well.
8. Regions tend to live longer in RegJava, than in ML.
9. Fewer regions are used in RegJava than in ML.
10. Most regions in RegJava have more than one element.

The claims 1 and 5 follow directly from the definition of the concept of regions. Since regions are explicitly created, expanded and deallocated, it is possible to calculate at every point during a program execution exactly the size of the memory that is in use. With garbage collection this is not possible, since garbage collection does not have such deterministic behaviour. Since creation, expansion and deallocation of regions are all done in constant time, it should also be possible to reason about execution speeds in a region framework, and in theory it is.

When running programs in the C++ region framework on normal (not real-time) operating systems, that theory does not quite apply. As will be seen in the "life" example, programs that use a lot of memory take longer to run than programs that use less memory, even though these programs have the overhead of memory management. This has to do with the fact that the operating system itself spends time on virtual memory management. The extra time is spent swapping pages in and out from the disk-cache, allocating more and more pages to the process and doing extra internal cache operations.

On a system with no such virtual memory management, processes from the operating system, or on a real-time system, the assumption that RegJava programs have predictable speed behaviour should hold by definition. Wilson makes similar claims for some garbage collectors in [Wil94], given knowledge of the garbage amount generated by the program and the system capabilities, but the property is much more complex than the one offered by RegJava.

Claims 3, 4, 8, 9 and 10 are, in a sense, claims about "typical" programs. It is possible to construct simple examples, where each claim is true, and other simple examples where each

claim is false. If RegJava is used in an ML-like way, using only initial value assignments, and letting recursion handle most iterations, and ML is used in a Java-like way, updating references all over, and using only tail-recursion, then claims 8 and 9 are not likely to hold. If one is determined to construct a RegJava program with a lot of regions with only one element in each, is is very possible (in fact, one of our examples have this property). It is possible to construct RegJava programs with bad locality of reference, and RegJava programs that use a lot more memory than the equivalent garbage collected programs. The easiest way is to annotate badly, but even with good annotation strategies, some programs are not well suited to the region based approach. We feel, that our examples are "reasonable" in the sense that they illustrate different types of programs without too many absurdities. They are, however, very small programs, and they are written by hand. Machine generated code or very large systems may have other properties, than those that can be seen from our examples. We do not aim to *prove* the claims 3, 4, 8, 9 and 10 in this paper, just to illustrate whether they are justifiable points of view or not. (Actually our results indicate that claim 10 is not as reasonable as we initially thought)

Claim 3 is extremely hard to quantify in a reasonable manner. We have considered logging every single memory access, and doing statistics on the addresses used, but such measurements would interfere with the memory usage and speed properties of the program to an extreme degree, and so would effectively not say very much. Instead, we have settled on measuring the sizes of the accessible regions at our profiling points. This strategy has two major drawbacks. The accessible regions change at every context switch (method calls, letregions, returns), so if the program only performs few instructions between such shifts, the fact that each context only can access a small part of the live regions, may be misleading. Also, the live regions only state which regions *can* be accessed, not which regions actually *are* accessed in a given context, so the actual locality of reference may be better than indicated by our measurement for a program that pass a lot of region parameters to its methods, but only use a few of them.

We try to illustrate claim 4 by showing that the C++ region framework aggressively reuses memory. The CMM garbage collector starts by allocating 1 Mb of memory, and then allocates more memory if and when it becomes necessary. Since many of our examples does not use several megabytes of memory, the comparison to CMM is not really a fair one. There is no easy way of determining how much of the allocated memory, that is actually used by CMM.

We give examples that illustrate claims 6 and 7, but do no actual measurements, since it is possible to create arbitrarily bad scenarios in both cases.

We do not try to further justify claims 8 and 9 by rewriting our examples in the ML-kit. We feel that the fact that simply typed variables are not stored in regions in RegJava, would make the results of such rewritings hard to interpret in a reasonable way.

We have tried to rewrite our examples to use "bounded regions", that is, regions that have only one element of predictable size, and so can be stored directly on the stack, rather than in pages from the heap. This was only possible in two of our examples, "Ackermann" and "Mandelbrot", but in those cases it did make a considerable difference.

CMM is a garbage collection package for C++ presented in [AFI97], where they suggest that its performance is close to the best achievable with conservative collectors on stock hardware and without compiler support. CMM has several modes. We use its mark-and-sweep conservative garbage collecting mode and its generational garbage collector for time comparisons with our C++ region framework.

### 4.3.1 Memory Usage Measurements

We try to illustrate that RegJava only has a few accessible regions at a time, and that their total size may be much smaller than the total size of memory used. We measure total memory usage compared to accessible memory through the execution of example programs. The source code of the example programs are given in Appendix E. The timings given in the graphs are real-time running times of the profiled program, but since the time spent writing the profiling information is included, these timings do not match the timings of the unprofiled programs.

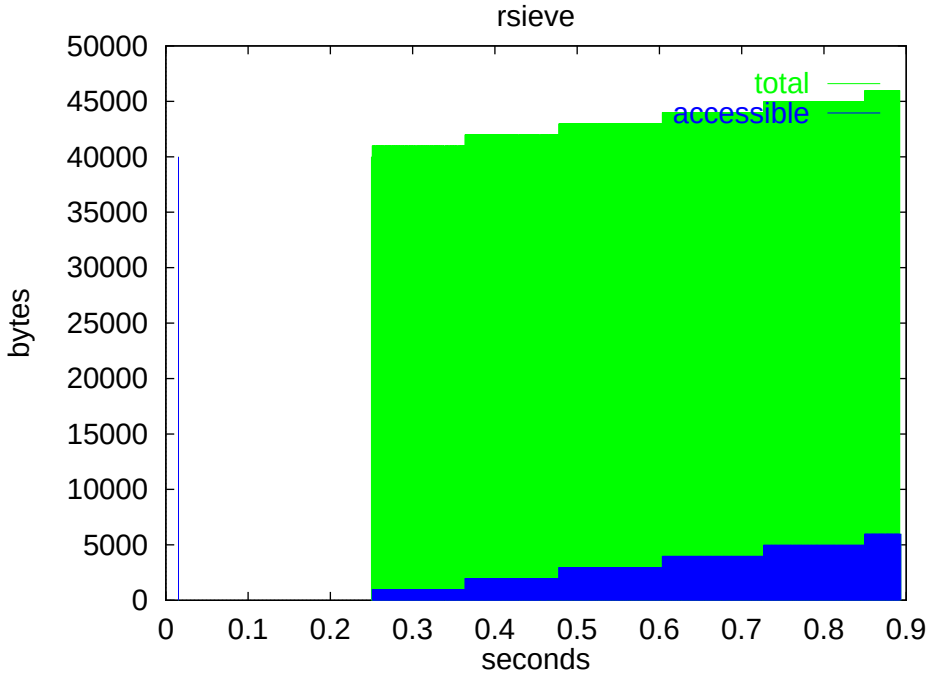
The sizes of the **total** memory given here, is the block size times the number of pages in the program's live regions. The **accessible** memory is the size of the pages, in the regions that are inside the scope, of the reference point. It is entirely possible, that many of the objects in the regions are actually not accessible, but that is beside the point.

#### Sieve of Eratosthenes

The Sieve of Eratosthenes is a method of generating prime numbers by removing all integers from a list, that has an already found prime as a factor. We implement the list of integers using objects.

Profiling an implementation with regions of the Sieve of Eratosthenes (a common benchmark), we get a pattern of memory usage like in figure 4.12.

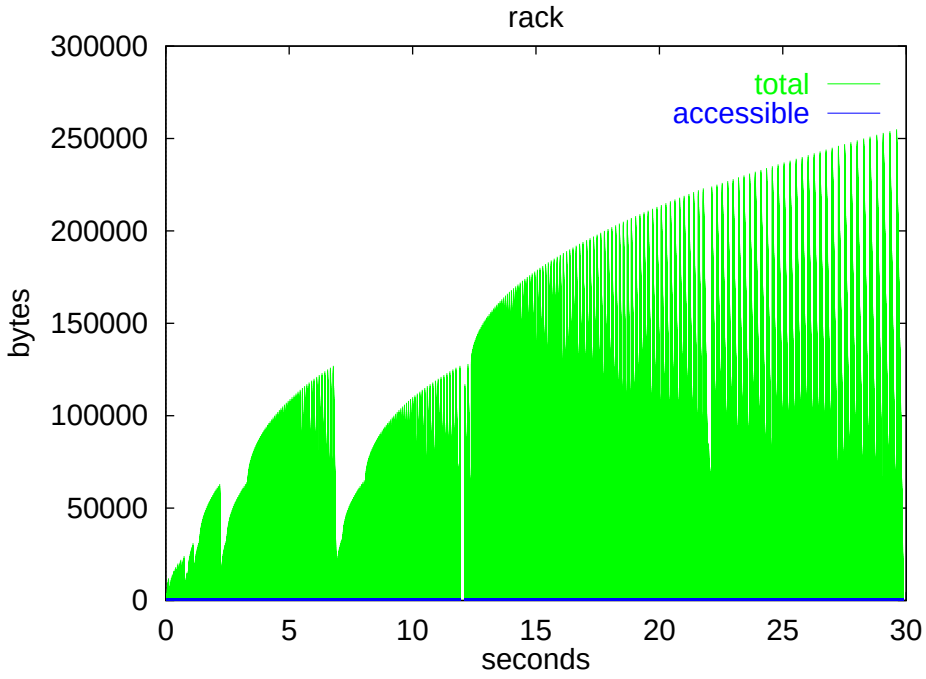
**Figure 4.12** Sieve of Eratosthenes




It is not too interesting. The empty part in the beginning is the buildup of the list, which we have not profiled. The **accessible** graph illustrates, that a typical method invocation only used a few of the total live regions. However, since the frame of reference shifts continuously in this program, this does not really say much about locality of reference.

### Ackermann

**Figure 4.13** Ackermann




Profiling a naive implementation of Ackermanns function gives the graph in figure 4.13. The **Accessible** plot is so small that it cannot be distinguished from the bottom line. Again, this does not say a lot about locality of reference, since the contexts are continually shifting.

In other ways, this graph is illustrative. Memory is aggressively reused, as seen by the many spikes in the graph, and on a much higher scale than a garbage collector would be likely to do. Note that some of the "subgraphs" are identical. This makes sense, since the implementation solves identical problems several times. We feel that this example is well suited to the region based approach, since a lot of temporary data structures are built, and then immediately destroyed.

### Merge Sort

Profiling a version of Merge-sort gives the graph in figure 4.14.

