Region-Based Model Abstraction

Jeremy Condit       James R. Larus       Sriram K. Rajamani
jcondit@cs.berkeley.edu   larus@microsoft.com   sriram@microsoft.com

Jakob Rehof
rehof@microsoft.com

August 2003

Technical Report
MSR-TR-2003-47

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com
Abstract

We present a new technique for abstracting programs to models suitable for state space exploration (e.g., using a model checker). This abstraction technique is based on a region type system in which regions are interpreted as sets of values. A major benefit of the region abstraction is that it soundly supports aggressive state space reduction and state size reduction in the presence of aliasing without relying on either imprecise global static alias analysis or a large pointer slice of the source program prior to model construction.

Region types themselves contain only locally sound alias information; however, our models are globally sound by computing dynamically with region names at model checking time. This technique effectively shifts part of the alias analysis to the model checker and leverages state space reduction as well as enhanced model precision. Region types also provide a flexible framework for adjusting the tradeoff between model precision and performance. We have used these techniques to implement a region-based model compiler for C#.

1 Introduction

Model checking [7] is an important technique for verifying temporal safety properties of software. Informally, a temporal safety property is one whose violation is witnessed by a finite execution trace of the program [20, 24]. A model checker attempts to check that a temporal safety property holds by systematically exploring the state space of a program. Software model checking has received much attention lately, and a number of recent overviews are available [4, 15, 17].

The advantage of model checking techniques is that they offer a high degree of precision and can therefore find non-trivial programming errors. However, in order to use a model checker on realistic software systems, we must confront the state explosion problem [7]; the number of possible states for a realistic software system is typically infinite or astronomically large, thus making a systematic exploration of these states impractical. Therefore, most model checkers attempt to decrease the state space of programs by constructing abstractions of them and by exploring these abstractions.

When generating abstractions, we identify certain data structures that are relevant to the property we wish to check. Expressions and statements involving those structures are added to our model, and the remaining expressions and statements are thrown away. However, relevant objects may be aliased through references. For example, suppose the field relevant_field is considered relevant in the following statements:

\[ \text{(*)} \quad \text{if(...) \, x = y; \ldots \quad x\.relevant\_field = true; \quad y\.relevant\_field = false; } \]

In this example, the statement (\*) may cause \(x\) and \(y\) to become aliased. If these two references are aliased, then \(x\.relevant\_field\) has the value \texttt{false} after executing the last two statements; otherwise, it has the value \texttt{true}. Thus, if we wish to throw away statement (\*), when generating a model, we must still account for the fact that \(x\) and \(y\) may be aliased.

Currently, the following strategies have been proposed for dealing with the aliasing problem in model abstraction: (i) global static alias analysis [2] computes alias sets prior to model construction and uses that information to represent alias approximations in the model; (ii) slicing [30, 16] includes in the model every assignment statement that could possibly affect aliasing of relevant data; and (iii) program restriction [20, 11] forbids aliasing between references to relevant values.

Unfortunately, all of these strategies have significant drawbacks. For software model checking, the main drawback to global static alias analysis is imprecision. If one has decided to pay the cost of running a model checker, it seems counterproductive to rely on a far less precise static analysis prior to model construction. Imprecision in the alias information can seriously decrease model precision, and in some cases, it can incur performance overhead. On the other hand, slicing may be too aggressive, because full slices can be quite large, which in turn causes the state space of the model to grow uncontrollably. Finally, the drawback to program restriction is that it decreases the application range of the analysis; in particular, it precludes analysis of legacy systems.

In this paper, we propose an alternative solution that uses a region type system [29] to address the problem of model abstraction in the presence of references. Static approximations to memory regions are introduced prior to model generation, but we defer region computation, including populating regions with values, to model checking time. This approach leads to a highly flow-sensitive, context-sensitive, and path-sensitive analysis, and it allows us to abstract deeply nested references in a principled way. The main contributions of this paper are as follows:

- We show that a region type system, originally conceived for region-based memory management, naturally supports program model abstraction by considering regions as sets of values with a nondeterministic choice operator. We prove region-based abstraction sound for checking safety properties.
- We show that our abstraction supports a number of extensions that allow for flexible tuning of the precision/performance tradeoff.
- We have implemented a region-based model compiler for C# [21], and we believe this technique to be applicable to other real-world languages such as Java, C, and C++.

The rest of this paper is organized as follows. In Section 2, we present an example that demonstrates the key ideas of the paper. Section 3 presents our model abstraction and discusses some of its properties. Next, in Section 4, we present dynamic semantics for our model and prove that our model abstraction is sound. Then, in Section 5, we discuss a number of extensions to our
model abstraction. Section 6 describes our implementation, and Section 7 discusses related work.

2 Region-based model abstraction

We propose using a region type system to improve model precision and performance in the presence of aliasing. Our work is based on RegJava [6], which is a region type system for a Java-like language; this work is in turn based on the region type system of Tofte and Talpin [29]. Both of these type systems were designed for the purpose of region-based memory management; in this work, we apply the region type system to the problem of software model generation instead.

A region type system associates a region variable with each object. The type system determines the scope and lifetime of region variables, guaranteeing that a region will outlive all objects allocated within it. At run time, region variables are bound to actual region names identifying regions in which objects can be allocated. This binding occurs at letregion statements, where region names are introduced, and at method invocation, where region names are passed as parameters.

For example, consider the following region-annotated class declarations:

```java
class Transaction{ρ} { class State[ρ] at ρ { ...
    ... State(ρ) s; ... relevant bool b;
    ... }
}
```

The Transaction class represents some complex transaction being performed by an application, and the State class represents some portion of the state for this transaction. The application uses the field b to make decisions that are relevant to a safety property that we wish to check; thus, the field has been annotated with the keyword relevant. We assume that these relevant annotations are already present in our source program; such annotations could be obtained from sources such as the programmer, program analyses, or counterexample-driven refinement. Furthermore, we assume that the source program is well-typed with respect to these annotations; for example, a variable of relevant type can only be assigned to another variable of relevant type.

Because the State class contains a relevant value, we must track references to objects of type State. Thus, our system associates a region variable ρ with this class, as indicated by the “at ρ” syntax. However, we do not require that all objects of a particular class be associated with the same region variable. Therefore, the class is parameterized over its region variable(s), as indicated by the square brackets; this parameterized class must be instantiated with actual region variables at every occurrence within the source program. The Transaction class does not contain any relevant fields, so it does not have a region variable of its own; however, it has a region parameter ρ’ because it contains a field s whose type State(ρ’) is instantiated with ρ’. All of these region annotations can be derived automatically from a program annotated with relevant qualifiers.

Now consider the region-annotated program shown in Figure 1. We wish to construct a model of this program that tracks the field b; in particular, we wish to model statements (1), (2), and (3) concretely. That is, because the tests in those statements are performed on expressions (e.g., y.s.b) with relevant type (relevant bool), our model abstraction preserves their values at statements (1), (2), and (3) (and elsewhere). The letregion annotations indicate the scope of region

```java
letregion ρ1 { letregion ρ2 { Transaction(ρ1) x;
      Transaction(ρ2) y;
      if (...) { Transaction(ρ1) t1 = new Transaction(ρ1)();
        x = t1; x.s = new State(ρ1)(); x.s.b = true;
      } else { Transaction(ρ1) t2 = new Transaction(ρ1)();
        x = t2; x.s = new State(ρ1)(); x.s.b = false;
      }
      if (...) { Transaction(ρ2) t3 = new Transaction(ρ2)();
        y = t3; y.s = new State(ρ2)(); y.s.b = true;
      } else { Transaction(ρ2) t4 = new Transaction(ρ2)();
        y = t4; y.s = new State(ρ2)(); y.s.b = false;
      }
      (1) if (x.s.b) ...
      (2) if (y.s.b) ...
      (3) if (x.s.b) ...
    }
  }
}
```

Figure 1: Example region annotated program

variables ρ1 and ρ2. Program variables x and y are assigned distinct instances of the type Transaction[ρ], instantiated on ρ1 for x and on ρ2 for y. However, the region variable associated with the types of t1 and t2 must be identical to the region variable used with the type of x. This identification of region variables is required by the assignments x = t1 and x = t2. Similarly, y, t3 and t4 use region variable ρ2. Once again, all of these region annotations can be inferred.

Our model abstraction technique relies on the region type system’s static semantics, which require potentially aliased objects to be allocated in the same region. For example, the statements x = t1 and x = t2 force all three of these variables to use the same region variable ρ1. Since the region ρ1 contains all objects to which these variables can possibly refer, we can approximate any of these variables by simply choosing an object nondeterministically from ρ1; the model checker will then explore all possible choices. Thus, we can replace uses of these variables with nondeterministic choice operators (over ρ1), and we can eliminate the assignment statements from the model entirely.

In addition to this static component, our abstraction technique also has an essential dynamic component. As
dictated by the region semantics, we allocate objects in regions at run time; since regions are sets of references in our system, we simply add every newly allocated object to the corresponding set. The region type system also requires that region names be passed as parameters to methods. Thus, at model checking time, each method invocation can have a different set of region arguments, allowing the model checker to generate context-sensitive alias information across procedure boundaries.

In essence, we use the static semantics of the region type system to determine the scope of alias sets (i.e., regions), and we use the dynamic semantics of the region type system to populate these sets (i.e., regions) with actual object references. By exploiting this phase distinction, we can rely on the model checker to compute and use flow-sensitive and path-sensitive alias information.

The main principles for model construction are: (i) operations on values of relevant type are modeled concretely; (ii) operations on references to objects containing relevant fields are modeled abstractly via operations on their associated region sets; and (iii) operations on all other references are discarded. Based on the region-annotated program above, we can generate the following sound model:

\[
\begin{align*}
\rho_1 &= \text{new Set}; \\
\rho_2 &= \text{new Set}; \\
\text{if } (...) \{ \\
& \quad \rho_1 = \rho_1 + \{\text{new State}\}; \quad \text{choose}(\rho_1).b = \text{true}; \\
& \quad \text{else} \{ \\
& \quad \quad \rho_1 = \rho_1 + \{\text{new State}\}; \quad \text{choose}(\rho_1).b = \text{false}; \\
& \quad \} \\
\text{if } (...) \{ \\
& \quad \rho_2 = \rho_2 + \{\text{new State}\}; \quad \text{choose}(\rho_2).b = \text{true}; \\
& \quad \text{else} \{ \\
& \quad \quad \rho_2 = \rho_2 + \{\text{new State}\}; \quad \text{choose}(\rho_2).b = \text{false}; \\
& \quad \} \\
(1) \text{if}(\text{choose}(\rho_1).b) ...; \\
(2) \text{if}(\text{choose}(\rho_2).b) ...; \\
(3) \text{if}(\text{choose}(\rho_1).b) ...;
\end{align*}
\]

Here, letregion statements are translated into allocations of new sets of memory references. Thus, letregion \( \rho_1 \) becomes \( \rho_1 = \text{new Set}, \) binding \( \rho_1 \) to a new reference to an empty set. Operations on objects of type \( \text{State}(\rho) \) are represented via operations on region \( \rho. \) For example, the statement \( x.z = \text{new State}() \) becomes

\[ \rho_1 = \rho_1 + \{\text{new State}\} \]

where a new State object is added to the set bound to \( \rho_1. \) All other operations are discarded; for example, all assignments between objects of type Transaction are eliminated from the model. This approximation is sound because the region type system has taken these statements into account via equality constraints on region parameters (e.g., identifying the parameters of the types of \( x, t_1 \) and \( t_2 \)). However, note that a model checker executing this particular model will never have to put more than one object into each set, even if we assume that all tests of the form if (...) are non-deterministic in the model. Hence, at statements (1), (2), and (3), the choice expressions \( \text{choose}(\rho_1) \) can only return one possible object, so there will only be one path to explore through these statements.

