ADVANCED FLOW-BASED TYPE SYSTEMS FOR OBJECT-ORIENTED LANGUAGES

FLORIN CRACIUN

(M.Sc., Technical University of Cluj-Napoca, Romania)

(B.Sc., Technical University of Cluj-Napoca, Romania)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2008

ACKNOWLEDGEMENTS

First of all, I would like to deeply thank my supervisor Professor Wei-Ngan Chin, who has been a constant source of advice, guidance and encouragement. This dissertation clearly represents an outgrowth of his research vision. His enormous energy and dedication as well as his combined theoretical and practical sense will always remain a model.

I am very grateful to Professor Siau-Cheng Khoo for his generous and timely help, for useful discussions, which influenced my work, and for his kindness in general.

I would like to express my gratitude to committee members Professor Jens Palsberg, Professor Siau-Cheng Khoo, Professor Martin Henz, and Professor Roland Yap for the interest and time they granted to this work. Their feedback and comments helped me better understand the weaknesses and strengths of this work.

I would also like to thank my co-authors without whom many parts of this text and other joint work would not have been possible: Professor Wei-Ngan Chin, Professor Siau-Cheng Khoo, Professor Martin Rinard, Dr. Shengchao Qin, Corneliu Popeea and Hong Yaw Goh.

I also want to thank Razvan Voicu, Corneliu Popeea, Cristina David, Huu Hai Nguyen, Mihail Asavoae, Mariuca Asavoae, Dana N. Xu, Wang Meng, Zhu Ping, David Lo, Stefan Andrei, Saswat Anand, Andrei Hagiescu, Alexandru Stefan, Cristian Gherghina for being great friends and colleagues throughout the years, and contributing to a fun and exciting environment, in and out of office. Special thanks to my best colleague Corneliu Popeea for our many technical discussions.

I am deeply thankful to my parents for their continued love and support. They have done whatever they could to ensure that I had the best education possible. This work is dedicated to them. Finally, I would like to thank my dearest Ioana for her constant encouragement and support.

TABLE OF CONTENTS

SUMMARY

This dissertation proposes two advanced type systems to improve two aspects of software quality, namely memory safety via region types and software reusability via generic types. Our type systems are designed in the context of a Java-like object-oriented language. Their two main ingredients consist of a simple flow analysis and a set of partially-ordered type annotations. Flow analysis captures type annotations in a flow-insensitive manner through the program code, but summarizes a parameterized flow at each method boundary. Subtyping of annotated types provides the direction of flows. With it, the type rules generate flow (subtyping) constraints among the annotated types.

Our first type system addresses the problem of a safe compile-time region-based memory management. We have formulated and implemented an automatic region type inference system. To provide an inference method that is both precise and practical, we support classes and methods that are region-polymorphic, with region-polymorphic recursion for methods. One challenging aspect is to ensure region safety in the presence of features such as class inheritance, method overriding, and downcast operations. Our region inference rules can handle these object-oriented features safely without creating dangling references. Initial experimental results are encouraging, as programs based on our inferred regions have been able to reuse a significant amount of memory, especially for cases when data are not live throughout the execution.

Our second type system addresses the problem of software reusability (genericity) in a type safe way. We propose a novel flow-based approach for the variant parametric types. Variant parametric types represent the successful result of combining subtype polymorphism with parametric polymorphism to support a more flexible subtyping for the object-oriented paradigm. A key feature of this combination is the variance. We have formulated and implemented a novel framework based on flow analysis and modular type checking to provide a simple but accurate model for capturing variant parametric types. Our scheme fully supports casting for variant parametric types with a special reflection mechanism, called cast capture to handle objects with unknown types. Experiments indicate that more downcasts can be eliminated by our approach, even when it is compared against the type system of Java 1.5.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Improving software quality is one of the most challenging problems facing software industry today. Software engineering methods, development tools, and programming languages all work together to accomplish this goal. Software quality consists of many aspects, however this dissertation focuses on only two of them, namely *memory safety via region types* and *software reuse via generic types*.

An important component of development tools used to improve the software quality is static program analysis. Static program analysis, as defined by Nielson et al. in [132], can be regarded as a collection of “compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program”.

Design and implementation of type systems has been one of the most active fields in static program analysis research over the last years. Among the multitude of proposals for statically-checked program annotations, types are the most pervasive. Type checking has been received with open arms by the software industry. Nearly all mainstream languages have been equipped with type systems to detect errors at compile time. In many languages, programmers must include type annotations in their source code. On top of these type annotations a large number of type-based analyses have been developed [141].

1.1 Thesis

In the context of developing novel sophisticated type-based program analyses for object-oriented languages we propose the following thesis: *a simple flow analysis tracing partially-ordered type annotations can produce advanced type systems with practical benefits for object-oriented languages*.

Standard type systems ensure simple safety properties at compile time. The specification of these properties is given by the types’ semantics. More complex safety properties are enforced by advanced type systems and their related static analyses. Advanced type systems can be obtained by augmenting the semantics of the standard types with additional static information.

A common approach is to decorate the standard types with some annotations.

In the context of the object-oriented languages, our type decoration consists in parameterizing a class type with additional annotations that can refer to a property of the object itself but also to the properties of the object fields. An annotation can take values from a partially-ordered domain, without being restricted to atomic properties. The partial order is used to define a *subtyping relation on the annotated types*.

The main ingredient is *a simple flow analysis*, by which we mean an analysis that is *flow-insensitive* inside the method body and *context-sensitive* at the method boundaries. A flow-insensitive analysis ignores the order of updates and therefore it can be considered to model all statements interleavings. A context-sensitive flow analysis can distinguish between different calling contexts of a method and does not allow information from one caller to propagate erroneously to another caller of the same method.

The main role of flow analysis is to trace the properties denoted by the type annotations through the source code level terms. However our *simple* flow analysis is limited to proving only program properties which are true throughout the whole execution of a method. The flow analysis is directly encoded in the type rules of the advanced type system. The subtyping relation of the annotated types provides suitable directions for the flow. As a consequence, the type rules of the resulting advanced type system generate *flow (subtyping) constraints* among the annotations (rather than equalities).

A type system has *practical benefits* if it can satisfy the following basic requirements defined by Cardelli in [32]: *decidably verifiable*, *transparent* and *enforceable*. The first property means that the typechecking algorithm can ensure that a program is well typed. Indeed, most type systems are simple enough for typechecking to be *decidable*. A typechecking algorithm is decidable if it is able to automatically verify that the types provided by the programmer (assuming that the programmer supplies sufficient type information) are correct and that the program indeed has the specified type. However, in the case of advanced type systems where more complex properties are verified, the typechecking algorithms may not be able to take a decision (namely either accept or reject the program) and therefore they may not terminate. However if sound approximations can be applied for the non-terminating situations, those type systems still have practical benefits. The idea is to trade off completeness for the possibility to verify more complex properties. The second property, *transparency* ensures that the programmer is able to predict whether a program typechecks and the reason for the failure when the typechecking fails. Annotated types can be quite complex. However we believe that the use of flow analysis guided by subtyping is a natural and easy way to understand them. The third property, *enforceable* refers to the possibility of run-time checking of those type declarations which cannot be statically verified.

Type-based program analyses are based on the type checking and/or type inference algorithms developed for the advanced type systems. Using the properties of type-based analyses described by Palsberg in [141], we introduce the requirements for the type checking and type inference algorithms to have *practical benefits* as follows: *simplicity*, *efficiency*, *precision* and *correctness*. Simplicity ensures that the algorithms are easy to implement and integrate into a compiler. Efficiency ensures that the algorithms can be scaled to larger programs. Precision is very important. However, algorithms which are less precise but computationally cheaper, may be preferable in practice. We have already taken such a decision by adopting a flow-insensitive analysis rather than a more precise flow-sensitive one. Correctness is proven using standard type theory techniques, namely it can be stated as a type soundness theorem. The well-understood method for proving type soundness based on proving type preservation and progress can be extended to annotated types.

1.2 Applications

The overall goal of our dissertation is to prove our thesis by developing advanced flow-based type systems that improve on software quality. In the context of Java-like object-oriented languages, our dissertation addresses two important applications towards this goal, as described next.

1.2.1 Safe Region-based Memory Management

Modern object-oriented programming languages provide a run-time system that automatically reclaims memory using tracing garbage collection [203]. A correct garbage collector can guarantee that the memory is not collecting too early, and also that all memory is eventually reclaimed if the program terminates. However the space and time requirements of garbage-collected programs are very difficult to estimate in practice. Therefore real-time and embedded software tries to avoid the use of garbage collectors. Many different solutions were proposed for these problems such as either real-time garbage collectors, or safe manual memory management, or safe automatic compile-time memory management.

In the context of a safe automatic compile-time memory management, *our goal was to develop an automatic region type inference system for object-oriented languages*. Region-based memory management systems allocate each new object into a program-specified *region*, with the entire set of objects in each region deallocated simultaneously when the region is deleted. The basic ideas of a region type system and the first region type inference algorithm for a simply typed lambda calculus have been proposed in Tofte and Talpin's seminal work [191]. Later on, several projects have investigated the use of region type systems for Java-like object-oriented languages [41, 23] and C-like imperative languages [80], but without providing an automatic region type inference. They have mostly focused on region type checking, which requires an additional effort for the programmer to augment the program with *region annotations*.

1.2.2 Software Reusability via Better Generic Types

In object-oriented programming a large software is built by combining different small objects into a large object, thus making the software reusability (also called *genericity*) one of the most important issue. Traditionally, most mainstream object-oriented languages, such as Java, C++ and C#, have relied on *subtype polymorphism* to support software reusability. Subtype polymorphism is a nominal relation, based on a class hierarchy declared by the programmer. This mechanism is convenient since it allows storage of objects via safe upcast into generic data structure. However it is not expressive enough because the converse process of retrieving objects from the generic data structure requires the programmers to insert explicit type casts for downcast testing at run-time. This results in losing the benefits of static type checking (safety at compile time) and also in incurring the run-time overheads. To address these shortcomings, there have been several recent proposals (amongst the Java [24], C++ templates, and C# [107] communities) for *parametric polymorphism* to be supported. Parametric polymorphism allows *parametric types* and supports *structural subtyping*. Parametric types can be obtained by adding type parameters to class types. In general, type parameters denote the types of the object fields. However structural subtyping has been restricted to invariant subtyping because fields reading and fields writing are based on opposite flows that change the subtyping direction. To address this shortcoming, *variant parametric types* (or VPT, in short) have been developed [104]. Variant parametric types represent a successful result of combining subtype polymorphism with parametric polymorphism to support a more flexible subtyping for the object-oriented paradigm. The key feature of this combination is the *variance*. Depending on how the fields are accessed, each variance denotes a covariant, a contravariant, an invariant, or a bivariant subtyping. Variant parametric types have been adopted in Java 5 [194, 78] under the name *wildcard types* by improving the original VPT proposal [104].

In this context, *our goal was to develop a novel flow-based approach for variant parametric types*. The current model of variant parametric types is based on existential types. We believe that flow analysis is more easy to understand by the programmers and it can also improve the precision of typechecking.

1.3 Our Methodology

We use a common methodology to accomplish our goals. *Our methodology is designed for type-based value flow analyses which are performed on a Java-like object-oriented language*. This section presents the main aspects of our approach and concludes with our methodology's key steps.

Our Applications as Type-based Value Flow Analyses. A *value flow analysis* can answer the question "whether any value appearing at a program point, $P1$, flows to another program point, $P2$". In general, a flow analysis assumes that each subexpression e of a program is labeled with a label L. Thus, the above question becomes "whether $L1$ flows into $L2$", where $L1$ and $L2$ are the labels of program subexpressions $e1$ and $e2$, respectively. Moreover, a *type-based value flow analysis* assumes that the subexpressions labels also decorate the program types and therefore it computes the program values flow from the type derivation of the program. More concretely, the possible flow between two subexpression $e1$ and $e2$ is computed by comparing their derived types. However, a type-based flow analysis is not restricted to tracing program points labels, as it can also trace more complex static information over the value flow. The static information can decorate the types generating the *annotated types*. We modeled both our applications as type-based value flow analyses.

The first application, *region analysis*, traces the regions (namely the memory zones where the objects are allocated) throughout the program using the region types associated to each program object. At each program point, it can conservatively compute the set of live regions, namely the memory zones which are still possibly required by the program. The set of live regions is computed by analysing the region type of the program point expression and the region types of the free program variables. The regions that are not live can be deallocated.

The second application, *genericity analysis*, traces the content of the generic data structures throughout the program using the generic types. The analysis can conservatively compute the values which may be read/written from/into each generic data structure. Based on the types of these values, a more precise generic type is computed for the content of each program generic data structure. As a result, a part of the type casts inserted by the programmers (requiring run-time checks) can be proven to be redundant at compile time.

From Annotated Types to Flow (Subtyping) Constraints. Type annotations can take values from a finite or infinite domain (not restricted to atomic properties), e.g. $\{\mathtt{a}_1, \mathtt{a}_2...\}$. The domain is *partially-ordered*, namely there is a reflexive, transitive and anti-symmetric ordering relation (not necessary a lattice) on it. The ordering relation $<:_{\mathtt{a}}$ defines a *subtyping relation* on the annotations, e.g. $\mathtt{a}_1 <:_{\mathtt{a}} \mathtt{a}_2$.

Our type annotation consists in parameterizing a class type with additional annotations. For example, given a class declaration `class Cell` {`Object fst`; }, the annotated class declaration is `class Cell`$\langle\mathtt{a}_1, \mathtt{a}_2\rangle$ {`Object`$\langle\mathtt{a}_2\rangle$ `fst`; }, where $\mathtt{a}_1$ denotes a property of the object, while $\mathtt{a}_2$ denotes a property of the object field `fst`. Note that $\mathtt{a}_1$ and $\mathtt{a}_2$ are annotation variables. Therefore, the annotated class declaration has *polymorphic annotations* such that each instance of that class can use different annotations, e.g. `Cell`$\langle\mathtt{a}_1, \mathtt{a}_2\rangle$, `Cell`$\langle\mathtt{a}_3, \mathtt{a}_4\rangle$. Polymorphic annotations allow us to distinguish unrelated instances of the same class.

Class subtyping is also extended to take into account the annotations. Annotated types subtyping is expressed in terms of *subtyping constraints*. For example, `Object`$\langle\mathtt{a}_1\rangle <:$ `Object`$\langle\mathtt{a}_2\rangle$ holds if $\mathtt{a}_1 <:_{\mathtt{a}} \mathtt{a}_2$ holds. In general, subtyping constraints may contain both annotations and types.

Using subtyping constraints, program value flow can be expressed as an asymmetric relation, namely subtyping can capture not only the flow, but also its direction. For example, given a function of type `Object`$\langle\mathtt{a}_2\rangle \rightarrow$ `Object`$\langle\mathtt{a}_3\rangle$ and an argument of type `Object`$\langle\mathtt{a}_1\rangle$, standard language semantics state there is a flow from the argument to the function's domain, not vice versa. With subtyping, the argument type is a subtype of the domain type, namely `Object`$\langle\mathtt{a}_1\rangle <:$ `Object`$\langle\mathtt{a}_2\rangle$, which in turn is satisfied if $\mathtt{a}_1 <:_{\mathtt{a}} \mathtt{a}_2$. The subtyping constraint $\mathtt{a}_1 <:_{\mathtt{a}} \mathtt{a}_2$ becomes a *flow constraint* expressing that values arising at expressions characterized by the property $\mathtt{a}_1$ flow to expressions characterized by the property $\mathtt{a}_2$. Without using subtyping, value flow is captured as a symmetric relation, meaning that the argument and the function's domain have the same type. If two expressions have the same type, then there is a potential flow from the first expression to the second expression, and also vice versa. Thus, without using subtyping the flow is imprecisely captured.

Modularity. Modularity is admitted to be the key property of a static analysis that allows it to scale to large programs. Another important benefit is that modular analyses support *separate compilation*.

Modularity concept has many different definitions in the literature, this dissertation uses the definition found in [118]: "a static analysis is modular if a program can be decomposed into components (*decomposability*) which are analyzed separately (*understandability*) and whose results can be merged together in order to obtain a result valid for the whole program (*composability*)". In [118] the modularity is defined *at the class level* since that approach looks for the class invariants which are preserved by all class methods. However in our approach we exploit the modularity *at the method level*. Thus we split the class invariant into two parts: one part that has the same role as the class invariant of [118], namely it has to be preserved by each instance of the class and the second part that capture the effect from invoking a method. The second part is contained in the method precondition and has to be preserved only by those class instances which may invoke that method. Given the following class declaration, *class invariant* and *method precondition* are specified after the keyword `where` at the class level and the method level, respectively:

```
class Cell⟨A1,A2⟩ where A1 <:a A2 {
  Object⟨A2⟩ fst;
  void set⟨A1,A2,A3⟩(Object⟨A3⟩ o) where A3 <:a A2 {this.fst=o;} }
```

A class invariant expresses a relation among the class annotations. A method precondition expresses a relation among the method's visible annotations, namely the annotations of the method receiver, method arguments and method result. Method body may contain other annotations for the local declarations, but those are not visible out of the method. Thus, a method precondition is a *polymorphic summary parameterized in terms of the visible annotations*. The visible annotations usually occur as the method's annotations parameters (e.g. $set\langle \mathtt{A}_1, \mathtt{A}_2, \mathtt{A}_4\rangle$).

We adopt a *summary-based* approach in order to have *context-sensitive* analyses. A context-sensitive analysis can distinguish between different calling contexts of a method and does not allow information from one caller to propagate erroneously to another caller of the same method. For example, considering the following code fragment:

$\texttt{Cell}\langle A_1, A_2\rangle\ \texttt{c1; Cell}\langle A_3, A_4\rangle\ \texttt{c2; Object}\langle A_5\rangle\ \texttt{o1; Object}\langle A_6\rangle\ \texttt{o2;}$

$\quad \texttt{...}\quad \texttt{c1.set}\langle A_1', A_2', A_3'\rangle\texttt{(o1);}$

$\texttt{//}A_3' <:_a A_2' \wedge \texttt{Cell}\langle A_1, A_2\rangle <: \texttt{Cell}\langle A_1', A_2'\rangle \wedge \texttt{Object}\langle A_5\rangle <: \texttt{Object}\langle A_3'\rangle$

$\quad \texttt{...}\quad \texttt{c2.set}\langle A_1'', A_2'', A_3''\rangle\texttt{(o2);}$

$\texttt{//}A_3'' <:_a A_2'' \wedge \texttt{Cell}\langle A_1, A_2\rangle <: \texttt{Cell}\langle A_1'', A_2''\rangle \wedge \texttt{Object}\langle A_6\rangle <: \texttt{Object}\langle A_3''\rangle$

The corresponding flow subtyping constraints are marked as comments after each method call. At each call site of the method `set`, the method summary is instantiated with fresh annotation variables. The link between the current call context and the fresh method summary is performed by subtyping. Thus the current types of method receiver and method arguments are subtypes of the formal types of the method receiver and arguments. The formal types are expressed in terms of the fresh annotation variables.

Our *type checking analyses* are designed in a *modular fashion on a per method basis*. The type annotations (including the method preconditions) are provided by the programmer based on the following modularity principle: type annotations appearing in the method header should depend only on the method body, while each call site should be a specific instance of the method's type declaration. This principle is also important for easier understanding of type annotations.

We aim for *interprocedural type inference analyses* that infer all the type annotations including the method's signatures. We design our *type inference analyses as summary-based analyses guided by the dependency graphs*. Each method is analyzed once to produce a polymorphic parameterized summary that can be specialized for use at all of the call sites that may invoke that method. A dependency graph can order the methods such that when a method is analyzed, the summaries of all the methods that it invokes are known.

Simplicity. There is an important distinction between *flow-insensitive* analyses, which tend to be simple and efficient, and *flow-sensitive* analyses, which are more precise but usually do not scale well to large programs. Flow-insensitive analyses can prove properties about a piece of code that are true throughout the whole execution of that code. In contrast, flow-sensitive analyses can prove properties that may change from one program point to another. An analysis is considered to be flow-sensitive or flow-insensitive based on whether or not it takes into account the order of destructive updates.

Flow-insensitive analyses ignore the order of updates and consider all possible interleavings of statements. In addition, the types of values remain the same everywhere in the program. Applying a flow-insensitive analysis on the following code fragment:

```
    //{x:Object<a0>, x1:Object<a1>, x2:Object<a2> }
x=x1; // a1 <:a a0
    //{x:Object<a0>, x1:Object<a1>, x2:Object<a2> }
x=x2; // a2 <:a a0
    //{x:Object<a0>, x1:Object<a1>, x2:Object<a2> }
```

produces two flow constraints (marked as comments after each assignment). Those two flow constraints approximate the possible interleavings of the assignments. As can be seen, the types of x, x1 and x2 are the same before and after each assignment (the types are specified inside the curly brackets).

In contrast, *flow-sensitive* analyses take into account the order of updates and perform strong updates. Applying a flow-sensitive analysis on the same code fragment:

```
    //{x:Object<a0>, x1:Object<a1>, x2:Object<a2> }
x=x1; // a1 <:a a0
    //{x:Object<a1>, x1:Object<a1>, x2:Object<a2> }
x=x2; // a2 <:a a1
    //{x:Object<a2>, x1:Object<a1>, x2:Object<a2> }
```

produce two flow constraints which take into account the fact that the analysis performs strong updating (annotated type of x is changing after each assignment). Modeling strong updates requires must alias information that usually can be computed by complex global analyses.

In general, there are two aspects of the flow: the flow through program variables (shown by above example) and the flow through the object fields. In the case of flow through program variables, flow-insensitive analyses can produce the same results as those of flow-sensitive analyses, if the programs are written in Static Single Assignment (SSA) form.

Our approach employs a simple flow-insensitive analysis to collect the flow constraints and therefore it can avoid the aliasing problem. Another direct consequence is that in our approach the *method precondition holds throughout the whole method execution*, namely it holds at the method entry-point, but also the method exit-point. The method caller must ensure the method precondition at the method entry-point, while the method itself must ensure its precondition at its exit-point. Thus our analyses do not require a separation between a method precondition (holding only at the method entry-point) and a method postcondition (holding only at the method exit-point).

Object-Oriented Features. Three main features characterize object-oriented languages: *class inheritance*, *method overriding*, and *downcasting*.

Class inheritance allows a class to be extended with new features to create a subclass such that the subclass can be used in place of the original class. Thus, the annotated type that corresponds to the subclass should be a subtype of the annotated type corresponding to the original class. In addition, the invariant of each subclass should be a *strengthening* of the parent class' invariant.

Each *overriding method* should be a *subtype* of its overridden method, which means that overridden's method precondition should *imply* the overriding method's precondition [116, 36]. This safety condition may affect the inference analyses. An additional dependency, that indicates that overridden method *depends* on its overriding method, must therefore be added to the dependency graph to guide the inference process. As a consequence, the inference analyses typically require the whole class hierarchy to be known.

In general, a *downcast* operation may be type unsafe if the object in question is not the subtype that was expected. For the case of the annotated types, a downcast may also be unsafe because the actual annotations of the object in question are not in subtype relation with those annotations which were expected.

Key Steps. Since our type-based flow analyses eventually produce and solve flow subtyping constraints, we can regard them as *constraint-based analyses*. Aiken [6] defines a constraint-based analysis as consisting of two parts: *constraint generation*, that is the analysis specification, and *constraint resolution*, that is the analysis implementation. We use a similar approach, but we focus more on the analysis specification part defining the following key steps:

1. design the semantics and domain of type annotations,

2. design the ordering relation on the type annotations domain (defining the annotations subtyping relation),

3. design the rules to annotate the types,

4. design the subtyping rules of the annotated types,
5. design the flow (subtyping) constraints language,
6. design the simplification rules of the flow (subtyping) constraints,
7. design the type system (type checking) rules, and
8. design the type inference rules.

Since the type system is the target of the inference algorithm, the type checking rules are always defined first. In addition, we use the type checking system to help prove the correctness of the inference algorithm, and validate its execution runs.

1.4 Technical Contributions

This dissertation is based on the materials published in [40, 39, 46, 45, 47] and it makes two main technical contributions which are highlighted below:

1. **A Region Type Inference System for a Java-like Object-Oriented Language**

   - **Region Type System:** We have formulated and implemented a region type system as a target for our region type inference. The region type system guarantees that well-typed programs use lexically scoped regions and do not create dangling references in the store and on the stack. Although our type system is similar to SafeJava's type system of Boyapati et al. [23], there are three main differences: (1) we isolated the object encapsulation issue in our type system, (2) we added support for region subtyping by adapting the region subtyping principle from Cyclone [80], and (3) we provided a rigorous soundness proof for our region type system (note that SafeJava does not provide a formal proof for its region type system).
   - **Region Type Inference:** We have formulated and implemented the *first region type inference system for a Java-like object-oriented language*. Our inference analysis is designed as a summary-based flow-insensitive analysis that automatically infers all the region annotations of the classes and methods. To provide an inference algorithm that is both precise and practical, we support classes and methods that are region-polymorphic, with region-polymorphic recursion for methods. Object-oriented features such as class inheritance, method overriding, and downcast operations are fully

handled by our analysis. We have also proven that our region type inference algorithm is correct with respect to our region type system.