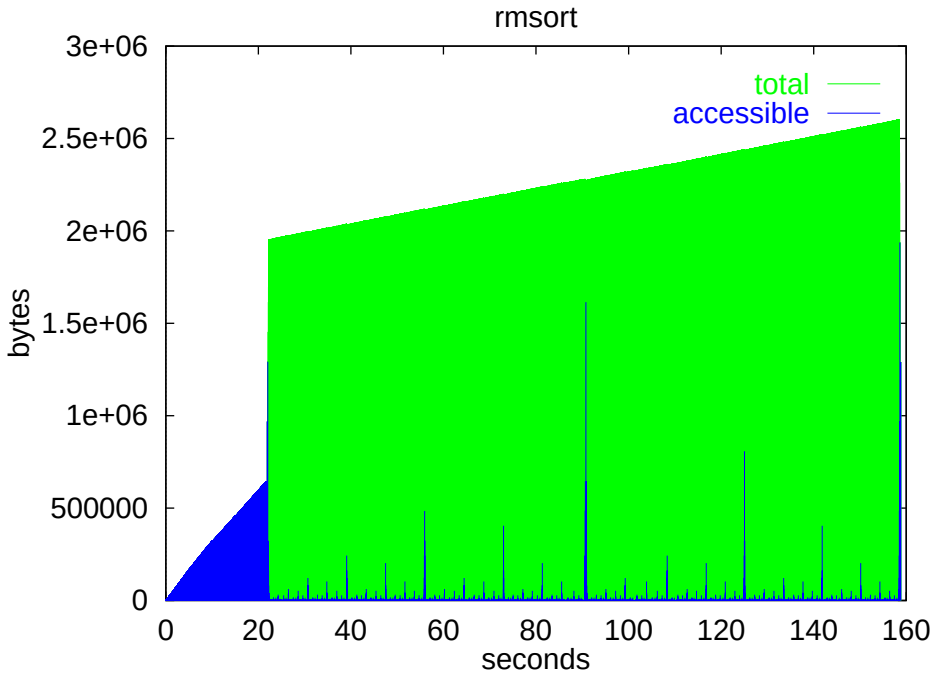
In this case, the low ratio of **accessible** to **total** as well as the low absolute values of **accessible** during most of the execution actually *is* indicative of good locality of reference, since the program keeps the same scope for longer periods of time.

The first 20 seconds is spent building the data that are to be sorted.

The timing information is absolute, and so includes the time spent writing profiling information. In this case, most of the time is spent writing profiling information, so the absolute running times of the unprofiled program is not comparable to the running time of the profiled program.

Deallocation of regions is not visible in the graph, though some *does* happen during the execution. The deallocated regions are too small, compared to total memory usage, to be

**Figure 4.14** Merge Sort




visible.

### Mandelbrot

A profiled version version of a Mandelbrot program gives the graph in figure 4.15. Complex numbers are stored in objects.

This is not a very helpful graph. It does seem that memory is reused intensely and that the accessible regions pretty much match the live regions, but too much is going on to really give a certain interpretation of the graph.

A detail study of a small part is slightly more informative, and is given in figure 4.16. In each point, a data-structure of up to 30 kb is built up during the iterations, and then torn down again, when it has been decided whether the point is in the Mandelbrot set or not. This is pretty much what should be expected of the program. No surprises.

### Life

Finally, Conway's game of Life has been profiled.

Conways life is a model of cellular life, moving from generation to generation according to strict rules (a cellular automaton). The cells lives in a grid, where each cell has 8 neighbours. A dead cell will spring to life if it has exactly three neighbours. A living cell will stay alive, if it has two or three living neighbours, otherwise it will die, either of loneliness or overcrowding.

In a naive implementation with only one region, we have a serious memory leak, as seen in figure 4.17.

The computer accessed the harddisc several times during the execution of the program, presumably to write to the swap file. The generations were printed to a fast device (/dev/null),

**Figure 4.15** Mandelbrot

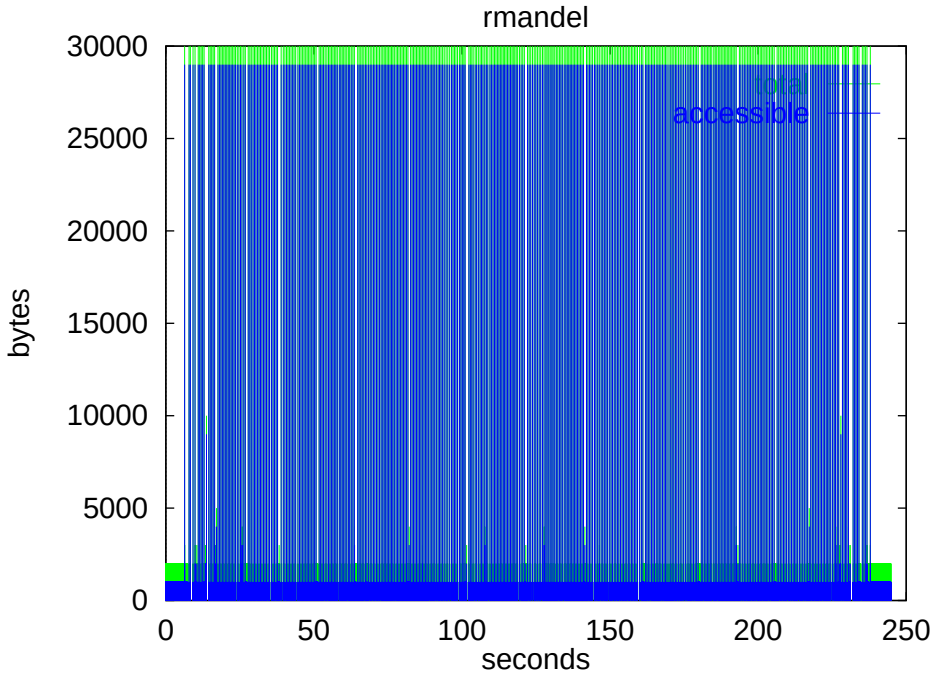



**Figure 4.16** Mandelbrot in detail

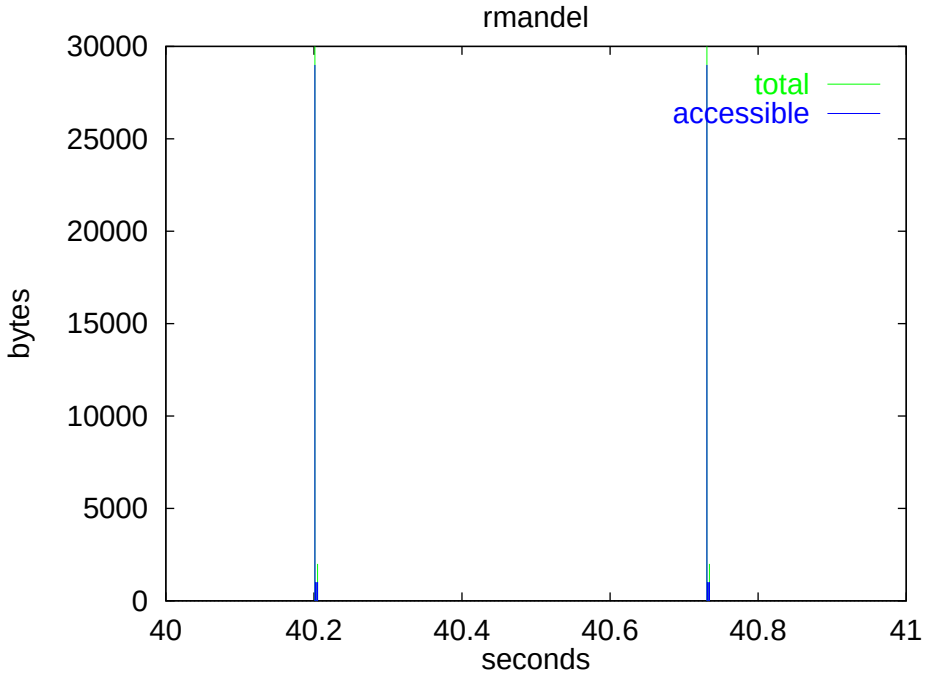
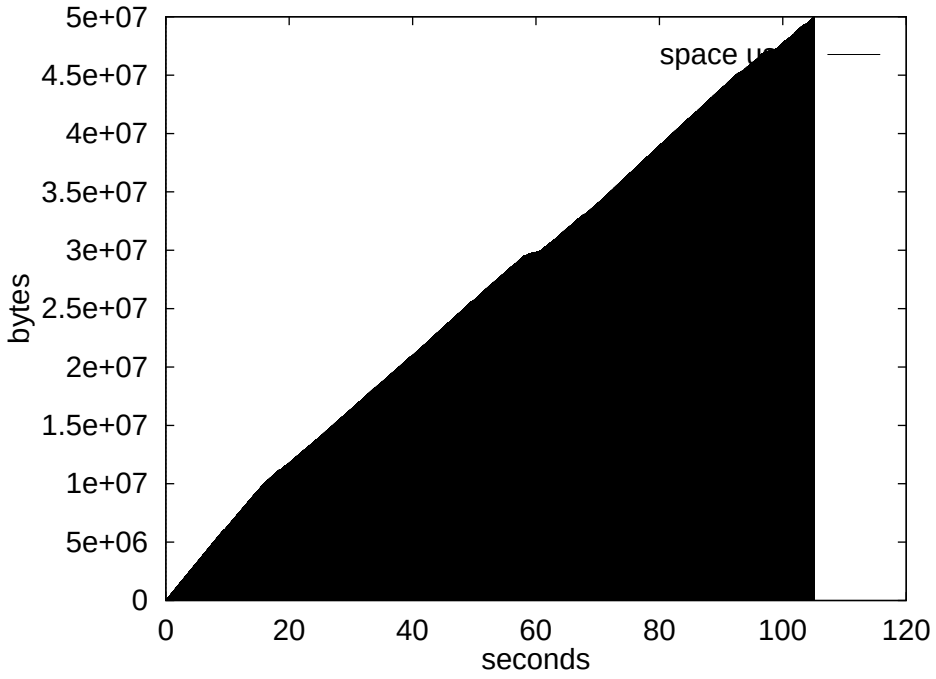


**Figure 4.17** Life - memory leak




while the profiling was dumped to a file. Note the "bump" on the curve. Presumably, some overhead (like using the swap-file on disc), is incurred at this point.

In an implementation with two regions, one of which is overwritten, and the other deallocated and reallocated between generations, the memory usage is much more acceptable, as seen in figure 4.18. Note the difference in the memory scales.

### 4.3.2 Speed Measurements

We use the CMM garbage collector for C++, as described in [AFI97], both in its default mode as a generational garbage collector, and in another mode as a conservative, mark-and-sweep garbage collector. We compare the run-time of RegJava to whichever of these methods has the best result for the particular example.

**REG** Programs use only unbounded regions.

**REG-B** Programs use bounded regions wherever possible, and otherwise they use unbounded regions.

**GC** Programs are rewritten to use the default generational garbage collector in CMM.

**GC-MSW** Programs use the conservative, mark-and-sweep garbage collector in CMM.

In figure 4.19 we give the measured execution times, and in the next figures we give, for each of our examples, a graphical comparison between the running times with regions and with garbage collection.