Our abstraction allows us to soundly abstract away from non-relevant data. For example, in the case of the type State, we can elide all fields other than the field \( b \) in our model, because only that field has relevant type. Hence, to be more precise, we could write

\[ \rho_1 = \rho_1 + \{\text{new State}\} \]

to indicate that the model type State is an abstraction of the concrete program type.

The scheme for region-based model construction outlined above can be refined in several ways. For example, we can represent any level of slicing we wish by declaring more object references to be relevant. Also, we can relax some of the restrictions of the region type system; instead of using equality constraints between region variables as a model-reduction principle, we can introduce assignments between regions of the form \( \rho = \rho + \rho' \). These topics will be discussed in Section 3 and Section 5.

2.1 Alternatives

We now consider two alternative approaches to this problem in order to demonstrate the advantages of our approach. In particular, we consider the result of using slicing or alias analysis to construct a model of this program.\footnote{Note that slicing and alias analysis are not mutually exclusive; for example, slicing in the presence of aliasing often relies on alias analysis to compute the slices.}

The scenarios we present are simplified for clarity.

Slicing. Program slicing [30, 5] can be used to produce very precise models [35]; however, slicing algorithms can cause large portions of the program to be included in the model. In this example, if we use a typical slicing algorithm to slice with respect to statements (1), (2), and (3), then we would include the whole program fragment in the model. In particular, we must model objects of type Transaction as well as objects of type State, which is far more expensive in terms of program state than a technique that only models objects of type State. Furthermore, this approach results in a large model state space because we include all statements that manipulate these objects (e.g., assignments), thus creating more transitions in the model. In general, if the source program accesses relevant fields through a long chain of object references or if the program uses long chains of control dependencies (which we have ignored in the example), then constructing a model using program slicing can lead to uncontrollably large states and state spaces.

Alias analysis. Alias analysis [2, 9, 27] is a powerful technique for reducing program size; however, static alias analysis can also lead to an undesirable loss of model precision. In our example, a typical alias analysis would determine that \( x \) could be aliased to objects \( t_1 \) or \( t_2 \) and that \( y \) could be aliased to objects \( t_3 \) or \( t_4 \). Using the alias approximation, we can eliminate large parts of the program; for example, the assignment statements
Figure 2: Modified RegJava expression and statement syntax.

Involving $x$ and $y$ need not appear in the model, since the static alias information accounts for their effects.

However, the price for the alias approximation is that we must now consider all possible alias assignments to $x$ and $y$. A natural way to do this would be to nondeterministically choose a value from the alias set when dereferencing $x$ or $y$. We could then model statements (1), (2), and (3) with the following code, assuming we represent the objects allocated into $t_1, t_2, t_3,$ and $t_4$ by those names:

(1) if (choose([$t_1.b, t_2.b$])) ...;
(2) if (choose([$t_3.b, t_4.b$])) ...;
(3) if (choose([$t_1.b, t_2.b$])) ...;

Assuming that the allocation statements are guarded by a conditional in our model, then one of the objects in each set must be undefined on a given path. Thus, each set will consist of one true or false value and one undefined value (which could be true or false), so the choose() operator will select from the set {true, false}. Because these nondeterministic choices are performed in sequence, we must explore eight possible paths through the program; in general, repeated nondeterministic choices can incur an exponential blow-up in the state space of the model. Furthermore, alias analysis cannot easily preserve the correlation between the choices made at statements (1) and (3). Thus, the alias approximation can lead to model imprecision as well as performance degradation.

3 Abstraction

In this section we formalize our abstraction technique. First, we give an overview of RegJava and its type system [6]. Then, we present and discuss our abstraction function.

3.1 RegJava

RegJava is a Java-like object-oriented language extended with a sound region type system. Figure 2 shows the relevant expression and statement syntax for RegJava. Expressions consist of variable access and object field access; we have omitted expressions such as constants and comparisons. Statements consist of field update, variable update, object creation, method invocation, method return, and region binding; we have omitted statements such as conditionals and sequencing.

Some of RegJava’s syntactic constructs feature a vector of region variables $\bar{\rho}$. For object creation, this vector contains the region variables with which the class $A$ is being instantiated. For method invocation, this vector contains the region variables that will be passed as arguments to the method.

Note that we have made a slight modification to RegJava’s syntax: we require that expressions have no side effects. Thus, operations such as object creation and method invocation are now considered to be statements instead of expressions. This change has no impact on the soundness of the RegJava type system.

RegJava base types include int and bool, as expected. Object types in RegJava have the following form:

$$[x_i : \tau_i, \phi_i | \bar{\rho} = [\phi_i | \bar{\rho}]]$$

The $x_i$ are the fields of the object, and the $m_i$ are the methods of the object. The $\tau_i$ and the $\phi_i$ are the types of the fields and objects, respectively. $\rho$ is a region variable, and each object has a region in which it can be found. In our system, the region variable $\rho$ can be omitted on object types that do not contain relevant fields (i.e., objects that will not be modeled directly).

The program type $\Pi$ is a mapping from class names to region-parameterized class types. These class types can be instantiated with a tuple of region variables $\bar{\rho}$ to yield an object type as described above. We write $\Pi(A, \bar{\rho})$ to indicate the object type of the class named $A$ when instantiated on region variables $\bar{\rho}$.

3.2 Abstraction function

We now extend RegJava by adding a nondeterministic choice expression, which takes two syntactic forms. The first form is written choose($\rho, \tau$); in this form, the choice operator nondeterministically chooses an object in region $\rho$ whose type is a subtype of (or equal to) $\tau$. The second form of this operator is written choose($\tau$); this operator chooses an arbitrary value of the type $\tau$. We will refer to this extended language as RegJava+Choice.

Now, we will give a basic model abstraction function that translates RegJava into RegJava+Choice. This abstraction is intended to preserve the behavior of the

\footnote{We could have chosen alternative ways of modeling the references to $x$ and $y$, of course, but the nondeterministic interpretation is natural and useful for model checking [15], and less expensive schemes would typically lead to other kinds of imprecision.}

\footnote{This latter form serves the same purpose as the value $\top$ in other modeling languages.}
source program on relevant values, possibly by overapproximating the original program.

Our abstraction maps well-typed RegJava expressions \( e \) and statements \( s \) into corresponding RegJava+Choice expressions \([e] \) and statements \([s] \). Since the translation depends on the typing of the RegJava program, we decorate the source program with explicit types (e.g., \( x^e \) and \( e^e \)). The translation distinguishes between three kinds of types: (i) Relevant types are annotated as such in the original program. (ii) References to relevant types are object types that contain one or more relevant fields. The predicate \( \text{RR}(\tau, \rho) \) used in some of the rules means that \( \tau \) is a reference type with relevant fields; that is, \( \text{RR}(\tau, \rho) \) is true if and only if \( \tau = \ldots x : \text{relevant} \tau' \ldots [0] \rho \), for some field \( x \) and some type \( \tau' \). (iii) Irrelevant types are all remaining types. Notice that any data object (i.e., not just fields but also locals and parameters) can be declared as relevant by qualifying their types as such.

Figure 3 gives the definition of our abstraction function for a representative subset of RegJava’s expressions and statements. Rules (1) and (2) give the translation for expressions. The abstraction treats values of relevant type concretely, using the RegJava operations performed on the values in the source program. It treats references to relevant values abstractly, by choosing nondeterministically from the regions where the references are located (i.e., \( \text{choose}(\rho, \tau) \), where \( \rho \) is the region variable in the value’s reference type \([\ldots] \rho \)). Finally, irrelevant values are modeled by the nondeterministic construct \( \text{choose}(\tau) \). Note that because expressions with irrelevant type are only used within the translation of conditional statements (not shown), this case will always yield \( \text{choose} \text{(boolean)} \); thus, in practice, we will never need to choose a value from an infinite type. Finally, note that because expressions are free of side effects, we can omit the recursive translation \( e^e \) in case (2) when \( \tau \) is not relevant.

Since we assume that expressions have no side effects, we must give rules (3) and (4) to show the translation of assignment statements. Assignments that do not involve relevant values are skipped, using the instruction skip.

Rule (5) gives the translation for object creation. If the object is relevant, we create and assign it. If the type is a reference to a relevant type, then we simply create the object and store it in the region associated with the object’s region type. We write \( \text{RR}(\tau, \rho) \) if and only if \( \tau = \ldots x : \text{relevant} \tau' \ldots [0] \rho \) for some field \( x \) and type \( \tau' \) (so, \( \rho \) is the region of a reference to an object that contains at least one relevant field). If the expression \( \text{new } A(\rho) \) has type \( \tau \) with \( \text{RR}(\tau, \rho) \), then that expression creates a new object and stores its reference in region \( \rho \). We do not create objects of irrelevant type.

Rules (6) and (7) show method invocation and return. The two cases for these rules indicate that we only store the return value of the method if the return value is relevant. Also, we omit the method’s object and its argument if their types are not relevant; for the sake of clarity, we have described this part of the translation informally. Note that if the object type, argument type, or return type is a reference to a relevant type, the corresponding region variables will be passed to the method as part of \( \rho \). If the type \( \tau_0 \) of the receiver object is not relevant, we translate

\[ [e^e], m(\rho)([\tau_1]) = \text{choose_type}(\tau_0, m(\rho)([\tau_1])) \]

where \( \text{choose_type}(\tau_0) \) nondeterministically chooses a subtype of \( \tau_0 \), thereby soundly overapproximating the dynamic dispatch. This scheme can be improved, as discussed in Section 5.3.

Finally, rule (8) shows that letregion statements are translated into corresponding letregion statements, indicating where regions (sets) should be created and destroyed.

Note that the RegJava+Choice program still contains new operators and letregion statements. In our models, and in our examples, we translate these constructs into more primitive operations on sets. Thus, \( \text{new } A(\rho) \) becomes

\[ \rho = \rho + \{ \text{new } A \} \]

where \( \text{new } A(\rho) \) \([\ldots] \rho \). In addition, letregion \( \rho \) in \( s \) becomes

\[ \rho = \rho \cup \{ s \} \]

\[ \rho = \text{null} \]

3.3 Discussion

In this section, we consider a number of examples that demonstrate how and why our abstraction works with respect to data, procedures, and control. We also discuss our handling of inheritance, and we consider the issue of generating a finite model.