- **Experimental Validation:** We have implemented a prototype of our region inference system and we have run some experiments on medium-sized benchmarks. Preliminary results that we have obtained are encouraging. The programs based on our inferred regions were able to reuse significant amount of memory for most of the cases where data was not live throughout the execution. The experiments suggest that our results are competitive when compared to those that are hand annotated by human experts, and comparable also to the approach based on non-lexically scoped regions with no-dangling-access [37]. The experiments also suggest that our region inference analysis is fast in terms of analysis time and reasonable with respect to the number of region parameters.

2. **A Flow-based Approach for the Variant Parametric Types**

- **Flow-based Approach**: Our framework is based on a value flow analysis which can concisely and intuitively capture flow of values on a per method basis. We use variance annotations primarily to predict the flows of values, and not for access control. In contrast, the existing approaches [103, 193] view variant parametric type system as a special case of the existential type system with subtyping.
- **Modular Type Checking**: Each method is specified with a flow constraint (and variant parametric types) that is used to predict the value flows that may occur in the method's body. We verify each method separately to ensure that the predicted accesses, flow constraint and variant parametric typings are efficiently and safely checked. In contrast, the existing approaches [103, 193] use a typechecking per class approach rather than a per method approach.
- **Casting and Cast Capture**: Our system supports full casting for variant parametric types. In contrast, Java 1.5 restricts the downcast mechanism to the outer type constructor [128]. We also advocate a novel cast capture mechanism, that uses reflection technique to handle objects with unknown types in a type-safe way. Cast capture mechanism help us obtain more precise generic typings for several JDK 1.5 libraries.

- **Experimental Validation**: We have implemented a prototype of our variant parametric type checker and we have run the experiments on a suite of Java libraries and some large-sized Java applications. The experiments suggest that more downcasts can be eliminated by our approach, even when it is compared against the state-of-the-art type system from Java 1.5. On average, we are able to eliminate 87.9% of the casts from non-generic Java 1.4 application code, that means 12.9% more casts than wildcard-generic Java 1.5 application code.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows.

Chapter 2 provides basic background about the underlying technologies of our work: type systems, type-based flow analyses, and flow subtyping constraints. It also introduces a core object-oriented Java-like language, called Core-Java on top of which we have developed our work.

Part I of our dissertation, consisting of Chapter 3 and Chapter 4, presents our first application, a safe region-based memory management for a Java-like language. Chapter 3 introduces the main concepts and formalizes our region type system. Chapter 4 presents our region inference, the experimental results and concludes with a discussion of related work.

Part II of our dissertation, consisting of Chapter 5, presents our second application, a better genericity for a Java-like language. Chapter 5 presents our flow-based approach for typechecking variant parametric types, the experimental results and concludes with a discussion of related work.

Part III of our dissertation, consisting of Chapter 6, concludes the dissertation and also discusses some perspectives for future work.

CHAPTER 2

UNDERLYING TECHNOLOGIES

In this chapter we provide a brief coverage of the underlying technologies used in our work: type systems, type-based flow analyses, and flow subtyping constraints. Section 2.1 provides some basic background on type system and introduces a standard type system for a core object-oriented Java-like language (called Core-Java). Section 2.2 provides a background on type-based flow analyses and illustrates the main concepts using our two applications. Section 2.3 provides a background on flow subtyping constraints solving.

2.1 Standard Type Systems

Type systems for programming languages are designed to provide several important functions:

- *Safety*: The main purpose of a type system is the prevention of run-time errors when executing a program. Type systems are used to distinguish between well-typed and ill-typed programs. This can be summarized by Milner's famous slogan: *Well-typed programs cannot go wrong* [121].

- *Optimization*: A type system can provide additional information to a compiler in order to support various optimizations (e.g. make runtime testing unnecessary).

- *Documentation*: Type annotations can be used as a form of documentation.

- *Abstraction*: Types force programmers to think at a higher level of abstraction in programming.

Languages like Haskell, ML, Java, C++, and C# are *typed languages* since the program variables can be given types. Typed languages may enforce *static checking* by rejecting all programs that are potentially unsafe at compile time. In contrast untyped languages like Lisp may enforce *dynamic checking* by performing run-time checks. A language is type *sound* if any given well-typed program does not produce a run-time error. Therefore a type sound language does not require run-time checks. Since type systems are not expressive enough to capture all kinds of properties, typed languages may also use a mixture of run-time and static checks. For

P	$::= \mathit{def}^*$	(**program**)
def	$::=$ **class** cn_1 **extends** cn_2 **implements** cn^*	(**class decl**)
	$\{(\tau\ f)^*\ \mathit{meth}^*\}$	(**class body**)
	$\mid$ **interface** cn_1 **extends** cn_2	(**interface decl**)
	$\{(\tau\ mn((\tau\ v)^*)$ **throws** $cn^*\ \{\})^*\}$	(**interface body**)
prim	$::=$ **int** $\mid$ **boolean** $\mid$ **void**	(**primitive type**)
τ	$::=\ cn \mid \mathit{prim} \mid \perp$	(**type**)
meth	$::= \tau\ mn((\tau\ v)^*)$ **throws** $cn^*\ \{e\}$	(**method decl**)
lhs	$::= v \mid v.f$	(**location**)
e	$::=$ **null** $\mid \mathit{lhs} \mid k$	(**expression**)
	$\mid \{(\tau\ v)\ e\}$	(**block decl**)
	$\mid$ **new** $cn(v^*) \mid \mathit{lhs} = e$	
	$\mid v.mn(v^*) \mid e_1\ ;\ e_2 \mid (cn)v$	
	$\mid$ **if** v **then** e_1 **else** e_2 $\mid$ **while** $v\ e$	
	$\mid$ **throw** v $\mid$ **try** e **catch** $(c\ v\ e)$	

$cn \in$ *class/interface names* $\quad mn \in$ *method names*
$f \in$ *field names* $\quad v \in$ *variable names*
$k \in$ *integer or boolean constants*

Figure 2.1: The Syntax of Core-Java

example, Java requires run-time checks for the cast operations. More technical issues that arise from the study of type systems can be found in [32, 151, 26].

In this dissertation we explore *object-oriented* Java-like languages. Figure 2.1 shows the syntax of our core object-oriented language, called Core-Java. Core-Java is designed in the same minimalist spirit as the pure functional calculus Featherweight Java [102], but it supports imperative features (assignments). In contrast to the other imperative calculi for Java (e.g. Middleweight Java [15]), Core-Java does not allow statements, remaining an expression-oriented calculus. The expression-oriented calculi are more suitable for the type-based analyses, since they make easier the formulation of the static and dynamic semantics. The full syntax of Core-Java and the translation rules of Java programs into Core-Java programs are given in [45]. We use the following Core-Java example:

```
class Cell extends Object {
        Object fst;
        Object getFst() {fst}
        void set(Object o) {fst=o}
}
class Pair extends Cell {
        Object snd;
        Object getSnd() {snd}
```

$$\frac{\begin{array}{c}\textbf{class}\ \mathit{cn}\ \textbf{extends}\ \mathit{cn}'\ \textbf{implements}\ \mathit{cn}_1..\mathit{cn}_k \cdots \in P \\ P \vdash \mathit{cn}' {<:} \mathit{cn}'' \vee P \vdash \mathit{cn}_1 {<:} \mathit{cn}'' \vee \ldots \vee P \vdash \mathit{cn}_k {<:} \mathit{cn}''\end{array}}{P \vdash \mathit{cn} {<:} \mathit{cn}''}\ [\textbf{SubClass}] \qquad \frac{}{P \vdash \bot {<:} \mathit{cn}}\ [\textbf{Bottom}]$$

$$\frac{}{P \vdash \mathit{cn} {<:} \mathit{Object}}\ [\textbf{Top}] \qquad \frac{}{P \vdash \tau {<:} \tau}\ [\textbf{Reflexivity}] \qquad \frac{P \vdash \tau_1 {<:} \tau_2 \quad P \vdash \tau_2 {<:} \tau_3}{P \vdash \tau_1 {<:} \tau_3}\ [\textbf{Transitivity}]$$

Figure 2.2: Subtyping Rules

```
        void set(Object o) {fst=o;snd=o}
}
```

to illustrate some of the key features of the object-oriented languages as follows:

- *class-based languages*: A class forms a template for the generation of new objects. It consists of fields and methods. A new object is created by **new** expression that invokes a constructor. A field is accessed using an expression of the form `v.f` where `v` denotes an object and `f` is a field name. To invoke a method, an expression of the form `v.mn(`v^*`)` is used, where `v` denotes an object.

- *inheritance allows reuse of implementation*: Each class declaration specifies its *superclass* after the keyword `extends`. The class `Pair`, called *subclass* of the class `Cell`, *inherits* `Cell`'s definitions of the fields (e.g. `fst`) and methods (e.g. `getFst`). A subclass can also *override* an inherited method definition. For instance the class `Pair` overrides the method `set` of the class `Cell`.

- *types and subtyping*: Each class declaration introduces a new type of the same name as the class. For example, objects instantiated from class `Cell` belong to the type `Cell`. The subclass relations induces a subtyping relation. For instance `Cell` is a *supertype* of `Pair`, and, conversely, `Pair` is a *subtype* of `Cell`. The class `Object` serves as the top type, which is the supertype of all types, while type $\bot$ is the subtype of all types. Subtyping guarantees the *principle of safe substitution* [115]: if `S` is a subtype of `T` then any expression of type `S` can be safely used in any context that expects an expression of type `T`. For example, considering the expression `v.set(o)` where the variable `v` is

$$\frac{\begin{array}{c}[\underline{\textbf{PROG}}]\\ \mathit{WFClasses}(P) \quad P = \mathit{def}_{i:1..n} \quad \mathit{FieldsOnce}(\mathit{def})_{i:1..n} \ \ \mathit{MethodsOnce}(\mathit{def})_{i:1..n} \\ P \vdash \mathit{InheritanceOK}(\mathit{def})_{i:1..n} \ \ P \vdash_{\mathit{def}} \mathit{def}_{i:1..n}\end{array}}{\vdash P}$$

$$\frac{\begin{array}{c}[\underline{\textbf{CLASS}}]\\ \mathit{def}{=}\textbf{class}\ \mathit{cn}\ \textbf{extends}\ c\ \textbf{implements}\ c_1..c_n\ \{\mathit{field}_{1..p}\ \mathit{meth}_{1..q}\} \\ P \vdash \mathit{InterfaceOK}(c_i, \{\mathit{meth}_1, .., \mathit{meth}_q\}) \ \ i = 1..n \\ P; \{\mathit{this} : \mathit{cn}\} \vdash_{\mathit{meth}} \mathit{meth}_i \ \ i = 1..q\end{array}}{P \vdash_{\mathit{def}} \mathit{def}}$$

$$\frac{\begin{array}{c}[\underline{\textbf{METH}}]\\ P; \Gamma + (v_j : \tau_j)_{j:1..p} \vdash e : \tau_0\end{array}}{P; \Gamma \vdash_{\mathit{meth}} \tau_0\ \mathit{mn}((\tau_j\ v_j)_{j:1..p})\{e\}} \qquad \frac{\begin{array}{c}[\underline{\textbf{BLOCK}}]\\ P; \Gamma + (v : \tau') \vdash e : \tau\end{array}}{P; \Gamma \vdash \{(\tau'\ v)\ e\} : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{NULL}}]\\ P \vdash \bot{<:}\tau\end{array}}{P; \Gamma; R; \varphi \vdash \textbf{null} : \tau}$$

$$\frac{\begin{array}{c}[\underline{\textbf{VAR}}]\\ (v : \tau') \in \Gamma \\ P \vdash \tau'{<:}\tau\end{array}}{P; \Gamma \vdash v : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{FIELD}}]\\ (v : \mathit{cn}) \in \Gamma \\ (\tau'\ f) \in \mathit{fieldlist}(P, \mathit{cn}) \ \ P \vdash \tau'{<:}\tau\end{array}}{P; \Gamma \vdash v.f : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{RC-NEW}}]\\ P \vdash \mathit{cn}{<:}\tau \ \ \mathit{fieldlist}(P, \mathit{cn}) = (\tau_i\ f_i)_{i:1..p} \\ (v_i : \tau_i') \in \Gamma \ \ P \vdash \tau_i' <: \tau_i \ i = 1..p\end{array}}{P; \Gamma \vdash \textbf{new}\ \mathit{cn}(v_1, .., v_p) : \tau}$$

$$\frac{\begin{array}{c}[\underline{\textbf{ASSIGN}}]\\ P; \Gamma \vdash_i \mathit{lhs} : \tau \quad P; \Gamma \vdash e : \tau\end{array}}{P; \Gamma \vdash \mathit{lhs} = e : \textbf{void}} \qquad \frac{\begin{array}{c}[\underline{\textbf{GET-VAR}}]\\ (v : \tau) \in \Gamma\end{array}}{P; \Gamma \vdash_i v : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{GET-FIELD}}]\\ (v : \mathit{cn}) \in \Gamma \quad (\tau\ f) \in \mathit{fieldlist}(P, \mathit{cn})\end{array}}{P; \Gamma \vdash_i v.f : \tau}$$

$$\frac{\begin{array}{c}[\underline{\textbf{CAST}}]\\ P \vdash \mathit{cn}{<:}\tau\end{array}}{P; \Gamma \vdash (\mathit{cn})v : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{SEQ}}]\\ P; \Gamma \vdash e_1 : \mathit{Object} \quad P; \Gamma \vdash e_2 : \tau\end{array}}{P; \Gamma \vdash e_1\textbf{;}e_2 : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{LOOP}}]\\ \Gamma \vdash v : \textbf{boolean} \quad P; \Gamma \vdash e : \textbf{void}\end{array}}{P; \Gamma \vdash \textbf{while}\ v\ e : \textbf{void}}$$

$$\frac{\begin{array}{c}[\underline{\textbf{IF}}]\\ \Gamma \vdash v : \textbf{boolean} \\ P; \Gamma \vdash e_1 : \tau \quad P; \Gamma \vdash e_2 : \tau\end{array}}{P; \Gamma \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau} \qquad \frac{\begin{array}{c}[\underline{\textbf{INVOKE}}]\\ (v_0 : \mathit{cn}){\in}\Gamma \quad P{\vdash}(\tau'\ \mathit{mn}((\tau_i\ v_i)_{i:1..n})\ \{e\}) \in \mathit{cn} \\ (v_i' : \tau_i') \in \Gamma \quad P \vdash \tau_i'{<:}\tau_i \quad i = 1..n \quad P \vdash \tau'{<:}\tau\end{array}}{P; \Gamma \vdash v_0.\mathit{mn}(v_1'..v_n') : \tau}$$

Figure 2.3: A fragment of the Type Rules

given the type `Cell`. The variable `v` can be replaced by an object either of type `Cell` or type `Pair` and the method invocation is correctly executed. It depends on the run-time type of the object which method `set` (either from class `Cell` or `Pair`) is executed. This mechanism is called *dynamic dispatch*. However the subtyping based on subclass is not flexible. For example if an object of a class `Triple` has the method `set`, it is not allowed to substitute the object for `v` unless `Triple` is a subclass of `Cell`. To improve flexibility, Java has introduced *interfaces*. Core-Java supports *multiple inheritance* through interfaces in the same restricted way as that supported by the Java language. Each class may extend from only a single superclass but may implement multiple interfaces.

The type system of Core-Java consists of the following main *judgments*:

$$\frac{P \vdash mbr \in_D cn}{P \vdash mbr \in cn} \qquad \frac{\begin{array}{c} mbr{=}field|meth \\ \textbf{class}\ cn...\{...mbr...\}{\in}P \end{array}}{P \vdash mbr \in_D cn} \qquad \frac{\begin{array}{c} \textbf{class}\ cn\ \textbf{extends}\ cn'...{\in}P \\ P{\vdash}mbr{\in}cn' \quad \neg(P{\vdash}mbr{\in_D}cn) \end{array}}{P \vdash mbr \in cn}$$

$$\frac{}{fieldlist(P, Object){=_{def}}[\,]} \qquad \frac{\textbf{class}\ cn_1 \textbf{extends}\ cn_2..\{(\tau_i\ f_i)_{i:1..p}..\}{\in}P}{fieldlist(P, cn_1){=_{def}}fieldlist(P, cn_2){+\!\!+}[(\tau_i)\ f_i]_{i=1}^{p}}$$

$$\frac{\begin{array}{c} P{=}def_{1..n} \qquad def_i{=}\textbf{class}\ cn_i\ \textbf{extends}\ cn_{i'}... \\ IR{=}\{(cn_i, cn_{i'}) \mid 1{\le}i{\le}n\} \quad ID{=}\{(cn_i, cn_i) \mid 1{\le}i{\le}n\} \\ TransClosure(IR){\cap}ID{=}\emptyset \quad \forall i, j{:}i{\ne}j \cdot cn_i{\ne}cn_j \end{array}}{WFClasses(P)} \qquad \frac{\begin{array}{c} def{=}\textbf{class}\ cn...\{(fd_j)_{j:1..p}...\} \\ \forall j, l{:}j{\ne}l \cdot name(fd_j){\ne}name(fd_l) \end{array}}{FieldsOnce(def)}$$

$$\frac{\begin{array}{c} def{=}\textbf{class}\ cn...\{...(m_j)_{j:1..q}\} \\ \forall j, l{:}j{\ne}l{\cdot}name(m_j){\ne}name(m_l) \end{array}}{MethodsOnce(def)} \qquad \frac{\begin{array}{c} def{=}\textbf{class}\ cn\ \textbf{extends}\ cn'...\ \{fd_{1..p}\ meth_{1..q}\} \\ \forall j{\in}1..q{\cdot}\exists meth'{\cdot}P{\vdash}meth'{\in}cn'{\wedge}name(meth'){=}name(meth_j) \\ {\Rightarrow}(P{\vdash}OverridesOK(meth_j, meth')) \end{array}}{P \vdash InheritanceOK(def)}$$

$$\frac{\begin{array}{c} \textbf{interface}\ c\ \textbf{extends}\ ..\ \{meth'_{1..n}\} \\ \{meth'_{1..n}\} \subset \{meth_1, .., meth_q\} \end{array}}{P \vdash InterfaceOK(c, \{meth_1, .., meth_q\})} \qquad \frac{\begin{array}{c} meth = \tau_0\, mn((\tau_i\ v_i)_{i:1..m})... \\ meth' = \tau_0'\, mn((\tau_i\ v_i)_{i:1..m})... \\ P \vdash \tau_0{<:}\tau_0' \end{array}}{P \vdash OverridesOK(meth, meth')}$$

Figure 2.4: A fragment of the Auxiliary Type Rules

- $P \vdash \tau_1{<:}\tau_2$ is the *subtyping judgment* denoting that the type τ_1 is a subtype of the type τ_2 with respect to the program *P*. In our type systems the program *P* is regarded as a class table that contains all the class definitions. *Subtyping relation* of class types is defined in Figure 2.2 as a reflexive and transitive relation.

- $\vdash P$ denoting that a program *P* is well-typed. The type rule [PROG] of Figure 2.3 asserts the validity of this judgment. The predicates (defined in Figure 2.4) in the rule premise are used to capture the standard well-formedness conditions for the object-oriented programs (such as no duplicate definitions of classes, no cycle in the class hierarchy, no duplicate definitions of fields, no duplicate definitions of methods).

- $P \vdash_{def} def$ denoting that a class declaration *def* is well-typed. The type rule [CLASS] of Figure 2.3 asserts the validity of this judgment.

- $P; \Gamma \vdash_{meth} meth$ denoting that a method *meth* is well-typed with respect to the program *P*, and the type environment Γ. The type rule [METH] of Figure 2.3 asserts the validity of this judgment.

- $P; \Gamma \vdash e : \tau$ denoting that the type τ is the expected type of the expression e with respect to the program *P*, and the type environment Γ. Validity of this judgment is defined by the rules of Figure 2.3. These rules are *type checking rules* which verify whether the given type τ is a valid type for the expression e with respect to the program *P*, and the type environment Γ.

- $P;\Gamma \vdash_i \mathit{lhs} : \tau$ denoting that the type τ is the derived type of the expression lhs with respect to the program *P*, and the type environment Γ. Validity of this judgment is defined by the rules $[\textsc{get-var}]$ and $[\textsc{get-field}]$ of Figure 2.3. These two rules are *type inference rules* which derive a valid type τ for the expression lhs with respect to the program *P*, and the type environment Γ.

Figure 2.4 shows the method *overriding rule* adopted in Java, where the overriding method $meth$ and the overridden method $meth'$ have the same types for their parameters, while the type of the overriding method result is a subtype of the type of the overridden method result. However, our advanced type systems described in this dissertation use a more general rule that requires the overriding method to be a subtype of the overridden method. As proven in [36] for an object-oriented language, the function subtyping is sound if the parameters (the receiver) that drive dynamic method selection are covariant, the normal parameters are contra-variant, and the result is covariant. In general, the function subtyping rule requires that all the parameters are *contra-variant*, and the result is *covariant*, as follows:

$$\frac{\vdash \tau_1' <: \tau_1 \quad \vdash \tau_2 <: \tau_2'}{\vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Given an unary type constructor F, the covariant subtyping, contra-variant subtyping, and invariant subtyping are defined as follows:

$$\underline{[\textbf{Covariant}]} \qquad \underline{[\textbf{Contra-variant}]} \qquad \underline{[\textbf{Invariant}]}$$

$$\frac{\vdash \tau_1 <: \tau_2}{\vdash F(\tau_1) <: F(\tau_2)} \qquad \frac{\vdash \tau_2 <: \tau_1}{\vdash F(\tau_1) <: F(\tau_2)} \qquad \frac{\vdash \tau_1 <: \tau_2 \quad \vdash \tau_2 <: \tau_1}{\vdash F(\tau_1) <: F(\tau_2)}$$

In general covariant subtyping is used for reading, contra-variant subtyping is used for writing, while invariant subtyping is used for both reading and writing.

Another important feature of an object-oriented language is exception handling. This feature is used to handle unusual conditions that may lead to errors, unless some remedial actions are taken. In Core-Java an exception may be generated either by `throw` expression or by calling a method that is supposed to throw exceptions and it is handled by `try..catch` expression. To manage the different categories of flow, the type rules are extended to a pair of types, *(normal execution type, exceptional type)* similar with [55] to represent the type of an expression:

$P; \Gamma \vdash e : \tau_n \# \tau_a$, where τ_n is the *normal type* that characterizes normal execution of the expression and τ_a that is the *exceptional type* that characterizes the exceptional execution of e.

In our dissertation we prove the *soundness* of a type system using the proof techniques from [204, 151] based on an *operational semantics*. The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. The soundness theorem consists of two properties that make a strong connection between static semantics (type system) and dynamic semantics (operational semantics):

- *Type Preservation or Subject Reduction* ensures that the well-typedness of a program is preserved under the evaluation rules of the language.
- *Progress* ensures that a well-typed program never gets stuck, that means it never gets into a state where no further evaluation rules are possible.

Note that well-typedness is related to the type system, while getting stuck is a property of the operational semantics.

Another important issue of a type system is the *type inference*. The type checking rules shown in Figure 2.3 depend on the explicit type annotations of the variable and method declarations. Type inference is the problem of finding a type for an expression within a given type system, when the type environment is given. The most general type that can be found, if any, is called *principal type*. Type inference is *sound* if the derived type is a valid type for the given expression with respect to the given type system. Whenever there is a type for the given expression with respect to a given type system, its corresponding type inference algorithm is said to be *complete* if it can derive that type. *Type reconstruction* consists in starting with an untyped expression and computing a type environment, a type annotated version of that expression, and a type for the annotated expression with respect to the computed type environment. The solution that imposes minimal assumptions on the free variables of the given untyped expression is called *principal typings*. In the presence of subtyping and polymorphism, type inference is either difficult [8, 135, 105, 66, 59, 195, 157] or even undecidable [201, 108, 92].

Traditionally, most mainstream object-oriented languages such as Java, C++ and C#, have provided only *inclusion (or subtyping) polymorphism* supported by class inheritance. While this mechanism allows the convenient storage of objects via safe upcast into generic data structures, the converse process of retrieving objects from the same data structure requires downcast testing, which incurs run-time overheads and is possibly unsafe. For example, an `Integer` object can be safely stored in the field `fst` (of type `Object`) of a `Cell` object, as follows:

```
Integer example(Cell cell, Integer a){
        Integer b;
        cell.fst=a; //safe upcast
        b=(Integer) cell.fst; //explicit downcast
        b }
```

However the field `fst` can only be read as an object of the same type as `fst`'s type, namely `Object`. Therefore an explicit downcast to `Integer` is required. Note that this cast cannot be checked by the type system (see rule [CAST] from Figure 2.3). This check is instead postponed to run time.

To address the shortcomings of inclusion polymorphism, there have been several recent proposals (amongst the Java [24] and C# [107] communities) for parametric types to be supported. Here, each class is allowed to carry a list of type parameters for its fields:

```
  class Cell⟨A⟩ { A fst; ...}
```

Each class type parameter `A` can either be instantiated or left as a *type variable*. With such parameterized class declarations, we may then define specialized instances, such as `Cell⟨Integer⟩` or `Cell⟨Float⟩`, which contain more specific type information for the fields of each class instance. Thus the explicit downcast of the previous example becomes redundant:

```
A example⟨A⟩(Cell⟨A⟩ cell, A a){
        A b;
        cell.fst=a; //safe
        b=cell.fst; //safe
        b }
```

A method which declares a type variable in its signature is called *generic method*. When a generic method is invoked, it requires type parameters to be provided (e.g. `example⟨Integer⟩ (cell,a)`).