**Figure 4.18** Life - no memory leak

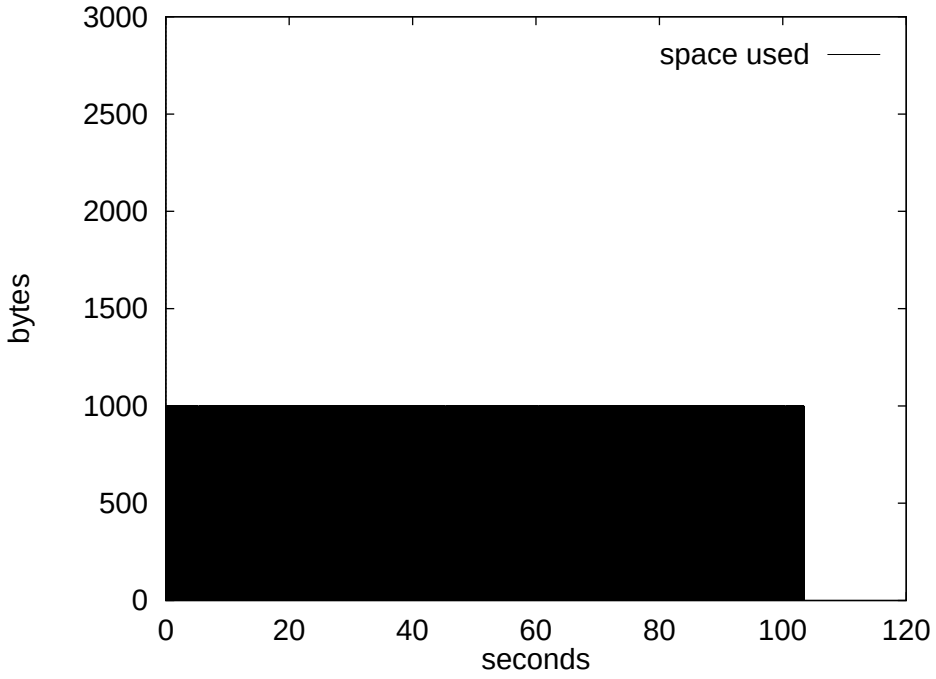
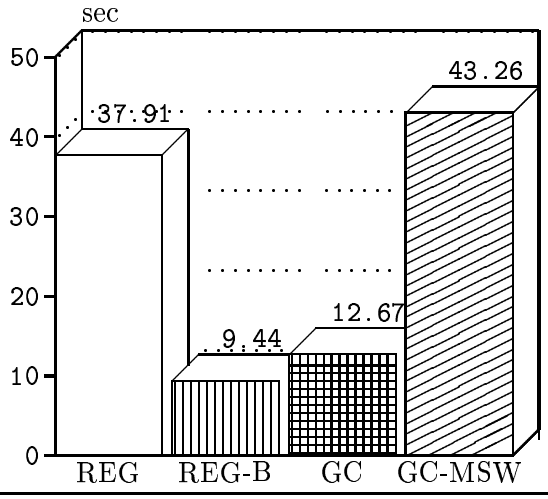



**Figure 4.19** Execution times

| | REG | B-REG | GC | GC-MSW |
|---|---|---|---|---|
| **ack** | 37.91 | 9.44 | 12.67 | 43.26 |
| **sieve** | 25.94 | — | 27.34 | 27.09 |
| **msort** | 14.11 | — | 76.79 | 27.91 |
| **mandel** | 25.05 | 15.31 | 38.80 | 22.47 |
| **life** | 48.94 | — | 51.77 | 48.74 |

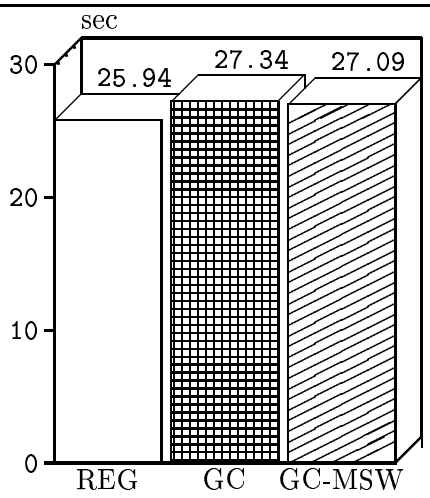
As can be seen by figure 4.20, for Ackermann CMM has much better running times with generational garbage collecting, than the C++ region framework. When the program is rewritten to use bounded regions, the performance is improved to match that of CMM.

**Figure 4.20** Ackermann's function



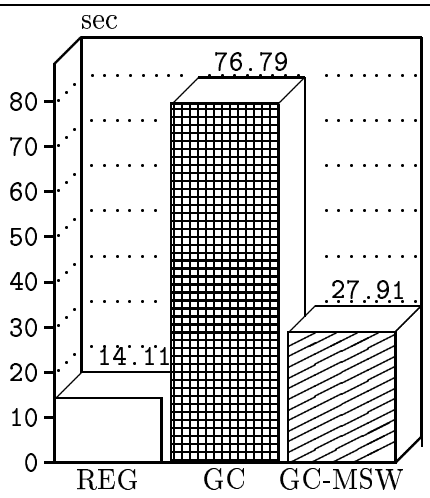

Eratosthenes' sieve runs in pretty much the same time in the C++ region framework and with garbage collection. This is a little surprising, since no memory is deallocated in the C++ region framework version until the very end of the execution, whereas garbage collection techniques might well do some clean-up "on the fly". Perhaps the total memory usage is so low, that garbage collecting simply is not done. Perhaps garbage collection *is* done, but its benefits with respect to locality is outweighed by the time spent doing the garbage collection.

**Figure 4.21** Eratosthenes' sieve



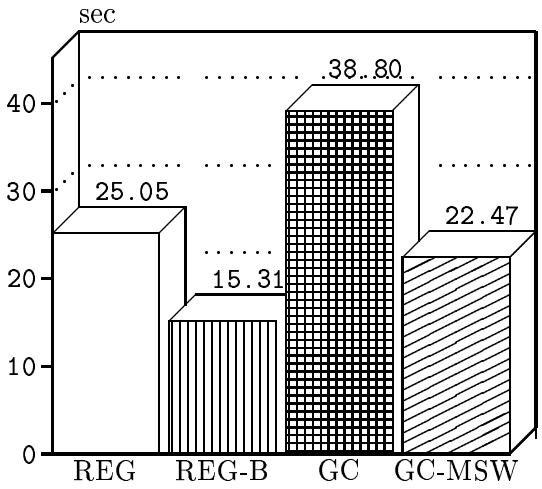

Mergesort runs slightly faster in the region based C++ framework, than with garbage collection. This is likely due to better locality of reference in the region-based version. For some reason, the generational garbage collector is extremely inefficient with this program. We do not know why.

**Figure 4.22** Merge sort




For the Mandelbrot example, the region framework runs in time comparable to the runtimes achieved with garbage collection. Changing the program to use bounded regions whenever possible gives some improvement, but nowhere near as much as was gained in the Ackermann example.

**Figure 4.23** Mandelbrot

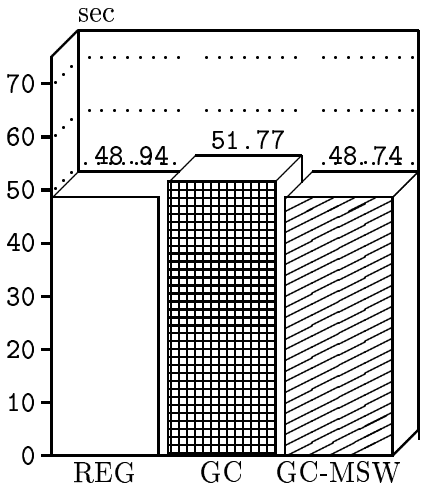



With the Life example (rlife3.cc in Appendix E, the optimised version), the region framework performs about the same as the garbage collected versions. It is worth noting, though, that the naive version only took 62.08 seconds to complete, so the speed gains from memory management even in this extremely memory consuming program, are not all that high. From a memory resource perspective, though, the graphs 4.17 and 4.18 speak for themselves.

### 4.3.3 Conclusion

In this section we briefly summarise what we have learned about our hypothesis.

**Figure 4.24** Convay's Game of Life




**RegJava has speed that can be reasoned about.**

This is not quite true, at least not on systems with caching. Programs that use more memory tend to run for a longer time than equivalent programs, that use less memory. On real-time systems, where the maximum execution times of each operation is well-known it is, by definition, possible to set upper bounds for the execution of RegJava programs.

**RegJava ensures smooth operation.**

Our memory graphs confirm this hypothesis. As they should.

**RegJava has good locality of reference.**

This depends very much on the program. If the program has rapid context switching, locality of reference may be arbitrarily bad. For "normal" programs, whatever that is, we still think that it is a fair statement.

**RegJava use less memory than garbage collection**

While cases can be constructed where garbage collection perform in much less memory than RegJava, RegJava does have very aggressive memory reclamation strategy, and is able to outperform garbage collection in many situations.

**RegJava has predictable memory usage**

This is indisputable.

**RegJava handles container classes poorly.**

We have given an example in `rstack.cc`. This is a problem with region based memory management in general, not just RegJava. We can not see any obvious solution.

**RegJava handles "trapped" dangling pointers well.**

While the block-structure of Java ensures that dangling pointers mostly cannot occur, it is possible to introduce them through subsumption. Such programs can not be handled well by garbage collection, but RegJava aggressively reclaims the memory.

**Regions tend to live longer in RegJava, than in ML.**

Well, perhaps. We have no indication either way.

**Fewer regions are used in RegJava than in ML.**

Again, we really cannot tell.

**Most regions in RegJava have more than one element.**

In two of our examples this was not so. While this cannot be said to conclusively falsify the hypothesis, it surely weakens it somewhat. Our experiments also indicate, that there are significant speed gains to be achieved for programs that can use bounded regions.

**Summary**

All in all, the results achievable with regions seem to have about the speed as garbage collection with CMM. Better results may be achievable as the with the addition of further analyses. It does seem, that good results can be achieved for some programs using bounded regions, and so hypothesis 10 does not so reasonable as we believed. The degree of memory reuse differ greatly in our examples, from Mandelbrot and Ackermann, where memory is aggressively reused, to Mergesort, where memory is not reused at all. The region model is better suited to the first type of programs, than to the second, obviously. Some of our programs, notably Ackermann and mergesort appear to have good locality of reference, where others look less good (Mandelbrot).

We include example programs rdemo1.cc and rstack.cc in appendix E to illustrate hypotheses 7 and 6, respectively.

# Chapter 5

# Future Work

In this thesis we have presented what we freely admit is a humble beginning of the project of creating a working Java with region based memory management. In this chapter we present our ideas about what it will take to complete the project. We discuss the language structures that are part of Java, but not part of RegJava. We discuss what modifications will have to be made in JVM and JVM interpreters in order to use regions. We very briefly discuss what further work is necessary if a realistic compiler is to be achieved.

## 5.1 The rest of Java

In chapter 2 we discussed the selection of a suitable subset of language features for RegJava. At this point we return to the excluded language structures and discuss if and how our model for region based memory management can be extended to cover them.

### 5.1.1 Simple extensions

We removed a number of constructs from RegJava, not because we expected that they would cause problems with respect to the region model, but simply because we wanted a simple and minimal language to work with. Most of these can be added to RegJava without much trouble.