Data abstraction. We first illustrate important properties of the data representation in region-based abstraction. Recall the classes \text{Transaction}[\rho] \] and \text{State}[\rho] \] from the example in Section 2. According to our classification in section 3.2, the \text{Transaction} class is an irrelevant type, the \text{State} class is the type of a reference to a relevant type, and the field \( b \) inside the \text{State} class has a relevant type.

A key idea is to use region variable parameters on class types to eliminate operations on irrelevant references. Even though \text{Transaction}[\rho] \] is an irrelevant type, it still carries the region parameter \( \rho \), because the \text{State} class is instantiated at \( \rho \) inside it. Thus, regions associated with references to relevant fields will transitively accrue up through the class containment hierarchy. The type system uses these parameters to take into account the effects that computations on irrelevant references may have on references to relevant values. As a result, we can remove statements affecting irrelevant types entirely. For instance, the assignments \( x = t_1 \) and \( x = t_2 \) in our example have the effect of identifying the region parameters on the \text{Transaction} types of \( t_1 \) and \( t_2 \). Therefore, the \text{State}[\rho] \] objects within them will be allocated into the same region (set) in the abstracted model, and thus these assignment statements can be removed from the model.

To see how the abstraction function achieves this goal, consider the model generated for the expression \( x.a.b \),
where \( x \) is of type \( \text{Transaction}(\rho) \) and field \( s \) is of type \( \text{State}(\rho) \):

\[
[x, s, b] \quad =_{(2.1)} \quad \text{choose}(\rho, \text{State}).b
\]

\[
[x, s].b \quad =_{(2.2)} \quad \text{choose}(\rho, \text{State}).b
\]

Referring to the defining equations for abstraction in Figure 3, the in equality labeled (2.1) above uses the first case of defining equation (2) in Figure 3, because the type of \( b \) is relevant; the second equality, labeled (2.2), uses the second case of the same defining equation, because \( x, s \) has the type of a reference to a relevant value. Note that the variable \( x \), which has an irrelevant type, does not appear in the resulting model.

Now consider the following example:

\[
[x = t1] =_{(4.2)} \text{skip}
\]

This example uses the second case of equation (4) in Figure 3, because the type of \( x \) is not relevant.

Despite the fact that we have eliminated these irrelevant statements and variables from the model, their effects will be taken into account by the expression \( \text{choose}(\rho, \text{State}).b \). Nondeterminism causes a model checker to explore all possible choices; therefore, all possible effects on references to relevant values will be explored, even though the effects happened through irrelevant references.

The precision of a model naturally depends on which values are declared as relevant; moreover, distinct values of the same type can be declared differently. For example, suppose we have the following classes:

\[
\text{class } C[\rho] \text{ at } \rho \{ \text{relevant } \tau : x : \}
\]
\[
\text{class } D[\rho_1, \rho_2] \text{ at } \rho_2 \{ C(\rho_1) : c : \}
\]
\[
\text{class } E[\rho_3, \rho_4] \text{ at } \rho_4 \{ \text{relevant } C(\rho_3) : c : \}
\]

Class \( D \) contains a reference to class \( C \), which contains a relevant type. However, class \( E \) contains a reference to class \( C \) that is itself relevant. So, if we have a variable \( d \) of type \( D(\rho_1, \rho_2) \) and a variable \( e \) of type \( E(\rho_3, \rho_4) \), then we have \([d, c].x = \text{choose}(\rho_1, C(\rho_1)).x\) and \([e, c].x = \text{choose}(\rho_4, E(\rho_3, \rho_4)).x\). The latter is potentially more precise than the former.

**Procedural abstraction.** At compile time, region
type systems do not produce interprocedurally sound alias information on region variables. However, the dynamic component of the region type system makes this alias information sound. In essence, soundness is achieved by making methods polymorphic with respect to region variables; thus, each call site for a method can instantiate that method on a different set of region variables.

For example, consider the method `foo`:

```c
void foo(\rho_1, \rho_2)(\text{State}(\rho_1) x, \text{State}(\rho_2) y) {
    x.b = \text{true}; y.b = \text{false};
}
```

The region parameters \(\rho_1\) and \(\rho_2\) are distinct, but they might become bound to identical regions at a call site such as `foo(\rho_1, \rho_1)(s, s)`. Thus, these region variables do not provide interprocedurally sound alias information in and of themselves, because there is no indication that \(\rho_1\) might be the same region as \(\rho_2\). At run time, however, the appropriate region names will be passed as parameters to this function. Thus, the model of `foo` is a function of regions (sets):

```c
void foo(Set(\rho_1), Set(\rho_2)) {
    choose(\rho_1).b = \text{true}; choose(\rho_2).b = \text{false};
}
```

If the \(x\) and \(y\) variables in the original program were aliased at a particular call site, then \(\rho_1\) and \(\rho_2\) will be bound to same region for the corresponding call site in the model; thus, our alias information will be sound.

We could avoid passing region variables to methods at run time if we performed a nondeterministic choice on each region parameter prior to invoking each method. However, this approach results in far too many premature choices and effectively isolates the region abstraction within each procedure.

**Control abstraction.** Our control flow abstraction is implicit in our translation of expressions. Control flow is governed by the boolean expressions that appear within if and while statements, and these expressions must have type `relevant bool` or simply `bool`. In the former case, the result will be a concrete value, so the model checker will deterministically choose one path through the conditional statement. In the latter case, our abstraction function will translate the expression to `choose(bool)`, this nondeterministic choice will cause the model checker to explore both paths.

**Inheritance.** RegJava's subtyping rule demands that if \(\tau\) is a subtype of \(\tau'\), then both \(\tau\) and \(\tau'\) must have the same region parameter \(\rho\). Thus, a single region can contain objects of a number of different types. In order to account for this possibility, our `choose(\rho, \tau)` operator selects only those objects in the region that are compatible with the static type of the original expression, \(\tau\). This approach ensures that we do not violate the soundness of the RegJava type system by returning an incompatible type from our choice operator. Otherwise, RegJava's inheritance scheme can be used in RegJava+Choice without further modification.

**Finiteness.** Our abstraction does not necessarily yield a finite model. However, the issue of generating a finite model is orthogonal to our abstraction technique, because other abstraction functions intended to generate a finite model can be composed with our set abstraction. For example, if we have a set \(\rho\) containing references to objects containing a relevant integer, and we have an abstraction function \(\ast\) that maps objects containing integers to the objects containing the values odd or even, then we can simply apply this abstraction to \(\rho\) pointwise, yielding \(\rho' = \{ x \ast x : x \in \rho\}\).

## 4 Semantics and soundness

### 4.1 Operational semantics

In order to explain the meaning of RegJava+Choice programs, it is necessary to briefly consider the RegJava semantics [5], which are defined in terms of the following reduction relation:

\[
\sigma, V, R \vdash p \rightarrow v, \sigma', V'
\]

In this relation, \(p\) is a program phrase, either an expression or a statement, that evaluates to a value \(v\). Values are either constants (such as boolean truth values or numbers), or addresses, or one of the special values `error` or `normal`. Addresses are pairs \((r, o)\), where \(r\) is a region name and \(o\) is an offset. Stores \(\sigma\) map region names to regions, which are finite maps from offsets to values; thus, \(\sigma(r)\) is a region and \(\sigma(r)[o]\) is a value in the region named \(r\) at offset \(o\). We write \((r, o) \in \text{Dom}(\sigma)\) as shorthand for \(r \in \text{Dom}(\sigma)\) and \(o \in \text{Dom}(\sigma(r))\). Value environments \(V\) are finite maps from program variables, parameters and field names to values; for example, \(V(x)\) denotes a value bound to \(x\). Such bindings are introduced by executing local declarations, method calls, or field updates. Region environments \(R\) map region variables to region names, so \(R(\rho)\) denotes a region name \(r\) bound to \(\rho\). These bindings are introduced by `letregion` statements and method calls.

Objects in the store are modeled by triples of the form \(<A, V, r'>\>, where \(A\) is a class name, \(V\) binds field names to values, and \(r'\) is region names bound to the formal region parameters of class \(A\).

It is an important property of region semantics that static region variables are mapped to actual regions dynamically. Since our model abstraction exploits this property in an essential way, we briefly explain it. The mapping is created when executing the statement `letregion` \(\rho\) in \(s\) with the following semantic rule:

\[
\frac{\Gamma \vdash r \notin \text{Dom}(\sigma) \quad \sigma \vdash (r \rightarrow \emptyset) \quad V, R + \{ \rho \rightarrow r \} \vdash s \rightarrow v, \sigma_1, V_1}{\sigma, V, R \vdash \text{letregion} \ \rho \quad s \rightarrow v, \sigma_1[\{ r \} \setminus V]}.
\]

The statement \(s\) is executed in an environment in which \(\rho\) is mapped to a fresh region name \(r\). In the rule above, this mapping is achieved by extending the region environment \(R\) with the binding \(\rho \rightarrow r\). Moreover, the current store \(\sigma\) is extended to \(\sigma + \{ r \rightarrow \emptyset \}\), where \(\emptyset\) is the empty region (i.e., the empty map from offsets to values). After executing \(s\), the region \(r\) is deallocated.
We define the semantics of RegJava+Choice by a similar relation:
\[ \sigma, V, R \vdash \rho \mapsto v, \sigma', V' \]
This relation is defined by taking all the rules of RegJava (with \( \rightarrow \) replaced by \( \mapsto \)) and adding the rules:

\begin{align*}
(\text{Choose}) & \quad R(\rho) = \tau \quad (r, o) \in \text{Dom}(\sigma) \\
& \quad \sigma(r)(o) = < A, V, \bar{r} > \\
& \quad \exists \bar{\rho}, \Pi(A, \bar{\rho}) < \tau \\
& \quad \sigma, V, R \vdash \text{choose}(\rho, \tau) \mapsto (r, o), \sigma, V
\end{align*}

\begin{align*}
(\text{Choose Type}) & \quad \exists \bar{\rho}, \Pi(A, \bar{\rho}) < \tau \\
& \quad \sigma, V, R \vdash A.m(e) \mapsto v, \bar{\sigma}, V' \\
& \quad \sigma, V, R \vdash \text{choose_type}(\tau).m(e) \mapsto v, \bar{\sigma}, V'
\end{align*}

\begin{align*}
(\text{Choose True}) & \quad \sigma, V, R \vdash \text{choose(boolean)} \mapsto \text{true}, \sigma, V \\
(\text{Choose False}) & \quad \sigma, V, R \vdash \text{choose(boolean)} \mapsto \text{false}, \sigma, V \\
(\text{Skip}) & \quad \sigma, V, R \vdash \text{skip} \mapsto \text{null}, \sigma, V
\end{align*}

Note that the operator \( \text{choose}(\rho, \tau) \) is nondeterministic by choosing over addresses \((r, o)\) in rule \((\text{Choose})\), and the rule \((\text{Choose Type})\) nondeterministically chooses a type \( A \) from the subtypes of \( \tau \).

4.2 Data abstraction
We do not require our abstraction to preserve variables and fields that are not relevant. In order to make this notion precise, it is necessary to define abstractions on some semantic objects of RegJava.