Though parametric types can coexist with class subtyping, an invariant subtyping is required for the type parameters. For example, the subtyping relation `Cell⟨t1⟩ <:Cell⟨t2⟩` is allowed only when `t1=t2`. Invariant subtyping is required because field reading and field writing are based on opposite flows that change the directions of the subtyping. This requirement limits the re-usability of programs based on parametric types. In the second part of the dissertation (starting with Chapter 5) we present advanced techniques that allow a more flexible subtyping.

Existential types can also be used for object encoding to hide the types of object states [25, 2, 44, 153]. In general (bounded) existential types represent a type-theoretic basis of abstract data types [123, 33]. An existential type is syntactically a type of the form $\exists X.T$, with the existential quantifier on a type variable X. Since X is regarding as something unknown, existential types can be used to hide some information (encapsulation of the abstract data types implementation). A value of an existential type $\exists X.T$ is constructed by a pair of a type U and a value v of type $[U/X]T$ (type T where U is substituted for the type variable X). Such a pair is often written $\texttt{pack}\ [U, v]\ \texttt{as}\ \exists X.T$. Since U witnesses the existence of X, U is called a witness type. A value of an existential type can be used by an expression of the form $\texttt{open}\ p\ \texttt{as}\ [X, x]\ \texttt{in}\ e$. It unpacks a package p, binds the type variable X and the value variable x to the witness type and the implementation, respectively, and evaluates e. Bounded existential types [33] allow existential type variables to have upper bounds. For instance the type $\exists X{<:}S.T$ means the type T where X is some subtype of S. Bounded existential types correspond to abstract types, where partial information of the implementation type is available. Subtyping of bounded existential types is defined as follows:

$$\frac{\vdash S_1{<:}S_2 \quad X <: S_1 \vdash T_1{<:}T_2}{\vdash \exists X{<:}S_1.T_1 <: \exists X{<:}S_2.T_2}$$

2.2 From Type Systems to Flow Analyses

Type-based analysis is an approach to static analysis of programs that assumes that the programs are well-typed [132, 141]. Type-based analyses provide a natural separation between the specification given by the type system and the implementation of the analysis. The types serve as an infrastructure on top of which more complicated but efficient program analyses can be built. Standard techniques from type theory can be applied to reason about the soundness and completeness of the analyses.

Type-based analyses (and in general program analyses) require information about the possible *flow of data* within the program and the possible *control paths* through the program. These two kinds of information are computed by *value flow analysis*. Therefore, some flow analysis is at the conceptual and technical core of most of the type-based analyses.

Flow analysis considers a value generated or constructed at some program point, traces its flow through the program, and computes all the places where it may be used or deconstructed. The values can be any kind of data: atomic data such as integers, structured data such as records, or higher-order data such as function closures. The flow analysis must be *sound*: whenever a value flow exists from a program point to another, the analysis must predict this. However the analysis is not necessary *complete*: it may predict spurious flows from a program point to another, which do not exist at execution time. An exact flow analysis that is formulated as a decision problem is undecidable by Rice's theorem [166].

Flow analysis for primitive values, called *data flow analysis*, has been used from the early years of compilers [5]. Reynolds was first to study a flow analysis for records and tuples, calling it *data set analysis* [165]. A similar flow analysis for structured values, called *value flow analysis* was developed later by Schwartz [171]. Flow analysis for function closures has been developed by Sestoft [173, 172] and Shivers [174, 175]. Sestoft has called it *closure analysis*, while Shivers has called it *control flow analysis*. Shivers has also introduced a hierarchy of *kCFA* of *polyvariant flow analyses*. Polyvariance allows several descriptions for a definition, one for each context in which it is used. *Polymorphic flow analysis* was developed by Dussart, Henglein, and Mossin in [58, 95, 126]. Palsberg and Schwartzbach [145, 146, 144] have introduced flow analysis for object-oriented languages. In imperative languages, flow analysis was studied by Horowitz, Reps and Sagiv [164, 168].

The equivalence between a type system and a flow analysis has been investigated by Palsberg and O'Keefe [142] and Heintze [87]. Palsberg and O'Keefe have studied which type information could be inferred from flow information. They have proven the equivalence between a monomorphic flow analysis and a type system with recursive subtyping. From the other direction, Heintze has studied which flow information could be inferred from a type derivation. In essence, both approaches [142, 87] have proven the equivalence between a constraint based analysis and a subtype based analysis.

In our dissertation, we adopted the approach of type-based flow analysis and we extended the expressiveness of a type system by *annotating* the standard types with extra static information. The static information is referred either as a *flow label* or as a *flow property* in [126], as a *type qualifier* in [67, 68], or as an *annotation*. A classical example of annotations comes from binding-time analysis [58] that uses two type annotations: *static* denoting values known at compile time and *dynamic* denoting values which may not be known until run-time.

The approach taken in our flow-based type systems is similar to Foster's *flow-insensitive type qualifiers* [67] and Solberg's type annotations [177]. However, our annotations are not restricted to atomic properties. In addition, we consider our annotations more suitable for the object-oriented languages. The annotations are interpreted operationally as tags for the objects. An object tag denotes a property of the object (including its fields). Our type systems model the *flow of the annotations* through a program in order to estimate the program objects properties at compile time. Type annotations are related to each other by a *partial order* [50], that allows a *subtyping relation over annotations*. This allows a greater precision of the analysis since the subtyping relation can produce constraints rather than equalities.

Our type-based analyses are based on the type checking and type inference rules of our advanced type systems. Both type rules eventually produce flow subtyping constraints. Our approach is similar to constraint-based approach for flow analyses, introduced by Palsberg in [139]. However, we capture the flow of values on *a per method basis* rather than for the entire program. Intuitively, our type checking process starts with a derivation tree of a Core-Java method, where all annotated types (including the method precondition) are given by the programmer. Using the method precondition and the annotated types of the method signature (namely the annotated types of the method receiver, method arguments and method result), the type checking rules verify the annotated types of each method body subexpression. In contrast, our inference process starts with a Core-Java method, where all types are annotated with fresh annotation variables, that each occurs only once. The method precondition is unknown at the beginning. The type inference rules collect a set of flow subtyping constraints by analysing each method body subexpression. This constraint set represents the *principal flow annotation* that gives the most general description of the method. However, as in [126], we are interested to find the *minimal principal flow annotation* that corresponds to solving all the local flow information that does not depend on input or free variables. In our case, this corresponds to the inference of the method precondition by localizing all the annotation variables which do not occur in the annotated types of the method signature (namely method receiver, method arguments and method results).

In our approach an annotated class declaration may also contain a class invariant, that expresses in terms of flow subtyping constraints a safety condition that has to be preserved by each instance of that class. It can also be regarded as a well-formed condition of the annotated type.

Our first application, described in Part I, is based on a *region type system*. We construct region types by adding *polymorphic region annotations* directly to the standard *monomorphic types* of Core-Java (Figure 2.1), without changing the structure of the underlying Core-Java type system. The general form of a region type is $\texttt{cn}\langle\texttt{r1..rn}\rangle$, where `cn` is a class name and the annotations `r1..rn` are region variables. The first region variable `r1` is used to store the object itself, while the rest of the region variables `r2..rn` are used to store the object fields. At run-time the region variables are instantiated with memory regions.

Memory is organized as a stack of memory regions, on which the memory regions are allocated and deallocated (Figure 3.3). The stack induces an ordering relation among the memory regions lifetimes such that the memory regions with longer lifetimes (older regions) are allocated at the bottom of the stack, while the memory regions with shorter lifetimes (younger regions) are at the top of the stack. At static time, we use an outlive relation among region variables, denoted by $\succeq$, to model the runtime ordering relation among memory regions, such that $\texttt{r1}\succeq\texttt{r2}$ means that the region variable `r1` denotes a memory region whose lifetime is not shorter than the lifetime of the memory region denoted by the region variable `r2`. In addition, our programs use lexically scoped region variables.

The region subtyping principle is based on the outlive relation, as follows: wherever a region is expected, it is always safe to provide a region with a longer or equal lifetime. This principle is used to define the following region type subtyping relation: $\texttt{cn}\langle\texttt{r1..rn}\rangle <:\texttt{cn}\langle\texttt{r1}'\texttt{..rn}'\rangle$ holds if $\texttt{r1}\succeq\texttt{r1}'$ and $\texttt{r2=r2}'\texttt{,..,rn=rn}'$ hold. Since the first region is reserved exclusively for the object itself, we can use region subtyping for it. However, the object fields are mutable and therefore an invariant subtyping is required for their regions.

In summary, in the case of region types, the flow subtyping constraints denote the relations among the region lifetimes. A class invariant expresses a no-dangling reference requirement, that ensures that each class object never references another object stored in a region with a shorter lifetime. A method precondition expresses the outlive relations among the method signature regions (namely the regions which annotate the method receiver, method arguments and method result). Method body may allocate and deallocate local regions, but the only non-local regions that it can used are those occurring in the method signature. Therefore the method precondition reflects how the method body uses non-local regions, namely it specifies how non-local regions must be organized on the region stack before the method execution. This region stack organization remains the same after the method execution, since we use lexically scoped regions. However some of the method signature regions may contain additional objects, allocated during the method execution. The region type checking rules ensure that the regions are properly used without creating dangling references. The type inference rules localize the regions which are no longer required (namely there is not any reference to them from the stack and from the other regions). In case of typechecking, the region localization is done by the programmer.

Our second application, described in Part II, is based on a *variant parametric type system*. Variant parametric types can be obtained from Core-Java standard types (Figure 2.1) in two steps. The first step translates Core-Java monomorphic types into *parametric types*, as we illustrated at the end of Section 2.1. The general form of a parametric type is $\mathtt{cn}\langle T_1 \mathinner{..} T_n\rangle$, where the annotations $T_1 \mathinner{..} T_n$ are either type variables or parametric types. The annotations $T_1 \mathinner{..} T_n$ denote the types of the class `cn`'s fields. The second step generates variant parametric types by decorating the parametric types with *variance* annotations. The general form of a variant parametric type is $\mathtt{cn}\langle \alpha_1 T_1 \mathinner{..} \alpha_n T_n\rangle$, where $\alpha_1 \mathinner{..} \alpha_n$ are either variance variables or variance values (such as $\circledast$, $\oplus$, $\ominus$, $\odot$) denoting the direction of the flow for the class `cn`'s fields. For example, $\mathtt{Cell}\langle\oplus T_1\rangle$ denotes that the class `Cell`'s field `fst` is subject to a read-only access that corresponds to a flow-out; $\mathtt{Cell}\langle\ominus T_1\rangle$ denotes that the class `Cell`'s field `fst` is subject to a write-only access that corresponds to a flow-in; $\mathtt{Cell}\langle\odot T_1\rangle$ denotes that the class `Cell`'s field `fst` is subject to a read-write access that corresponds to a flow-in and a flow-out; while $\mathtt{Cell}\langle\circledast T_1\rangle$ denotes that the class `Cell`'s field `fst` is not accessed. However, there are some exceptional flows, that are discussed later in Chapter 5. There is also an ordering relation among variance values such that $\odot\mathtt{<:}\oplus\mathtt{<:}\circledast$ and $\odot\mathtt{<:}\ominus\mathtt{<:}\circledast$.

As we mentioned before, parametric types use an invariant subtyping, namely $\mathtt{Cell}\langle T_1\rangle$ $\mathtt{<:}\mathtt{Cell}\langle T_2\rangle$ holds if $T_1{=}T_2$ holds. The variance annotations make subtyping more flexible such that $\oplus$ denotes a covariant subtyping, $\ominus$ denotes a contra variant subtyping, while $\odot$ denotes an invariant subtyping. For example, $\mathtt{Cell}\langle\oplus T_1\rangle$ $\mathtt{<:}\mathtt{Cell}\langle\oplus T_2\rangle$ holds if $T_1\mathtt{<:}T_2$ holds, while $\mathtt{Cell}\langle\ominus T_1\rangle$ $\mathtt{<:}\mathtt{Cell}\langle\ominus T_2\rangle$ holds if $T_2\mathtt{<:}T_1$ holds. In summary, in the case of variant parametric types, the flow subtyping constraints denote relations among the type variables. The type variables represent the types of the values which can be read/written from/into generic data structures. For example, the type $\mathtt{Cell}\langle\oplus T\rangle$ denotes that the field `fst` of class `Cell` contains a value whose type is a subtype of `T`; the type $\mathtt{Cell}\langle\ominus T\rangle$ denotes that the field `fst` of class `Cell` contains a value whose type is a supertype of `T`; the type $\mathtt{Cell}\langle\odot T\rangle$ denotes that the field `fst` of class `Cell` contains a value whose type is `T`; while the type $\mathtt{Cell}\langle\circledast T\rangle$ denotes that the field `fst` of class `Cell` contains a value whose type is unknown. Therefore, it is safe to read a value of any supertype of `T` from `Cell`$\langle\oplus T\rangle$; to write a value of any subtype of `T` into `Cell`$\langle\ominus T\rangle$; to read and write a value of type `T` to `Cell`$\langle\odot T\rangle$; and to read a value of type `Object` and to write a value of type $\perp$ to `Cell`$\langle\circledast T\rangle$. These more precise types allow the type system to prove that some program type casts are redundant.

A method precondition expresses the subtyping relations among the type variables occurring in the types of the method signature (namely the types of method receiver, method arguments, and method result). These subtyping relations capture all possible value flows that may occur in the method body. Method body may contain some local type variables, but they do not escape into method precondition. Type checking rules assume that all variance annotations are given by the programmer. Checking process works in two steps, first it collects the method body flow and then it verifies whether the method precondition entails the collected flow. Type inference process is more complex since the variance annotations are not known at the beginning.

2.3 Flow (Subtyping) Constraints Solving

Type-based flow analyses can be regarded as *constraint-based analyses*, consisting of two parts: constraint generation and constraint resolution. The constraint generation is done by both type checking and type inference rules, since they eventually produce flow (subtyping) constraints. These constraints require a *constraint solver* that is able to perform the following three operations: *constraint simplification* that reduces the redundant information, *constraint satisfiability* that checks whether a system of constraints has a solution, and *constraint entailment* that checks whether a system of constraints implies another system of constraints. For our flow-based type systems, we designed and implemented our constraint solvers by employing techniques from different research areas such as constrained types [135, 147, 176, 59], recursive types [110, 140, 73], polymorphic types [9], constraint simplification [195, 157, 158], subtype entailment [96, 183, 182, 181], set constraints [85, 8, 6], and mixed constraints [65]. The remainder of this section presents several aspects of the constraint solvers and concludes with a discussion about our work.

Subtyping. In general a *subtyping constraint* is an inequality of the form $\tau_1 <: \tau_2$, where τ_1 and τ_2 are type expressions which may contain type variables. A *constraint system* (or *constraint set*) is a conjunction of a finite set of subtyping constraints. In subtype systems, types are typically interpreted as *trees* over some base elements [110]. The base elements can be drawn from a *lattice* or a *partial order* [50]. *Simple types* [122] are interpreted over finite trees, while *recursive types* [11] are interpreted over *regular tree*, that are possibly infinite trees with finitely many sub-terms. Type expressions that are either constants or type variables are referred to as *atomic types* since they have no complex syntactic structure. Note that subtyping over atomic types is referred to as *atomic subtyping*. Two subtype orders arise naturally in practice: the *structural subtype order* and *non-structural subtype order*. Structural subtyping allows only types with the same shape to be related. They are related by some additional structural rules besides the subtype relation of the base elements. An example of structural rule is the subtyping rule [Func] of Figure 2.2, that compares two function types. Non-structural subtyping allows the existence of two additional types, the *smallest type* $\perp$ and the *largest type* $\top$. Besides the structural rules, two rules are added, which essentially say that $\perp$ is smaller than any type, while $\top$ is larger than any type (e.g. the rules [Bottom] and [Top] of Figure 2.2).

Constraint Satisfiability. Constraint satisfiability answers the question whether a constraint system have solutions. Hoang and Mitchell [101] proved that the typability (namely whether a given term has a type) is equivalent to the satisfiability of a conjunction of atomic formulas in the language of structural subtyping constraints. A constraint system is *satisfiable* if there is a valuation that satisfies each constraint of the system. A valuation is a mapping from type variables to ground types (namely types expressions without type variables). A valuation satisfies a constraint if by applying the valuation on that constraint we obtain a new constraint that holds in the lattice of ground types. A detailed discussion of several algorithms, that check the satisfiability of subtyping constraints, can be found in Rehof's thesis [162]. The algorithms are based on the idea of checking consistency in the closure of the constraints with respect to some closure rules. Subtype orderings generated from lattices have PTIME satisfiability problems: atomic subtype satisfiability (Lincoln and Mitchell [114], Tiuryn [187], Rehof and Mogensen [163]), finite structural subtype satisfiability (Tiuryn [187]), recursive structural subtype satisfiability (Rehof [162]), recursive non-structural subtype satisfiability (Palsberg and O'Keefe [142], Pottier [157]), and finite non-structural subtype satisfiability (Kozen, Palsberg, and Schwartzbach [109], Palsberg, Wand, and O'Keefe [148]). Figure 2.5 presents the complexity results of lattice-based subtype satisfiability as were summarized by Rehof in his thesis [162].

	structural subtyping	non-structural subtyping
atomic types	$O(n)$	$O(n)$
finite types	$O(n)$	$O(n^3)$
recursive types	$O(n^3)$	$O(n^3)$

Figure 2.5: Lattice-based Subtype Satisfiability Complexity

In general, when partially-ordered sets (posets) are allowed (rather than lattices), satisfiability problems become more complex. Pratt and Tiuryn [159] have proven that atomic subtype satisfiability is NP-hard. Benke [14] has also tried to characterize the structure of posets (e.g. n-crowns) for which the atomic satisfiability problem is tractable. Tiuryn [187] has proven that finite structural subtype satisfiability is PSPACE-hard, and then Frey [69] has shown that it is in PSPACE and therefore PSPACE-complete. Tiuryn and Wand [188] have shown that recursive structural subtype satisfiability is DEXPTIME. Recently, Niehren, Priesnitz, and Su [131] have proven that finite non-structural satisfiability is PSPACE-complete, recursive structural satisfiability is DEXPTIME-hard, and recursive non-structural satisfiability is DEXPTIME-complete. Figure 2.6 presents the complexity results on subtype satisfiability over posets as were summarized by Niehren, Priesnitz, and Su [131].

	structural subtyping	non-structural subtyping
finite types	$PSPACE{-}complete$	$PSPACE{-}complete$
recursive types	$DEXPTIME{-}complete$	$DEXPTIME{-}complete$

Figure 2.6: Complexity of Subtype Satisfiability over Posets

Constraint Entailment. Constraint entailment answers the question whether a system of constraints C_1 implies (or *entails*) another system of constraints C_2. We say that C_1 entails C_2 if all valuations, that hold for C_1, also hold for C_2. Entailment based subtyping is a key problem in constraint simplification, as it can be used to support, justify and reason about powerful simplification techniques. In general it can be used to check whether a particular constraint $\tau_1 <: \tau_2$ holds in a given system of constraints. Henglein and Rehof [96, 97, 162] have done a systematic study of the subtyping entailment complexity. Figure 2.7 shows their results as were summarized by Rehof in his thesis [162]. The complexity class above the line indicates an upper bound, while the class below the line indicates a lower bound. The question marks indicate that no upper bounds for non-structural entailment are known. However, Henglein and Rehof conjectured that non-structural entailment is in PSPACE.

Niehren and Priesnitz [129, 130] have proven that the non-structural subtype entailment in the

	structural subtyping	non-structural subtyping
atomic types	$O(n)$	$O(n)$
finite types	$\frac{coNP}{coNP}$	$\frac{?}{PSPACE}$
recursive types	$\frac{PSPACE}{PSPACE}$	$\frac{?}{PSPACE}$

Figure 2.7: Subtyping Entailment Complexity

presence of $\bot$, $\top$, and a single non-constant type constructor is PSPACE-complete if $\bot$ and $\top$ do not appear explicitly in the constraints.

In order to take into account the quantifiers, Su, Aiken, Niehren, and Priesnitz [183] have studied issues relating to the *first-order theory of subtyping constraints*. The constraint entailment discussed so far is in the universal fragment ($\forall$*-fragment*) of the first-order theory. Let be C a conjunction of basic constraints, the entailment $C \models x{<:}y$ holds iff the universal formula $\forall x_1..x_n.(C \implies (x{<:}y))$ is valid, where $x_1..x_n$ are the variables free in $C \cup \{x{<:}y\}$. A more powerful entailment is the *existential entailment* represented as $C_1 \models \exists x_1..x_n.C_2$, where $fv(C_2) \cap \{x_1,..,x_n\} = \emptyset$ and $fv(C)$ denotes the free variables of C. The existential entailment holds if for every solution of C_1, there exists a solution for C_2 such that both solutions coincide on the variables $fv(C_2) \setminus \{x_1..x_n\}$. Existential entailment is important for the simplification of the *constrained types* [195, 8, 9]. A constrained type $\tau \setminus C$ consists of a type τ restricted by a constraint set C. Here only the variables appearing in the type τ are important, the other variables appearing only in C should be eliminated by the existential quantifier. Both the existential entailment and the constrained types subtyping are in the $\forall\exists$*-fragment* of the first-order theory. Thus, the existential entailment $C_1 \models \exists x_1..x_n.C_2$ is represented by the following formula in the $\forall\exists$-fragment: $\forall y_1..y_m \exists x_1..x_n.(C_1 \implies C_2)$, where $y_1..y_m$ are the variables in $fv(C_1) \cup (fv(C_2) \setminus \{x_1..x_n\})$. Su et al [183, 181] have proven that the first-order theory of non-structural subtyping constraints is undecidable for both finite and infinite trees and for any type signature with at least one binary type constructor and a least element $\bot$. They have also shown that first-order theory of structural and non-structural subtyping constraints with unary function symbols is decidable for both finite and infinite trees. Kuncak and Rinard [111] have proven that first-order theory of structural subtyping of non-recursive types is decidable.

There are still a lot of *open problems* in this area, but among them, the most important are the decidability and exact complexity of non-structural subtype entailment, existential entailment, and subtyping constrained types.

Constraint simplification. Constraint simplification consists of transformations on constraint sets that aim at removing the redundant information. The redundant information can be defined in the context of typings as the *unnecessary degrees of freedom* [162]. There are two ways to allow types to have a higher degree of freedom than simple types: *parametric polymorphism*, that has the ability to abstract a type with respect to a type variable, and *subtyping*, that enriches the typing judgments with constraint sets. The simplification transformations must satisfy some *soundness conditions* which ensure the preservation of the typings information content. A powerful condition based on the existential entailment was used in [195, 157]: two constraint sets are *observationally equivalent* if replacing one with the other does not affect the results of an analysis. As was argued by Aiken, Wimmers and Palsberg in [9], there are three benefits of simplification: (1) *efficiency*: reducing the number of gathered constraints may speed up the analyses, especially the type inference; (2) *readability*: it reduces the size of type representation; (3) *transparency*: it makes the information content of a type more explicit. However, Pottier has shown in [158] that efficiency and readability are conflicting goals. If the goal is efficiency, the most succinct representation is not necessarily the easiest to deal with (e.g. it may not preserve some invariants used by the analysis).

Fuh and Mishra [71] have developed simplification techniques for simple constraints between variables and base types. Aiken, Wimmers and Palsberg [9] have considered the number of distinct type variables as a measure of freedom degree. They have developed a sound and complete variable elimination algorithm to simplify quantified recursive and non-recursive types in the presence of subtyping. They have also extended their algorithm to type languages with intersection and union types and to type languages with constrained types. These two extensions are sound but not complete. Pottier [157] and Trifonov and Smith [195] have developed sound but not complete algorithms to simplify polymorphic constrained types. Both algorithms have a non-structural recursive entailment at their core. Flanagan and Felleisen [66] have developed practical techniques for simplifying set constraints in the context of a static debugging for Scheme.

Constraint resolution algorithms take an initial set of constraints and repeatedly transform it by applying some resolution rules until the constraint set is in a solved form. These algorithms can also be regarded as simplification algorithms. In general, the resolution algorithms are based on transitively closed constraint graphs which are incrementally built each time when a new constraint is added. Much progress has been made on developing scalable algorithms to handle large sets of constraints. Fahndrich and Aiken [62] have proposed several simplification techniques to reduce the memory requirements of the constraint graphs. However the simplifications are performed only at some points since they are relatively expensive to compute. Their results are similar to those obtained before by Heintze [86]. However Heintze has applied different techniques. Fahndrich et al. [63] have developed an algorithm to perform cycle detection and elimination at every update of the constraint graph. All variables on such a cycle are equal in all solutions of the constraints and therefore they can be collapsed to a single variable. Su et al. [180] have proposed a technique called projection merging that can be used in conjunction with cycle elimination to obtain more scalable analyses. The technique consists of merging many upper bounds on a variable into a single upper bound. Heintze and McAllester [89, 88] have also developed a technique that resembles projection merging. Heintze and Tardieu [84] have proposed an efficient algorithm for implementing dynamic transitive closure. Their algorithm maintain a pre-transitive graph, namely a graph that is not transitively closed. When information about a node is requested, a reachability computation is performed.