#### `for` and `do` loops

Besides the `while` statement, Java has two extra iterative constructs; `do` and `for`. While these construct are certainly useful from a practical perspective, from a semantic perspective they are not interesting. They can be considered to be simple "syntactic sugar" for the `while` construct, and so it should pose no problems for the model.

#### `switch` statements

Similarly, the `switch` statement in Java can be viewed as "syntactic sugar" for a series of `if` statements, and so it should pose no problems for the model.

### Conditional operators

Java has conditional operators, that look like this

```
test ? trueresult : falseresult;
```

where `test` is a boolean expression, and `trueresult` and `falseresult` are expressions. This can easily be handled by production rules similar to those used for if statements, but on the expression level instead.

### Method parameters

In RegJava we only allow one parameter in method definition and method calls. If more parameters are to be allowed (or, indeed, methods are allowed to take no parameters), the method types will will have the form $\forall\vec{\rho}.(\tau_1,\ldots,\tau_n) \xrightarrow{\Delta} \tau'$, where $(\tau_1,\ldots,\tau_n)$ is the list of parameter types, and $\tau'$ is the return type of the method. In the local environment used for typing (and executing) the statement block all the parameters will have to be bound. Other than that, nothing changes.

### Type `void`

In RegJava we insist that methods return values as explained in section 2.3.10. It should pose no problems to allow for methods with no return value. A way of doing this, could be to create a type matching the value noval in the dynamic semantics. Java allows return statements without expressions, and it would be natural to allow this in RegJava if `void` methods were allowed.

### Further simple types

Java has the types `byte`, `short`, `long`, `float`, `double` and `char`, that are not part of RegJava. These types should follow the same scheme as for int and boolean with respect to regions, i.e. they should not be region-annotated. Variables of these types should be kept on the stack, and in object fields they should be stored in the main region of the object. Adding these types to RegJava also implies adding some implicit and explicit conversion operators for numbers. We see no problems with this.

### `break` and `continue` statements

Java has language structures for breaking out of loops. The `break` statement ends an iteration, and the `continue` statement skips to the next iteration of a loop. The statements has optional labels allowing them to exit several loops at once.

One way of adding the simple constructs to RegJava could be to add more dummy values like noval to the dynamic semantic and letting the execution of sequence and while constructions be dependent on these values. In this manner it would be insured that letregion statements would be terminated correctly.

Labels are much more problematic. Adding a label to a loop is no problem, but where should the label from a `break` or `continue` statement be kept? A new special environment? If a good mechanism for exception handling is developed, it is perhaps easier to consider a label on a loop as syntactic sugar for the `try`-construct setting up exception handling, and labeled `break`s or `continue`s could then be considered `throws`.

### 5.1.2 Object-oriented language constructs

Java has a number of object-oriented language constructs that we have excluded from RegJava, either because they are uninteresting with respect to regions or because they are extremely hard to fit into a region based model.

#### `static` fields and methods

In Java it is possible to declare fields within each class (class variables), that are global for all classes, rather than local to each object. Such fields are declared with the `static` keyword.

Similarly, methods can be declared to be class methods using the `static` keyword. Such methods do not have a `this` reference and can be invoked with the class name rather than a particular object as the basic entity.

Since classes are global to an entire program, there is no point in placing class fields in a region. Instead, they could be placed in a special environment corresponding to normal static memory usage. Static methods are much simpler than object methods, since there are no object regions to consider, and no binding of `this`. In Java, classes are represented as objects and can be loaded and unloaded dynamically. If RegJava were to have this functionality, it might make sense to place class fields in a global region, rather than in a special environment. This is by no means a trivial extension to RegJava, however. We will discuss it in section 5.2.

#### Arrays

In Java arrays are defined using the bracket (`[]`) syntax. Arrays are objects and are allocated with the `new` statements.

```
int [] temps = new int [e]
```

allocates an array of $e$ integers, where $e$ is an integer expression. This makes it impossible in general to statically determine the size of arrays. As long as RegJava only use unbounded regions anyway, this is not a problem. If bounded regions are introduced, only arrays whose sizes *can* be determined statically could be put in bounded regions. A notation for array types might be $[\tau]@\rho$, where $\tau$ is the type of the components of the array, and $\rho$ is the region the array is to be kept in. This notation seems natural since array are sometimes regarded as object with a number of unnamed fields of the same type.

#### Interfaces

Java supports an alternate subtype hierarchy based on *interfaces*. An variable or field may have an interface type rather than a "proper" class type, and then it is expected that the actual class of the object is one that implements the interface, i.e. has the required fields and methods described in the interface type definition and is a subclass of a class defined to `implement` the interface.

Since RegJava generally use structural types rather than a formal type hierarchy (except in the `new` command), this should present no problem. The usual RegJava subtyping mechanism should be able to handle interfaces with no modifications. If a type is "longer" than another, but the fields and methods they share have the same types, RegJava considers the "longer" type to be a subtype of the shorter type, without regard for whether the longer type is actually *defined* as a subclass of the shorter type. However, region annotations are considered part

of the types, and at this point, region annotations only are "propagated" inside the proper class hierarchy, in the sense that methods that override each other are forced to share method signatures (or almost, see the static semantics). With interfaces, types that are unrelated in the hierarchy may have method types that "override" each other, and so region types suddenly become interdependent across the region hierarchy.

### Constructors

Constructors are special methods that are always called when a new object is created. They are named like the class they appear in and take their arguments from the `new` statement. We see no problem in modifying the `new` statement of RegJava to find and call constructors on an object after its creation. (Of course, this violates the idea that object construction run in constant time. A constructor can run as long as it likes. But predictability is maintained since the programmer defines what the constructor is to do.) The grammar of RegJava is, in a way, prepared for constructors, since the current grammar for the `new` statement:

$$\texttt{new}[\vec{\rho}]\ A[\vec{p}](e_1, \ldots, e_n)$$

where $\vec{p}$ are the actual region arguments to the constructor and $e_1, \ldots, e_n$ are the actual value arguments.

### `public/private/protected/package` declarations

Java support several schemes for protection of methods and instance variables. RegJava only has what corresponds to the `public` scheme. Here a field or a method in a class can be accessed from anywhere. In the package (the default in Java) scheme, methods and fields can be accessed from other classes within the same package, but not from outside it. Since RegJava has no concept of packages, this is conceptually the same as `public` protection from our perspective. `private` fields and methods are accessible only from the class itself, and `protected` methods and fields can be accessed from the class and its subclasses.

We have not studied this thoroughly, but we believe it would be possible to handle these access modifiers the way it is suggested in [AC96]. They use different types for each class; one to be used only in the definition of the class itself, one to be used in subclasses, and one to be used everywhere else.

### `abstract` classes and methods

Abstract classes are classes that can never be instantiated but provide common information to their subtypes. Abstract methods are methods definitions without implementations, requiring implementation of the method to be put in subclasses. Abstract classes can be handled like any other RegJava class. They should be given proper region annotations, to be inherited by subclasses, and method annotations should propagate through them. The fact that no instances of abstract classes can be created causes no problem.

Abstract methods should be typed through the implementations in the subclasses only, since there is no contribution from the abstract method itself.

### `final` methods, fields and classes

If a method is declared `final` in Java, it cannot be overridden. The `final` keyword can also be used to protect entire classes against subtyping. It might be a good idea to extend the RegJava type scheme with optional annotations on classes and methods marking them as `abstract` or `final`, and optional `final` annotations on fields. Other than that we do not see any problems with final methods and classes.

The `final` keyword can be used to create variables that cannot be changed. This is possible to do in RegJava. If the `final` information is stored with the variable types, the static analysis can disallow other than initial assignments to such variables.

### Field initialisation

In Java, it is possible to initialise fields in objects explicitly, and all new instances of the methods then have have their initialisation done before the constructor is run. Class fields are initialised when the class is loaded. Since RegJava has no way of allocating regions in expressions, such initialisers would have to use only object regions in RegJava. Alternatively, an expression form of the letregion statement might be worth considering.

### Overloading

In Java, different methods can share the same name if their signatures (parameter number, parameter types, return types) differ. Since this is just "syntactic sugar" (the signatures effectively become part of the method names), RegJava could be extended with this functionality without too much trouble.

### The `super` keyword

In Java, it is possible to access a superclass's methods rather than the method belonging in the class itself, using `super` keyword, and it is possible to use the constructor for the superclass, by calling the `super()` method. Methods called with the `super` keyword use static dispatch, but unlike class methods, they bind the `this` keyword. If RegJava is extended with static methods, implementing the `super` keyword should not pose any problems.

### `instanceof` and type-casting of objects

In Java, it is possible to type-cast an object variable to a subtype if the actual object contained in the variable is of the subtype or deeper in the type hierarchy. The `instanceof` operator examine the actual type of an object to see if it matches a particular class or interface. In RegJava, a construction that type-casts an object to a subclass is possible, but it will have the side effect with regards to annotation of forcing all regions required by the subclass to be live at the point of the type-cast. The `instanceof` operator is not a problem without type-casting, since it can just check the class name in the object representation in the store (the $A$ in $<A, \tilde{V}, \vec{r}>$).

### `finalize()` methods

In Java, special `finalize()` methods can be added to classes. These methods are run before instances of the classes are deallocated—possibly by a garbage collector. It is possible to

add such a functionality to RegJava, but it will require keeping a list of objects that have `finalize()` methods with each region, and walking through the list when the region is deallocated. As with constructors, this would destroy the property that region deallocation is always done in constant time, but again the programmer would know exactly which objects had this property. It would also be necessary to restrict what could be done in the `finalize()` method. In Java, it is actually possible to save the objects from deallocation, by creating an external reference to `this`. In RegJava that would not work. If the object should be saved, a `clone()` operation would have to be used.

#### Methods and fields in the `Object` class

In RegJava, the *Object* class (the top class in the class hierarchy) has no objects or fields. In Java, a number of methods are defined in Object and inherited by all classes. Specifically, they are `clone()`, `equals()`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyall()`, `toString()`, and three versions of `wait()`.

We see a serious problem with adding these methods to *Object* in RegJava. This is because of the way the signature of overridden methods in superclasses are affected by the signature of the overriding methods in subclasses. Since the signatures of the methods in *Object* should not be something that change from program to program, it would means that the overridden methods of *Object* would be limited in their behaviour. In particular, the methods `clone()` and `equals()` could not be made to do a deep traversal of an object hierarchy if this hierarchy was spread over multiple regions.

A solution may be to remove these functionalities from the class Object. Instead, they should be implemented outside the class hierarchy as special functions applicable to all objects.

#### Native methods

In Java it is possible to link Java programs with methods written in other languages. Such methods are called native methods. Typically, the Java compiler (actually the **javah** tool) generates stubs for native methods, telling them where to find their parameters and whatever else is in the scope of the method. It should make no difference here that a region based memory model is used. How a native method should handle creation (and deallocation) of new objects is tricky, though, since the automatic annotation mechanisms of RegJava presumably would not extend to the new methods. Either such methods should then be hand-annotated by someone understands region discipline, or special tools should be created to assist in the proper region annotation of native methods.