Abstract variable environments only need to preserve relevant values. For a typed variable environment \( V \) we define \( V^# \) by
\[ V^# = \{ (x^\tau, v) | V(x^\tau) = v \text{ and } \tau \text{ is relevant} \} \]
We say that variable environments \( V_1 \) and \( V_2 \) are equivalent, written \( V_1 \approx V_2 \), if and only if \( \text{Dom}(V_1^#) = \text{Dom}(V_2^#) \) and \( V_1^#(x^\tau) = V_2^#(x^\tau) \) for all \( x^\tau \in \text{Dom}(V_1^#) \).

Abstract objects may leave out non-relevant fields. Hence, we define equivalence on objects, denoted \( \approx_o \), by defining \( < A_1, V_1, \bar{r}_1 > \approx_o < A_2, V_2, \bar{r}_2 > \) if and only if \( A_1 = A_2, \bar{r}_1 = \bar{r}_2 \) and \( V_1 \approx_v V_2 \).

Abstract stores preserve only relevant references and references containing relevant fields. For a store \( \sigma \), we define the abstract store \( \sigma^# \) by
\[ \sigma^# = \{ (a, < A, V^#(a), \bar{r}^a > ) | \sigma(a) = < A, V, \bar{r} > \text{ and } A \text{ is relevant or } A \text{ has a relevant field} \} \]
We say that store \( \sigma_1 \) and \( \sigma_2 \) are equivalent, written \( \sigma_1 \approx_v \sigma_2 \), if and only if \( \text{Dom}(\sigma_1^#) = \text{Dom}(\sigma_2^#) \), and \( \sigma_1^#(a) \approx_v \sigma_2^#(a) \) for all \( a \in \text{Dom}(\sigma_1^#) \).

4.3 Soundness
Now, we are prepared to state our soundness lemmas and theorem. Essentially, we argue that any execution trace from the original RegJava program will be considered by the nondeterministic execution of the RegJava+Choice program. The fundamental observation is that it is always sound to replace an object reference with a non-deterministic choice over its region, since one possible result of the choice must be the object itself. Since we do not require the RegJava+Choice program to preserve variables and fields that are not relevant, we require soundness properties to be formulated out of relevant variables and fields.

Our soundness result uses soundness of RegJava [6]. As is usually the case in region-based type systems, RegJava soundness relies on assumptions about certain consistency relations between the static typing of a program and its dynamic execution environment. Hence, when considering, say, a well typed statement \( s \) with typing \( \Pi, \Delta, E \vdash s : \tau \) and an execution \( \sigma, V, R \vdash s \rightarrow v, \bar{\sigma}', V' \), we are only interested in cases where the dynamic environment given by the data \( \sigma, V, R \) respect the static environment given by the data \( \Pi, \Delta, E \). We say that a typing \( \Pi, \Delta, E \vdash s : \tau \) is a well typing \( g \) with respect to dynamic environment \( \sigma, V, R \) if an appropriate set of consistency relations hold. A similar notion is defined for expression typings. The precise definition can be found in Section A.3.1. Proofs of the following lemmas and theorem are placed in Appendix A.

We first state two lemmas about preservation of values of expressions. In both lemmas, we use the assumption that expressions \( e \) are free of side effects, so evaluating expressions will not change the store or value environment.

The following lemma states that references to relevant data are preserved when evaluating abstract expressions in an abstract store and an abstract value environment, provided the abstract store and environment are equivalent to the concrete ones:

**Lemma 1** Assume that \( \Pi, \Delta, E \vdash e : \tau \) is a well typing with respect to \( \sigma, V, R \) such that
\[ \begin{align*}
1. \quad & \sigma, V, R \vdash e \rightarrow (r, o), \sigma, V \\
2. \quad & RR(\tau, \rho) \\
3. \quad & \sigma \approx_v \bar{\sigma} \text{ and } V \approx_v \bar{V}
\end{align*} \]
Then \( \bar{\sigma}, \bar{V}, R \vdash e^# \rightarrow (r, o), \bar{\sigma}, \bar{V} \).

The following lemma states that relevant values are preserved by equivalent abstract stores and value environments:

**Lemma 2** Assume that \( \Pi, \Delta, E \vdash e : \tau \) is a well typing with respect to \( \sigma, V, R \) such that
1. $\sigma, V, R \vdash e^* \rightarrow v, \sigma, V$

2. $\tau$ is relevant

3. $\sigma \approx_\tau \hat{\sigma}$ and $V \approx_\tau \hat{V}$

Then $\hat{\sigma}, \hat{V}, R \vdash [e^*] \rightarrow v, \hat{\sigma}, \hat{V}$.

Our soundness theorem states that relevant values are preserved by our abstraction under evaluation in an abstract store and an abstract environment; provided the abstract store and environment are equivalent to the original store and environment. Moreover, the equivalences are preserved in the stores and environments after evaluation. The last property is an inductive strengthening needed to prove soundness for sequential composition.

THEOREM 1 (Soundness) Let $p$ be a well-typed expression or statement in RegJava. Assume $\Pi, \Delta, E : p : \tau$ is a well typing with respect to $\sigma_0, V_0, R$ such that

(i) $\sigma_0, V_0, R \vdash p \rightarrow v_1^*, \sigma_1, V_1$

(ii) $\sigma_0 \approx_\tau \hat{\sigma_0}$ and $V_0 \approx_\tau \hat{V_0}$

Then there exist $\hat{\sigma_1}, \hat{V_1}$ such that

1. $\sigma_0, \hat{V_0}, R \vdash [p] \rightarrow v_1^*, \hat{\sigma_1}, \hat{V_1}$

2. if $\tau$ is relevant, then $v_1^* = \hat{v_1}^*$

3. $\sigma_1 \approx_\tau \hat{\sigma_1}$

4. $V_1 \approx_\tau \hat{V_1}$

A proof of Theorem 1 can be found in Appendix A. The theorem implies that our abstraction is sound for checking safety properties on relevant data.

5 Extensions

A key advantage of our region-based model abstraction technique is that it offers a high degree of flexibility when considering the tradeoff between precision and performance. By making slight adjustments to the translation scheme and to the region-based type system, we can offer the programmer a number of options for tuning the model generation process. In this section, we discuss two such extensions to our technique. Finally, we discuss an extension for optimizing our abstraction with respect to dynamic dispatch.

5.1 Optimizing choose()

Our first extension involves the placement of choose() expressions. The translation given in Section 3 offers a straightforward approach to the placement of these expressions, but in some cases, we can make significant improvements. Consider the following code:

```java
class State[\rho] at \rho {
    relevant int a;
    relevant int b;
}

... State(\rho). s = ...;

s.a = 0;

s.b = 1;
```

Using our original model generation technique, the resulting model will be:

```java
choose(\rho). a = 0;
choose(\rho). b = 1;
```

Note that this model contains two distinct nondeterministic choice operators. If the region $\rho$ contains more than one object, then these operators may return different objects. In this case, the assignment statements will apply to fields of different objects, which cannot take place in the original program; in essence, the model does not capture the correlation between these two statements. A more precise model would be:

```java
State(\rho). s = choose(\rho);

s.a = 0;

s.b = 1;
```

By introducing a local variable, we have preserved some information about the correlation between these two assignment statements; that is, the model captures the fact that both assignments affect fields of the same object. Note that this approach also yields a more efficient model, because there are fewer paths to explore.

To implement this optimization, we can alter the translation rules so that they model concretely any local variable that contains a reference to a relevant type. For each of these local variables, the new translation will ensure that the nondeterministic choice takes place only once per definition of the variable. We omit the details of this translation in this paper.

Unfortunately, picking an appropriate place to introduce the choose() expression can be tricky. As long as a choose() expression is placed between each definition of the variable and its first subsequent use, the model will be sound. However, there can still be several options for where to place the choose expression. In general, delaying a choice operation will reduce the size of the program that must be explored after each possible choice, but a delayed choice can also increase the number of possible choices to explore.

It is quite possible to generalize this technique beyond local variables; indeed, there are a number of possible interprocedural optimizations. We plan to explore these possibilities in future work.

5.2 Relaxing region identification

Our second extension offers a more dynamic approach to region-based model abstraction. In our original model generation scheme, we removed assignments between
references of abstract type, counting on the identification of their region variables to account for the possible aliasing created by the assignment statement. However, we can avoid identifying these region variables if we introduce additional statements to the model. Consider the following region-annotated source program:

```c
void foo[ρ][State(ρ) s1, State(ρ) s2] {
  State(ρ) tmp;
  if (…) { tmp = s1; }
  else { tmp = s2; }
  tmp.a = 0;
}
```

In this example, assignments between the three variables s1, s2, and tmp require that their types (and thus their region variables) be equal. The resulting model would be:

```c
void foo(Set ρ) {
  choose(ρ).a = 0;
}
```

This model is relatively simple, but the price for this simplicity is imprecision. All objects to which s1 and s2 could point will now be placed in a single region; furthermore, this equality constraint will propagate to the arguments at all call sites of the function foo. Thus, regions will become larger in size and fewer in number, resulting in a less precise and less efficient model.

An alternative to this approach is to relax our notion of type equality such that we consider two types to be compatible even if their region variables differ. The new region-annotated program is:

```c
void foo[ρ1, ρ2][State(ρ1) s1, State(ρ2) s2] {
  letregion ρm = new Set;
  if (…) { ρm = ρm + ρ1; }
  else { ρm = ρm + ρ2; }
  choose(ρm).a = 0;
}
```

To preserve soundness, though, we must update the regions at model checking time in order to account for the results of the assignment statements. In this example, the resulting model would be:

```c
void foo(Set ρ1, Set ρ2) {
  Set ρm = new Set;
  if (…) { ρm = ρm + ρ1; }
  else { ρm = ρm + ρ2; }
  choose(ρm).a = 0;
}
```

In this revised model, we avoid identifying these three region variables. Instead, we create a new region ρm corresponding to the tmp variable, and we copy the contents of ρ1 or ρ2 at model checking time. As a result, there are fewer choices to explore at the nondeterministic `choose()` expression, and there is more granularity in the region abstraction elsewhere in the program.

Note that this technique is entirely local. For each assignment statement in the program, we can choose independently whether to introduce a region variable equality constraint or whether to introduce a region update statement into the model. An alternative approach would be to mark additional fields as relevant; however, the type system would then require that this annotation spread to other variables elsewhere in the program. Thus, this extension provides the programmer with a high degree of flexibility in tuning the resulting model.

In essence, this extension allows references within a region to escape the scope of the region. When using region-based type systems for memory management, this approach is unsound; we must insist that each object be located in exactly one region. However, when we interpret regions as sets for the purpose of model abstraction, it is sound to allow objects to belong to more than one region.