An important source of inefficiency in polymorphic constraint-based analyses stems from computing instances of constraints. In general each function is separately analyzed and the information about that function is summarized as a constraint set. At different call sites of the function, the constraint set is instantiated with fresh variables and duplicated. The constraint set duplication is necessary in order to distinguish the different call sites. Thus, the number of resulting constraints grows very fast even if the underlying types are small. Two solutions have been proposed: *instantiation constraints* (Fahndrich et al. [64], Rehof and Fahndrich [161]) and *constraint abstractions* (Gustavsson and Svenningsson [81]). Both solutions make the substitution instantiation a syntactic construct in the constraint language. Instantiation constraints are a form of constraints similar to Henglein's semi-unification constraints [92] but they are annotated with an instantiation site and a polarity. Constraint abstractions allow the constraints to compactly express substitution instantiation. The main difference is that the constraint abstractions provide more structure and a notion of local scope. In the case of the instantiation constraints, the scope of a variable is the entire set of constraints.

Our work. Our first type system from Part I, the *region type system* generates region constraints. Region constraints (see Figure 3.2 of Chapter 3) are atomic constraints containing only region variables. Region variables can be instantiated with the runtime regions from the runtime stack of memory regions. At runtime, there is only one stack of memory regions, on which the memory regions are allocated and deallocated. There is always at least one memory region on the runtime stack (namely the heap, the first region memory that is allocated at the beginning of the program execution and deallocated at the end of the program execution). Among the regions is defined an outlive relation as a partial ordering (see Section 3.2). Since every subset of runtime memory regions has a least upper bound with respect to the outlive relation, the runtime memory regions form a lattice. Our region constraints are in the domain of *lattice-based atomic subtyping*. Therefore region constraint satisfiability and region constraint entailment can be checked in linear time. We use constraint abstractions for region inference (especially for fixed point iterations). We also employ variable elimination techniques to localize the regions either for a *letreg* block, or at the method boundary.

Our second type system from Part II, the *variant parametric type system* produces more complex constraints (see Figure 5.3 and Figure 5.4 of Chapter 5). Variant parametric subtyping is in the domain of *lattice-based recursive non-structural subtyping*. Variant parametric subtyping satisfiability is $O(n^3)$. The non-structural subtyping entailment is still an open problem. Our constraint solver proves the entailment $\forall V_G \cdot (\psi_1 \implies X_i{<:}Y_i)$ by contradiction using the falsity of the formula $\forall V_G \cdot (\psi_1 \wedge \mathit{notsub}(X_i, Y_i))$, where $\mathit{notsub}(t_1, t_2)$ represents negation of subtyping relation. Our deduction mechanism detects falsity based on pair of constraints of the form $t_1{<:}t_2$ and $\mathit{notsub}(t_1, t_2)$. This is a sound approximation of the entailment problem. We can further extend deduction mechanism with the techniques of *case analysis* and *inductive proving* (similar to those presented by Pottier in his thesis [158]) especially for recursive types. Nevertheless, from our experience working with large sets of Java library and application codes that have been annotated and checked with variant parametric types, we have yet to encounter real examples which require such extensions.

PART I

Safe Region-Based Memory Management

CHAPTER 3

REGION-BASED MEMORY MANAGEMENT

3.1 Introduction

Region-based memory management has been developed as an alternative approach to explicit allocation/deallocation (e.g. malloc and free), and automatic garbage collection techniques [203, 106]. Region-based systems allocate each new object into a program-specified *region* [191], with the entire set of objects in each region deallocated simultaneously when the region is deleted. Various studies have shown that region-based memory management can provide memory management with good real-time performance. Individual object deallocation is accurate but time unpredictable, while region deletion presents a better temporal behavior, at the cost of some space overhead. Data locality may also improve when related objects are placed together in the same region. Classifying objects into regions based on their lifetimes may deliver better memory utilization if regions are deleted in a timely manner.

3.1.1 Region Issues

Regions have been introduced and used for decades in practice [167, 83]. However the original proposals (e.g. arenas in [83]) were *unsafe*: deleting a region may create dangling references that are subsequently accessed. Moreover some well-known applications such as the apache web server and the gcc compiler (before version v3) have been written using unsafe region libraries [74].

The main safety issue of the region-based memory management is represented by the *dangling references*. A reference from (an object in) one region to (an object in) another region is considered to be *dangling* if the latter region has a shorter lifetime than the former. A region has a *shorter lifetime* than another region if it is deleted before the latter. Using a dangling reference to access memory is unsafe because the accessed memory may have been recycled to store other objects.

We distinguish two approaches to ensure region-based memory safety. The first approach, called dynamic safety uses *runtime checks* to guarantee the region safety at runtime. In contrast the second approach, called static safety, ensures the region safety at compile time by using either a type system or a static analysis. The first approach is more flexible, but the runtime checks may introduce a large runtime overhead. Some systems combine dynamic and static safety either to be more flexible or to reduce the runtime overhead. For example, the system from [75] prevents unsafe region deletions by maintaining a count of references to each region. A region type system in [76, 74], may significantly reduce the cost of reference-counting.

Researchers have identified two approaches to ensure region safety at compile time. The first approach allows the program to create dangling references, but uses a type and effect system to ensure that the program never uses a dangling reference to access memory [191, 18, 192, 41, 80, 37]. The second approach uses a type system to prevent the program from creating dangling references at all [23]. The first approach (*no-dangling-access*) may yield more precise region lifetimes, but the latter approach (*no-dangling*) is required by the Real-Time Specification for Java (RTSJ) [19] and also makes easier the co-existence of the region-based memory management with garbage collection. For example, in ML Kit [82, 60], the original region typing rules [191] were strengthened to forbid dangling pointers in order to make possible a memory discipline that combines the regions and a copying garbage collection within regions. However, in Cyclone [80] dangling references are allowed, but a conservative garbage collector was used to reclaim the objects allocated into the heap region.

Another important issue of the region-based memory management is the accuracy of the regions' lifetimes to model the lifetimes of the program objects in order to avoid *memory leaks*. A region conservatively approximates the lifetimes of all its objects. Therefore storing objects with different lifetimes in the same region may potentially lead to a considerable amount of wasted memory, especially in recursions and loops. An extreme situation occurs when no memory is ever reclaimed because all objects are placed into only one region that is alive throughout the execution.

Based on the discipline imposed on the region lifetimes we can distinguish two kinds of regions: *lexically scoped regions* and *not lexically scoped regions*. A *lexically scoped region* r is introduced by the expression $letreg\ r\ in\ e$ such that the lifetime of the region r is the scope of the expression e [191, 192, 41, 80]. The $letregion$ construct is aligned with the program's expression hierarchy, thus all region allocations and deallocations follow a stack discipline. The problem of this approach is that in practice object lifetimes do not follow a stack discipline.

Therefore a number of optimizations have been proposed on top of the lexically scoped regions. In ML Kit [190, 18] a storage mode analysis determines when it is possible to reset a region (to deallocate the region content) prior to deallocation of the region itself. In [7] the region variable introduction is separated from the region allocation and deallocation. Thus the region allocation is postponed until just before its first access, while the region deallocation is postponed just after the last access. However these optimizations come at the cost of a *region aliasing* analysis and may require rewriting of the original program. An advantage of stack discipline is that it induces an *outlive* relationship on regions, which, in turn, provides a region subtyping discipline on pointer types [80].

Not lexically scoped regions do not require a stack discipline to be allocated and deallocated. They can improve the precision of the computed lifetimes at the expense of solving an additional problem: region deallocation must take into account region aliasing. Not lexically scoped regions decouples region creation from region removal. Moreover multiple region variables may denote the same region. Therefore wherever a region is deallocated, it is required to check whether there are no other region aliases of that region. Several solutions have been proposed such as either a reference counting on regions that may incur noticeable runtime overhead [75], or a combination of a region type system with a runtime reference counting on regions [76, 199, 93], or a complex region points-to analysis [37].

Another important issue of region-based memory management is the *region size*. The size of a region is the maximal size of the objects that may be allocated in that region. We can distinguish two kinds of regions *finite regions* and *infinite regions*. Finite regions contain a finite number of objects, while infinite regions hold an unbounded number of objects. An advantage of finite lexically scoped regions is that they can be directly allocated on the runtime system stack. For example, in ML Kit [190, 18] it was developed an analysis (called multiplicity inference analysis) to determine suitable finite regions, where possible.

3.1.2 Motivation and Goal

Several projects have recently investigated the use of region-based memory management for Java-based languages [41, 23, 19, 51]. Most of these projects have focused on region checking [41, 23], which requires manual effort to augment the program with *region annotations*. An issue is that region annotations may impose considerable mental overhead for the programmer and raise compatibility issues with legacy code. In addition, the quality of the annotations may vary, with potentially suboptimal outcomes for less experienced programmers. On the other hand, Real-Time Specification for Java (RTSJ) [19] allows the programmers to explicitly use scoped memory areas in order to avoid the garbage collector for time-critical tasks. However RTSJ requires *run-time checks* to ensure the safety of the memory management. In addition, programming for the RTSJ is so complex that it forces the Java programmer to adopt *new coding habits* [155]. The proposal in [51] provides an automatic translation of Java code into Real-Time Java using a dynamic analysis to determine the lifetime of an object. However the translation may not be *sound* as the dynamic analysis may miss some execution paths that create and use dangling references.

In this context our goal was to develop *an automatic region type inference system for object-oriented languages* that should meet the following requirements:

- *A Sound Type-based Analysis*: The compiler should automatically augments the programs with region type annotations such that the region annotated programs use a region-based memory management at runtime. Region safety should be ensured at compile time without using runtime checks.
- *Convenient*: The region-based memory management should be transparent for the programmers. This means that programmers should not be required to provide region annotations or to rewrite the source programs in order to obtain region friendly programs.
- *Scalable*: The region analysis should be fast and simple.
- *Precise as much as possible*: The precision is defined as the ability to reuse memory as soon as possible. Since it is hard to define the best solution for the region annotations, we expect that the results of the inferred programs to be competitive with those of the programs hand annotated by the human experts.
- *Easy integration with a garbage collector*: The inferred regions should safely co-exist with any kind of garbage collector.
- *Support for object-oriented features*: The region inference rules must ensure region safety in the presence of the main object-oriented features such as *class inheritance*, *method overriding*, and *downcast operations*.

Figure 3.1: Region System Overview

3.1.3 Solution and Contributions

We provide a systematic formulation of a region type inference system for the core subset of Java, called Core-Java. Our solution uses *lexically-scoped regions* approach to impose stack discipline on regions and *no-dangling references* approach to ensure region safety. Lexically-scoped regions makes the region analysis simpler by avoiding the region aliasing problem. They also allow us to define subtyping on the region types. No-dangling references approach is required by Real-Time Specification for Java [19] and also makes easier the co-existence with any garbage collection strategy. Although no-dangling references approach seems to be less precise, it has a little effect on overall memory behavior as shown in [60]. Our entire region system is depicted in the diagram of Figure 3.1.

In summary, this part of our dissertation makes a number of technical contributions explained below:

- **Region Type System:** We have formulated and implemented a region type system for a core subset of Java as the target for region inference. The region type system guarantees that well-typed programs use lexically scoped regions and do not create dangling references in the store and on the stack. Although our type system is similar to SafeJava' type system of Boyapati et al. [23], there are three main differences: (1) we isolated out the object encapsulation issue in our type system, (2) we added support for region subtyping by adapting the region subtyping principle from Cyclone [80], and (3) we provided a rigorous soundness proof for our region type system (note that SafeJava does not provide a formal proof for its region type system).

  ◦ **Region Lifetime Constraints:** Our region type rules prevent dangling references by requiring the target object of each reference to live at least as long as the source object. We formalise this requirement explicitly through region lifetime constraints, with support for region subtyping.

    ◦ **Safety Proof:** We have proven that our type system is safe. Safety implies that the regions follow a stack discipline and they can be deallocated only when there are not external references to their objects. In addition, the objects' fields and the program variables cannot contain references to a non-existing region. Although our safety property (lexically scoped regions and no dangling references) is similar to safety property preserved by the region type system of Elsman [60], our proof is different. The proof from [60] is based on a small-step contextual semantics, while our proof explicitly represents the heap as a stack of regions and keeps a consistency relationship between the static and dynamic semantics. Moreover the region type system from [60] is designed for a functional language.

- **Region Inference:** We have formulated and implemented a novel summary-based flow-insensitive analysis to automatically infer region annotations for a core subset of Java. The result of our region inference is correct with respect to our region type system. Object-oriented features such as class subtyping, method overriding, and downcast operations are fully handled by our analysis.

    ◦ **Summary-based Flow-Insensitive Analysis:** Our region inference is designed as a summary-based flow insensitive analysis for classes and methods. The summary of a class is the class invariant, while the summary of a method is the method precondition. Due to a fairly complex inter-dependency between classes and methods, the analysis is required to process the classes and methods in some particular order given by a dependency graph. In a research performed concurrently with ours, Cherem and Rugina [37] have developed a three-stage region inference algorithm for Java. That algorithm relies on a flow analysis to propagate unifications between regions in an interprocedural manner. Using the no-dangling-access principle, that inference produces programs that use non-lexically scoped regions different than our lexically scoped regions. While our inference system is based on a region type system where object and field subtyping could be supported, the approach of [37] directly generates region handles (the run-time structures needed to allocate an object into a region), and uses points-to analysis and liveness analysis to determine when regions can be deallocated.

  - **Region Polymorphism:** We support classes and methods with region polymorphism: region-polymorphic recursion for methods, and region-monomorphic recursion for classes. These features provide an inference algorithm that is precise and yet efficient.
  - **Class Inheritance and Method Overriding:** Our inference analysis provides an improved solution for class subtyping and method overriding. Previous region types systems for Java [41] require "phantom regions" to support inheritance with downcasting and method overriding, which may cause a loss in lifetime precision.
  - **Downcast Safety:** We provide a compile-time analysis which ensures that downcast operations are region-safe. Previous proposals [21] require runtime checks for downcast operations.
  - **Correctness:** We proved that our inference algorithm is sound and complete with respect to our region type system.
  - **Runtime Regions:** Since only the handles of the regions that may be written with new objects are required at the runtime, we designed a type-based analysis that simplifies the region annotations and generates a corresponding program with such region handles.

- **Experimental Validation:** We have implemented a prototype of our region inference system and we have run some experiments on medium-sized benchmarks. Preliminary results that we have obtained are encouraging. The programs based on our inferred regions were able to reuse significant amount of memory for most of the cases where data was not live throughout the execution. The experiments suggest that our results are competitive when compared to those that are hand annotated by human experts, and comparable also to the approach based on non-lexically scoped regions with no-dangling-access [37]. The experiments also suggest that our region inference analysis is fast in terms of analysis time and reasonable with respect to the number of region parameters.

3.1.4 Organization of Part I

Part I of our dissertation is mainly based on papers published in [40, 47]. It consists of two chapters organized as follows.

$$\begin{aligned} \textit{prim} &::= \textbf{int} \mid \textbf{boolean} \mid \textbf{void} \\ \tau &::= cn \mid \textit{prim} \mid \bot \\ t &::= \tau\langle r^* \rangle \mid \bot \end{aligned}$$

(a) Regions Types

$$\begin{aligned} \varphi &::= r_1 \succeq r_2 \mid r_1 = r_2 \mid \textit{true} \\ &\quad \mid \varphi_1 \wedge \varphi_2 \mid q\langle r_1, .., r_n \rangle \\ q &::= cn \mid cn.mn \\ r &\in \textit{region variable names} \end{aligned}$$

(b) Region Lifetime Constraints

$$Q ::= \{(q\langle r_1, .., r_n \rangle = \varphi)^+\}$$

(c) Constraint Abstractions

Figure 3.2: Region Types and Lifetime Constraints

In Chapter 3 we introduce the main concepts and formalise our region type system. Section 3.2 introduces the region types and the region lifetime constraints. Section 3.3 describes the region-based memory model used by our type system. The region annotations principles for classes and methods are presented in Section 3.4, while the region subtyping principle is defined in Section 3.5. Section 3.6 presents Core-Java, our object-oriented core language, and the type rules of our region type system. In Section 3.7 we formulate the dynamic semantics and prove the safety properties of our region type system.

Chapter 4 presents our region inference, the experimental results and concludes with some remarks and a discussion of related work. Section 4.1 introduces the inference algorithm using a simple example. Then we formalise the main inference rules as follows: rules for classes in Section 4.2, rules for expressions in Section 4.3, rules for region localization in Section 4.4, rules for method overriding in Section 4.5, and rules for dependency graph in Section 4.7. Section 4.8 formulates and proves the correctness of the inference algorithm with respect to our region type system. Section 4.9 discusses an extension of region subtyping. Section 4.10 presents the experimental results obtained using our prototype, while Section 4.11 discusses the related work. In addition, Appendix A.4 presents the rules for downcasting, Appendix A.5 discusses the runtime region analyses, while Appendix A.6 discusses other Java features.

3.2 Regions Types

To support region-based memory management, our region inference algorithm adds region parameters and constraints to each class and its methods. Each class definition is parameterized with one or more regions to form a *region type*, denoted by *t* in Figure 3.2(a). For instance, a region type $cn\langle r_1, ..., r_n\rangle$ is a class name cn annotated with region parameters $r_1...r_n$. Parameterization allows us to obtain a region-polymorphic type for each class whose fields can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which the instance object of this class is allocated. The fields of the objects, if any, are allocated in the other regions $r_2...r_n$ which should *outlive* the region of the object. This is expressed by the constraint $\bigwedge_{i=2}^{n}(r_i \succeq r_1)$, which captures the property that the regions of the fields (in $r_2...r_n$) should have lifetimes no shorter than the lifetime of the region (namely r_1) of the object that refers to them. This condition, called *no-dangling requirement*, prevents dangling references completely, as it guarantees that each object never references another object in a younger region. The first region r_1 of an object region type can not be used by the region types of the object fields because our region subtyping rules (Section 3.5) assume this invariant and would be unsound otherwise.

We do not require region parameters for primitive types, since primitive values can be copied and stored directly in the stack or they are part of an object. In order to keep the same notation, we use *prim*$\langle\rangle$ to denote a region annotated primitive type. Although null values are of object type, they are regarded as primitive values. The type of a null value is denoted by $\perp$.

Figure 3.2(b) presents the syntax of region lifetime constraints. Our algorithm infers region constraints of two forms $r_1{\succeq}r_2$ and $r_1{=}r_2$. The constraint $r_1{\succeq}r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2, while the constraint $r_1{=}r_2$ denotes that r_1 and r_2 must be the same region. The outlive relation $\succeq$ is a transitive relation such that if $r_1{\succeq}r_2$ and $r_3{\succeq}r_1$ then $r_3{\succeq}r_2$. There is also a relation between equality and outliving such that $r_1{=}r_2$ iff $r_1{\succeq}r_2$ and $r_2{\succeq}r_1$. We assume that the region constraint φ is always closed by transitivity such that the transitivity is performed each time a new constraint ($r_1{\succeq}r_2$ or $r_1{=}r_2$) is added to the constraint φ. Given a constraint φ and a set of regions R, the notation $\varphi\backslash R$ (or $\varphi{-}R$) denotes the elimination from φ of all region constraints which use regions from R.

We also use constraint abstractions ([81]) of the form $q\langle r_1, .., r_n\rangle = \varphi$ to capture a parameterized constraint (Figure 3.2(c)). A constraint abstraction is in a *closed-form* when its body

Figure 3.3: Memory Model based on Lexical Regions

φ contains only simple lifetime constraints ($r_i{\succeq}r_j$ or $r_i{=}r_j$) between the regions from its head ($r_i, r_j{\in}\{r_1, .., r_n\}$).

For uniformity, the region constraint of each class and method are each captured with *one* constraint abstraction, denoted by a singleton set Q. The region constraint of each class is also known as the *class invariant* and is denoted using *cn*$\langle r_1, .., r_n\rangle$. The region constraint of each method is also known as the *method precondition* and is denoted using *cn.mn*$\langle r_1, .., r_n\rangle$. Note that mn denotes a method name, while cn denotes a class name. Constraint abstractions act as intermediate forms in our summary-based analysis (see Section 4.1). They can be inlined after the fixpoint analysis has been applied to obtain the closed-form formulae of the recursive constraints (this fixpoint mechanism is described later in Section 4.5).

In the case of mutually recursive methods (or classes), the fixpoint analysis is applied to a set of constraint abstractions (see Section 4.7). This set is the union of the singleton sets corresponding to the mutually recursive methods (or classes). Note that there is one singleton set (consisting of one constraint abstraction) per method (class).

3.3 Region-Based Memory Model

We adopt a memory model based on a stack of regions, as illustrated in Figure 3.3. Regions are memory blocks that are introduced and disposed by the construct `letreg r in e`, where the region `r` can only be used to allocate objects in the expression `e`. Our region inference algorithm localizes the regions by introducing `letreg` constructs in the original program code. The older regions (with longer lifetime) are allocated at the bottom of the stack while the younger regions (shorter lifetime) are at the top.

We can also allow a single `heap` region that conceptually lives forever. For instance, the first region `r0` at the bottom of the stack can be reserved to denote a global heap with unlimited lifetime, that is $\forall \texttt{r} \cdot \texttt{r0} \succeq \texttt{r}$. From a type-checking and inference perspective it is equivalent to put a `letreg` around the body of the main method to introduce the region `r0`.

Our region inference computes the region where each object is allocated. Region objects are deallocated when the region is popped from the stack of regions. For a method that allocates objects into a region, our system infers the region handles (the run-time structures needed to allocate an object in a region) that may need passing to that method at its call sites.

Lifetime constraints of the regions from Figure 3.3 can be expressed as $\texttt{r0} \succeq \texttt{r1} \wedge \texttt{r1} \succeq \texttt{r2} \wedge \texttt{r2} \succeq \texttt{r3} \wedge \texttt{r3} \succeq \texttt{r4}$. Figure 3.3 shows two kinds of references: non-dangling references (drawn using normal lines) and possible dangling references (drawn using dashed lines). Non-dangling references originate from objects placed in a younger region and point to objects placed either in an older region or inside the same region. Possible dangling references occur when objects placed in an older region point to objects placed in a younger region. Possible dangling references turn into dangling references when the younger region is deallocated. Our region inference algorithm disallows the references from the older regions to the younger regions, totally preventing the dangling references.

3.4 Regions Annotations

Our region inference algorithm adds region parameters and constraints to each class and its methods. There are a number of ways to perform such region annotations. The following principles guide our approach:

- Keep the regions of fields in each class (and the regions of the parameters and results of each method) *distinct*, where possible.

- Keep the region constraints on classes and methods *separate*. Region constraints on a class capture the expected class invariant (including the no-dangling requirement) on the regions of each instance of the class. Region constraints on a method denote the precondition for invoking the method that follows from the effect of assignments inside the respective method.

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1 {
   Object⟨r2⟩ fst;
   Object⟨r3⟩ snd;
   Object⟨r4⟩ getFst⟨r1,r2,r3,r4⟩() where r2⪰r4
     {return fst;}
   void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o) where r4⪰r3
     {snd=o;}
   Pair⟨r4,r5,r6⟩ cloneRev⟨r1,r2,r3,r4,r5,r6⟩() where r2⪰r6∧r3⪰r5
     { Pair⟨r4,r5,r6⟩ tmp;
      tmp =new Pair⟨r4,r5,r6⟩(null,null);
      tmp.fst=snd; tmp.snd=fst; return tmp;}
   void swap⟨r1,r2,r3⟩() where r2=r3
     { Object⟨r2⟩ tmp=fst; fst=snd; snd=tmp; }
}
```

Figure 3.4: Pair Class

The first principle allows more region polymorphism, where applicable. The second principle places the region constraints that must hold for every instance of a given class in the class, while the region constraints of the method is to capture the method's effects. Placing region constraints with methods where possible allows these constraints to be selectively applied to only those objects which may invoke the methods. We shall see how this idea improves the precision of region lifetimes.

3.4.1 Regions for Field Declarations

Consider the `Pair` class in Figure 3.4. As there are two fields in this class, we introduce a distinct region for each of them, `r2` for `fst` field and `r3` for `snd` field. The `Pair` object is placed in the region `r1`. To ensure that every `Pair` instance satisfies the no-dangling requirement, we also add `r2⪰r1∧r3⪰r1` to the class invariant. In general the class invariant of a class consists of the no-dangling requirement for the region type of the current class, the no-dangling requirements for the fields' region types, and the class invariant of the parent class (see Section 4.2). Sometimes the class invariant could be strengthened with region constraints from the methods (see Section 4.6).

Next consider the `List` class with `next` as its recursive field in Figure 3.5. There are many different ways of annotating such recursive fields; the best choice depends on how the objects are manipulated. To keep matters simple, we use a special form of region-monomorphic recursion for class declarations, similar to Tofte/Birkedal's handling of data structures [191, 18, 189], but with support for region subtyping. We introduce a distinct region for all the recursive fields.