### 5.1.3 Exception handling

The Java `try/catch` and `throw` exception handling structures can disrupt flow-of-control severely. We have not studied how to extend RegJava with exceptions. We believe that, although it is probably not easy, it can be done. We base this confidence mainly on the fact that the ML-Kit with regions can handle exceptions.

### 5.1.4 Threads

In Java, many threads can execute a program at once, sharing objects. This works well with garbage collection since objects are only deallocated when no threads can access them. We

are not sure if threads could be incorporated into the region-based model. It may be done by introducing some kind of reference-counting semaphore on every region that has to be shared by multiple threads. But this would certainly destroy the stack-like properties of the model.

## 5.2 Modules and separate compilation

For the typing of RegJava programs we assume that all the of the program is known. In general all classes used by a program should be known if good annotations are to be achieved. In Java, though, modular design is a very important feature. It should be possible to force programs to use classes with given, unchangeable region signatures. The resulting annotations may well be very inefficient, though.

## 5.3 JVM and bytecode

Although we feel that Java is an interesting language in its own right, its huge popularity has greatly been due to the portability that stems from compiling it to bytecode. The bytecode is run by a machine-dependent interpreter called a Java Virtual Machine (JVM). Therefore, if a region-based Java language is to be as popular as its originator, we must address this portability issue.

A specification of the JVM as designed by SUN can be found in [LY97]. Unfortunately, for reasons unknown to us, this specification explicitly demands that a JVM-implementation use some kind of garbage collection for managing dynamic memory. So it is impossible to implement a JVM that is based entirely on regions while at the same time living up to the specification.

Anyway, we find that the bytecode and the class file format of the specification are quite well-suited for extensions. We believe that it would be possible to extend the bytecode with two extra instructions for creating and destroying regions. In addition to this, the bytecode instructions for invoking methods and for creating objects should be enhanced to also handle region parameters. The region parameters could just be placed on the operand stack before or after the other parameters.

The class file format enables compiler-writers and JVM-designers to add extra information using so called *attributes* that are given an identifying name. If a certain JVM does not recognise a certain attribute name, it is obliged to silently ignore that attribute. Since there are no limitations on the kind of information stored in attributes, our basic idea is, therefore, to encode all the region annotations of both types and instructions into one or more attributes.

For instance, a naive way of doing this would be simply to keep two versions of both the type information and the code of methods: one version is the normal type information and code that lives up to SUN's specification, the other is a region-annotated version encoded in attributes. We could then design and implement a JVM that was aware of the attributes containing the region-annotated version. If it loads a class with region attributes, it would execute the class using regions. If it loads a class with no region attributes, it would fall back to the normal function of a JVM with garbage collection. Conversely, if a class file with region attributes is loaded by a normal JVM, this JVM simply ignores the region attributes and executes the class with garbage collection.

We strongly believe that the possibilities of the idea, described above, should be further investigated. One of the major reasons that new, promising programming methodologies do

not spread to a wider audience is that people do not like to make too many changes at once. If they can keep using old Java programs along with new ones using regions, the transition would be less painful and therefore more likely to succeed.

## 5.4 A realistic compiler

Besides extending RegJava to the rest of Java, there are some significant problems, that must be addressed, before a realistic compiler for RegJava will have any practical use. Here we list the major needs that we see.

**Good automatic annotations**
: This is the first and most comprehensive of the static analyses. Good results were achieved for ML. With luck, the basic methods can be reused in RegJava.

**Multiplicity analysis**
: Our measurements indicate that for some programs significant speed gains can be achieved by placing data in bounded regions. An analysis to do this automatically would probably be worthwhile.

**Tools for measuring memory usage and rewriting**
: In the ML-Kit, profiling tools are very much a part of gaining good performance from regions. This would likely also be the case with RegJava.

**Resetting of regions**
: Many of the situations where region discipline is really inefficient can be remedied by automatic or manual resetting of regions, i.e. emptying regions of data without removing them from the region stack. In the ML-Kit this is a very important analysis.

# Chapter 6

# Conclusion

We have presented a Java-like language in which objects are explicitly put in regions. The language has constructs for allocating, updating and deallocating regions, as well as region types for objects.

We have presented static and dynamic semantics for a small subset of Java annotated with regions, called RegJava. The semantics use a variable-based model, as is fitting for an imperative language, and so is very different from the semantics given for ML in [TT97] that use a reference-based memory model, as is suitable for a functional language. The static semantics ensures that well-typed programs use regions safely. The dynamic semantics is intentional with respect to a region-based store.

We further have presented a proof of soundness for these semantics, stating that well-typed programs do not go wrong. The proof is by induction on the static semantics and make the unproven assumption that statements cannot create new references to regions that are not live during the statement evaluation.

A "pseudo-implementation" of our model written in C++ was then described as an instance of a concrete model for implementing regions, and we compared this instance to garbage collection for some small examples. We feel that the C++ region framework is useful in itself as an alternative to doing manual memory management or retrofitting C++ with a garbage collector. The C++ framework has performance that is pretty much comparable to CMM, which is an impressive result considering the state of the region model technology.

We finally have presented our considerations about extending RegJava to the full Java language, and adding automatic region annotations, resetting regions, and multiplicity analysis. We believe that the most troublesome constructs in Java will be exception handling, threads and packages.

The major contribution of this thesis is a working model for region-based memory management for Java-like languages. We would have liked to also develop and implement the static analysis needed for automatic region annotation, but time would not allow it. As we indicated in the preface, the proof in chapter 3 has taken much more time and energy, than we initially had estimated. We have learned that defining language, semantics and soundness proofs together is harder than we would have imagined, particularly when the desired result is something as new and experimental as Java with regions.

# Appendix A

# Grammar of RegJava

| | | | |
|---|---|---|---|
| $\mathcal{P}$ | ::= | $\mathcal{C}_1 \cdots \mathcal{C}_n$ | program |
| $\mathcal{C}$ | ::= | class $A[\vec{\rho}]$ at $\rho$ extends $B[\vec{p}]$ $\{b_i \ ^{i \in 1 \ldots n}\}$ | class definition |
| $b$ | ::= | $\tau_2 \ m[\vec{\rho}](\tau_1 \ x) \ \{s\}$ | method definition |
| | \| | $\tau \ x;$ | field definition |
| $s$ | ::= | $\epsilon$ | empty statement |
| | \| | $e$ | statement expression |
| | \| | $\{s\}$ | block |
| | \| | $s_1; s_2$ | sequence |
| | \| | let $x : \tau = e$ in $s$ | variable declaration |
| | \| | if $e$ then $s_1$ else $s_2$ | conditional |
| | \| | return $e$ | return |
| | \| | letregion $\rho$ in $s$ | letregion |
| | \| | while $e$ do $s$ | while loop |
| $e$ | ::= | $x$ | variable |
| | \| | this | self parameter |
| | \| | $i$ | integer constant |
| | \| | false | boolean constant |
| | \| | true | boolean constant |
| | \| | null | null constant |
| | \| | $(e)$ | parenthesize |
| | \| | $e_1 + e_2$ | addition |
| | \| | $e_1 == e_2$ | equality |
| | \| | $e.x$ | field selection |
| | \| | $e_1.m[\vec{\rho}](e_2)$ | method invocation |
| | \| | $e_1.x = e_2$ | field update |
| | \| | $x = e$ | variable assignment |
| | \| | new$[\vec{\rho}] \ A()$ | object construction |

# Appendix B

# Static semantics of RegJava

## B.1 Programs and classes

(Reg Program)

$$\frac{\Pi = \{A_1 \mapsto \Gamma_1, \ldots, A_n \mapsto \Gamma_n\} \quad \Pi, A_1, \Delta \vdash \mathcal{C}_1 : \Gamma_1 \quad \cdots \quad \Pi, A_n, \Delta \vdash \mathcal{C}_n : \Gamma_n}{\Delta \vdash \mathcal{C}_1 \cdots \mathcal{C}_n : \Pi}$$

(Reg Class)

$$\frac{\begin{array}{c}\tau = [x_i : \tau_i{}^{\,i \in 1 \ldots k} \mid m_i : \phi_i{}^{\,i \in k+1 \ldots n}]@\rho \\ \Pi(B, \vec{p}) = \tau' \qquad \vdash \tau <: \tau' \\ \Delta \cap \mathrm{frv}(\vec{\rho}) = \emptyset \qquad \rho \in \Delta \cup \mathrm{frv}(\vec{\rho}) \\ \Delta \cup \mathrm{frv}(\vec{\rho}) \vdash b_i : \tau_i \qquad \forall i \in 1 \ldots k \\ \Pi, \Delta \cup \mathrm{frv}(\vec{\rho}), \tau \vdash b_i : \phi_i \qquad \forall i \in k+1 \ldots n\end{array}}{\Pi, A, \Delta \vdash \mathsf{class}\ A[\vec{\rho}]\ \mathsf{at}\ \rho\ \mathsf{extends}\ B[\vec{p}]\ \{b_i{}^{\,i \in 1 \ldots n}\} : \lambda\vec{\rho}.\tau}$$

(Reg Field)

$$\frac{\mathrm{frv}(\tau) \subseteq \Delta}{\Delta \vdash \tau\ x : \tau}$$

(Reg Method)

$$\frac{\Pi, \Delta', \{\mathsf{this} : \tau\} + \{x : \tau_1\} \vdash s : \tau_2 \quad \mathrm{frv}(\tau_1) \cup \mathrm{frv}(\tau_2) \cup \mathrm{frv}(\tau) \subseteq \Delta' \quad \Delta' \subseteq \Delta \cup \mathrm{frv}(\vec{\rho})}{\Pi, \Delta, \tau \vdash \tau_2\ m[\vec{\rho}](\tau_1\ x)\ \{s\} : \forall\vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2}$$

## B.2 Statements

(RegStm Empty)

$$\frac{}{\Pi, \Delta, E \vdash \epsilon : \tau}$$

(RegStm Exp)

$$\frac{\Pi, \Delta, E \vdash e : \tau'}{\Pi, \Delta, E \vdash e : \tau}$$

(RegStm Sequence)

$$\frac{\Pi, \Delta, E \vdash s_1 : \tau \quad \Pi, \Delta, E \vdash s_2 : \tau}{\Pi, \Delta, E \vdash s_1; s_2 : \tau}$$

(RegStm Return)

$$\frac{\Pi, \Delta, E \vdash e : \tau}{\Pi, \Delta, E \vdash \mathsf{return}\ e : \tau}$$

(RegStm Cond)
$$\frac{\Pi, \Delta, E \vdash e : \mathsf{boolean} \quad \Pi, \Delta, E \vdash s_1 : \tau \quad \Pi, \Delta, E \vdash s_2 : \tau}{\Pi, \Delta, E \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \tau}$$

(RegStm Declaration)
$$\frac{\Pi, \Delta, E \vdash e : \tau' \quad \Pi, \Delta, E + \{x : \tau'\} \vdash s : \tau \quad \mathrm{frv}(\tau') \subseteq \Delta}{\Pi, \Delta, E \vdash \mathsf{let}\ x : \tau' = e\ \mathsf{in}\ s : \tau}$$