### 5.3 Optimizing dynamic dispatch

In Section 3.2 we have mentioned the translation of method invocation that approximates dynamic dispatch by

\[ [e_0^\rho]m(\tilde{ρ})([e_1^\rho]) = \begin{cases} [e_0^\rho]m(\tilde{ρ})([e_1^\rho]) & \text{if } \tau_0 \text{ is relevant} \\ \text{choose(ρ, } \tau_0).m(\tilde{ρ})([e_1]) & \text{otherwise} \end{cases} \]

for the case where \( \tau_0 \) is not relevant. One possible optimization is to use a receiver class analysis to reduce the number of types that must be considered. However, we can optimize this translation further by taking advantage of our region-based abstraction technique. Consider this new rule for the translation of an invocation \( e_0^\rho.m(\tilde{ρ})(e_1^\rho) \):

\[ [e_0^\rho].m(\tilde{ρ})([e_1]) = \begin{cases} [e_0^\rho].m(\tilde{ρ})([e_1]) & \text{if } \tau_0 \text{ is relevant} \\ \text{choose(ρ, } \tau_0).m(\tilde{ρ})([e_1]) & \text{otherwise} \end{cases} \]

Note that we use the operator `choose(ρ, τ)` in the case of an irrelevant type; that is, we have decided to use our region-based abstraction for irrelevant object types as well as for references to relevant types. This approach allows us to generate and use flow-sensitive, path-sensitive, and context-sensitive receiver class information. We can choose to use this technique on a per-type basis.

This optimization turns out to be a special case of our abstraction technique in which regions are used to track type information at run time. Because irrelevant object types do not contain any relevant fields, the only property that can distinguish one object from another is its dynamic type. Thus, if a region contains objects of irrelevant type, then the only information that region tracks is type information. These regions are efficient from a model-checking perspective because the number of elements in a region is limited by the number of possible object types.
Finally, note that not all method calls need to be represented in a model. Only calls to methods that have effects on relevant data or regions with references to relevant data need to be represented. For example, a method which only reads from such data or regions does not need to be called. Static effects, such as read(p) or write(p), are a standard component in region-based type and effect systems [29, 1] and can be readily integrated into our system to help guide model extraction for method calls.

6 Implementation

We have implemented a model compiler for a core subset of C# based on the region abstraction described in this paper. Given a C# program with relevant annotations, our compiler produces a model written in ZING, the input language of a software model checker developed at Microsoft Research [3]. The ZING language supports dynamic creation of sets with nondeterministic choice and references to structures, and it has built-in functions and exceptions. Moreover, ZING supports parallel processes and message passing.

6.1 Region-based model compiler

Our model compiler is implemented as an extension to Microsoft’s research compiler infrastructure for C# [14]. We extend the C# type checker to accept the type qualifier relevant such that if r is a valid C# type, then relevant r is also a valid type. After type checking, we insert several phases. The first phase decorates the C# type structure with fresh region variables and parameterizes class types with region parameters. We only insert regions for variables that are references to relevant types. Class parameterization requires a fixpoint computation over the class type structure, because recursive C# types may introduce mutual dependencies between region parameters among several types. The second phase performs region inference. We implement polymorphic type inference by collecting and solving equality constraints and instantiation constraints [12]. The third phase inserts let region statements. Finally, the fourth phase generates ZING models based on the region-annotated program.

6.2 Example application

We have applied our technique to an example application that illustrates the use of our model compiler in a case where aggressive model reduction is both necessary and possible. Our example C# program is a client of three web services, which it uses to make a travel arrangement involving reservations for a flight, a hotel, and a car rental. Because the services are asynchronous, the system is concurrent. We wish to check that the client does not deadlock and that it does not violate temporal constraints on the order of service object method invocations. Due to concurrency, interleaving can create an exponential explosion in the state space of the model; therefore, it is essential to reduce the model’s state space. Model reduction is possible, in turn, because the property we wish to check only involves the communication behavior of the client. Below we discuss the region-annotated version of the source from which the model was constructed; this region-annotated source is an intermediate form generated and consumed by our compiler.

The client is implemented by an object called ArrangeTravel, and it contains three kinds of structures that are relevant for its communication behavior:

```java
public class ArrangeTravel[p2, p1, p0] at p2 {
    ... 
    private relevant FlyMeEx_airline;
    private relevant RentMeEx_carRental;
    private relevant StayMeEx_hotel;
    AgentState(p1) SessionState;
    Itinerary(p0) SessionItinerary;
}
```

The relevant fields hold proxy objects for the services. The region variables p0, p1, and p2 are inserted at references to objects containing relevant fields. The client uses an AgentState class to keep track of the state of the transaction it is performing with the services. This class contains many irrelevant fields that are elided in the model and a few boolean fields holding transaction state:

```java
class AgentState[p1] at p1 {
    ... 
    public relevant bool PlanesQuired = false;
    public relevant bool CarRquired = false;
    public relevant bool HotelQuired = false;
}
```

Another class, itinerary, is used to record which services need to be invoked. It contains several irrelevant fields, but two of these fields are used by the client to make decisions about which services should be invoked for a particular transaction:

```java
public class itinerary[p0] at p0 {
    ... 
    public relevant bool NeedCar = false;
    public relevant bool NeedHotel = false;
}
```

Finally, the ArrangeTravel class contains several methods that use these structures and pass them around to other methods. For example:

```java
public bool Cancel[p31, p32, p33]() {
    itinerary[p31] itin = SessionItinerary;
    AgentState[p32] state = Session_state;
    return cancelReserve[p31, p32, p33](itin, state);
}
```

Here, p31, p32, and p33 represent the this-parameter, where p31 is the region in which the this reference is allocated and p32 and p33 are region parameters on relevant fields inside the this-object. Notice that the reference to this has a region variable (p31) because it contains relevant fields (the proxy objects). Notice also that in the model we generate, the variables itinerary and state will be elided; instead, regions will be passed at the call to cancelReserve.
When checking the application, we found a constraint violation in the `cancelReservation` method. The error occurs in a situation where more than one of the services return an exception message. The method's error handling is quite complicated but was correctly modeled in the generated model. The problem was caused by an incorrect combination of tests on some of the transaction state structures.

Unfortunately, our example is not complex enough to demonstrate the full capabilities of our techniques with respect to alias computation. However, we believe they will be essential as we begin to apply our abstraction tool to complex C# programs.

7 Related work

Our work builds on region type systems for region-based memory management [29], but we reinterpret regions as sets with nondeterministic choice for the purpose of generating program models. To the best of our knowledge, this application of region type systems has not been proposed before. Our implementation of region-based model abstraction for C# relies on the JregJava [6] region type system.

Our work has been inspired by Nielsen and Nielsen’s use of region-based type and effect systems in concurrent behavior analysis [1, 22]. However, they use region types in a different manner than we do; they generate type constraints instead of generating models for a model checker. Their analysis is not path sensitive and does not exploit dynamic operations on regions as our technique does.

Slicing [19, 30] is often used in software model construction. A distinguishing property of our technique is that it computes abstract slices that can be limited to include only operations that happen at any bounded level of indirection to relevant data; in addition, our technique leverages the power of the model checker to compute these abstract slices. In the extreme, our technique can produce full program slices if sufficiently many data structures are considered relevant. However, our abstraction technique can also result in less precise models if too few structures are considered relevant, because relevant types can be control dependent on irrelevant types.

Program slicing can be used to automatically generate relevance information. Our system assumes that relevant fields and variables have already been identified, either by the programmer or by a separate analysis. We could compose our abstraction with a control dependency analysis, in which case our technique could be used to cut off the slice at any desired level of indirection to the slicing criterion.

The Bandera tool set [15] uses several abstraction techniques, including slicing. Slicing is reported to generate state spaces that are sometimes intractably large [16]. Bandera also composes slicing with abstract interpretation to reduce state spaces. Our techniques are complementary, because we specifically target the problem of eliminating long chains of references to relevant data in the presence of aliasing.

The SLAM tools [4] use counterexample-driven refinement to generate models iteratively. This technique can be seen as a demand-driven, iterative form of program slicing, which includes nondeterministic control abstraction. In each iteration, global static alias analysis is used to generate sound models. Our techniques do not rely on iterative refinement, although they could be put to use in such a framework.

A large number of program analysis tools rely on static global alias information. Global alias information can be used to scale analyses to large programs [10] and has also been used in model abstraction [8]. Our technique is relevant for analyses that can afford to use a model checker, but it attempts to eliminate model imprecision caused by global alias analysis by leveraging the model checker.

In some regards, our work is related to program specialization in the presence of pointers [18] and to class hierarchy slicing [28], but these techniques do not produce abstract models for a model checker.

Powerful analyses of the heap, such as shape analyses [13, 23], have focused on modeling unbounded data structures. Our techniques are orthogonal to these analyses, because we focus on solving the problem of model reduction in the presence of aliasing and references by using a dynamic set abstraction. However, we have remarked that our set-based abstraction composes naturally with many other abstractions, and it would be tempting to study its composition with heap analyses.

8 Conclusion

In this paper, we have presented a region-based technique for performing model abstraction in the presence of aliasing. Our technique has a static phase and a dynamic phase: the static type system determines the scope of the alias sets, and the dynamic phase populates these sets with objects. As a result, the model generation phase can construct small models, and the model checking phase can generate precise alias information. We have also shown that by relaxing some of the assumptions of traditional region type systems (namely, the requirement that compatible types must use identical region variables), we can adjust the tradeoff between model precision and performance in a flexible manner. Coping with aliasing is an unavoidable problem when attempting to apply a model checker to large software systems. We believe that this model generation technique will be essential when attempting to verify safety properties of real-world code.

Acknowledgements

We would like to thank Manuel Fähndrich for stimulating discussions about region-based model abstraction.
Expressions

\[
\begin{align*}
\text{(REGEXP TRUE)} & \quad \Pi, \Delta, E \vdash \text{true} : \text{bool} \\
\text{(REGEXP FALSE)} & \quad \Pi, \Delta, E \vdash \text{false} : \text{bool} \\
\text{(REGEXP NULL)} & \quad \Pi, \Delta, E \vdash \text{null} : \text{null} \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGEXP x)} & \quad E(x) = \tau \quad \text{hrv}(\tau) \subseteq \Delta \\
\Pi, \Delta, E & \vdash x \in x : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGEXP FIELD)} & \quad \Pi, \Delta, E \vdash e : [x : \tau]@\rho \\
\rho & \in \Delta \quad \text{hrv}(\tau) \subseteq \Delta \\
\Pi, \Delta, E & \vdash e.x : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGEXP Sub)} & \quad \Pi, \Delta, E \vdash e : \tau \\
\tau & \subseteq \tau' \quad \text{hrv}(\tau') \subseteq \Delta \\
\Pi, \Delta, E & \vdash e : \tau' \\
\end{align*}
\]