```
class List⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r3⪰r1∧r2⪰r3∧r2⪰r1 {
   Object⟨r2⟩ value;
   List⟨r3,r2,r3⟩ next;
Object⟨r4⟩ getValue⟨r1,r2,r3,r4⟩() where r2⪰r4
     { return value; }
List⟨r4,r5,r6⟩ getNext⟨r1,r2,r3,r4,r5,r6⟩() where r5=r2∧r6=r3
     { return next; }
void setNext⟨r1,r2,r3,r4,r5,r6⟩(List⟨r4,r5,r6⟩ o)where r5=r2∧r6=r3=r4
   { next = o; }
}
```

Figure 3.5: List Class

This approach ensures that each recursive field has the same annotation as its class, except for its first region. Given a recursive class declaration with region type cn$\langle r_1, r^*, r_n\rangle$, where r_n is the region for recursive fields, we annotate each of the recursive fields as cn$\langle r_n, r^*, r_n\rangle$. The reason for allowing region polymorphic data recursion on the first region parameter is to support object region subtyping (see Section 3.5), while the use of region monomorphic data recursion on the other fields helps to simplify our region inference mechanism. This combination can be implemented cheaply and is also critical for supporting another form of region subtyping based on immutable fields (see Section 4.9). Mutually recursive class declarations are similarly handled (see Section 4.7). In the case of the `List` class, region `r3` is reserved specially for the recursive `next` field, as illustrated in Figure 3.5. To ensure that every `List` object satisfies the no-dangling requirement, we add `r2⪰r1∧r3⪰r1` to the class invariant. We also add the no-dangling requirement for the region type of the recursive field, as `r2⪰r3∧r3⪰r3`. Based on the above guidelines, the constraint abstractions for the `Pair` and `List` classes are:

```
Pair<r1,r2,r3> = r2⪰r1∧r3⪰r1
List<r1,r2,r3> = r3⪰r1∧r2⪰r3∧r2⪰r1
```

3.4.2 Regions for Method Declarations

For each method declaration, we provide a set of regions to support the method parameters (including the receiver) and the method result. For simplicity, no other externally defined regions are made available for a method. Thus, all regions used in a method either are mapped to these region parameters or are localised by `letreg` in the method body. Region localisation is described later in Section 4.4.

We also provide region lifetime constraints over such region parameters and the regions of `this` object. These constraints naturally depend on how the method manipulates the objects. Consider the `getFst`, `setSnd` and `cloneRev` methods of the `Pair` class. We introduce a set of distinct region parameters for the methods' parameters, and the results, as shown in Figure 3.4. The receiver regions are taken from the class definition. Moreover, the region (lifetime) constraints are based on the possible operations of the respective methods. For example, due to an assignment operation and region subtyping, we have r4⪰r3 for `setSnd`, while r2⪰r6∧r3⪰r5 are due to copying by the `cloneRev` method.

Consider the `swap` method. A region constraint r2=r3 is present due to the swapping operation on the receiver object. Though this constraint is exclusively on the regions of the current object, we associate the constraint with the method. In this way, only those objects that might call the method are required to satisfy this constraint.

The region constraint for a method also contains the class invariants of its parameters including the receiver and its result. For example, the region constraint for `cloneRev` implicitly includes the class invariant r6⪰r4∧r5⪰r4 of the resulting type Pair⟨r4,r5,r6⟩ and the class invariant r2⪰r1∧r3⪰r1 of the receiver Pair⟨r1,r2,r3⟩. For simplicity, we omit the presentation of such constraints in this dissertation. These constraints can be easily recovered from the method's type signature. Except for this omission, the constraint abstractions for the various methods of the `Pair` class are as shown below:

```
Pair.getFst<r1,r2,r3,r4>          = r2⪰r4
Pair.setSnd<r1,r2,r3,r4>          = r4⪰r3
Pair.cloneRev<r1,r2,r3,r4,r5,r6> = r2⪰r6∧r3⪰r5
Pair.swap<r1,r2,r3>               = r2=r3
```

Note that the first three regions r1,r2,r3 are the regions of the current receiver. The receiver regions could be also omitted from the method region parameters list since they are recovered from the region type of the receiver during the type checking. However the region inference algorithm generates them by default in front of the method region parameters list (Section 4.5).

3.4.3 Regions for Subclass Declarations

Each subclass typically augments its superclass with additional fields and methods. Correspondingly, the regions of each subclass are *extended* from its superclass, its invariant represents a *strengthening* from the invariant of its superclass. These requirements are needed to support class subsumption. Consider:

```
  class A⟨r1..rm⟩ extends Object⟨r1⟩ where φA...
  class B⟨r1..rm..rn⟩ extends A⟨r1..rm⟩ where φB...
```

We expect the regions of the subclass `B`, namely ⟨`r1..rm..rn`⟩, to be an extension from the regions of `A`, namely ⟨`r1..rm`⟩, with `n`≥`m`. Likewise, the region invariant of φ_B is a strengthening of φ_A, with the logical implication $\varphi_B \Rightarrow \varphi_A$. These requirements allow an object of the `B` class to be safely passed to any location that expects an `A` object, as the invariant of the latter holds by implication.

Method overriding poses another challenge which requires subtyping of functions to be taken into account. In general, the method of a subclass is required to be a subtype of the overridden method. As it was proved in [36], in object-oriented languages the function subtyping is sound if the parameters (the receiver) that drive dynamic method selection are covariant, the normal parameters are contravariant, and the result is covariant.

Consider a method `mn` in class `A` that is overridden by another method `mn` from the `B` subclass. Let us assume that the region type of class `A` is `A`⟨`r1..rm`⟩, while the region type of class `B` is `B`⟨`r1..rm..rn`⟩. Let us also assume a class `X` with its region type `X`⟨`x1..xp`⟩ and a class `Y` with its region type `Y`⟨`y1..yq`⟩:

```
Y⟨y1..yq⟩ A⟨r1..rm⟩.mn⟨r1..rm,x1..xp,y1..yq⟩ (X⟨x1..xp⟩ a)
   where φA.mn
Y⟨y1..yq⟩ B⟨r1..rm..rn⟩.mn⟨r1..rm..rn,x1..xp,y1..yq⟩(X⟨x1..xp⟩ a)
    where φB.mn
```

The constraints $\varphi_{A.mn}$ and $\varphi_{B.mn}$ are the preconditions for the region parameters ⟨`r1..rm,x1..xp,y1..yq`⟩ of `A.mn` and ⟨ `r1..rm..rn,x1..xp,y1..yq`⟩ of `B.mn`, respectively. These parameters must be contravariant for function subtyping, requiring $\varphi_{A.mn} \Rightarrow \varphi_{B.mn}$. With the class invariant, φ_B of class `B` (as the receiver), it is also safe to weaken this soundness check to $\varphi_B \wedge \varphi_{A.mn} \Rightarrow \varphi_{B.mn}$. The class invariant of `B` can be used as this method is only invoked when the current receiver is of the `B` class or any other subclass of `B`. Hence, strengthening φ_B may help the method satisfy this soundness check. Its inclusion is critical to our approach for handling method overriding without phantom regions. Phantom regions is the solution adopted by a previous approach [41]. A comparison between phantom regions and our approach is shown in Appendix A.7.

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle \cdots \in P' \\ n{\geq}m{\geq}p \quad \vdash cn'\langle x_{1..m}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi\end{array}}{\vdash cn\langle x_{1..n}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi}\,[\textbf{SubClass}] \qquad \frac{}{\vdash \bot{<:}cn\langle x_{1..n}\rangle,\ \mathit{true}}\,[\textbf{Null}]$$

$$\frac{\varphi = \bigwedge_{i=1}^{n}(x_i{=}\hat{x}_i)}{\vdash \tau\langle x_{1..n}\rangle <:_{inv} \tau\langle \hat{x}_{1..n}\rangle,\ \varphi}\,[\textbf{InvRegSub}] \qquad \frac{\varphi = (x_1{\succeq}\hat{x}_1) \wedge \bigwedge_{i=2}^{n}(x_i{=}\hat{x}_i)}{\vdash \tau\langle x_{1..n}\rangle <:_{obj} \tau\langle \hat{x}_{1..n}\rangle,\ \varphi}\,[\textbf{ObjRegSub}]$$

Figure 3.6: Region Subtyping Rules

Method overriding is particularly challenging for region inference. We introduce some techniques to ensure the compliance of the overriding checks in Section 4.6, after the basic region inference method has been presented.

3.5 Region Subtyping Principle

The *region subtyping principle* allows an object from a region with longer lifetime to be assigned to a location where a region with a shorter lifetime is expected. This concept was pioneered in Cyclone [80]. We use the region subtyping, where applicable, to improve the precision of the regions' lifetimes. Figure 3.6 presents two versions of the region subtyping rules; the versions differ in the precision of the regions' lifetimes: invariant (region) subtyping (rule [**InvRegSub**]) and object (region) subtyping (rule [**ObjRegSub**]). The first kind of subtyping was used in [23] and [41]. The second kind was introduced in [80]. A further extension of the region subtyping to immutable fields is presented in Section 4.9.

To discuss the technical differences, we introduce the general form of the region subtyping relation:

$$\vdash t_1 <: t_2,\ \varphi$$

which establishes that t_1 is a subtype of t_2 and infers a region lifetime constraint φ.

The class subtyping rule used by both kinds of the region subtyping is described by the rule [**SubClass**] of Figure 3.6. Note that P' denotes the region annotated program. The rule is applied until the left hand side region type has the same class name cn'' as the right hand side region type. Then the specific region subtyping rule is called according to the context. A subclass can have more regions than its superclass, $n{\geq}m$. The class subtyping rule does not impose any constraint on the subclass's additional regions, $x_{m+1..n}$, though these regions are required during downcasting. In order to support downcasting, Appendix A.4 will modify the class subtyping rule.

A special case of the subtyping rule is described by the rule [**Null**] of Figure 3.6. This rule ensures that a value of type $\perp$ can be assigned to any object without imposing any restriction on the region type of that object.

3.5.1 Invariant Region Subtyping

The rule [**InvRegSub**] of Figure 3.6 defines the invariant region subtyping. We use $\bigwedge_{i=1}^{n}(x_i{=}\hat{x}_i)$ to denote invariant subtyping on the region of the objects and their fields' regions.

3.5.2 Object Region Subtyping

Object region subtyping relies on the fact that once an object is allocated in a particular region, it stays within the same region and never migrates to another region. This property allows us to apply covariant subtyping to the region of the current object. However, the object fields are mutable (in general) and must therefore use invariant subtyping to ensure the soundness of subsumption. By reserving the first region exclusively for the region of each object, we can therefore use the subtyping rule [**ObjRegSub**] from Figure 3.6. Note that $x_1{\succeq}\hat{x}_1$ allows an object in a region with a longer lifetime to be assigned to a location that expects objects in a region with a shorter lifetime. For the other regions (that are used by the fields), a stronger invariant constraint $\bigwedge_{i=2}^{n}(x_i{=}\hat{x}_i)$ would be used instead to allow field mutability.

One situation where the object region subtyping is better than the invariant region subtyping is the following:

```
void foo (Object a, Object b)
   { Object tmp; if ...  then tmp=a else tmp=b;}
```

Without object subtyping, the dual assignments of both `a` and `b` to `tmp` would cause their regions to be coalesced together and generate the constraint $r_a{=}r_b$ (where r_a and r_b are the regions for `a` and `b`). With object subtyping, regions of `a` and `b` may be different, as long as they both outlive the region of `tmp`. Therefore in our approach we use object region subtyping rule for assignments and to support pass-by-value mechanism of the parameters.

3.6 Region Type System

In this section we formalise the region type system that is the target of our region inference algorithm. To ease the formulation, Section 3.6.1 introduces Core-Java, an object-oriented core language. Some Java programs can be automatically translated into a Core-Java program by our translator [45] and then the region inference algorithm is applied to it. The region type system can be used either to check type safety of the user-supplied programs written in region-annotated Core-Java, or to type check the region-annotated programs generated by our region inference algorithm. The latter was especially useful during the debugging phase of region inference system implementation.

3.6.1 A Fragment of Core-Java

For simplicity, we start the presentation with a fragment of Core-Java language (Figure 3.7(a)). Multiple inheritance and exceptions are discussed in Appendix A.6, while casting is presented in Appendix A.4.

Core-Java assignment evaluates to a `void` value instead of the value of the right hand side expression. The sequence $e_1; e_2$ evaluates to the value of the expression e_2, while a block expression $\{(\tau\ \ v)\ \ e\}$ evaluates to the value of e. Core-Java uses the pass-by-value mechanism. The suffix notation s^* denotes a list of zero or more distinct syntactic terms separated by appropriate separators, while s^+ represents a list of one or more distinct syntactic terms. The syntactic terms could be `v`, `r`, `(t v)`, etc. For example, $(t\ \ v)^*$ denotes $(t_1\ \ v_1, \ldots, t_n\ \ v_n)$ where $n{\geq}0$. Note that `this` is a reserved variable referring to the current object.

Figure 3.7(b) presents region-annotated Core-Java, the target language of our region inference system. This language extends Core-Java with region types and region constraints for each class and method. In addition, `letreg` declarations introduce local regions with lexical scopes. Every class declaration in the target language is parameterized with one or more regions; the first region parameter refers to the region in which the current object of the class is stored, while the remaining regions are used to store the class fields. The invariant associated with every class expresses mainly the no-dangling requirement. The instance methods of a subclass can override the instance methods of the superclass. Every method in the target language is decorated with zero or more region parameters; these parameters capture the regions used by each method's parameters (including `this`) and result. Each method also has a region lifetime constraint that is consistent with the operations performed in the method body.

3.6.2 Region Checking Rules

Our region type system guarantees that region-annotated Core-Java programs running using the region-based memory model described in Section 3.3 never create dangling references. To avoid

P	$::=$ *def**	(**program**)
def	$::=$ **class** cn_1 **extends** cn_2	(**class decl**)
	$\{(\tau\, f)^*\ meth^*\}$	(**class body**)
prim	$::=$ **int** $\mid$ **boolean** $\mid$ **void**	(**prim type**)
τ	$::=\ cn \mid prim \mid \bot$	(**type**)
meth	$::=\tau\ mn((\tau\ v)^*)\ \{e\}$	(**method decl**)
lhs	$::= v \mid v.f$	(**location**)
e	$::=$ **null** $\mid k \mid lhs$	(**expression**)
	$\mid \{(\tau\ v)\ \ e\}$	(**block decl**)
	$\mid$ **new** $cn(v^*) \mid lhs = e$	
	$\mid v.mn(v^*) \mid e_1\ ;\ e_2$	
	$\mid$ **if** v **then** e_1 **else** e_2	
	$\mid$ **while** $v\ e$	(**loop**)

(a) The Source Language

P	$::=$ *def**	(**region ann program**)
def	$::=$ **class** ca_1 **extends** ca_2	(**region ann class decl**)
	where φ	(**class invariant**)
	$\{(t\, f)^*\ meth^*\}$	(**class body**)
ca	$::=\ cn\langle r^+\rangle$	(**region ann class**)
t	$::=\ \tau\langle r^*\rangle$	(**region type**)
meth	$::= t\ mn\langle r^*\rangle((t\ v)^*)$	(**region ann meth**)
	where φ	(**meth precondition**)
	$\{e\}$	(**meth body**)
e	$::=$ **null** $\mid k \mid lhs$	(**region ann expression**)
	$\mid \{(t\ v)\ e\}$	(**region ann block**)
	$\mid$ **new** $ca(v^*) \mid lhs = e$	
	$\mid v.mn\langle r^*\rangle(v^*) \mid e_1\ ;\ e_2$	
	$\mid$ **if** v **then** e_1 **else** e_2	
	$\mid$ **while** $v\ e$	
	$\mid$ **letreg** r **in** e	(**region declaration**)

(b) The Target Language

$cn\ \in$ *class names*	$r\ \in$ *region variable names*
$mn\ \in$ *method names*	$\varphi\ \in$ *region constraints*
$f\ \in$ *field names*	$v\ \in$ *variable names*
$k\ \in$ *integer or boolean constants*	

Figure 3.7: A Fragment of Core-Java Syntax. Multiple inheritance and exceptions are discussed in Appendix A.6, while casting is presented in Appendix A.4.

[**RC–PROG**]
$$\frac{\begin{array}{c} \mathit{WFClasses}(P) \\ P = \mathit{def}_{i:1..n} \\ \mathit{FieldsOnce}(\mathit{def})_{i:1..n} \\ \mathit{MethodsOnce}(\mathit{def})_{i:1..n} \\ P \vdash \mathit{InheritanceOK}(\mathit{def})_{i:1..n} \\ P \vdash_{\mathit{def}} \mathit{def}_{i:1..n} \end{array}}{\vdash P}$$

[**RC–CLASS**]
$$\frac{\begin{array}{c} def = \textbf{class}\ cn\langle r_{1..n}\rangle \textbf{extends}\ c\langle r_{1..m}\rangle \\ \textbf{where}\ \varphi\ \{\mathit{field}_{1..p}\ \mathit{meth}_{1..q}\} \\ r_1 \notin \bigcup_{i=1}^{p}\ \mathit{reg}(\mathit{field}_i) \\ \varphi \Rightarrow r_i \succeq r_1 \quad i = 2..n \quad R = \{r_1, \ldots, r_n\} \\ P; \{\mathit{this} : cn\langle r_{1..n}\rangle\}; R; \varphi \vdash_{\mathit{meth}} \mathit{meth}_i \quad i = 1..q \\ P; R; \varphi \vdash_{\mathit{field}} \mathit{field}_i \quad i = 1..p \end{array}}{P \vdash_{\mathit{def}} def}$$

[**RC–METH**]
$$\frac{\begin{array}{c} \Gamma' = \Gamma + (v_j : t_j)_{j:1..p} \quad R' = R \cup \{r_1, \ldots, r_m\} \\ \varphi' = \varphi \wedge \varphi_0 \qquad P; R'; \varphi' \vdash_{\mathit{type}} t_j,\ j = 0..p \\ P; \Gamma'; R'; \varphi' \vdash e : t_0' \qquad P; R'; \varphi' \vdash t_0' <: t_0 \end{array}}{P; \Gamma; R; \varphi \vdash_{\mathit{meth}} t_0\ mn\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p})\textbf{where}\ \varphi_0\ \{e\}}$$

[**RC–EB**]
$$\frac{\begin{array}{c} P; R; \varphi \vdash_{\mathit{type}} t' \\ \Gamma' = \Gamma + (v : t') \\ P; \Gamma'; R; \varphi \vdash e : t \end{array}}{P; \Gamma; R; \varphi \vdash \{(t'\ v)\ e\} : t}$$

[**RC–CONS1**]
$$\frac{}{P; \Gamma; R; \varphi \vdash \textbf{null} : \bot}$$

[**RC–CONS2**]
$$\frac{}{P; \Gamma; R; \varphi \vdash k : \mathit{prim}\langle\rangle}$$

[**RC–VAR**]
$$\frac{(v : t) \in \Gamma}{P; \Gamma; R; \varphi \vdash v : t}$$

[**RC–FD**]
$$\frac{\begin{array}{c} (v : cn\langle a_{1..n}\rangle) \in \Gamma \\ (t\,f) \in \mathit{fieldlist}(cn\langle r_{1..n}\rangle) \end{array}}{P; \Gamma; R; \varphi \vdash v.f : [r_1 \mapsto a_1 ... r_n \mapsto a_n]t}$$

[**RC–NEW**]
$$\frac{\begin{array}{c} P; R; \varphi \vdash_{\mathit{type}} cn\langle r_{1..n}\rangle \\ \mathit{fieldlist}(cn\langle r_{1..n}\rangle) = (t_i\,f_i)_{i:1..p} \\ (v_i : t_i') \in \Gamma \quad P; R; \varphi \vdash t_i' <: t_i \quad i = 1..p \end{array}}{P; \Gamma; R; \varphi \vdash \textbf{new}\ cn\langle r_{1..n}\rangle(v_1, .., v_p) : cn\langle r_{1..n}\rangle}$$

[**RC–IF**]
$$\frac{\begin{array}{c} v : \textbf{boolean}\langle\rangle \in \Gamma \\ P; \Gamma; R; \varphi \vdash e_1 : t_1 \quad P; R; \varphi \vdash t_1 <: t \\ P; \Gamma; R; \varphi \vdash e_2 : t_2 \quad P; R; \varphi \vdash t_2 <: t \end{array}}{P; \Gamma; R; \varphi \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : t}$$

[**RC–ASSGN**]
$$\frac{\begin{array}{c} P; \Gamma; R; \varphi \vdash \mathit{lhs} : t \\ P; \Gamma; R; \varphi \vdash e : t' \\ P; R; \varphi \vdash t' <: t \end{array}}{P; \Gamma; R; \varphi \vdash \mathit{lhs} = e : \textbf{void}}$$

[**RC–SEQ**]
$$\frac{\begin{array}{c} P; \Gamma; R; \varphi \vdash e_1 : t_1 \\ P; \Gamma; R; \varphi \vdash e_2 : t_2 \end{array}}{P; \Gamma; R; \varphi \vdash e_1\textbf{;}e_2 : t_2}$$

[**RC–LOOP**]
$$\frac{\begin{array}{c} v : \textbf{boolean}\langle\rangle \in \Gamma \\ P; \Gamma; R; \varphi \vdash e : \textbf{void} \end{array}}{P; \Gamma; R; \varphi \vdash \textbf{while}\ v\ e : \textbf{void}}$$

[**RC–INVOKE**]
$$\frac{\begin{array}{c} (v_0 : cn\langle a^+\rangle) \in \Gamma \quad P; R; \varphi \vdash_{\mathit{type}} cn\langle a^+\rangle \\ P \vdash (t\ mn\langle a^+ r'^+\rangle((t_i\ v_i)_{i:1..n})\textbf{where}\ \varphi_0\ \{e\}) \in cn\langle a^+\rangle \\ (v_i' : t_i')_{i:1..n} \in \Gamma \quad a'^+ \in R \quad \rho = [r'^+ \mapsto a'^+] \\ \varphi \Rightarrow \rho\,\varphi_0 \quad P; R; \varphi \vdash t_i' <: \rho\,t_i \quad i = 1..n \end{array}}{P; \Gamma; R; \varphi \vdash v_0.mn\langle a^+ a'^+\rangle(v_1'..v_n') : \rho\,t}$$

[**RC–LETR**]
$$\frac{\begin{array}{c} a = \mathit{fresh}() \\ \varphi' = \varphi \wedge \bigwedge_{r' \in R}(r' \succeq a) \\ P; \Gamma; R \cup \{a\}; \varphi' \vdash [r \mapsto a]e : t \\ \mathit{reg}(t) \subseteq R \end{array}}{P; \Gamma; R; \varphi \vdash \textbf{letreg}\ r\ \textbf{in}\ e : t}$$

$\rho t,\ \rho\varphi,\ \rho e$	region substitution on a type, a constraint, and an expression
$\mathit{reg}(\mathit{field}),\ \mathit{reg}(t)$	computes the region variables of a field (or a type) (see Figure 4.6)
$\mathit{fieldlist}(cn\langle r_{1..n}\rangle)$	computes all fields of *cn* and their region types according to regions $r_{1..n}$ (see Figure 4.6)
$\mathit{fresh}()$	returns one or more new/unused region names

Figure 3.8: Region Type Checking Rules

$$\frac{P = ...\mathit{def}...}{\mathit{def} \in P} \qquad \frac{P \vdash \mathit{mbr} \in_D \mathit{cn}\langle r_{1..n}\rangle}{P \vdash \mathit{mbr} \in \mathit{cn}\langle r_{1..n}\rangle} \qquad \frac{\mathit{mbr}{=}\mathit{field}|\mathit{meth}\ \ \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle ...\{...\mathit{mbr}...\}{\in}P}{P \vdash \mathit{mbr} \in_D \mathit{cn}\langle r_{1..n}\rangle}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..m}\rangle ...{\in}P \\ P \vdash \mathit{mbr} \in \mathit{cn}'\langle r_{1..m}\rangle \ \ \neg(P \vdash \mathit{mbr}{\in_D}\mathit{cn}\langle r_{1..n}\rangle)\end{array}}{P \vdash \mathit{mbr} \in \mathit{cn}\langle r_{1..n}\rangle} \qquad \frac{\begin{array}{c}\vdash t <: t', \varphi' \quad \varphi{\Rightarrow}\varphi' \\ P;R;\varphi \vdash_{\mathit{type}} t \quad P;R;\varphi \vdash_{\mathit{type}} t'\end{array}}{P;R;\varphi \vdash t{<:}t'}$$

$$\frac{P;R \vdash_{\mathit{constr}} t, \varphi' \quad \varphi{\Rightarrow}\varphi'}{P;R;\varphi \vdash_{\mathit{type}} t} \qquad \frac{}{P;R \vdash_{\mathit{constr}} \mathit{prim}\langle\rangle,\ \mathit{true}} \qquad \frac{r \in R}{P;R \vdash_{\mathit{constr}} \mathit{Object}\langle r\rangle,\ \mathit{true}}$$

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

$$\frac{\begin{array}{c}P{=}\mathit{def}_{1..n} \qquad \mathit{def}_i{=}\textbf{class}\ \mathit{cn}_i\langle...\rangle\ \textbf{extends}\ \mathit{cn}_{i'}\langle...\rangle... \\ \mathit{IR}{=}\{(\mathit{cn}_i, \mathit{cn}_{i'}) \mid 1{\le}i{\le}n\} \quad \mathit{ID}{=}\{(\mathit{cn}_i, \mathit{cn}_i) \mid 1{\le}i{\le}n\} \\ \mathit{TransClosure}(\mathit{IR}){\cap}\mathit{ID}{=}\emptyset \quad \forall i, j{:}i{\neq}j \cdot \mathit{cn}_i{\neq}\mathit{cn}_j\end{array}}{\mathit{WFClasses}(P)}$$

$$\frac{\begin{array}{c}\mathit{def}{=}\textbf{class}\ \mathit{cn}\langle...\rangle...\{(\mathit{fd}_j)_{j:1..p}...\} \\ \forall j, l{:}j{\neq}l \cdot \mathit{name}(\mathit{fd}_j){\neq}\mathit{name}(\mathit{fd}_l)\end{array}}{\mathit{FieldsOnce}(\mathit{def})} \qquad \frac{\begin{array}{c}\mathit{def}{=}\textbf{class}\ \mathit{cn}\langle...\rangle...\{...(m_j)_{j:1..q}\} \\ \forall j, l{:}j{\neq}l \cdot \mathit{name}(m_j){\neq}\mathit{name}(m_l)\end{array}}{\mathit{MethodsOnce}(\mathit{def})}$$

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

$$\frac{\begin{array}{c}\mathit{meth} = t_0\ \mathit{mn}\langle x_{1..p(p+1)..q}, r_{1..n}\rangle((t\,v)_{i:1..m})\ \textbf{where}\ \varphi\ ... \\ \mathit{meth}' = t_0\ \mathit{mn}\langle x_{1..p}, r_{1..n}\rangle((t\,v)_{i:1..m})\ \textbf{where}\ \varphi'... \quad \varphi_0{\wedge}\varphi'{\Rightarrow}\varphi\end{array}}{P; \varphi_0 \vdash \mathit{OverridesOK}(\mathit{meth}, \mathit{meth}')}$$

Figure 3.9: Auxiliary Region Checking Rules

variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase. Region type checking rules are depicted in Figure 3.8, with some auxiliary rules in Figure 3.9. Judgments of the following forms are employed:

- $\vdash P$ denoting that a program P is well-typed.
- $P \vdash_{\mathit{def}} \mathit{def}$ denoting that a class declaration *def* is well-formed.
- $P; \Gamma; R; \varphi \vdash_{\mathit{meth}} \mathit{meth}$ denoting that a method *meth* is well-defined with respect to the program P, the type environment Γ, the set of live regions R, and region constraint φ.
- $P; \Gamma; R; \varphi \vdash e : t$ denoting that an expression e is well-typed with respect to the program P, the type environment Γ, the set of live regions R, and region constraint φ.
- $P; R; \varphi \vdash_{\mathit{type}} t$ denoting that a type t is well-formed, namely, the regions of the type t are

from the set of the live regions R, and the invariant of the type t is satisfied by the constraint context φ.

- $P; R \vdash_{constr} t,\ \varphi$ denoting that the regions of the type t are from the set of the live regions R, while φ is the invariant of the type t.
- $P; R; \varphi \vdash_{field}$ *field* denoting that the type of a field *field* is well-formed with respect to $\vdash_{type}$ judgment.
- $P; R; \varphi \vdash t{<:}t'$ denoting that the type t is a subtype of the type t', namely both types are well-formed and the region constraint of the subtyping relation (defined in Section 3.5) is satisfied by the constraint context φ.

The rule [RC−PROG] denotes that a region-annotated program is well-typed if all declared classes are well-typed. The predicates in the premise are used to capture the standard well-formedness conditions for the object-oriented programs, as follows:

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

These predicates are formulated in Figure 3.9, where the last two rules are used to check the soundness of the class subtyping and method overriding. Take note that $\varphi \Rightarrow \varphi'$ in the *InheritanceOK(def)* rule is to support the soundness of the class subtyping, while $\varphi_0 \wedge \varphi' \Rightarrow \varphi$ in the last rule is to ensure the soundness of the method overriding.

The rule [RC−CLASS] indicates that a class is well-formed if all its fields and methods are well-formed, and the class invariant ensures the necessary lifetime relations among class region parameters. In addition, the rule does not allow the first region of the class to be used by the region types of the fields. Using the first region on a field would break the object (region) subtyping (rule [ObjRegSub] of Figure 3.6). Function *reg* returns the region variables of a field type (see Figure 4.6).

The rule [RC−METH] checks the well-formedness of a method declaration. Each region type is checked to be well-formed, that means its regions are in the current set of live regions and its invariant is satisfied by the current constraint context. The method body is checked using the type relation for expressions such that the gathered type has to be a subtype of the declared type.

Our type relation for expressions is defined in a syntax-directed fashion. Take note that region constraints of the variables are not checked at their uses ([RC−VAR]), but at their declaration sites ([RC−EB]). The region invariant of an object is also checked when that object is created ([RC−NEW]). In the rule for object creation ([RC−NEW]), the function $\mathit{fieldlist}(cn\langle x_{1..n}\rangle)$ returns a list comprising all declared and inherited fields of the class $cn\langle x_{1..n}\rangle$ and their region types according to the regions $x_1..x_n$ of the class cn (see Figure 4.6). They are organized in an order determined by the constructor function.

The rule [RC−INVOKE] is used to check a method call. It ensures that the method region parameters are live regions and the method precondition is satisfied by the current constraint context as $\varphi \Rightarrow \rho\varphi_0$. A substitution ρ is computed for the method's formal region parameters. The current arguments are also checked to be subtypes of the method's formal parameters.

The rule [RC−LETR] is used to check a local region declaration. The local expression is checked with an extra live region a (that is a fresh region), and an extra constraint $\bigwedge_{r' \in R}(r' \succeq a)$ that ensures that new introduced region is on the top of the region stack. The rule uses a region substitution on the expressions. Note that the region substitutions on expressions, constraints and types are defined as expected. The gathered region type of the local expression is checked to contain only live regions (from R excepting a). This guarantees that the localized region a does not escape. Function $reg(t)$ returns all region variables of t (see Figure 4.6).

3.7 Formalism

First, we define the dynamic semantics of the target language. Then we show that our region type system for the target language is sound, meaning that the programs accepted by the type system do not create dangling pointers.

3.7.1 Dynamic Semantics

We define the dynamic semantics of region-annotated Core-Java as a small-step rewriting relation from machine states to machine states. A machine state has the form $\langle \varpi, \Pi \rangle [e]$ where ϖ is the heap organized as a stack of regions, Π is the variable environment, and e is the current program. Our dynamic semantics was inspired by the previous work on abstract models of memory management [125] and region-based memory management [41, 80]. The following notations are used:

Region Variables :	r, a	$\in$	*RegVar*
Offset :	o	$\in$	*Offset*
Locations :	ℓ *or* (r, o)	$\in$	*Location*=*RegVar*×*Offset*
Primitive Values :	k | `null`	$\in$	*Prim*
Values :	δ	$\in$	*Value* = *Prim* ⊎ *Location*
Variable Environment :	Π	$\in$	*VEnv* = *Var* $\rightharpoonup_{\text{fin}}$ *Value*
Field Environment :	V	$\in$	*FEnv* = *FieldName* $\rightharpoonup_{\text{fin}}$ *Value*
Object Values :	$cn\langle r^*\rangle(V)$	$\in$	*ObjVal* = *ClassName* × $(RegVar)^n$ × *FEnv*
Store :	ϖ	$\in$	*Store* = $[\,]\,\vert\,[r{\mapsto}Rgn]Store$
Runtime Regions :	*Rgn*	$\in$	*Region* = *Offset* $\rightharpoonup_{\text{fin}}$ *ObjVal*

Regions are identified by region variables. We assume a denumerably infinite set of region variables, *RegVar*. The store ϖ is organized as a stack, that defines an ordered map from region variables, r to runtime regions *Rgn*. The notation $[r{\mapsto}Rgn]\varpi$ denotes a stack with the region r on the top, while $[\,]$ denotes an empty store. The store can only be extended with new region variables. A runtime region *Rgn* is an unordered finite map from offsets to object values. We assume a denumerably infinite set of offsets, *Offset* for each runtime region *Rgn*.

The set of values that can be assigned to variables and fields is denoted by *Value*. Such a value is either a primitive value (a constant or a null value) or it is a location in the store. A location consists of a pair of a region variable and an offset.

An object value consists of a region type $cn\langle r^*\rangle$, and a field environment V mapping field names to values. V is not really an environment since it can only be updated, never extended. An update of field f with value δ is written as $V{+}\{f{\mapsto}\delta\}$.

The variable environment Π is a mapping $Var \rightharpoonup_{\text{fin}} Value$, while the type environment Γ that corresponds to the runtime variable environment is also a mapping $Var \rightharpoonup_{\text{fin}} Type$. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase.

Notation $f\colon A \rightharpoonup_{\text{fin}} B$ denotes a partial function from A to B with a finite domain, written $A{=}dom(f)$. We write $f{+}\{a \mapsto b\}$ for the function like f but mapping a to b (if $a{\in}dom(f)$ and

$f(a){=}c$ then

$(f{+}\{a\mapsto b\})(a){=}b$).

The notation $\{\}$ (or $\emptyset$) stands for an undefined function. Given a function $f\colon A\rightharpoonup_{\text{fin}} B$, the notation $f{-}C$ denotes the function $f_1 : (A{-}C)\rightharpoonup_{\text{fin}} B$ such that $\forall x{\in}(A{-}C)\cdot f_1(x)=f(x)$.

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

$$e ::= \ldots \mid (r,o) \mid \texttt{ret}(v,e) \mid \texttt{retr}(r,e)$$

The expression $\texttt{ret}(v,e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The variable associated with `ret` denotes either a block local variable or a method receiver or a method parameter. This variable is popped from the variable environment at the end of the block's evaluation. In the case of a method invocation there are multiple nested $\texttt{ret}s$ which pop off from the variable environment the receiver and the method parameters at the end of the method's evaluation. The expression $\texttt{retr}(r,e)$ is used to pop off the top, r of the store stack at the end of expression e evaluation.

Dynamic semantics rules of region annotated Core-Java are shown in Appendix A.1. The evaluation judgment is of the form:

$$\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']$$

where ϖ (ϖ') denotes the store before (after) evaluation, while Π (Π') denotes the variable environment before (after) evaluation.

The store ϖ organized as a stack establishes the outlive relations among regions at runtime. The function $\mathit{ord}(\varpi)$ returns the outlive relations for a given store, the function $\mathit{dom}(\varpi)$ returns the set of the store regions, while the function $\mathit{location\_dom}(\varpi)$ returns the set of all locations from the store. They are defined as follows:

$\mathit{ord}([r_1{\mapsto}\mathit{Rgn}_1][r_2{\mapsto}\mathit{Rgn}_2]\varpi){=}_{def}(r_2{\succeq}r_1){\wedge}\mathit{ord}([r_2{\mapsto}\mathit{Rgn}_2]\varpi)$

$\mathit{ord}([r{\mapsto}\mathit{Rgn}]) =_{def} \mathit{true} \quad \mathit{ord}([\,]) =_{def} \mathit{true}$

$\mathit{dom}([r{\mapsto}\mathit{Rgn}]\varpi){=}_{def}\{r\}{\cup}\mathit{dom}(\varpi) \quad \mathit{dom}([r{\mapsto}\emptyset]\varpi){=}_{def}\{r\}{\cup}\mathit{dom}(\varpi) \quad \mathit{dom}([\,]){=}_{def}\emptyset$

$\mathit{location\_dom}(\varpi){=}_{def}\{(r,o)\mid \varpi{=}\varpi_1[r{\mapsto}\mathit{Rgn}]\varpi_2 \wedge \mathit{Rgn}{\neq}\emptyset \wedge o{\in}\mathit{dom}(\mathit{Rgn})\}$

Notation $\varpi(r)(o)$ denotes an access into the region r at the offset o, as follows:

$$\varpi(r)(o){=}_{def}\mathit{Rgn}(o) \;\; \mathit{where} \;\; \varpi{=}\varpi_1[r{\mapsto}\mathit{Rgn}]\varpi_2$$

We define the meaning of *no-dangling references* property at runtime. The property refers to two kinds of references: (1) references from variable environment to store locations, and (2) references from store locations to store locations. Note that the notion of *no-dangling references* was introduced in Section 3.3, and a reference is formalized as a location (r, o) in this section.

Definition 3.7.1.1. *(live location) A location* (r, o) *is* live *with respect to a store* ϖ*, if* $r \in dom(\varpi)$.

Definition 3.7.1.2. *(no-dangling)*

1. *A variable environment* Π *is* no-dangling *with respect to a store* ϖ *if for all* $v \in dom(\Pi)$, $\Pi(v)$ *is either a primitive value, or a live location* (r, o) *with respect to* ϖ.

2. *A runtime store* ϖ *is* no-dangling *if each region* $r_1 \in dom(\varpi)$ *contains only references to regions older than itself, that means that for each location* (r_1, o) $\in location\_dom(\varpi)$ *containing an object value* $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$ *that object satisfies the* non-dangling requirement for a class, *such that* $ord(\varpi) \Rightarrow \bigwedge_{i:2..n}(r_i \succeq r_1)$ *and the current values of the fields are either primitives or references to regions older than those expected by the region type* $cn\langle r_{1..n}\rangle$*, as follows:*

   $$\forall f \in dom(V)\ .\ V(f){=}(r_f, o_f) \quad ord(\varpi) \Rightarrow r_f \succeq fieldregion(cn\langle r_{1..n}\rangle, f)$$

   Function $fieldregion(cn\langle r_{1..n}\rangle, f)$ *computes the region type of the class field f and then returns its first region where the field is expected to be stored.*

The dynamic semantics evaluation rules may yield two possible runtime errors, namely:

$$Error ::= \texttt{nullerr} \mid \texttt{danglingerr}$$

The first error `nullerr` is due to null pointers (by accessing fields or methods of null objects). The second error `danglingerr` is reported when a store updating operation or a variable environment updating operation creates a dangling reference.

Our dynamic semantics rules use runtime checks to guarantee that a `danglingerr` error is reported (and the execution is aborted) whenever the program evaluation tries to create a dangling reference. There are five situations that require no-dangling reference checks at runtime:

- creation of a new object value, where we check the class invariant, mainly whether the fields regions outlive the region of the object. We also check whether the initial values of the fields are stored in regions that outlive the corresponding expected regions.
- updating an object field, where we check whether the new value is stored into a region that outlives the expected region for that field. The expected region of an object field is computed from the regions of the object value.

- updating a variable from the variable environment with a new location, where we check whether the new location is live.
- deallocation of a region, where we check whether the computed result, the variable environment, and the store locations do not contain references to the deallocated region. We also check whether the deallocated region is on the top of the current store.
- calling a method, where we check whether the method's region arguments are in the current store.

The static semantics of the language is also extended to include the new intermediate expressions. The process requires introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics [151]. In our case, it is denoted by: $\Sigma \in \mathit{StoreType} = \mathit{RegVar} \rightharpoonup_{\text{fin}} \mathit{Offset} \rightharpoonup_{\text{fin}} \mathit{Type}$. The judgments of static semantics are extended with store typing, as follows:

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

For a store typing $\Sigma : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type}$, a region r, a location (r,o), and a type t we introduce the following notations:

$$\begin{array}{l}
\mathit{dom}(\Sigma){=}R \qquad \Sigma(r)(o){=}f(o),\ \mathit{where}\ f{=}\Sigma(r) \\
\mathit{location\_dom}(\Sigma){=}_{def}\{(r,o) \mid r{\in}\mathit{dom}(\Sigma) \wedge f{=}\Sigma(r) \wedge f{\neq}\emptyset \wedge o{\in}\mathit{dom}(f)\} \\
\Sigma{-}r{=}_{def}\Sigma_1\ \mathit{such\ that}\ \Sigma_1 : (R{-}\{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_1(r'){=}\Sigma(r') \\
\Sigma{+}r{=}_{def}\Sigma_2\ \mathit{such\ that}\ \Sigma_2 : (R{\cup}\{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge \Sigma_2(r){=}\emptyset \wedge \forall r'{\in}R \cdot \Sigma_2(r'){=}\Sigma(r') \\
\Sigma{-}(r,o){=}_{def}\Sigma_3\ \mathit{such\ that}\ \Sigma_3 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r{\in}R \wedge \Sigma_3(r){=}\Sigma(r){-}\{o\} \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_3(r'){=}\Sigma(r') \\
\Sigma{+}((r,o) : t){=}_{def}\Sigma_4\ \mathit{such\ that}\ \Sigma_4 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r{\in}R \wedge \Sigma_4(r){=}\Sigma(r){+}\{o{\mapsto}t\} \wedge \forall r'{\in}(R{-}r) \cdot \Sigma_4(r'){=}\Sigma(r')
\end{array}$$

Definition 3.7.1.3. *The function vars(e) computes the set of all program variables which occur in the expression e, excepting those variables introduced by e's block subexpressions, as follows:*

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{vars}(e) \ =_{def} \ \mathit{case\ e\ of}} \\
\quad \texttt{ret}(v,e) & \rightarrow & \{v\} \cup \mathit{vars}(e) \\
\quad \{(t\ v)\ \ e\} & \rightarrow & \mathit{vars}(e) \setminus \{v\} \\
\quad \texttt{retr}(r,e) \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{vars}(e) \\
\quad v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow & \{v\} \cup \mathit{vars}(e) \\
\quad v.f \mid v & \rightarrow & \{v\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \{v\} \cup \mathit{vars}(e_1) \cup \mathit{vars}(e_2) \\
\quad e_1\ \textbf{;}\ e_2 & \rightarrow & \mathit{vars}(e_1) \cup \mathit{vars}(e_2) \\
\quad \textbf{new}\ cn\langle r^+\rangle(v^*) & \rightarrow & \{v^*\} \\
\quad v.mn\langle r^*\rangle(v^*) & \rightarrow & \{v\} \cup \{v^*\} \\
\quad \mathit{otherwise} & \rightarrow & \emptyset
\end{array}$$

Definition 3.7.1.4. *The function retvars(e) computes the set of all program variables which occur in the ret subexpressions of the expression e, as follows:*

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{retvars}(e) \ =_{def} \ \mathit{case\ e\ of}} \\
\quad \texttt{ret}(v,e) & \rightarrow & \{v\} \cup \mathit{retvars}(e) \\
\quad \texttt{retr}(r,e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & \mathit{retvars}(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{retvars}(e) \\
\quad e_1\ \textbf{;}\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \mathit{retvars}(e_1) \cup \mathit{retvars}(e_2) \\
\quad \mathit{otherwise} & \rightarrow & \emptyset
\end{array}$$

Definition 3.7.1.5. *The function regs(e) computes the set of all region variables which occur in the expression e, excepting those regions introduced by e's letreg subexpressions, as follows:*

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{regs}(e) \ =_{def} \ \mathit{case\ e\ of}} \\
\quad \{(t\ v)\ \ e\} & \rightarrow & \mathit{reg}(t) \cup \mathit{regs}(e) \\
\quad \texttt{retr}(r,e) & \rightarrow & \{r\} \cup \mathit{regs}(e) \\
\quad \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{regs}(e) \setminus \{r\} \\
\quad \texttt{ret}(v,e) \mid v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow & \mathit{regs}(e) \\
\quad (r,o) & \rightarrow & \{r\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid e_1\ \textbf{;}\ e_2 & \rightarrow & \mathit{regs}(e_1) \cup \mathit{regs}(e_2) \\
\quad \textbf{new}\ cn\langle r^+\rangle(v^*) \mid v.mn\langle r^+\rangle(v^*) & \rightarrow & \{r^+\} \\
\quad \mathit{otherwise} & \rightarrow & \emptyset
\end{array}$$

where reg(t) is defined in the Figure 4.6.

Definition 3.7.1.6. *The function retregs(e) computes the set of all region variables which occur* *in the retr subexpressions of the expression e, as follows:*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{retregs}(e) =_{def} \ \mathit{case\ e\ of}} \\
\quad \texttt{retr}(r,e) & \rightarrow & \{r\} \cup \mathit{retregs}(e) \\
\quad \texttt{ret}(v,e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & \mathit{retregs}(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{retregs}(e) \\
\quad e_1\ \textbf{;}\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \mathit{retregs}(e_1) \cup \mathit{retregs}(e_2) \\
\quad \mathit{otherwise} & \rightarrow & \emptyset
\end{array}
$$

Definition 3.7.1.7. *(valid program)*

1. *An expression* e *is a* valid expression *if the predicate valid*(e) *holds, where valid*(e) *is defined as follows:*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{valid}(e) =_{def} \ \mathit{case\ e\ of}} \\
\quad \{(t\ v)\ e\} & \rightarrow & \mathit{retvars}(e){=}\emptyset \wedge \mathit{retregs}(e){=}\emptyset \\
\quad \mathit{lhs} = e & \rightarrow & \mathit{retvars}(e) \cap \mathit{vars}(\mathit{lhs}){=}\emptyset \wedge \mathit{valid}(e) \\
\quad e_1\ \textbf{;}\ e_2 & \rightarrow & \mathit{retregs}(e_2){=}\emptyset \wedge \mathit{retvars}(e_2){=}\emptyset \wedge \mathit{valid}(e_1) \\
 & & \wedge \mathit{retvars}(e_1) \cap \mathit{vars}(e_2){=}\emptyset \wedge \mathit{retregs}(e_1) \cap \mathit{regs}(e_2){=}\emptyset \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \mathit{retregs}(e_1){=}\emptyset \wedge \mathit{retvars}(e_1){=}\emptyset \\
 & & \wedge \mathit{retregs}(e_2){=}\emptyset \wedge \mathit{retvars}(e_2){=}\emptyset \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{retregs}(e){=}\emptyset \wedge \mathit{retvars}(e){=}\emptyset \\
\quad \texttt{ret}(v,e) & \rightarrow & v \notin \mathit{retvars}(e) \wedge \mathit{valid}(e) \\
\quad \texttt{retr}(r,e) & \rightarrow & r \notin \mathit{retregs}(e) \wedge \mathit{valid}(e) \\
\quad \mathit{otherwise} & \rightarrow & \mathit{true}
\end{array}
$$

2. *A method is a* valid method *if the method's body* e*, is a valid block expression such that retvars*$(e){=}\emptyset$ *and retregs*$(e){=}\emptyset$*.*

3. *A class is a* valid class *if all the class's methods are valid methods.*

4. *A program is a* valid program *if all the program's classes are valid classes.*

Note that a source language Core-Java program is by default a valid program since it does not contain any intermediate expression.

Definition 3.7.1.8. *Using the evaluation rules from Appendix A.1, the function lvar*(e) *estimates* *the set of variables which may be popped off from the variable environment* Π *during the evaluation of the valid expression* e *(note that only* $\texttt{ret}(v,e)$ *may affect* Π*), as follows:*

$$\begin{array}{lll} \mathit{lvar}(e) =_{def} \mathit{case}\ e\ \mathit{of} & & \\ \quad \texttt{ret}(v,e) & \rightarrow & \{v\} \cup \mathit{lvar}(e) \\ \quad \texttt{retr}(r,e) \mid \mathit{lhs} = e \mid e\,;e_1 & \rightarrow & \mathit{lvar}(e) \\ \quad \mathit{otherwise} & \rightarrow & \emptyset \end{array}$$

Definition 3.7.1.9. *Using the evaluation rules from Appendix A.1, the function* $\mathit{lreg}(e)$ *estimates the set of regions which may be popped off from the store* ϖ *during the evaluation of the valid expression* e *(note that only* $\texttt{retr}(r,e)$ *may affect* ϖ*), as follows:*

$$\begin{array}{lll} \mathit{lreg}(e) =_{def} \mathit{case}\ e\ \mathit{of} & & \\ \quad \texttt{retr}(r,e) & \rightarrow & \{r\} \cup \mathit{lreg}(e) \\ \quad \texttt{ret}(v,e) \mid \mathit{lhs} = e \mid e\,;e_1 & \rightarrow & \mathit{lreg}(e) \\ \quad \mathit{otherwise} & \rightarrow & \emptyset \end{array}$$

Definition 3.7.1.10. *Using the evaluation rules from Appendix A.1, the function* $\mathit{lloc}(e)$ *estimates the new location which may be created into an existing region during one evaluation step of the valid expression* e *(note that only* **new** *may create a new location), as follows:*

$$\begin{array}{lll} \mathit{lloc}(e) =_{def} \mathit{case}\ e\ \mathit{of} & & \\ \quad \mathbf{new}\ cn\langle r_1,..,r_n\rangle(v^*) & \rightarrow & \{(r_1,o)\} \\ \quad \texttt{ret}(v,e) \mid \texttt{retr}(r,e) \mid \mathit{lhs} = e \mid e\,;e_1 & \rightarrow & \mathit{lloc}(e) \\ \quad \mathit{otherwise} & \rightarrow & \emptyset \end{array}$$

where the offset o *of the region* r *is the offset where the next allocation in* r *is done.*

The judgments of the new intermediate expressions are presented in Figure 3.10. They assume that the expressions are valid with respect to the Definition 3.7.1.7. The first two rules [RC−LOCATION] and [RC−ObjVal] are used to type the store, either a location or an object value (i.e. a location's content). Rule [RC−ObjVal] preserves the same invariants as those of the rule [RC−NEW]. Rule [RC−RET] ensures that the variable to be popped off, v is in the current environment Γ. The subsumption rule, [SUBSUMPTION], simplifies the next theorems and their proofs.

Rule [RC−RETR] is similar to rule [RC−LETR], but it takes into account the evaluation of the expression $\texttt{retr}(r,e)$. The first check ensures that the region to be deallocated, a is in R. The R_t denotes the regions of R which are different than a and are not younger than a. Note that $\mathit{lreg}(e)$ denotes the regions which are younger than a. The second check ensures that our type system uses only lexically scoped regions such that the region to be deallocated, a is always on

$$\frac{r\in R \quad \Sigma(r)(o)=t}{P;\Gamma;R;\varphi;\Sigma \vdash (r,o) : t}\ \boxed{\text{RC−LOCATION}}$$

$$\frac{P;R;\varphi\vdash_{type} cn\langle r_{1..n}\rangle \quad \mathit{fieldlist}(cn\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p} \quad P;\Gamma;R;\varphi;\Sigma\vdash V(f_i):t_i' \quad P;R;\varphi\vdash t_i' <: t_i \quad i=1..p}{P;\Gamma;R;\varphi;\Sigma\vdash cn\langle r_{1..n}\rangle(V) : cn\langle r_{1..n}\rangle}\ \boxed{\text{RC−ObjVal}}$$

$$\frac{v\in\Gamma \quad P;\Gamma;R;\varphi;\Sigma\vdash e:t}{P;\Gamma;R;\varphi;\Sigma\vdash \mathtt{ret}(v,e):t}\ \boxed{\text{RC−RET}}$$

$$\frac{P;\Gamma;R;\varphi;\Sigma\vdash e:t' \quad P;R;\varphi\vdash t' <: t}{P;\Gamma;R;\varphi;\Sigma\vdash e:t}\ \boxed{\text{SUBSUMPTION}}$$

$$\frac{a\in R \quad R_t=R-lreg(e)-\{a\} \quad \varphi\Rightarrow\bigwedge_{r\in R_t}(r\succeq a) \quad reg(t)\subseteq R_t \quad reg(\Gamma-lvar(e))\subseteq R_t \quad P;\Gamma;R;\varphi;\Sigma\vdash e:t}{P;\Gamma;R;\varphi;\Sigma\vdash \mathtt{retr}(a,e):t}\ \boxed{\text{RC−RETR}}$$

$reg(t)$	computes the region variables of a type (see Figure 4.6)
$\mathit{fieldlist}(cn\langle r_{1..n}\rangle)$	computes all fields of *cn* and their region types according to regions $r_{1..n}$ (see Figure 4.6)

Figure 3.10: Region Type Checking Rules for Valid Intermediate Expressions

the top of the regions stack. The third and the fourth check ensure that the region a and the regions younger than a do not escape either through the result or through the live variables of the type environment. Note that $lvar(e)$ denotes the local variables of the expression e which are deallocated from the variable environment during the evaluation of e.

3.7.2 Safety Proof

By using the standard techniques found in [204] we show that a valid program well-typed by the type system we have presented, never creates dangling references. In what follows, we formulate the type preservation theorem and the progress theorem. The soundness of our static semantics relies on the following consistency relationship between the static and dynamic semantics.

Definition 3.7.2.1. *(consistency) A run-time environment* (ϖ,Π) *is* consistent *with a static environment* $(\Gamma,R,\varphi,\Sigma)$*, written* $\Gamma,R,\varphi,\Sigma\models\langle\varpi,\Pi\rangle$*, if the following judgment holds:*

$$\frac{\begin{array}{c} dom(\Gamma)=dom(\Pi) \quad \forall v\in dom(\Pi)\cdot P;\Gamma;R;\varphi;\Sigma\vdash\Pi(v):\Gamma(v) \quad reg(\Gamma)\subseteq R \\ location\_dom(\Sigma)=location\_dom(\varpi) \quad dom(\Sigma)=dom(\varpi) \quad R=dom(\varpi) \quad ord(\varpi)\Rightarrow\varphi \\ \forall(r,o)\in location\_dom(\varpi)\cdot P;\Gamma;R;\varphi;\Sigma\vdash\varpi(r)(o):\Sigma(r)(o)\end{array}}{\Gamma,R,\varphi,\Sigma\models\langle\varpi,\Pi\rangle}$$

Note that $\varpi(r)(o)$ returns an object value $cn\langle r^*\rangle(V)$ whose type is $cn\langle r^*\rangle$. In our instrumented operational semantics an object value and its type are stored together.

The subject reduction theorem ensures that the type is preserved during the evaluation of a valid program, as follows:

Theorem 3.7.2.1. *(Subject Reduction): If*

$$valid(e) \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \models \langle \varpi, \Pi \rangle$$
$$\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$$

then there exist Σ', Γ', R', *and* φ', *such that*

$$(\Sigma' - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$$
$$\Gamma' - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$$
$$R' - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$$
$$\varphi' - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$$
$$\Gamma', R', \varphi', \Sigma' \models \langle \varpi', \Pi' \rangle$$
$$valid(e') \quad P;\Gamma';R';\varphi';\Sigma' \vdash e' : t.$$

Proof: By structural induction on e. The detailed proof is in Appendix A.2.2.

Although the hypothesis of the above theorem contains an evaluation relation, the proof does not use the runtime checks associated with the evaluation rules to prove that the result of the evaluation (result and dynamic environment) is well-typed, valid and consistent.

The following theorem guarantees that the evaluation of a valid program cannot generate `danglingerr` errors, by proving that those runtime checks are redundant for a well-typed valid program (the runtime checks are proved by the static semantics).

Theorem 3.7.2.2. *(Progress) If*

$$valid(e) \quad P;\Gamma;R;\varphi;\Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \models \langle \varpi, \Pi \rangle$$

then either

- *e is a value, or*
- $\langle \varpi, \Pi \rangle[e] \hookrightarrow$`nullerr` *or*
- *there exist* ϖ', Π', e' *such that* $\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$.

Proof: By induction over the depth of the type derivation for expression e. The detailed proof is in Appendix A.2.3.

We conclude with the following soundness theorem for Core-Java. The theorem states that if a valid program is well-typed and is evaluated in a runtime environment consistent with the static environment, the result is (1) either an error different than dangling error, (2) or a value, (3) or the program diverges. The evaluation never reports dangling errors, namely the program never creates dangling references.

Theorem 3.7.2.3. *(Soundness) Given a well-typed valid Core-Java program P=def* and the main function* $(void\ \mathbf{main}(void)\{e_0\})\in P$*, where* e_0 *is a well-typed valid closed term (without free regions and free variables), such that* $retvars(e_0)=\emptyset \wedge retregs(e_0)=\emptyset$ *and* $P;\Gamma_0;R_0;\varphi_0;\Sigma_0 \vdash e_0 : void$*, where* $\Gamma_0=\emptyset$*,* $R_0=\emptyset$*,* $\varphi_0=true$*, and* $\Sigma_0=\emptyset$ *. Starting from the initial runtime environment* $\langle\varpi_0,\Pi_0\rangle$*, where* $\varpi_0=[\,]$*,* $\Pi_0=\emptyset$*, such that* $\Gamma_0,R_0,\varphi_0,\Sigma_0 \models \langle\varpi_0,\Pi_0\rangle$*. Then either*

$$(1) \qquad \langle\varpi_0,\Pi_0\rangle[e_0] \hookrightarrow^* \texttt{nullerr}$$

or there exist a store ϖ*, a variable environment* Π*, a value* δ*, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

$$(2)\ \ \langle\varpi_0,\Pi_0\rangle[e_0] \hookrightarrow^* \langle\varpi,\Pi\rangle[\delta] \quad \Gamma,R,\varphi,\Sigma \models \langle\varpi,\Pi\rangle \quad P;\Gamma;R;\varphi;\Sigma \vdash \delta : void$$

or for a store ϖ*, a variable environment* Π*, a valid expression* e*, a type environment* Γ*, a set of regions* R*, a region constraint* φ*, a store typing* Σ *such that*

$$\langle\varpi_0,\Pi_0\rangle[e_0] \hookrightarrow^* \langle\varpi,\Pi\rangle[e] \quad \Gamma,R,\varphi,\Sigma \models \langle\varpi,\Pi\rangle \quad P;\Gamma;R;\varphi;\Sigma \vdash e : void \quad valid(e)$$

there exist a store ϖ'*, a variable environment* Π'*, an expression* e'*, a type environment* Γ'*, a set of regions* R'*, a region constraint* φ'*, a store typing* Σ' *such that*

$$(3)\ \ \langle\varpi,\Pi\rangle[e] \hookrightarrow \langle\varpi',\Pi'\rangle[e'] \quad \Gamma',R',\varphi',\Sigma' \models \langle\varpi',\Pi'\rangle \ \ P;\Gamma';R';\varphi';\Sigma' \vdash e' : void \ \ valid(e')$$

Proof: The proof is an induction on the number of the reduction steps. We can repeatedly use the progress theorem (Theorem 3.7.2.2) to prove that there is a reduction step and then the preservation theorem (Theorem 3.7.2.1) to prove that the runtime environment after evaluation is still well-typed and the evaluation result is valid.

Although the type *void* typically denotes the empty type, here we assume *void* to be isomorphic to the type *unit*. Thus, the singleton value of type $void$ is $()$.

3.7.3 Comparison to Other Proofs

We have proven the safety properties of our region type system by induction. In previous effect-based region type system, Tofte and Talpin [192, 189, 17] and Christiansen and Velschow [41] made use of co-induction to prove the soundness. Their proof requires co-induction partly because they prove two properties at the same time: type soundness and translation soundness. The latter property guarantees that there exists a semantic relation between source program and its region-annotated counterpart. Our safety theorems are only focused on the problem of type soundness, thus are simpler to prove. A coinductive definition is required in their proof also because they use a big-step semantics where certain information is lost when deleting a region from the store, as discussed in [91, 29]. Our system uses a small-step operational semantics instrumented with regions which makes the consistency definition and the proof easier. Later, Calcagno [28] used stratified operational semantics to avoid co-induction in the proof of safety properties of a simple region calculus, while Helsen [90] introduced a special constant for defunct regions in their big-step semantics which made the soundness proof simpler. A similar proof with ours is the safety proof of Niss [133], that in addition to a simple functional language handles an imperative calculus, and like our proof avoids explicit co-induction by using store typing. An early example of a proof with store types is in [1]. Cyclone [80] also has an effect system used for a soundness proof and does not use coinduction. Elsman [60] has refined Tofte and Talpin's region type system in order to forbid the dangling references and proved by induction the safety for a small functional language. There are many differences between his proof and ours. His proof is based on a small-step contextual semantics [124], while in our proof we explicitly modelled the heap as a stack of regions and we used a consistency relation between the static and dynamic semantics. In addition Elsman used a syntax-directed containment relation to express the regions of the program values and also to force the stack dicipline for regions'allocation and deallocation. In our case the region requirements and the order among regions are expressed by the type system region constraints. However we also imposed a syntactic condition to restrict the valid non-source programs.

CHAPTER 4

REGION INFERENCE

4.1 Algorithm Overview

The goal of our region inference is to automatically set a region for each object of the input program such that the output program uses the region-based memory model described in Section 3.3 and never creates dangling references. The trivial solution is to put everything in one region (that could be a pre-allocated region that lives forever), but our region inference is going to aim for a better solution where better means to put objects into regions with shorter lifetime, whenever our system can guarantee that it is safe to do so. Given a Core-Java program `P`, our algorithm infers the appropriate region annotations for each class, method, and expression from `P` generating as output a region-annotated Core-Java program `P`'. The input program is assumed to be a *well-typed* Core-Java program. The region inference algorithm is designed as summary-based flow insensitive analysis and consists of the following main components:

1. A Dependency Graph to guide the Summary-Based Flow Insensitive Analysis.

   Our region inference is designed as a summary-based analysis [202] for classes and methods. The summary of the class consists of the class region parameters and the class invariant, while the summary of the method consists of the method region parameters and the method precondition. In general the class invariant corresponds to the non-dangling requirement, while the method precondition encapsulates the method effect. Our analysis traverses each class (method) only once to collect the summary. The summary is expressed as a constraint abstraction that usually is not in a closed-form. The gathered summary may contain the constraint abstractions of other classes and/or methods on which the current class/method depends. In order to compute the closed-form of the current class/method summary, it is required to process the classes and methods in some particular order given by the complex inter-dependency between classes and methods. Therefore a dependency graph of classes and methods is built. The final dependency graph has the classes and methods organized into a hierarchy of strongly connected components (SCCs). Each SCC is separately analysed, first the classes and then the methods. The summaries of classes (methods) in the same SCC have cyclic dependencies, so they are computed simultaneously by a fixed point iteration. In contrast, the summaries of classes (methods) in different SCCs have hierarchical dependencies (or no dependence at all), and hence are computed by bottom-up traversal on SCCs, without iteration. Section 4.7 presents more details about the dependencies and also discusses the case of the mutually-dependent classes/methods. The method overriding aspects are presented in Section 4.6. For simplicity, from Section 4.1 to Section 4.5, we assume that each SCC contains at most one class and one method.

2. Inference for a Class.

   The region parameters and the class invariant of a class are inferred. The algorithm is based on the principles introduced in Section 3.4 and consists of the following steps, formalized in Section 4.2:

   (a) Inherit the regions and the class invariant of the superclass.

   (b) Reserve the first region for the receiver object (`this`).

   (c) Compute the regions for the fields. The recursive fields should have the same region annotation as the class, except for the first region. All recursive fields reuse the same special region as their first region. This special region is the last region parameter of the class.

   (d) Add the fields' regions to the region annotation of the class.

   (e) Add the fields' constraints (corresponding to the class invariants of the fields' region types) to the class invariant.

   (f) Add the no-dangling requirements to the class invariant.

3. Inference for a Method.

   The method region parameters and the method precondition are inferred. The algorithm consists of the following steps:

   (a) Compute the region parameters of the method based on the region parameters of the method's parameters (including `this`) and the method result (Section 4.5).

   (b) Gather constraints from the method body and compute the method precondition. The method precondition naturally depends on how the method manipulates the

(a) Acyclic Data Structure