(RegStm Letregion)
$$\frac{\Pi, \Delta \cup \{\rho\}, E \vdash s : \tau \quad \rho \notin \Delta}{\Pi, \Delta, E \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s : \tau}$$

(RegStm While)
$$\frac{\Pi, \Delta, E \vdash e : \mathsf{boolean} \quad \Pi, \Delta, E \vdash s : \tau}{\Pi, \Delta, E \vdash \mathsf{while}\ e\ \mathsf{do}\ s : \tau}$$

## B.3 Expressions

(RegExp Sub)
$$\frac{\Pi, \Delta, E \vdash e : \tau \quad \vdash \tau <: \tau'}{\Delta, E \vdash e : \tau'}$$

(RegExp $x$)
$$\frac{E(x) = \tau}{\Pi, \Delta, E \vdash x : \tau}$$

(RegExp Int)
$$\frac{}{\Pi, \Delta, E \vdash i : \mathsf{int}}$$

(RegExp Null)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{null} : \mathsf{Null}}$$

(RegExp False)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{false} : \mathsf{boolean}}$$

(RegExp True)
$$\frac{}{\Pi, \Delta, E \vdash \mathsf{true} : \mathsf{boolean}}$$

(RegExp Plus)
$$\frac{\Pi, \Delta, E \vdash e_1 : \mathsf{int} \quad \Pi, \Delta, E \vdash e_2 : \mathsf{int}}{\Pi, \Delta, E \vdash e_1 + e_2 : \mathsf{int}}$$

(RegExp Equal)
$$\frac{\Pi, \Delta, E \vdash e_1 : \tau \quad \Pi, \Delta, E \vdash e_2 : \tau}{\Pi, \Delta, E \vdash e_1 == e_2 : \mathsf{boolean}}$$

(RegExp Assign)
$$\frac{E(x) = \tau \quad \Pi, \Delta, E \vdash e : \tau}{\Pi, \Delta, E \vdash x = e : \tau}$$

(RegExp Field)
$$\frac{\Pi, \Delta, E \vdash e : [x : \tau]@\rho \quad \rho \in \Delta}{\Pi, \Delta, E \vdash e.x : \tau}$$

(RegExp Update)
$$\frac{\Pi, \Delta, E \vdash e_1 : [x : \tau]@\rho \quad \Pi, \Delta, E \vdash e_2 : \tau \quad \rho \in \Delta}{\Pi, \Delta, E \vdash e_1.x = e_2 : \tau}$$

(RegExp New)
$$\frac{\Pi(A, \vec{\rho}) = \tau = [\cdots]@\rho \quad \mathrm{frv}(\vec{\rho}) \subseteq \Delta \quad \rho \in \Delta}{\Pi, \Delta, E \vdash \mathsf{new}[\vec{\rho}]\ A() : \tau}$$

(RegExp Method)
$$\frac{\begin{array}{c}\Pi, \Delta, E \vdash e_0 : [m : \forall \vec{\rho}.\tau_1 \xrightarrow{\Delta'} \tau_2]@\rho \\ \Pi, \Delta, E \vdash e_1 : S(\tau_1) \\ \rho \in \Delta \quad S(\Delta') \subseteq \Delta \quad S(\mathrm{frv}(\vec{\rho})) \subseteq \Delta\end{array}}{\Pi, \Delta, E \vdash e_0.m[S(\vec{\rho})](e_1) : S(\tau_2)}$$

# Appendix C

# Dynamic semantics of RegJava

## C.1 Statements

(DynStm Exp)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash e \to \mathsf{noval}, \sigma', V'}$$

(DynStm Return)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{return}\ e \to v, \sigma', V'}$$

(DynStm Decl)
$$\frac{\sigma, V, R \vdash e \to v, \sigma', V' \qquad \sigma', V' + \{x \mapsto v\}, R \vdash s \to v', \sigma'', V''}{\sigma, V, R \vdash \mathsf{let}\ x = e\ \mathsf{in}\ s \to v', \sigma'', V'' \setminus\setminus \{x\}}$$

(DynStm Letregion)
$$\frac{r \notin \mathrm{Dom}(\sigma) \qquad \sigma + \{r \mapsto \emptyset\}, V, R + \{\rho \mapsto r\} \vdash s \to v, \sigma', V'}{\sigma, V, R \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ s \to v, \sigma' \setminus\setminus \{r\}, V'}$$

(DynStm Cond False)
$$\frac{\begin{array}{c}\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V' \\ \sigma', V', R \vdash s_2 \to v, \sigma'', V''\end{array}}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Cond True)
$$\frac{\begin{array}{c}\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \\ \sigma', V', R \vdash s_1 \to v, \sigma'', V''\end{array}}{\sigma, V, R \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to v, \sigma'', V''}$$

(DynStm Sequence Return)
$$\frac{\sigma, V, R \vdash s_1 \to v, \sigma', V' \qquad v \neq \mathsf{noval}}{\sigma, V, R \vdash s_1; s_2 \to v, \sigma', V'}$$

(DynStm Sequence Cont)
$$\frac{\begin{array}{c}\sigma, V, R \vdash s_1 \to \mathsf{noval}, \sigma', V' \\ \sigma', V', R \vdash s_2 \to v', \sigma'', V''\end{array}}{\sigma, V, R \vdash s_1; s_2 \to v', \sigma'', V''}$$

(DynStm While False)
$$\frac{\sigma, V, R \vdash e \to \mathsf{false}, \sigma', V'}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to \mathsf{noval}, \sigma', V'}$$

(DynStm While True)

$$\frac{\sigma, V, R \vdash e \to \mathsf{true}, \sigma', V' \qquad \sigma', V', R \vdash s; \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}{\sigma, V, R \vdash \mathsf{while}\ e\ \mathsf{do}\ s \to v, \sigma'', V''}$$

## C.2 Expressions

(DynExp $x$)

$$\frac{V(x) = v}{\sigma, V, R \vdash x \to v, \sigma, V}$$

(DynExp Const)

$$\frac{}{\sigma, V, R \vdash c \to c, \sigma, V}$$

(DynExp Plus)

$$\frac{\sigma, V, R \vdash e_1 \to i_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to i_2, \sigma'', V''}{\sigma, V, R \vdash e_1 + e_2 \to i_1 + i_2, \sigma'', V''}$$

(DynExp Equal True)

$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \qquad v_1 = v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{true}, \sigma'', V''}$$

(DynExp Equal False)

$$\frac{\sigma, V, R \vdash e_1 \to v_1, \sigma', V' \qquad \sigma', V', R \vdash e_2 \to v_2, \sigma'', V'' \qquad v_1 \neq v_2}{\sigma, V, R \vdash e_1 == e_2 \to \mathsf{false}, \sigma'', V''}$$

(DynExp Assign)

$$\frac{\sigma, V, R \vdash e \to v, \sigma', V'}{\sigma, V, R \vdash x = e \to v, \sigma', V' + \{x \mapsto v\}}$$

(DynExp Field)

$$\frac{\sigma, V, R \vdash e \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, \vec{r}> \qquad \tilde{V}(x) = v}{\sigma, V, R \vdash e.x \to v, \sigma', V'}$$

(DynExp Update)

$$\frac{\sigma, V, R \vdash e_1 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, \vec{r}> \qquad \sigma', V', R \vdash e_2 \to v, \sigma'', V''}{\sigma, V, R \vdash e_1.x = e_2 \to v, \sigma'' + \{a \mapsto <A, \tilde{V} + \{x \to v\}, \vec{r}>\}, V''}$$

(DynExp Method) ($x$ and $s$ is the formal parameter and body of method $m$ in class $A$)

$$\frac{\begin{array}{c}\Pi(A) = \lambda\rho_1 \cdots \rho_k.[\cdots, m : \forall\rho_{k+1} \cdots \rho_n.\tau_1 \xrightarrow{\Delta} \tau 2, \cdots]@\rho \\ \sigma, V, R \vdash e_0 \to a, \sigma', V' \qquad \sigma'(a) = <A, \tilde{V}, r_1, \ldots, r_k> \qquad \sigma', V', R \vdash e_1 \to v, \sigma'', V'' \\ \tilde{R} = \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k, \rho_{k+1} \mapsto R(\rho'_1), \ldots, \rho_n \mapsto R(\rho'_n)\} \\ \sigma'', \{\mathsf{this} \mapsto a\} + \{x \mapsto v\}, R + \tilde{R}, \vdash s \to v', \sigma''', V'''\end{array}}{\sigma, V, R \vdash e_0.m[\rho'_{k+1}, \ldots, \rho'_n](e_1) \to v', \sigma''', V''}$$

(DynExp New)

$$\frac{\begin{array}{c}\Pi(A, \rho_1, \ldots, \rho_k) = [x_i : \tau_i\ ^{i \in 1 \ldots n}]@\rho \\ \tilde{V} = \{x_1 \mapsto \mathbf{init}(\tau_1), \ldots, x_n \mapsto \mathbf{init}(\tau_n)\} \qquad R(\rho) = r \qquad o \notin \mathrm{Dom}(\sigma(r))\end{array}}{\sigma, V, R \vdash \mathsf{new}\ A[\rho_1, \ldots, \rho_k]() \to (r, o), \sigma + \{(r, o) \mapsto <A, \tilde{V}, R(\rho_1), \ldots, R(\rho_k)>\}, V}$$

# Appendix D

# Region Framework in C++

In this appendix we present the simple region framework in C++ that we discussed in section 4.1.4. The code presented here is not exactly the code used for profiling the examples in section 4.3 because we have removed all profiling code.

The framework consists of one header file and one code file. The header file declares three classes. `Region` is the abstract base class for regions. `URegion` is an implementation of unbounded regions, and `BRegion` is an implementation of bounded regions.

Unbounded regions are declared like ordinary variables:

```
URegion rho1;
```

A bounded region can be declared using the macro `declareBRegion`:

```
declareBRegion(rho1, SomeType);
```

Here `rho1` is a new region variable, and `SomeType` is the type of the object that is eventually put into that region.