Statements

\[
\begin{align*}
\text{(REGSTM DEC)l} & \quad \Pi, \Delta, E \vdash e : \tau_1 \\
\Pi, \Delta, E + \{x : \tau_1\} & \vdash s : \tau \\
\text{fr}(\tau_1) & \subseteq \Delta \\
\Pi, \Delta, E & \vdash \text{let } x = e \text{ in } s : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM LETREGION)} & \quad \rho \notin \Delta \\
\Pi, \Delta \cup (\rho), E & \vdash s : \tau \\
\Pi, \Delta, E & \vdash \text{let region } \rho \text{ in } s : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM NEW)} & \quad \Pi(A, \rho) = \tau = [\ldots]@\rho \\
\Pi, \Delta, E & \vdash x : \tau \\
\rho & \in \Delta \quad \text{fr}(\tau) \subseteq \Delta \\
\Pi, \Delta, E & \vdash x = \text{new } A(\rho)(\ldots) : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM Assign)} & \quad \Pi, \Delta, E \vdash x : \tau \\
\Pi, \Delta, E & \vdash e : \tau \\
\Pi, \Delta, E & \vdash e_1 = e_2 : \tau \\
\Pi, \Delta, E & \vdash e_1.x = e_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM UPDATE)} & \quad \Pi, \Delta, E \vdash e_1 : \tau_1 = [x : \tau]@\rho \\
\Pi, \Delta, E & \vdash e_2 : \tau \\
\rho & \in \Delta \\
\Pi, \Delta, E & \vdash e_1.e_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM METHOD)} & \quad \Pi, \Delta, E \vdash e_0 : \ldots , m : \forall \rho \tau_1 \triangleright \tau_2 \ldots @\rho \\
\Pi, \Delta, E \vdash S(\tau_1) & \\
\Pi, \Delta, E & \vdash x : S(\tau_2) \\
\rho & \in \Delta \quad \text{fr}(\rho) \subseteq \Delta \\
\Pi, \Delta, E & \vdash x = e_0.m(S(\rho))(e_1) : S(\tau_2) \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM Cond)} & \quad \Pi, \Delta, E \vdash e : \text{bool} \\
\Pi, \Delta, E \vdash s_1 : \tau \\
\Pi, \Delta, E \vdash s_2 : \tau \\
\Pi, \Delta, E \vdash \text{if } e \{s_1\} \text{ else } \{s_2\} : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM Seq)} & \quad \Pi, \Delta, E \vdash s_1 : \tau \\
\Pi, \Delta, E \vdash s_2 : \tau \\
\Pi, \Delta, E \vdash s_1.s_2 : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM RETURN)} & \quad \Pi, \Delta, E \vdash \text{return } e : \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM REFL)} & \quad \vdash \tau <: \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM TRANS)} & \quad \vdash \tau_1 <: \tau_2 \quad \vdash \tau_2 <: \tau_3 \quad \vdash \tau_1 <: \tau_3 \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSTM null)} & \quad \vdash \text{null} <: [x_1 : \tau_1^{e_1, \ldots, \nu} \mid m_1 : \phi_1^{e_1, \ldots, \nu}]@\rho \\
\end{align*}
\]

\[
\begin{align*}
\text{(REGSUB OBJECT)} & \quad \tau = [x_1 : \tau_1^{e_1, \ldots, \nu} \mid m_1 : \forall \rho(\tau_1^{e_1, \ldots, \nu} \Delta) \tau_1'^{e_1, \ldots, \nu}]@\rho \\
\tau' = [x_1 : \tau_1^{e_1, \ldots, \nu} \mid m_1 : \forall \rho(\tau_1^{e_1, \ldots, \nu} \Delta) \tau_1'^{e_1, \ldots, \nu}]@\rho \\
\Delta & \subseteq \Delta' \quad \forall i \in 1 \ldots q \\
\vdash \tau <: \tau' \\
\end{align*}
\]

Figure 4: RegJava type rules (expressions and statements)

Figure 5: RegJava subtyping rules
A Appendix

A.1 Type rules

We show the core rules of the RegJava type system in Figure 4, which contains rules for expressions and statements. Figure 5 gives the RegJava subtyping rules. For convenience, we also include the rules for typing declarations (fields, methods, classes, and whole programs) shown in Figure 6. The remaining components of the full type system can be found in the RegJava report [6].

We have modified the RegJava typing rules slightly to accommodate a few syntactic changes to the expression and statement language; these changes were made to ease the presentation of our abstraction function. These changes are not essential to the properties of the type system, and in particular, the soundness proof in the RegJava report [6] can be trivially modified to accommodate them. Moreover, we have introduced new side conditions on rules (REGEXP x), (REGEXP FIELD) and (REGEXP SUB). The new side conditions are expressed by means of a function called “hrv.” In a type of the form $[\ldots]\varrho$, we say that $\varrho$ occurs in head position, and we refer to $\varrho$ as a head region variable. The function hrv collects the head region variable of a type, if any:

$$\text{hrv}(\tau) = \begin{cases} \{\varrho\} & \text{if } \tau = [\ldots]\varrho \\ \emptyset & \text{otherwise} \end{cases}$$

The side conditions ensure that for any expression of reference type $[\ldots]\varrho$, the region variable $\varrho$ is in the set $\Delta$ of free region variables. The reason for the addition of the side conditions is technical: they imply that we never need to apply the consistency rule (CON DANGLE) from Figure 7 (see Section A.3.1) in proving soundness of our abstraction of expressions (see Section A.3.2). We could have avoided these side conditions by making our abstraction function slightly more complicated. Notice that RegJava soundness as proven in the original report [6] obviously still holds for the modified type system, since we have added more restrictions to the system. Notice also that we can still type the same set of programs (modulo region annotations); in the worst case, the size of the region variable sets $\Delta$ will be somewhat larger.

A.2 Semantics

The operational semantics of RegJava is given in the original report [6] and will not be repeated here (however, the core rules of the RegJava semantics can be gleaned from our proof of Theorem 1 in Section A.3.4.) At the statement level, the semantics is defined by a reduction relation

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

and at the expression level a relation

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

We have introduced a few trivial changes to the RegJava language. For convenience in defining our abstraction function, we have removed side effects from the expression sublanguage; thus, all constructs with side effects appear at the statement level (e.g., assignment statements), and all expression-level reductions have the form $\sigma, V, R \vdash e \rightsquigarrow v, \sigma, V$ (i.e., $\sigma$ and $V$ remain unchanged). In addition, we have included the static method invocation form $A.m(e)$. It should be very obvious how to modify the semantic rules in the RegJava report [6] to accommodate our changes.

The operational semantics of RegJava+Choice define a reduction relation

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

for statements and a relation

$$\sigma, V, R \vdash e \rightsquigarrow v, \sigma, V$$

for expressions. These relations are defined by including the rules of RegJava (exchanging the relation symbol $\rightsquigarrow$ with $\rightsquigarrow$ in those rules) and by adding the following rules for nondeterministic choice and skip statements:
\[ \Pi, \sigma, R, \Delta \vdash \text{val} : \tau \]
\[ \Pi, \sigma, R, \Delta \vdash \text{null} : \text{null} \]

\[ \Pi, \sigma, R, \Delta \vdash \text{true} : \text{bool} \]
\[ \Pi, \sigma, R, \Delta \vdash \text{false} : \text{bool} \]

\[ \Pi, \sigma, R, \Delta \vdash v : \tau \Rightarrow \tau < : \tau' \]
\[ \Pi, \sigma, R, \Delta \vdash v : \tau' \]

\[ \rho \notin \Delta \]
\[ \Pi, \sigma, R, \Delta \vdash (r, o) : [x_i : \tau_i^{r_i} \in \{1 .. n \} \cdots] @ \rho \]

\[ \tau = [x_i : \tau_i \in \{1 .. n \} | m_i : \phi_i \in \{1 .. n \}] \]
\[ r \in \text{Dom}(\sigma) \]
\[ o \in \text{Dom}(\sigma(r)) \]
\[ R(\rho) = r \]
\[ \sigma(r, 0) = \langle A, \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}, \tau > \]
\[ \Pi, \sigma, R, \Delta \vdash v_i : \tau_i \]
\[ \text{for all } 1 \leq i \leq n \]
\[ A \in \text{Dom}(\Pi) \]
\[ \exists \rho. \Pi(A, \rho) = \tau \]

Figure 7: Consistency rules

\[ \langle \text{Choose} \rangle \]
\[ R(\rho) = r \]
\[ (r, o) \in \text{Dom}(\sigma) \]
\[ \sigma(r)(o) = \langle A, \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}, \tau > \]
\[ \exists \rho. \Pi(A, \rho) < \tau \]
\[ \sigma, V, R \vdash \text{choose}(\rho, \tau) \Rightarrow (r, o), \sigma, V \]

\[ \langle \text{Choose Type} \rangle \]
\[ \exists \rho. \Pi(A, \rho) < \tau \]
\[ \sigma, V, R \vdash A.m(e) \Rightarrow v, \sigma', V' \]
\[ \sigma, V, R \vdash \text{choose_type}(\tau), m(e) \Rightarrow v, \sigma', V' \]

\[ \langle \text{Choose True} \rangle \]
\[ \sigma, V, R \vdash \text{choose(bool)} \Rightarrow \text{true}, \sigma, V \]

\[ \langle \text{Choose False} \rangle \]
\[ \sigma, V, R \vdash \text{choose(bool)} \Rightarrow \text{false}, \sigma, V \]

\[ \langle \text{Skip} \rangle \]
\[ \sigma, V, R \vdash \text{skip} \Rightarrow \text{null}, \sigma, V \]

A.3 Soundness proof
A.3.1 Consistency

The relation
\[ \Pi, \sigma, R, \Delta \vdash v : \tau \]
is the relation co-inductively defined by the rules of Figure 7. This definition is taken from the RegJava report [6]. The relation is lifted to environments by defining
\[ \Pi, \sigma, R, \Delta \vdash V : E \] to hold if and only if
\[ \text{Dom}(V) = \text{Dom}(E) \]
\[ \Pi, \sigma, R, \Delta \vdash V(x) : E(x) \] for all \( x \in \text{Dom}(V) \)

In the context of program \( p \) and live region variables \( A_i \), we define the relation \( \text{Con}(\Pi, \Delta, E, \sigma, V, R) \) to hold if and only if the following conditions are satisfied:

1. \( A_i \vdash p : \Pi \)
2. \( \Pi, \sigma, R, \Delta \vdash v : E \)
3. \( \Delta \subseteq \text{Dom}(R) \)
4. \( \text{frv}(E) \subseteq \Delta \)

For a statement \( s \) in a typed program \( p \), we say that typing \( \Pi, \Delta, E \vdash s : \tau \) is a well typing with respect to a dynamic environment \( \sigma, V, R \) if and only if we have \( \text{Con}(\Pi, \Delta, E, \sigma, V, R) \) in the context of \( p \) and \( A_i \).

In the context of \( p \), \( \Delta \), the relation
\[ \Pi, \Delta, E \vdash s : \tau \]
is defined [6] to hold if and only if the following implication is true for all \( \sigma, V, R, v_0 \):

If the following conditions are satisfied

1. \( \text{Con}(\Pi, \Delta, E, \sigma, V, R) \)
2. \( \Pi, \sigma, R, \Delta \vdash v_0 : \tau_0 \)
3. \( \sigma, V, R \vdash s \Rightarrow v, \sigma', V' \)

then

1. \( \Pi, \sigma', R, \Delta \vdash v : \tau \)
2. \( \Pi, \sigma', R, \Delta \vdash v' : E \)
3. \( \Pi, \sigma', R, \Delta \vdash v_0 : \tau_0 \)

This definition is a slightly reformulated version of the definition given for RegJava [6], mainly because we have introduced the relation \( \text{Con}(\ldots) \) for convenience. Our definition is trivially equivalent to the one given in the RegJava report [6]. The relation on expressions
\[ \Pi, \Delta, E \vdash e : \tau \]
is defined analogously to the relation on statements, by substituting \( e \) for \( s \) in the definition for statements.