```
class Pair extends Object {
   Object fst;
   Object snd;
   void setSnd(Object o) {snd=o;}
   Pair example(){
    Pair p1,p2,p3,p4;
    p4 = new Pair(null,null);
    p3 = new Pair(p4,null);
    p2 = new Pair(null,p4);
    p1 = new Pair(p2,null);
    p1.setSnd(p3);
    p2}
}
```

(b) Source program

Figure 4.1: Core-Java input program

objects (Section 4.3). In the case of a recursive method, a fixpoint iteration is performed to compute the method precondition (Section 4.5).

(c) Introduce `letreg` expressions in the method body (Section 4.4). Only the global regions (used to annotate the method parameters, the method receiver and the method result) can escape outside of the method body. The regions younger than the global regions are localised by `letreg` expressions. The regions older than the global regions are made equivalent to suitable global regions.

4.1.1 An Example

We next illustrate our region inference algorithm with a simple example. Figure 4.1(b) shows the Core-Java input program. Consider a `Pair` class with two fields of type `Object`. For simplicity, this class has only two methods: `setSnd` that modifies the value of the second field and `example` that builds the acyclic data structure drawn in Figure 4.1(a). First we compute the dependency graph. Thus, we obtain the following dependencies:

- `Pair.example`→`Pair`: denotes that the method `Pair.example` makes use of the class `Pair` in its body

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

(a) Inference for the class

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1
{  ...
   void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o) where ...
     {snd=o;}
}
```

(b) Inference for the method region parameters

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1
{  ...
   void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o) where r4⪰r3
     {snd=o;} // r4⪰r3
}
```

(c) Inference for the method precondition

Figure 4.2: Inference of Pair Class and Pair.setSnd Method

- `Pair.example`→`Pair.setSnd`: denotes that the method `Pair.example` calls the method `Pair.setSnd`
- `Pair.setSnd`→`Pair`: denotes that the method `Pair.setSnd` makes use of the class `Pair` in its body

Based on the computed dependencies, we perform region inference in the following order: (i) the inference for the class `Pair`, (ii) the inference for the method `Pair.setSnd`, and (iii) the inference for the method `Pair.example`.

The result of the inference for the class `Pair` is shown in Figure 4.2(a). The class `Pair` is annotated with three region parameters: region `r1` is for `this`, region `r2` stores the first field `fst` and region `r3` stores the second field `snd`. The class invariant `r2⪰r1∧r3⪰r1` denotes the non-dangling requirement.

Figures 4.2(b) and 4.2(c) present the inference for the method `Pair.setSnd`. First the method region parameters are inferred based on the region types of the parameter `o` and the receiver `this`. Then the method body is analysed. The body generates the constraint `r4⪰r3` based on the object (region) subtyping for assignment (see Section 4.3). This constraint refers only to the method region parameters and therefore it becomes part of the method precondition.

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1
{  ...
   Pair⟨r5,r6,r7⟩ example⟨r1,r2,r3,r5,r6,r7⟩()
       where pre.Pair.example⟨r1,r2,r3,r5,r6,r7⟩ {
    Pair⟨l4,l4a,l4b⟩ p4;        //l4a⪰l4∧l4b⪰l4
    Pair⟨l3,l3a,l3b⟩ p3;        //l3a⪰l3∧l3b⪰l3
    Pair⟨l2,l2a,l2b⟩ p2;        //l2a⪰l2∧l2b⪰l2
    Pair⟨l1,l1a,l1b⟩ p1;        //l1a⪰l1∧l1b⪰l1
    p4 = new Pair⟨n4,n4a,n4b⟩(null,null);
    //n4⪰l4∧l4a=n4a∧l4b=n4b∧n4a⪰n4∧n4b⪰n4
    p3 = new Pair⟨n3,n3a,n3b⟩(p4,null);
    //l4⪰n3a∧n3⪰l3∧l3a=n3a∧l3b=n3b∧n3a⪰n3∧n3b⪰n3
    p2 = new Pair⟨n2,n2a,n2b⟩(null,p4);
    //l4⪰n2b∧n2⪰l2∧l2a=n2a∧l2b=n2b∧n2a⪰n2∧n2b⪰n2
    p1 = new Pair⟨n1,n1a,n1b⟩(p2,null);
    //l2⪰n1a∧n1⪰l1∧l1a=n1a∧l1b=n1b∧n1a⪰n1∧n1b⪰n1
    p1.setSnd⟨l1,l1a,l1b,l3⟩(p3);      //l3⪰l1b
    p2        //l2⪰r5∧l2a=r6∧l2b=r7
   }
}
```

Figure 4.3: Initial Region-Annotation of Pair.example Method

Gathered Region Constraints:

```
n2b=l2b∧l2b=r7∧n2a=l2a∧l2a=r6∧l1a=n1a∧l1b=n1b∧l4a=n4a∧l4b=n4b∧l3a=n3a
∧l3b=n3b∧l3⪰l1b∧l4a⪰l4∧l4b⪰l4∧l3a⪰l3∧l3b⪰l3∧r6⪰l2∧r7⪰l2∧l1a⪰l1
∧l1b⪰l1∧n4⪰l4∧l4a⪰n4∧l4b⪰n4 ∧l4⪰l3a∧n3⪰l3∧l3a⪰n3∧l3b⪰n3
∧l4⪰r7∧n2⪰l2∧r6⪰n2∧r7⪰n2∧l2⪰l1a ∧n1⪰l1∧l1a⪰n1∧l1b⪰n1∧l2⪰r5
```

Global and Local Regions:

```
GlobalRegions={r1,r2,r3,r5,r6,r7}
OutliveGlobalRegions={l4a,n4a,l4b,n4b,n4,l4,l2,n2,l2b,n2b,l2a,n2a}
LocalRegions={l1,n1,l1a,n1a,l1b,n1b,l3,n3,l3a,n3a,l3b,n3b}
```

Solutions for the Regions that outlive Global Regions

```
l2=n2=r5∧n2a=l2a=r6∧n2b=n4=n4a=n4b=l4=l4a=l4b=l2b=r7
```

Localising the Regions using Letreg

```
l1a=n1a=n1=l1=n1b=l1b=l3=n3=n3a=l3a=n3b=l3b=r
```