Objects are created in the region denoted by the region variable `rho1` by the following construct:

```
new(rho1) SomeType;
```

## D.1 Header file: `regmem.hh`

```
#include <stddef.h>
#include <assert.h>

#define REGION_PAGE_SIZE 1000

class Region {
  Region(Region&);              // copy protection
  void operator=(Region&);      // copy protection
public:
  Region() {}
```

```
  virtual ~Region() {}
  virtual void* alloc(size_t) = 0;
};

struct Page;

class PageList {
protected:
  Page* first;
  Page* last;
public:
  PageList();
  bool empty() const;
  void clear();
  void add(Page*);
  Page* top() const;
  Page* pop();
  void append(PageList&);
};

class URegion : public Region {
protected:
  size_t free;
  PageList pages;
  virtual void freemem();
public:
  URegion();
  virtual ~URegion();
  virtual void* alloc(size_t);
  void reset();
};

class BRegion : public Region {
protected:
  const size_t size;
  void* data;
public:
  BRegion(const size_t s, void* d) : size(s), data(d) {}
  virtual ~BRegion() {}
  virtual void* alloc(size_t);
};

#define declareBRegion(reg,type)\
char _regdata_##reg[sizeof(type)];\
BRegion reg(sizeof(type), _regdata_##reg)
```

```
inline void* operator new(size_t size, Region& rho) {
  assert(size <= REGION_PAGE_SIZE);
  return rho.alloc(size);
}
```

## D.2 Code file: regmem.cc

```
#include "regmem.hh"

struct Page {
  Page* next;
  Page* prev;
  char data[REGION_PAGE_SIZE];

  Page() {
    next = NULL;
    prev = NULL;
  }

};

class FreePages {
protected:
  PageList pages;

public:
  FreePages() {}

  Page* alloc() {
    if (pages.empty())
      return new Page();
    else
      return pages.pop();
  }

  void free(PageList& pl) {
    pages.append(pl);
  }

} free_pages;


// PageList

PageList::PageList() : first(NULL), last(NULL) {}
```

```
bool PageList::empty() const {
  return first == NULL;
}

void PageList::clear() {
  first = NULL;
  last = NULL;
}

void PageList::add(Page* p) {
  assert(p != NULL);
  p->prev = NULL;
  if (first == NULL) {
    p->next = NULL;
    last = p;
  }
  else {
    p->next = first;
    first->prev = p;
  }
  first = p;
}

Page* PageList::top() const {
  return first;
}

Page* PageList::pop() {
  assert(first != NULL);
  Page* p = first;
  first = first->next;
  if (first != NULL)
    first->prev = NULL;
  return p;
}

void PageList::append(PageList& pl) {
  if (pl.empty())
    return;
  if (first == NULL)
    last = pl.last;
  else {
    first->prev = pl.last;
    pl.last->next = first;
  }
  first = pl.first;
  pl.clear();
```

```
}

// URegion

URegion :: URegion () {
  free = 0;
}

URegion ::~ URegion () {
  freemem ();
}

void* URegion :: alloc (size_t s) {
  if (pages.empty() || (s > free)) {
    pages.add(free_pages.alloc());
    free = REGION_PAGE_SIZE;
  }
  size_t top = REGION_PAGE_SIZE - free;
  free -= s;
  return &(pages.top()->data[top]);
}

void URegion :: reset () {
  freemem ();
}

void URegion :: freemem () {
  free_pages.free(pages);
}


// BRegion

void* BRegion :: alloc (size_t s) {
  assert (s <= size);
  return data;
}
```

# Appendix E

# Source code of examples in C++

## E.1 Sieve of Eratosthenes: `rsieve.cc`

```
#include <iostream.h>
#include "regmem.hh"
#include "regprof.hh"
#include "sieve.hh"

class Node {
public:
  int i;
  Node* next;

  Node(int i, Node* n) {
    this->i = i;
    this->next = n;
  }

};

Node* /* @r1 */ copy(Region& r1, Node* ns) {
  PROFILE(profile.current_access(r1.getSize()));
  if (ns == NULL)
    return NULL;
  else
    { PROFILE(profile.current_access(r1.getSize()));
    return new(r1) Node(ns->i,copy(r1,ns->next)); }
}

Node* /* @r1 */ fromto(Region& r1, int from, int to) {
  if (from == to)
    return new(r1) Node(to, NULL);
  else
    return new(r1) Node(from, fromto(r1, from+1,to));
```

```
}

Node* remove(int p, Node* ns) {
  if (ns == NULL)
    return NULL;
  if ((ns->i % p) == 0)
    return remove(p, ns->next);
  ns->next = remove(p, ns->next);
  return ns;
}

Node* sieve(Node* ns) {
  Node* ps = ns;
  while (ns != NULL)
    ns = remove(ns->i, ns->next);
  return ps;
}

Node* /* @r1 */ primes(Region& r1, int max) {
  URegion r2;
  Node* /* r2 */ numbers = fromto(r2, 2, max);
  PROFILE(profile.current_access(r1.getSize()+r2.getSize()));
  return copy(r1,sieve(numbers));
}

int main() {
  {
    URegion r1;
    Node* /* r1 */ p = primes(r1, PRIMES);
    PROFILE(profile.current_access(r1.getSize()));
    while (p != NULL) {
      cout << p->i << " ";
      p = p->next;
    }
  }
  cout << "\n";
}
```

## E.2 Ackermann: `rack.cc`

```
#include <iostream.h>
#include "regmem.hh"
```

```
#include "ack.hh"

class Pair {
  PADDING;
public:
  int x;
  int y;

  Pair(int x, int y) {
    this->x = x;
    this->y = y;
  }

};

int ack(Pair* p) {
  if (p->x == 0)
    return p->y + 1;
  else if (p->y == 0)
    {
      URegion rho2;
      return ack(new(rho2) Pair(p->x - 1, 1));
    } // pop rho2
  else
    {
      URegion rho3;
      return ack(new(rho3) Pair(p->x - 1,
                 ack(new (rho3) Pair(p->x, p->y - 1))));
    } // pop rho3
}

int main() {
  int i, j;
  i = 0;
  while (i < 4) {
    j = 0;
    while (j < 11 && i * 10 + j < LIMIT) {
      {
    URegion rho1;
    cout << ack(new(rho1) Pair(i, j)) <<  "\t";
      } // pop rho1
      j = j + 1;
    }
    cout << "\n";
    i = i + 1;
  }
```

```
}
```

## E.3 Merge Sort: `rmsort.cc`

```
#include <iostream.h>
#include "regmem.hh"
#include "regprof.hh"

#include "msort.hh"

extern "C" int rand();

class List {
protected:
  Region& rho1;
public:
  List(Region& r1) : rho1(r1) {}

  virtual bool isNil() = 0;
  virtual int hd() = 0;
  virtual List* tl() = 0;

  virtual List* take(Region&, int);
  virtual List* drop(Region&, int);
  virtual List* copy(Region&);
  virtual int length();
  virtual List* msort(Region&);
  virtual List* merge(Region&, List*);
  virtual bool isSorted();

  virtual void print();

  static List* build(Region&, int);
};

class Nil : public List {
public:
  Nil(Region& r1) : List(r1) {}

  virtual bool isNil() { return true; }
  virtual int hd() { exit(1); return 0; }
  virtual List* tl() { exit(1); return NULL; }
};

class Cons : public List {
public:
  int data;
```

```
  List * tail;

  Cons(Region& r1, int d, List * t) : List(r1) {
    data = d;
    tail = t;
  }

  virtual bool isNil() { return false; }
  virtual int hd() { return data; }
  virtual List * tl() { return tail; }
};


List * /* @rho2 */ List::take(Region& rho2, int n) {
  if ((this->isNil()) || (n<=0))
    return new(rho2) Nil(rho2);
  else
    return new(rho2) Cons(rho2, this->hd(),
                this->tl()->take(rho2, n-1));
}


List * /* @rho2 */ List::drop(Region& rho2, int n) {
  if (this->isNil() || (n==0))
    return this->copy(rho2);
  else
    return this->tl()->drop(rho2, n-1);
}

List * /* @rho2 */ List::copy(Region& rho2) {
  if (this->isNil())
    return new(rho2) Nil(rho2);
  else
    return new(rho2) Cons(rho2, this->hd(),
                this->tl()->copy(rho2));
}

void List::print() {
  if (!this->isNil()) {
    cout << this->hd() << ' ';
    this->tl()->print();
  }
}

int List::length() {
  if (this->isNil())
    return 0;
```

```
  else
    return 1+this->tl()->length();
}

List* /* @rho2 */ List::merge(Region& rho2, List* xs) {
  if (this->isNil())
    return xs->copy(rho2);
  else if (xs->isNil())
    return this->copy(rho2);
  else if (this->hd() <= xs->hd())
    return new(rho2) Cons(rho2, this->hd(),
                this->tl()->merge(rho2,xs));
  else
    return new(rho2) Cons(rho2, xs->hd(),
                xs->tl()->merge(rho2,this));
}

List* /* @rho2 */ List::msort(Region& rho2) {
  int k = this->length() / 2;
  if (k==0)
    return this->copy(rho2);
  else {
    URegion rho3;
    List* /* @rho3 */ mxs = NULL;
    List* /* @rho3 */ mys = NULL;
    {
      URegion rho4;
      List* xs /* @rho4 */ = this->take(rho4, k);
      PROFILE(profile.current_access(rho4.getSize()+rho3.getSize()+
                      rho2.getSize()));
      List* /* @rho4 */ ys = this->drop(rho4, k);
      PROFILE(profile.current_access(rho4.getSize()+rho3.getSize()+
                      rho2.getSize()));

      mxs = xs->msort(rho3);
      PROFILE(profile.current_access(rho4.getSize()+rho3.getSize()+
                      rho2.getSize()));
      mys = ys->msort(rho3);
      PROFILE(profile.current_access(rho4.getSize()+rho3.getSize()+
                      rho2.getSize()));
    }
    return mxs->merge(rho2, mys);
  }
}

bool List::isSorted() {
  if ( (this->isNil()) || this->tl()->isNil() )
```

```
    return true;
  else
    return (this->hd() <= this->tl()->hd())
      && (this->tl()->isSorted());
}

List* /* @rho3 */ List::build(Region& rho3, int n) {
  List* xs = new(rho3) Nil(rho3);
  while (n > 0) {
    int x = rand()%1000;
    xs = new(rho3) Cons(rho3, x, xs);
    PROFILE(profile.current_access(rho3.getSize()));
    n = n - 1;
  }
  return xs;
}



int main() {
  URegion rho4;
  List* /* @rho4 */ xs = List::build(rho4, LENGTH);
    PROFILE(profile.current_access(rho4.getSize()));
//    cout << "\nxs =";
//    xs->print();
  if (xs->isSorted())
    cout << "\nxs is sorted!\n";
  else
    cout << "\nxs is NOT sorted!\n";

  List* /* @rho4 */ mxs = xs->msort(rho4);
    PROFILE(profile.current_access(rho4.getSize()));
//    cout << "\nmxs = ";
//    mxs->print();
  if (mxs->isSorted())
    cout << "\nmxs is sorted!\n";
  else
    cout << "\nmxs is NOT sorted!\n";
  return 0;
}
```