The following soundness theorem is proven for RegJava [6]:

**Theorem 2 (RegJava type soundness)** For any \( \Pi, \Delta, E, s, e \) and \( \tau \):

1. If \( \Pi, \Delta, E \vdash s : \tau \), then \( \Pi, \Delta, E \vdash s : \tau \)
2. If \( \Pi, \Delta, E \vdash e : \tau \), then \( \Pi, \Delta, E \vdash e : \tau \)

In the RegJava report [6], similar soundness results are also proven for consistency relations defined on class declarations, method declarations, and whole programs. In this report, we confine ourselves to considering soundness at the statement and expression levels, since it is easy to prove the remaining soundness properties in analogy with the proof in the RegJava report [6].
A.3.2 Proof of Lemma 1

In the following formulation of Lemma 1 we unfold the definition of being a well-typed with respect to a dynamic environment \( \sigma, V, R \).

**LEMMA 1** Assume

1. \( \Pi, \Delta, E \vdash e : \tau \)
2. \( \sigma, V, R \vdash e \rightarrow (r, o), \sigma, V \)
3. \( \text{Con}(\Pi, \Delta, E, \sigma, V, R) \)
4. \( \text{RR}(\tau, \rho) \)
5. \( \sigma \equiv \hat{\sigma} \) and \( V \equiv \hat{V} \).

Then \( \hat{\sigma}, \hat{V}, R \vdash [e'] \rightarrow (r, o), \hat{\sigma}, \hat{V} \).

**PROOF:** By soundness of RegJava, it follows from assumption 1, assumption 2, and assumption 3 that we have

\[ \Pi, \sigma, R, \Delta \models (r, o) : \tau \]  

(1)

From assumption 5, we get \( \tau = [\ldots] \hat{\tau} \), and Lemma 3 implies \( \rho \in \Delta \). Then (1) implies that, for some \( \tau' < \tau \), we have:

\[ R(\rho) = r, r \in \text{Dom}(\sigma), o \in \text{Dom}(\sigma(r)) \]
\[ \sigma(r, o) = \langle A, V', \hat{r}' \rangle \]
\[ \exists \hat{\rho}(A, \hat{\rho}) = \tau' \]

An application of rule (CHOOSE) proves the lemma. □

A.3.3 Proof of Lemma 2

In the following formulation of Lemma 2 we unfold the definition of being a well-typed with respect to a dynamic environment \( \sigma, V, R \).

**LEMMA 2** Assume

1. \( \Pi, \Delta, E \vdash e : \tau \)
2. \( \sigma, V, R \vdash e \rightarrow v, \sigma, V \)
3. \( \text{Con}(\Pi, \Delta, E, \sigma, V, R) \)
4. \( \tau \) is relevant
5. \( \sigma \equiv \hat{\sigma} \) and \( V \equiv \hat{V} \).

Then \( \hat{\sigma}, \hat{V}, R \vdash [e'] \rightarrow v, \hat{\sigma}, \hat{V} \).

**PROOF:** We proceed by cases over the syntactic form of \( e \):

- If \( e = x \)
  - Because \( \tau \) is relevant by assumption 4, we have \([e'] = x'\), and \( \sigma, V, R \vdash e \rightarrow v, \sigma, V \) is derived by an application of rule (DYNEXP x):
    \[ \frac{}{\sigma, V, R \vdash e \rightarrow v, \sigma, V} \]
  - Because \( \hat{V} \approx_v V \) and \( \tau \) is relevant, it follows from assumption 5 that \( \hat{V}(x') = v \), and the lemma is proven by an application of rule (DYNEXP x) in RegJava+Choice.

- If \( e = e_1, x \)
  - Because \( \tau \) is relevant, we have \([e] = [e_1', x'] = [e_1'] x'\) with \( \Pi, \Delta, E \vdash e_1 : \tau_1 \), and we know that \( \sigma, V, R \vdash e \rightarrow v, \sigma, V \) is derived by an application of rule (DYNEXP FIELD):
    \[ \frac{}{\sigma, V, R \vdash e_1 \rightarrow a, \sigma, V} \]
    \[ \frac{\sigma \equiv \hat{\sigma} \quad V \equiv \hat{V} \quad \tau \equiv \hat{\tau} \quad \tau \equiv \hat{\tau}}{\sigma, V, R \vdash e_1, x \rightarrow v, \sigma, V} \]
  - where \( a = (r, o) \) for some \( r, o \). Because \( \tau \) is relevant (by assumption 4), we have \( \text{RR}(\tau_1, \rho) \) for some \( \rho \). It then follows from Lemma 1 that
    \[ \hat{\sigma}, \hat{V}, R \vdash [e_1'] \rightarrow a, \hat{\sigma}, \hat{V} \]
  - Because \( \text{RR}(\tau_1, \rho) \) holds, it must be the case that \( A \) contains a relevant field, and hence \( \hat{\sigma} \equiv \sigma \) implies \( a \in \text{Dom}(\hat{\sigma}) \) with \( \hat{\sigma}(a) = \langle A, V''', \hat{r}' > \) with \( V'' \approx_v V'' \).
  - Because \( \tau \) is relevant, it follows from \( V'(x) = v \) and \( V'' \approx_v V' \) that \( V''(x') = v \). An application of rule (DYNEXP FIELD) in RegJava+Choice then proves the lemma. □

A.3.4 Proof of Theorem 1

In the following formulation of Theorem 1 we unfold the definition of being a well-typed with respect to a dynamic environment \( \sigma_0, V_0, R \).

**THEOREM 1** (*Soundness*) Let \( s \) be a well-typed statement with respect to \( \sigma_0, V_0, R \vdash s \rightarrow v_1, \sigma_1, V_1 \) in the context of typed program \( p \), such that:

(i) \( \Pi, \Delta, E \vdash s : \tau \)
(ii) \( \sigma_0, V_0, R \vdash s \rightarrow v_1, \sigma_1, V_1 \)
(iii) \( \text{Con}(\Pi, \Delta, E, \sigma_0, V_0, R) \)
(iv) \( \sigma_0 \equiv \hat{\sigma} \) and \( V_0 \equiv \hat{V} \)

Then there exist \( \hat{v}_1, \hat{\sigma}_1, \hat{V}_1 \) such that:

1. \( \hat{\sigma}_0, \hat{V}_0, R \vdash [s] \rightarrow \hat{v}_1, \hat{\sigma}_1, \hat{V}_1 \)
2. \( \text{if } \tau \text{ is relevant, then } v_1 = \hat{v}_1 \)
3. \( \sigma_1 \equiv \hat{\sigma}_1 \) and \( V_1 \equiv \hat{V}_1 \)

**PROOF:** By induction on the derivation of (ii). We proceed by cases over the last rule used in the derivation. We consider the core subset of the rules defining
the operational semantics of RegJava [6], leaving out remaining cases, for which the proof is easily extended by analogy to the proof of the rules considered here.

**[DYNSTM DECL]**

We have a derivation of the form:

\[
\sigma_0, V_0, R \vdash e \rightarrow v, \sigma_0, V_0 \\
\sigma_0, V_0, e \rightarrow v, R \vdash s \rightarrow v_1, \sigma_1, V_1 \\
\sigma_0, V_0, E \vdash \text{let } x = e \text{ in } s \rightarrow v_1, \sigma_1, V_1 \{x\}
\]

with typing

\[
\Pi, \Delta, E \vdash e : \tau_1 \\
\Pi, \Delta, E + \{x : \tau_1\} \vdash s : \tau \\
\Pi, \Delta, E \vdash \text{let } x = e \text{ in } s : \tau
\]

By soundness of RegJava and assumptions (ii) and (iii), we can conclude

\[
\Pi, \sigma_0, R, \Delta \models v : \tau_1
\]

Moreover, from (ii) we get

\[
\Pi, \sigma_0, R, \Delta \models V_0 : E
\]

From (2) and (3) it follows that

\[
\Pi, \sigma_0, R, \Delta \models V_0 + \{x \rightarrow v\} : E + \{x \rightarrow \tau_1\}
\]

Using (iii) together with (4) it follows that

\[
\text{Con}(\Pi, \Delta, E + \{x \rightarrow \tau_1\}, \sigma_0, V_0 + \{x \rightarrow v\}, R)
\]

We consider the case where \(\tau_1\) is relevant. We have

\[
\{\text{let } x = e \text{ in } [s]\} = \{\text{let } x = [e^\tau] \text{ in } [s]\}
\]

By Lemma 2 we have

\[
\sigma_0, \hat{V}_0, R \vdash [e^\tau] \rightarrow v^\tau, \sigma_0, \hat{V}_0
\]

Now, because of (iv), \(V_0 \approx_\tau \hat{V}_0\), and because \(\tau_1\) is relevant, we get

\[
V_0 + \{x^\tau \rightarrow v\} \approx_\tau \hat{V}_0 + \{x^\tau \rightarrow v\}
\]

Using (5) and (7) together with remaining assumptions, the inductive hypothesis is applicable to the reduction

\[
\sigma_0, V_0 + \{x \rightarrow v\}, R \vdash s \rightarrow v_1, \sigma_1, V_1
\]

and yields

\[
\sigma_0, \hat{V}_0 + \{x^\tau \rightarrow v\}, R \vdash [s] \rightarrow v_1, \sigma_1, \hat{V}_1
\]

such that \(\sigma_1 \approx_\tau \sigma_1\) and \(\hat{V}_1 \approx_\tau V_1\), and such that \(\tau\) relevant implies \(v_1 = v_1\). An application of rule (DYNSTM DECL) in RegJava+Choice then proves the theorem for this case.

Next, if \(\tau_1\) is not relevant, we have

\[
\{\text{let } x = e \text{ in } [s]\} = [s]
\]

In this case, we have \(\hat{V}_0 \approx_\tau V_0 + \{x \rightarrow v\}\), and so, by induction hypothesis,

\[
\sigma_0, \hat{V}_0, R \vdash [s] \rightarrow v_1, \sigma_1, \hat{V}_1
\]

with \(\sigma_1 \approx_\tau \sigma_1\) and \(\hat{V}_1 \approx_\tau V_1\). Because \(\tau_1\) is not relevant, it follows that \(V_1 \approx_\tau V_1 \{x\}\), thereby proving the theorem in this case.

**[DYNSTM LETREGION]**

We have a derivation of the form:

\[
r \notin \text{Dom}(\sigma_0) \\
\sigma_0 + \{r \rightarrow \emptyset\}, V_0, R + \{\rho \rightarrow r\} \vdash s \rightarrow v_1, \sigma_1, V_1 \\
\sigma_0, V_0, R \vdash \text{letregion } \rho \text{ in } s \rightarrow v_1, \sigma_1 \{r\}, V_1
\]

with typing

\[
\rho \notin \Delta \\
\Pi, \Delta \cup \{\rho\}, E \vdash s : \tau
\]

Because \(\rho \notin \Delta\) and \(\Delta \subseteq \text{Dom}(R)\) holds by (iii), we have \(\Delta \cup \{\rho\} \subseteq \text{Dom}(R + \{\rho \rightarrow r\})\). It follows that we have

\[
\text{Con}(\Pi, \Delta \cup \{\rho\}, E, \sigma_0 + \{r \rightarrow \emptyset\}, V_0, R + \{\rho \rightarrow r\})
\]

Moreover, because \(\sigma_0 \approx \sigma_0\), we get \(\sigma_0 + \{r \rightarrow \emptyset\} \approx_\tau \sigma_0 + \{r \rightarrow \emptyset\}\). Then the theorem follows by induction hypothesis followed by an application of rule (DYNSTM LETREGION) in RegJava+Choice.

**[DYNSTM NEW]**

We have a derivation of the form:

\[
\Pi(A, \rho_1, \ldots, \rho_k) = \{x_i : \tau_i\} \cup \rho \\
V = \{x_1 \rightarrow \text{init}(\tau_1), \ldots, x_n \rightarrow \text{init}(\tau_n)\} \\
R(\rho) = r \quad \rho \notin \text{Dom}(\sigma(r)) \\
\sigma_1 = \sigma_0 + \{(r, o) \rightarrow < A, V, R(\rho_1), \ldots, R(\rho_k)>\} \\
V_1 = V_0 + \{x \rightarrow (r, o)\} \\
\sigma_0, V_0, R \vdash x = \text{new } A(\rho_1, \ldots, \rho_k)() \rightarrow (r, o), \sigma_1, V_1
\]

with typing

\[
\Pi(A, \rho) = \tau = \{\ldots\} \cup \rho \\
\Pi, \Delta, E \vdash x : \tau \\
\rho \in \Delta \\
\text{frv}(\tau) \subseteq \Delta \\
\Pi, \Delta, E \vdash x = \text{new } A(\rho)() : \tau
\]

Because \(\sigma_0 \approx \sigma_0\), it follows that

\[
\sigma_0 + \{(r, o) \rightarrow < A, V, R(\rho_1), \ldots, R(\rho_k)>\} \approx_\tau \\
\sigma_0 + \{(r, o) \rightarrow < A, V, R(\rho_1), \ldots, R(\rho_k)>\}
\]

If \(\tau\) is relevant, we have

\[
[x^\tau = \text{new } A(\rho)()] = x^\tau = \text{new } A(\rho)()
\]

Since \(\hat{V}_0 \approx_\tau V_0\) and \(\tau\) is relevant, it follows that

\[
\hat{V}_0 + \{x^\tau \rightarrow (r, o)\} \approx_\tau V_0 + \{x^\tau \rightarrow (r, o)\}
\]

Using (10) and (11), the theorem follows by an application of rule (DYNSTM NEW1) in RegJava+Choice.
If \( \tau \) is not relevant, we have
\[
[x^r = \text{new } A(\bar{d})()] = \text{new } A(\bar{d})
\]
Here, we have \( \dot{V}_0 \approx_r V_0 \) implies
\[
\dot{V}_0 \approx_r V_0 + \{x^r \rightarrow (r, a)\} \tag{12}
\]
since \( \tau \) is not relevant. Using (10) and (12), the theorem follows by an application of rule (DynStm New2) in RegJava + Choice.

**[DynStm Assign]**

We have a derivation of the form:
\[
\pi_0, V_0, R \vdash e \rightarrow v, \pi_0, V_0
\]
with typing
\[
\Pi, \Delta, E \vdash x : \tau
\]
If \( \tau \) is relevant, we have
\[
[x^r = e^r] = x^r = [e^r]
\]
Because \( \dot{V}_0 \approx_r V_0 \), we have \( \dot{V}_0 + \{x^r \rightarrow v\} \approx_r V_0 + \{x^r \rightarrow v\} \). The theorem now follows from Lemma 2 together with an application of rule (DynStm Assign) in RegJava + Choice.

If \( \tau \) is not relevant, we have
\[
[x^r = e^r] = \text{skip}
\]
The theorem follows from the fact that, since \( \tau \) is not relevant, \( \dot{V}_0 \approx_r V_0 \) implies \( V_0 \approx_r V_0 + \{x^r \rightarrow v\} \).

**[DynStm Update]**

We have a derivation of the form:
\[
\pi_0, V_0, R \vdash e_1 \rightarrow a, \pi_0, V_0
\]
with typing
\[
\Pi, \Delta, E \vdash e_1 : \tau
\]
If \( \tau \) is relevant, we have
\[
[e^r_1, x^r = e^r_2] = [e^r_1] \times^r = [e^r_2]
\]
with RR(\( \tau, \rho \)). We can apply Lemma 2 to the reduction
\[
\sigma_0, V_0, R \vdash e_1 \rightarrow a, \pi_0, V_0
\]
and obtain
\[
\dot{\sigma}_0, \dot{V}_0, R \vdash [e^r_1] \rightarrow a, \dot{\sigma}_0, \dot{V}_0 \tag{13}
\]
Because \( \dot{\sigma}_0 \approx_r \dot{\sigma}_0 \), it follows from \( \sigma_0(a) = \langle A, V', \bar{r} \rangle \) that we have
\[
\dot{\sigma}_0(a) = \langle A, V'', \bar{r} \rangle \quad \text{for some } V'' \text{ with } V'' \approx_r V'
\]
Lemma 1 applied to the reduction
\[
\sigma_0, V_0, R \vdash e_2 \rightarrow v, \sigma_0, V_0
\]
yields
\[
\dot{\sigma}_0, \dot{V}_0, R \vdash [e^r_3] \rightarrow v, \dot{\sigma}_0, \dot{V}_0 \tag{15}
\]
Using (13), (14), and (15), we can apply rule (DynStm Update) in RegJava + Choice to get
\[
\dot{\sigma}_0, \dot{V}_0, R \vdash [e^r_1], x^r = [e^r_2] \rightarrow v, \dot{\sigma}_1, \dot{V}_0
\]
where
\[
\dot{\sigma}_1 = \dot{\sigma}_0 + \{a \rightarrow \langle A, V'' + \{x^r \rightarrow v\}, \bar{r} \rangle\}
\]
Using \( \dot{\sigma}_0 \approx_r \dot{\sigma}_0 \) and \( V'' \approx_r V' \), the theorem follows from (16).

If \( \tau \) is not relevant, we have
\[
[e^r_1, x^r = e^r_2] = \text{skip}
\]
Because \( \tau \) is not relevant, we have \( V' \approx_r V'' + \{x^r \rightarrow v\} \) and \( \dot{\sigma}_0 \approx_r \dot{\sigma}_0 + \{a \rightarrow \langle A, V'' + \{x^r \rightarrow v\}, \bar{r} \rangle\} \), from which the theorem follows in this case.

**[DynStm Method]**

We have a derivation of the form:
\[
\pi(A) = \{\lambda \rho_0 \ldots \rho_k : \ldots : \emptyset, \rho_{k+1} \ldots \rho_n, \tau_1 \rightarrow \tau_2, \ldots \} \Gamma \tag{14}
\]
\[
\pi_0, V_0, R \vdash e_0 \rightarrow a, \pi_0, V_0
\]
\[
\pi_0, V_0, R \vdash e_1 \rightarrow v, \pi_0, V_0
\]
\[
\sigma_0(a) = \langle A, V', \tau_1, \bar{r} \rangle
\]
\[
R' = \{\rho_1 \rightarrow \tau_1, \ldots, \rho_n \rightarrow \tau_n\}\}
\]
\[
\sigma_0(\text{this } \rightarrow a) + \{y \rightarrow v\}, R + R' \vdash s \rightarrow v_1, \sigma_1, V_1
\]
\[
\sigma_0, V_0, R \vdash x = e_0.m(\rho'_1, \ldots, \rho'_n)(e_1) \rightarrow v_1, \sigma_1, V_1 + \{x \rightarrow v\}
\]
with typing
\[
\Pi, \Delta, E \vdash e_0 : \tau = \{\ldots, m : \emptyset, \rho \rightarrow \tau_1 \rightarrow \tau_2, \ldots \} \Gamma
\]
\[
\Pi, \Delta, E \vdash e_1 : S(\tau_1)
\]
\[
\Pi, \Delta, E \vdash x : S(\tau_2)
\]
\[
\rho \in \Delta, S(\Delta) \subseteq \Delta, \quad S(\text{frv}(\rho_1)) \subseteq \Delta
\]
assuming first that the types \( \tau, \tau_1 \) and \( \tau_2 \) are all relevant. Using (iii), we can show that we have
\[
\text{Con}(\Pi, \Delta, E, \sigma_0, \{\text{this } \rightarrow a\} + \{y \rightarrow v\}, R + R') \tag{17}
\]
by repeating the argument for RegJava soundness in the case of method invocation. Rather than repeating this argument, we refer to [5] p. 72-76 for details. Using (17), we can apply induction hypothesis to the reduction
\[
\sigma_0, \{\text{this } \rightarrow a\} + \{y \rightarrow v\}, R + R' \vdash s \rightarrow v_1, \sigma_1, V_1
\]
to obtain
\[
\dot{\sigma}_0, \{\text{this } \rightarrow a\} + \{y \rightarrow v\}, R + R' \vdash [a] \rightarrow \dot{v}_1, \dot{\sigma}_1, \dot{V}_1
\]
such that \( \dot{V}_1 \approx_r V_1, \dot{\sigma}_1 \approx_r \sigma_1 \), and if \( \bar{r} \) is relevant, then \( \dot{v}_1 = v_1 \). Moreover, assuming that \( \tau \) and \( \tau_1 \) both have relevant types, induction hypothesis yields
\[
\dot{\sigma}_0, \dot{V}_0, R \vdash [e^r_1] \rightarrow a, \sigma_0, V_0
\]
(20)
and 
\[ \sigma_0, \tilde{V}_0, R \vdash [e_1] \rightsquigarrow_\Delta \sigma, \tilde{V}_0 \]

The theorem now follows from an application of rule (DYNSTM METHOD) in RegJava+Choice. If \( \tau \) and \( \tau_1 \) are not relevant, we have \( \emptyset \Rightarrow \{ \text{this} \Rightarrow a \} \cup \{ y \Rightarrow v \} \), and induction hypothesis applied to (18) yields 
\[ \sigma_0, \emptyset, R + R' \vdash [\emptyset] \rightsquigarrow_\Delta \sigma_1, \tilde{V}_1 \]

with \( \tilde{v}_1 = v_1 \) if \( \tau_1 \) is relevant. We have 
\[ x = c_0.m(\tilde{\tau})(e_1) = x = \text{choose type}(\tau).m(\tilde{\tau})(\cdot) \]

Using (22) and (CHOOSE TYPE) in RegJava+Choice, the theorem follows in this case, because the method \( m \) invoked in the RegJava program must be among the methods chosen by the operation \( \text{choose type}(\tau) \).

The remaining cases, where some of \( \tau \) or \( \tau_1 \) or \( \tau_2 \) are relevant, are proven by analogous arguments.

\[ \square \]

References