## E.4 Mandelbrot: `rmandel.cc`

```
#include <iostream.h>
#include "regmem.hh"
#include "regprof.hh"
```

```
#include "mandel.hh"


class complex {
private:
  double re, im;
  char dummy[10];

public:
  complex (double r = 0, double i = 0) {
    re = r;
    im = i;
  }

  complex (complex* c) {
    re = c->real();
    im = c->imag();
  }


  double real() {
    return re;
  }

  double imag() {
    return im;
  }

  complex*/*@rho1*/ plus (Region& rho1, complex*/*@rho2*/ r)
  {
    return new(rho1) complex(real()+r->real(), imag()+r->imag());
  }

  complex*/*@rho1*/ minus (Region& rho1, complex*/*@rho2*/ r)
  {
    return new(rho1) complex(real()-r->real(), imag()-r->imag());
  }

  complex*/*@rho1*/ times (Region& rho1, complex*/*@rho2*/ r)
  {
    return new(rho1) complex(real()*r->real()-imag()*r->imag(),
                  real()*r->imag()+imag()*r->real());
  }

  complex*/*@rho1*/ f(Region& rho1, complex*/*@rho2*/ z)
  {
    URegion rho3;
```

```
    return new(rho1) complex(z->times(rho3, z)->plus(rho3, this));
  }

  bool in_mandel()
  {
    URegion rho2;
    complex*/*@rho2*/ z = new(rho2) complex(0,0);
    PROFILE(profile.current_access(rho2.getSize()));
    for (int n=0; n < MAX_ITERATIONS; n++) {
      z = f(rho2, z);
      if ((z->real()*z->real()+z->imag()*z->imag()) >= 4)
    { PROFILE(profile.current_access(rho2.getSize()));
    return false;}
    }
    PROFILE(profile.current_access(rho2.getSize()));
    return true;
  }

  friend ostream& operator<<(ostream&,complex*);

};

inline ostream& operator<<(ostream& out, complex* c) {
  return out << '(' << c->real() << ',' << c->imag() << ')';
}



const double min_x = -2;
const double min_y = -2;

const double size = 4;

const double max_x = min_x+size;
const double max_y = min_y+size;

const double step_x = (max_x-min_x)/100;
const double step_y = (max_y-min_y)/50;

inline plot(char c) {
  cout << c;
}

void main()
{
  PROFILE(profile.current_access(0));
  for (double i=max_y; i>=min_y; i-=step_y) {
```

```
    for (double r=min_x; r<=max_x; r+=step_x) {
      PROFILE(profile.current_access(0));
      if ((r*r<=step_x*step_x/4) || (i*i<=step_y*step_y/4)) {
    plot('+');
    continue;
      }
      URegion rho1;
      complex*/*@rho1*/ c = new(rho1) complex(r,i);
      PROFILE(profile.current_access(rho1.getSize()));
      if (c->in_mandel())
    plot('*');
      else
    plot(' ');
      PROFILE(profile.current_access(rho1.getSize()));
    }
    plot('\n');
  }

}
```

## E.5 Naive life: `rlife.cc`

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include "regmem.hh"
#include "regprof.hh"
#include "life.hh"

class Generation {
protected:
  Region& rho1;
public:
  char cells[MAXROW+1][MAXCOL+1];

  Generation(Region& r1) : rho1(r1) {
    int r = 0;
    int c = 0;
    // make a border of live cells
    for (r=0; r<MAXROW; r++)
      cells[r][0] = cells[r][MAXCOL-1] = 1;
    for (c=0; c<MAXCOL; c++)
      cells[0][c] = cells[MAXROW-1][c] = 1;
  }

  virtual void init() {
    int r = 0;
```

```
    int c = 0;
    // put random values in the rest
    for (r=1; r<MAXROW-1; r++)
      for (c=1; c<MAXCOL-1; c++)
    cells[r][c] = abs(rand())%2;
  }

  virtual Generation* /* @rho2 */ nextGen(Region& rho2) {
    int r = 0;
    int c = 0;
    PROFILE(profile.current_access(rho2.getSize()));
    Generation* /* @rho2 */ gen = new(rho2) Generation(rho2);
    for (r=1; r<MAXROW-1; r++)
      for (c=1; c<MAXCOL-1; c++) {
    int n = this->neighbours(r,c);
    int cell = this->cells[r][c];
    if ((cell==1) && (n<2))
      gen->cells[r][c]=0; // dies of loneliness
    else if ((cell==1) && (n>3))
      gen->cells[r][c]=0; // dies of overcrowding
    else if ((cell==0) && (n==3))
      gen->cells[r][c]=1; // springs to life
    else
      gen->cells[r][c]=cell; // stays alive
      }
    return gen;
  }

  virtual int neighbours(int r, int c) {
    return cells[r-1][c-1] + cells[r-1][c] + cells[r-1][c+1]
      + cells[r][c-1] + cells[r][c+1]
      + cells[r+1][c-1] + cells[r+1][c] + cells[r+1][c+1];
  }

  virtual void print() {
    int r = 0;
    int c = 0;
    cout << "\n";
    for (r=0; r<MAXROW; r++) {
      for (c=0; c<MAXCOL; c++)
    cout << (cells[r][c]?'*':' ');
      cout << "\n";
    }
  }

};
```

```
void main() {
  srand(time(NULL));
  URegion rho1;
  Generation* /* @rho1 */ gen = new(rho1) Generation(rho1);
  gen->init();
  for (int i=1; i<=GENERATIONS; i++) {
    cout << i << ". generation:";
    gen->print();
    gen = gen->nextGen(rho1);
  }
}
```

## E.6 Optimized Life: `rlife3.cc`

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include "regmem.hh"
#include "regprof.hh"
#include "life.hh"

class Generation {
protected:
  Region& rho1;
public:
  char cells[MAXROW+1][MAXCOL+1];

  Generation(Region& r1) : rho1(r1) {
    int r = 0;
    int c = 0;
    // make a border of live cells
    for (r=0; r<MAXROW; r++)
      cells[r][0] = cells[r][MAXCOL-1] = 1;
    for (c=0; c<MAXCOL; c++)
      cells[0][c] = cells[MAXROW-1][c] = 1;
  }

  virtual void init() {
    int r = 0;
    int c = 0;
    // put random values in the rest
    for (r=1; r<MAXROW-1; r++)
      for (c=1; c<MAXCOL-1; c++)
```

```
    cells[r][c] = abs(rand())%2;
  }

  virtual Generation*/*@rho4*/ nextGen(Region& rho3, Region& rho4) {
    int r = 0;
    int c = 0;
    PROFILE(profile.current_access(rho3.getSize()+rho4.getSize()));
    Generation* /* @rho4 */ gen = new(rho4) Generation(rho4);
    for (r=1; r<MAXROW-1; r++)
      for (c=1; c<MAXCOL-1; c++) {
    int n = this->neighbours(r,c);
    int cell = this->cells[r][c];
    if ((cell==1) && (n<2))
      gen->cells[r][c]=0; // dies of loneliness
    else if ((cell==1) && (n>3))
      gen->cells[r][c]=0; // dies of overcrowding
    else if ((cell==0) && (n==3))
      gen->cells[r][c]=1; // springs to life
    else
      gen->cells[r][c]=cell; // stays alive
      }
    return gen;
  }

  virtual int neighbours(int r, int c) {
    return cells[r-1][c-1] + cells[r-1][c] + cells[r-1][c+1]
      + cells[r][c-1] + cells[r][c+1]
      + cells[r+1][c-1] + cells[r+1][c] + cells[r+1][c+1];
  }

  virtual void print() {
    int r = 0;
    int c = 0;
    cout << "\n";
    for (r=0; r<MAXROW; r++) {
      for (c=0; c<MAXCOL; c++)
    cout << (cells[r][c]?'*':' ');
      cout << "\n";
    }
  }
};


void main() {
  int r = 0;
  int c = 0;
  srand(time(NULL));
```

```
  URegion rho1;
  Generation* /* @rho1 */gen = new(rho1) Generation(rho1);
  gen->init();
  for (int i=1; i<=GENERATIONS; i++) {
    URegion rho2;
    cout << i << "␣generation:";
    gen->print();
    Generation* /* @rho2 */ gen2=gen->nextGen(rho1,rho2);
    for (r=1; r<MAXROW-1; r++) //Clone
      for (c=1; c<MAXCOL-1; c++) {
    gen->cells[r][c]=gen2->cells[r][c];
      };
  }; // end Region rho2
}
```

## E.7 Optimized Life: `rdemo1.cc`

```
#include "regmem.hh"
#include "regprof.hh"
#include <stdio.h>

// This code should illustrate
// that some code will always run better
// with regions than with garbage colloection

// data[rho1]=[val:char[1000]] @rho1
class data{
public:
char val[1000];
};

// A[rho1]=[data1:data[rho1]] @rho1
class A {
public:
data* data1;
};

//B[rho1,rho2]=[data1:data[rho1], data2:data[rho2]] @rho1
class B: public A {
public:
data* data2;
};
```

```
int main(){
  int n;
  URegion rho3;
  {
    A/*[rho3]*/ obj1=NULL;
    URegion rho4;
    B/*[rho3,rho4]*/ obj2=*new(rho3) B;
    obj2.data1=new(rho3) data;
    obj2.data2=new(rho4) data;
    obj1=obj2;
    for (n=1;n!=1000;n++) {
      PROFILE(profile.current_access(rho3.getSize()+
                    rho4.getSize()));
    }
  } //rho4 is deallocated

  //Here we have a dangling pointer in obj1 legally
  for (n=1;n!=1000;n++) {
    PROFILE(profile.current_access(rho3.getSize()));}

}
```

## E.8 Optimized Life: `rstack.cc`

```
#include <iostream.h>
#include "regmem.hh"
#include "stack.hh"

extern "C" int rand();


class Cell {
protected:
  Region& rho1;
public:
  int data;
  Cell* next;

  Cell(Region& r1, int d, Cell* n) : rho1(r1) {
    data = d;
    next = n;
  }

};
```

```
class Stack {
protected:
  Region& rho1;
public:
  Cell* tp;

  Stack(Region& r1) : rho1(r1) {
    tp = NULL;
  }

  virtual bool isEmpty() {
    return tp == NULL;
  }

  virtual void push(int i) {
    tp = new(rho1) Cell(rho1, i, tp);
  }

  virtual void pop() {
    tp = tp->next;
  }

  virtual int top() {
    return tp->data;
  }

};


int main() {
  URegion rho1;
  Stack* s = new(rho1) Stack(rho1);
  int i = 0;
  while (i<OUTER) {
    int j = 0;
    while (j<INNER) { s->push(j); j = j + 1; }
    while (!s->isEmpty()) s->pop();
    i = i + 1;
  }

  return 0;
}
```

# Bibliography


[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Acz77] Peter Aczel. An introduction to inductive definitions. In K. J. Barwise, editor, *The Handbook of Mathematical Logic*, Studies in Logic and Foundations of Mathematics. North Holland, 1977.

[AFI97] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. A customisable memory management framework for C++, 1997. The research described here has been funded in part by the ESPRIT Basic Research Action, project 6846, PoSSo: Polynomial System Solving and by ESPRIT Reactive LTR, project 21.024, FRISCO: A Framework for Integrated Symbolic Numeric Computing.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, USA, May 1996.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1996.

[FA95] Tito Flagella and Giusseppe Attardi. POSSO customisable memory management (CMM), November 1995.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1997.

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, September 1991.

[Str94] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, USA, second edition, 1994.

[TB96] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. Accepted for the Milner Festschrift (25 pages), November 1996.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. Transactions on Programming Languages and Systems (TOPLAS) (Accepted), November 1998.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science (DIKU), University of Copenhagen, April 1997.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.