Figure 4.4: Solving region constraints

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r2⪰r1 ∧ r3⪰r1
{  ...
Pair⟨r5,r6,r7⟩ example⟨r1,r2,r3,r5,r6,r7⟩()where r7⪰r5∧r6⪰r5{
   letreg r in {
    Pair⟨r7,r7,r7⟩ p4;
    Pair⟨r,r,r⟩ p3;
    Pair⟨r5,r6,r7⟩ p2;
    Pair⟨r,r,r⟩ p1;
    p4 = new Pair⟨r7,r7,r7⟩(null,null);
    p3 = new Pair⟨r,r,r⟩(p4,null);
    p2 = new Pair⟨r5,r6,r7⟩(null,p4);
    p1 = new Pair⟨r,r,r⟩(p2,null);
    p1.setSnd⟨r,r,r,r⟩(p3);
    p2}
}}
```

Figure 4.5: Region Inference Result for Pair.example Method

All of the dependencies of `Pair.example` were analysed, thus we can continue with the region inference of this method. As shown in Figure 4.3, our inference rules initially annotate each method parameter, local variable, and constructor with new distinct regions and proceed to gather the constraints from each sub-expression. A set of equality and outlive constraints are collected and simplified; these constraints can be applied to reduce the number of distinct regions. The only regions that can be used outside of the method body are the global regions (used to annotate the method parameters, object receiver and the result) `r1,r2,r3,r5,r6,r7`. Based on region lifetime constraints, our rule computes the regions that outlive the global regions. These regions escape the method and are made equivalent to suitable global regions. The rest of the regions can be localized and coalesced into a single region. The steps are highlighted in Figure 4.4, with the final result of region inference shown in Figure 4.5.

4.1.2 Inference Rules Summary

Our rules assume that the source program P is globally available. Some of our rules also assume that parts of the target (region-annotated) program P' are also available. This is possible as we perform region inference in stages, in accordance with the calling hierarchy with the help of the dependency graph. For simplicity, first we assume that there is at most one class and one method at each stage. The main judgments employed by our region inference are the following:

- $\vdash def \Rightarrow def', Q$ denoting the region inference for a class declaration (only fields) *def*; the

$$\frac{}{\mathit{fieldlist}(\textbf{Object}\langle r\rangle)=_{def}[\,]} \qquad \frac{\begin{array}{c}\textbf{class}\ cn_1\langle r_{1..n}\rangle\ \textbf{extends}\ cn_2\langle r_{1..m}\rangle..\{(t_i\ f_i)_{i:1..p}..\}\in P'\\ \ell=\mathit{fieldlist}(\rho\ cn_2\langle r_{1..m}\rangle)\quad \rho=[r_i\mapsto x_i]_{i=1}^{n}\end{array}}{\mathit{fieldlist}(cn_1\langle x_{1..n}\rangle)=_{def}\ell+\!\!+[(\rho\ t_i)\ f_i]_{i=1}^{p}}$$

$$\frac{}{\mathit{methlist}(\textbf{Object}\langle r\rangle)=_{def}[\,]} \qquad \frac{\begin{array}{c}\textbf{class}\ cn_1\langle r_{1..n}\rangle\ \textbf{extends}\ cn_2\langle r_{1..m}\rangle..\{..\mathit{meth}_{j:1..q}\}\in P'\\ \ell=\mathit{methlist}(\rho\ cn_2\langle r_{1..m}\rangle)\quad \rho=[r_i\mapsto x_i]_{i=1}^{n}\end{array}}{\mathit{methlist}(cn_1\langle x_{1..n}\rangle)=_{def}\ell+\!\!+[(\rho\ \mathit{meth}_j)]_{j=1}^{q}}$$

$$\frac{}{\mathit{recursivefieldlist}([\,],cn)=_{def}[\,]} \qquad \frac{\tau=cn\quad l=[(\tau\ f)]+\!\!+\mathit{recursivefieldlist}(\mathit{fdl},cn)}{\mathit{recursivefieldlist}([(\tau\ f)]+\!\!+\mathit{fdl},cn)=_{def}l}$$

$$\frac{}{\mathit{nonrecursivefieldlist}([\,],cn)=_{def}[\,]} \qquad \frac{\tau\neq cn\quad l=[(\tau\ f)]+\!\!+\mathit{nonrecursivefieldlist}(\mathit{fdl},cn)}{\mathit{nonrecursivefieldlist}([(\tau\ f)]+\!\!+\mathit{fdl},cn)=_{def}l}$$

$$st(\varphi,s_1,s_2)=_{def}\{x\mapsto s_x|x\in s_1\wedge s_x\in\{y\mid y\in s_2\wedge\varphi\Rightarrow(x=y)\}\}$$

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

$$\mathit{reg}(\{\})=_{def}\{\}\quad \mathit{reg}(\{v{:}\tau\langle r^*\rangle\}\cup\Gamma)=_{def}\{r^*\}\cup\mathit{reg}(\Gamma)\quad \mathit{reg}(\tau\langle r^*\rangle)=_{def}\{r^*\}$$

$$\mathit{reg}((\tau\langle r^*\rangle\ f))=_{def}\{r^*\}\quad \mathit{reg}(\mathit{true})=_{def}\{\}\quad \mathit{reg}(r_1=r_2)=_{def}\{r_1,r_2\}$$

$$\mathit{reg}(r_1\succeq r_2)=_{def}\{r_1,r_2\}\quad \mathit{reg}(q\langle r_1..r_n\rangle\}=_{def}\{r_1..r_n\}\quad \mathit{reg}(\varphi_1\wedge\varphi_2)=_{def}\mathit{reg}(\varphi_1)\cup\mathit{reg}(\varphi_2)$$

$\rho\ t,\ \rho\ \varphi,\ \rho\ e,$	region substitution on a type, a constraint, an expression,
$\rho\ \mathit{meth}$	and a method
$\mathit{fresh}()$	returns one or more new/unused region names
r_j^*	denotes a possible empty sequence $r_{j_1}..r_{j_n}$ where $n\geq 0$
r_j^+	denotes a non-empty sequence $r_{j_1}..r_{j_n}$ where $n>0$

Figure 4.6: Auxiliary Rules for Region Inference

inference result consists of the region-annotated class *def′* and the class invariant (as a constraint abstraction) *Q*.

- $\vdash\tau\Rrightarrow t,\varphi$ denoting that t and φ are the region type and the class invariant corresponding to a type τ with respect to the region-annotated program *P′*. Note that t and φ are generated using fresh region names.

- $\vdash t\Rrightarrow\varphi$ denoting that φ is the class invariant corresponding to the region type t with respect to the region-annotated program *P′*.

- $\vdash t{<:}t'\Rrightarrow\varphi$ denoting that φ is the region constraint corresponding to the subtype relation defined in Section 3.5. However φ also contains the class invariants of the region types t and t'.

- $\Gamma\vdash \mathit{meth}\Rrightarrow\mathit{meth}',Q$ denoting the region inference for a method *meth* with respect to the region type environment Γ; the inference result consists of the region-annotated method *meth′* and the method precondition (as a constraint abstraction) *Q*.

- $\Gamma \vdash e \Rrightarrow e' : t, \varphi$ denoting the region inference for an expression e with respect to the region type environment Γ; the inference result consists of the region-annotated expression e', its region type *t*, and the derived region constraint φ.

- $\Gamma \vdash eb \Rrightarrow eb' : t, \varphi$ denoting the region inference for a block expression eb with respect to the region type environment Γ; the inference result consists of the region-annotated block expression eb', its region type *t*, and the derived region constraint φ. This inference rule may introduce a **letreg** in eb' to localise the regions that do not escape the block. However the region localisation could be done for any kind of expression at any level, not only for the expression blocks. For simplicity, we formalize the region localisation only for the expression blocks.

- $\vdash def_{1..n} \Rrightarrow def'_{1..n}, Q_{1..n}$ denoting the region inference for the mutually-recursive class declarations (only fields) $def_{1..n}$; the inference result consists of the region-annotated classes $def'_{1..n}$ and the set of the constraint abstractions $Q_{1..n}$.

- $\Gamma \vdash meth_{1..n} \Rrightarrow meth'_{1..n}, Q_{1..n}$ denoting the region inference for the mutually-recursive methods $meth_{1..n}$ with respect to the region type environment Γ; the inference result consists of the region-annotated methods *meth'* and the set of the constraint abstractions $Q_{1..n}$.

Figure 4.6 presents some auxiliary functions used by the main inference rules. The first two functions $fieldlist(cn\langle r_{1..n}\rangle)$ and $methlist(cn\langle r_{1..n}\rangle)$ compute all (defined and inherited) fields and methods of the class *cn* with respect to the current region annotation of the class, $cn\langle r_{1..n}\rangle$.

The function $recursivefieldlist(fdl, cn)$ returns the sublist of all the fields with type *cn* from an input list of fields *fdl*. The function $nonrecursivefieldlist(fdl, cn)$ returns the sublist of all the fields whose types are different than *cn* from an input list of fields *fdl*. These two functions are used to separate out the recursive fields from the non-recursive fields for a given class *cn*.

Given two sets of regions s_1 and s_2 and a region constraint φ, $st(\varphi, s_1, s_2)$ computes (when it is possible) a region substitution for each region of s_1 to a region of s_2 based on the equality constraints from φ. Specifically, for each region x of s_1, a subset of s_2 is generated, such that the subset contains all the regions of s_2 which have the same lifetime as x with respect to the region constraint φ. If the generated subset is not empty, an element s_x is randomly selected from it and a corresponding substitution for x is generated.

The function $ors(\varphi, s_1, s_2)$ computes the regions of the set s_1 which outlive at least one region of the set s_2 with respect to the region constraint φ. The function $\overline{ors}(\varphi, s_1, s_2)$ computes the regions of the set s_1 which do not outlive any region of the set s_2 with respect to the region constraint φ.

The function *reg* is overloaded, it computes the regions of a region type environment Γ, the regions of a region type t, the regions of a field $(t\ f)$, and the regions of a constraint φ. Note that the region substitutions on a type, a constraint, an expression, a class, and a method are defined as expected.

4.2 Inference for a Class

Figure 4.7 presents the inference rules for a class declaration. The first three rules [RI−PRIM], [RI−OBJ], and [RI−OBJ] are instances of the judgment:

$$\vdash \tau \Rrightarrow t, \varphi$$

that takes a type τ as input and generates its region type, t and its region invariant φ with respect to the region-annotated program P'. Primitive types are not annotated with regions, while the reference types are annotated with one or more regions. The output contains fresh region names, generated by the function *fresh*().

The rules [RI−INV−1], [RI−INV−2], and [RI−INV−3] generate the region class invariant of a given region type with respect to the region-annotated program P'. The rule [RI−SUBTYPE] generates the region constraint corresponding to a region subtyping relation and the region class invariants of the subtyping relation components.

The inference rules for a class declaration are based on a judgment of the following form:

$$\vdash def \Rrightarrow def', Q$$

where *def* is the source code of the class declaration, *def'* is the region-annotated class declaration, while Q is a constraint abstraction capturing the region class invariant. The constraint abstraction is useful for mutually recursive class declarations (see Section 4.7). The goal of the region inference for a class is to compute the class region annotation and the class invariant.

The inference rule [RI−CLASS−1] is designed for a class that does not contain recursive fields. First, the rule calls the functions $recursivefieldlist(fd_i, cn_2)$ and $nonrecursivefieldlist(fd_i, cn_2)$ (defined in Figure 4.6) to separate out the non-recursive fields from the recursive fields. In this case the list of the recursive fields is empty. Then the region types and the class invariants of the

[RI−PRIM]

$$\frac{}{\vdash prim \Rrightarrow prim\langle\rangle, true}$$

[RI−OBJ]

$$\frac{r{=}fresh()}{\vdash \text{Object} \Rrightarrow \text{Object}\langle r\rangle, true}$$

[RI−CT]

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle ...\ \textbf{where}\ \varphi\ \{...\}{\in}P' \\ a_{1..n}{=}fresh()\end{array}}{\vdash cn \Rrightarrow cn\langle a_{1..n}\rangle, ([r_i{\mapsto}a_i]_{i:1..n}\ \varphi)}$$

[RI−INV−1]

$$\frac{}{\vdash prim\langle\rangle \Rrightarrow true}$$

[RI−INV−2]

$$\frac{}{\vdash \text{Object}\langle r\rangle \Rrightarrow true}$$

[RI−INV−3]

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle ...\ \textbf{where}\ \varphi\ \{...\}{\in}P'}{\vdash cn\langle a_{1..n}\rangle \Rrightarrow ([r_i{\mapsto}a_i]_{i:1..n}\ \varphi)}$$

[RI−SUBTYPE]

$$\frac{\vdash t <: t', \varphi_0 \quad \vdash t \Rrightarrow \varphi \quad \vdash t' \Rrightarrow \varphi'}{\vdash t{<:}t' \Rrightarrow \varphi_0 \wedge \varphi \wedge \varphi'}$$

[RI−CLASS−1]

$$\frac{\begin{array}{c} recursivefieldlist(fd_{i:1..n}, cn_2){=}[\,] \quad nonrecursivefieldlist(fd_{i:1..n}, cn_2){=}[(\tau_i\, f_i)_{i:1..n}] \\ \vdash cn_1 \Rrightarrow cn_1\langle a_{1..l}\rangle, \varphi_0 \quad \vdash \tau_i \Rrightarrow \tau_i\langle r_i^*\rangle, \varphi_i \ \ i{=}1..n \\ \varphi{=}\varphi_0 \wedge \bigwedge_{i=1}^{n}(\varphi_i \wedge r_i^* {\succeq} a_1) \ \ Q{=}\{cn_2\langle a_{1..l}, r_{1..p}^*\rangle{=}\varphi\} \end{array}}{\begin{array}{l} \vdash\ \textbf{class}\ cn_2\ \textbf{extends}\ cn_1\ \{fd_{i:1..n}\ ...\} \ \Rrightarrow \\ \quad \textbf{class}\ cn_2\langle a_{1..l}, r_{1..p}^*\rangle\ \textbf{extends}\ cn_1\langle a_{1..l}\rangle\ \textbf{where}\ \varphi\ \{(\tau_i\langle r_i^*\rangle\, f_i)_{i:1..n},\ ...\}, Q \end{array}}$$

[RI−CLASS−2]

$$\frac{\begin{array}{c} recursivefieldlist(fd_{i:1..n}, cn_2){=}[(cn_2\, f_i)_{i:p+1..n}] \quad nonrecursivefieldlist(fd_{i:1..n}, cn_2){=}[(\tau_i\, f_i)_{i:1..p}] \\ r{=}fresh() \quad \vdash cn_1 \Rrightarrow cn_1\langle a_{1..l}\rangle, \varphi_0 \quad \vdash \tau_i \Rrightarrow \tau_i\langle r_i^*\rangle, \varphi_i \ \ i{=}1..p \\ \varphi{=}\varphi_0 \wedge \bigwedge_{i=1}^{p}(\varphi_i \wedge r_i^* {\succeq} a_1 \wedge r_i^* {\succeq} r) \wedge \bigwedge_{i=2}^{l}(a_i {\succeq} r) \wedge r {\succeq} a_1 \ \ Q{=}\{cn_2\langle a_{1..l}, r_{1..p}^*, r\rangle{=}\varphi\} \end{array}}{\begin{array}{l} \vdash\ \textbf{class}\ cn_2\ \textbf{extends}\ cn_1\ \{fd_{i:1..n}\ ...\} \ \Rrightarrow \\ \quad \textbf{class}\ cn_2\langle a_{1..l}, r_{1..p}^*, r\rangle\ \textbf{extends}\ cn_1\langle a_{1..l}\rangle\ \textbf{where}\ \varphi \\ \qquad \{(\tau_i\langle r_i^*\rangle\, f_i)_{i:1..p},\ (cn_2\langle r, a_{2..l}, r_{1..p}^*, r\rangle\, f_i)_{i:p+1..n},\ ...\}, Q \end{array}}$$

Figure 4.7: Region Inference Rules for a Class

parent class and of the fields are generated (using fresh regions). At this stage of the inference, the dependency graph guarantees that the inference for the parent class and for the classes of the fields was already done and their region types are available. The region type of the current class is formed from the regions of the parent class followed by the regions of the fields. The invariant of the current class is computed as a conjunction of the parent class invariant, fields invariants, and the no-dangling requirement for the current class region type. Note that the no-dangling requirement captures the property that all regions of a region type should outlive the first region of that region type.

The inference rule [RI−CLASS−2] deals with a class with recursive fields (the result of the function $recursivefieldlist(fd_i, cn_2)$ is nonempty). The inference process of the non-recursive fields is similar with that described by the previous rule. In contrast the recursive fields have the same region annotation as the class, except for the first region that is a fresh region, r. However, the same region r is shared by all class recursive fields. The class invariant is similar with that computed by the rule [RI−CLASS−1], except the additional invariants of the recursive fields and the non-dangling requirement corresponding to the additional region r.

4.3 Inference for Expressions

The inference rules for expressions are based on the following judgment:

$$\Gamma \vdash e \Rrightarrow e' : t,\ \varphi$$

where e is the unannotated expression, e' is the region-annotated expression, $\mathtt{t}$ is the inferred region type, and φ is the derived region constraint. The inference is done with respect to the region type environment Γ. The syntax-directed inference rules for expressions are detailed in Figure 4.8.

The first two rules [RI−CONS1] and [RI−CONS2] infer the region types for constants, either primitives or null values. Note that null values are never used as a destination to store other objects.

Based on the type environment, the rules [RI−VAR] and [RI−FD] retrieve the region types for a variable and for an object field. In rule [RI−FD], the relation $\vdash (\mathtt{t}\ \ \mathtt{f}) \in \mathit{fieldlist}\,(\mathtt{cn}\langle x^*\rangle)$ (see Figure 4.6) computes the current region type of a field according to the region type of the field object.

Rule [RI−ASSGN] is the region inference rule for the assignment. It uses the region subtyping rule (from Section 3.5) to express that the inferred region type of the right hand side expression e is a subtype of the left hand side location `lhs`. The gathered region constraint consists of two parts: one from the inference for the expression e and the second from the region subtyping itself. Note that the region type of the left hand side location (variable or field) is directly retrieved from the environment and the corresponding region constraint is `true`.

Our region inference is flow-insensitive, thus the rule [RI−SEQ] does not take into account the order of the sequence when it combines the inference results of its components.

Rule [RI−NEW] is the region inference rule for the **new** operator. First it generates a fresh region type for the given class `cn`. Then it calls the function *fieldlist* (see Figure 4.6) to get the list of all fields and their region types according to the current region type associated to `cn`. The arguments of **new** are the initial values for the class fields. The rule assumes that there is a one-to-one correspondence between the arguments of **new** and the fields of class `cn`. We apply the region subtyping rules for region types of arguments and fields. The class invariant and the region subtyping constraints then form the gathered region constraint.

Rule [RI−IF] is the region inference rule for the conditional expression. First it infers the two region types for two branches. Second it computes the most specific region supertype t of

$$[\underline{\mathbf{RI{-}CONS1}}] \qquad \frac{}{\Gamma \vdash \textbf{null} \Rrightarrow \textbf{null} : \bot, \mathit{true}}$$

$$[\underline{\mathbf{RI{-}CONS2}}] \qquad \frac{}{\Gamma \ \vdash k \Rrightarrow k : \mathit{prim}\langle\rangle, \mathit{true}}$$

$$[\underline{\mathbf{RI{-}VAR}}] \qquad \frac{v : \tau\langle r^*\rangle \ \in \ \Gamma}{\Gamma \ \vdash v \Rrightarrow v : \tau\langle r^*\rangle, \ \mathit{true}}$$

$$[\underline{\mathbf{RI{-}FD}}] \qquad \frac{(v : \mathit{cn}\langle x^+\rangle) \in \Gamma \quad (t\,f) \in \mathit{fieldlist}(\mathit{cn}\langle x^+\rangle)}{\Gamma \ \vdash v.f \Rrightarrow v.f : t, \ \mathit{true}}$$

$$[\underline{\mathbf{RI{-}ASSGN}}] \qquad \frac{\begin{array}{c}\Gamma \ \vdash \mathit{lhs} \Rrightarrow \mathit{lhs} : t, \mathit{true} \\ \Gamma \ \vdash e \Rrightarrow e' : t', \varphi' \ \ \vdash t' {<:} t \Rrightarrow \varphi\end{array}}{\Gamma \vdash \mathit{lhs} = e \Rrightarrow \mathit{lhs} = e' : \textbf{void}, \varphi \wedge \varphi'}$$

$$[\underline{\mathbf{RI{-}SEQ}}] \qquad \frac{\begin{array}{c}\Gamma \ \vdash e_1 \Rrightarrow e_1' : t_1, \varphi_1 \\ \Gamma \ \vdash e_2 \Rrightarrow e_2' : t_2, \varphi_2\end{array}}{\Gamma \vdash e_1 \,\textbf{;}\, e_2 \Rrightarrow e_1' \,\textbf{;}\, e_2' : t_2, \varphi_1 \wedge \varphi_2}$$

$$[\underline{\mathbf{RI{-}NEW}}] \qquad \frac{\begin{array}{c}\vdash \mathit{cn} \Rrightarrow \mathit{cn}\langle x^+\rangle, \varphi_0 \quad \mathit{fieldlist}(\mathit{cn}\langle x^+\rangle) = [(t_i\, f_i)]_{i=1}^{p} \\ (v_i : t_i') \in \Gamma \ \ \vdash t_i' {<:} t_i \Rrightarrow \varphi_i \quad i = 1..p\end{array}}{\Gamma \vdash \textbf{new}\, \mathit{cn}(v_{1..p}) \Rrightarrow \textbf{new}\, \mathit{cn}\langle x^+\rangle(v_{1..p}) : \mathit{cn}\langle x^+\rangle, \varphi_0 \wedge \bigwedge_{i=1}^{p} \varphi_i}$$

$$[\underline{\mathbf{RI{-}IF}}] \qquad \frac{\begin{array}{c}(v_0 : \textbf{boolean}\langle\rangle) \in \Gamma \quad \Gamma \vdash e_1 \Rrightarrow e_1' : t_1, \varphi_1' \quad \Gamma \vdash e_2 \Rrightarrow e_2' : t_2, \varphi_2' \\ \vdash t_1 {<:} t \Rrightarrow \varphi_1 \ \ \vdash t_2 {<:} t \Rrightarrow \varphi_2 \ \ (\neg \exists \hat{t} \cdot (\vdash t_1 {<:} \hat{t} \Rrightarrow \hat{\varphi}_1) \wedge (\vdash t_2 {<:} \hat{t} \Rrightarrow \hat{\varphi}_2) \wedge (\vdash \hat{t} {<:} t \Rrightarrow \hat{\varphi}))\end{array}}{\Gamma \vdash \textbf{if}\, v_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \Rrightarrow \textbf{if}\, v_0\, \textbf{then}\, e_1'\, \textbf{else}\, e_2' : t, \varphi_1' \wedge \varphi_2' \wedge \varphi_1 \wedge \varphi_2}$$

$$[\underline{\mathbf{RI{-}INVOKE}}] \qquad \frac{\begin{array}{c}v_1' : \mathit{cn}\langle x_1'^+\rangle \in \Gamma \ \ (\tau_0\langle x_0^*\rangle \ mn\langle y^+\rangle((\tau_j\langle x_j^*\rangle \ v_j)_{j:2..p})\ \textbf{where}\ \hat{\varphi}\ \{e\}) \in \mathit{methlist}(\mathit{cn}\langle x_1'^+\rangle) \\ \vdash \mathit{cn}\langle x_1'^+\rangle \Rrightarrow \varphi_1 \quad (v_j' : \tau_j'\langle x_j'^*\rangle) \in \Gamma \quad \vdash \tau_j'\langle x_j'^*\rangle {<:} \tau_j\langle x_j^*\rangle \Rrightarrow \varphi_j \quad j = 2..p\end{array}}{\Gamma \ \vdash v_1'.mn(v_2', ..., v_p') \Rrightarrow v_1'.mn\langle y^+\rangle(v_2', ..., v_p') : \tau_0\langle x_0^*\rangle, \ \hat{\varphi} \wedge \bigwedge_{j=1}^{p} \varphi_j}$$

$$[\underline{\mathbf{RI{-}EB1}}] \qquad \frac{\begin{array}{c}\vdash \tau_1 \Rrightarrow \tau_1\langle x_1^*\rangle, \varphi_1 \qquad \Gamma, \{v_1 : \tau_1\langle x_1^*\rangle\} \vdash e \Rrightarrow e' : \tau\langle r^*\rangle, \varphi \\ \rho = \mathit{st}(\varphi \wedge \varphi_1, \{x_1^*\}, \mathit{reg}(\Gamma)) \\ \mathit{rs} = \overline{\mathit{ors}}(\varphi \wedge \varphi_1, \mathit{reg}(\varphi) \cup \{x_1^*\}, \{r^*\} \cup \mathit{reg}(\Gamma)) \\ \mathit{rs} = \emptyset\end{array}}{\Gamma \vdash \{(\tau_1\ v_1)\ e\} \Rrightarrow \rho\, \{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\} : \tau\langle \rho\, r^*\rangle, \rho(\varphi \wedge \varphi_1)}$$

$$[\underline{\mathbf{RI{-}EB2}}] \qquad \frac{\begin{array}{c}\vdash \tau_1 \Rrightarrow \tau_1\langle x_1^*\rangle, \varphi_1 \quad \Gamma, \{v_1 : \tau_1\langle x_1^*\rangle\} \vdash e \Rrightarrow e' : \tau\langle r^*\rangle, \varphi \\ \rho = \mathit{st}(\varphi \wedge \varphi_1, \{x_1^*\}, \mathit{reg}(\Gamma)) \\ \mathit{rs} = \overline{\mathit{ors}}(\varphi \wedge \varphi_1, \mathit{reg}(\varphi) \cup \{x_1^*\}, \{r^*\} \cup \mathit{reg}(\Gamma)) \\ \mathit{rs} \neq \emptyset \quad a = \mathit{fresh}() \quad \rho' = \{x \mapsto a \mid x \in \mathit{rs}\}\end{array}}{\Gamma \vdash \{(\tau_1\ v_1)\ e\} \Rrightarrow \textbf{letreg}\ a\ \textbf{in}\ \rho'\rho\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\} : \tau\langle \rho r^*\rangle, \rho((\varphi \wedge \varphi_1) \backslash \mathit{rs})}$$

$$[\underline{\mathbf{RI{-}LOOP}}] \qquad \frac{(v_0 : \textbf{boolean}\langle\rangle) \in \Gamma \quad \Gamma \vdash e \Rrightarrow e' : \textbf{void}, \varphi}{\Gamma \vdash \textbf{while}\ v_0\ e \Rrightarrow \textbf{while}\ v_0\ e' : \textbf{void}, \varphi}$$

Figure 4.8: Region Inference Rules for Expressions

the previous two region types. The inferred region constraints are conjoined.

Rule [RI−INVOKE] is the region inference rule for the instance method invocation. The function $methlist$ (defined in Figure 4.6) retrieves the region-annotated method mn with respect to the current region type of the receiver v_1'. The function automatically does the substitution of the receiver regions $cn\langle x_1'^+\rangle$ for the region-annotated method mn (region types and precondition). The other region names used by the region-annotated method are fresh region names. These fresh regions are instantiated by the region type subtyping rule applied for the current arguments.

4.4 Localising Regions

For simplicity, we present a set of rules that may introduce localised regions at expression blocks. However effective placement of local variable declarations, object allocations and expression blocks can affect region placement and the extent to which memory is effectively reused. Therefore our region inference algorithm can support region localisation for any kind of expression or for a group of expressions.

The key inference rules that govern this process for expression blocks are presented in Figure 4.8, rule [RI−EB1] and rule [RI−EB2]. The first rule deals with the case when region localisation is not possible, while the second rule introduces a `letreg` statement for the local objects. The goal of these two rules is to localise the regions that do not escape the expression block. Those regions that may escape the block can be traced to the regions that exist in either the region type environment or the region type of the expression block body. All regions that outlive these regions also escape.

The first part of the inference process (first three lines) is similar for both rules. First the rules [RI−EB1] and [RI−EB2] compute the fresh region type and the region invariant for the local variable declaration. Then this region type is used to infer the region type and the region constraint of the expression block body. Based on the region constraint of the block body and the region invariant of the local variable, the function st (defined in Figure 4.6) identifies which of the newly introduced regions are equivalent to the regions in the type environment Γ. The result is a substitution ρ from the newly introduced regions to the existing regions of Γ. We also use the function reg (shown in Figure 4.6) to extract the region variables either from a given constraint or from the type environment. The set of all regions which must escape the expression block consists of the regions from the type environment and the regions that appear

(a) Cyclic data structure

```
{Pair p1=new Pair(null,null);
 Pair p2=new Pair(p1,null);
 Pair p3=new Pair(null,null);
 p1.setSnd(p2);
 p2.setSnd(p3);
 p3 }
```

(b) Source program

```
{Pair⟨r1,r1a,r1b⟩ p1 =
   new Pair⟨r1,r1a,r1b⟩(null,null);
   //r1a⪰r1 ∧ r1b⪰r1
 Pair⟨r2,r2a,r2b⟩ p2 =
   new Pair⟨r2,r2a,r2b⟩(p1,null);
   // r2a⪰r2 ∧ r2b⪰r2 ∧ r1⪰r2a
 Pair⟨r3,r3a,r3b⟩ p3=
   new Pair⟨r3,r3a,r3b⟩(null,null);
   //r3a⪰r3 ∧ r3b⪰r3
 p1.setSnd⟨r1,r1a,r1b,r2⟩(p2); // r2⪰r1b
 p2.setSnd⟨r2,r2a,r2b,r3⟩(p3); // r3⪰r2b
 p3 }
```

(c) Initial region-annotated program

```
{ letreg r in {
 Pair⟨r,r,r⟩ p1 =
   new Pair⟨r,r,r⟩(null,null);
 Pair⟨r,r,r⟩ p2 =
   new Pair⟨r,r,r⟩(p1,null);
 Pair⟨r3,r3a,r3b⟩ p3=
   new Pair⟨r3,r3a,r3b⟩(null,null);
   //r3a⪰r3 ∧ r3b⪰r3
 p1.setSnd⟨r,r,r,r⟩(p2);
 p2.setSnd⟨r,r,r,r3⟩(p3);
 p3 }}
```

(d) Final region-annotated program

Figure 4.9: Example with Circular Structure

in the region type of the block body. The set of the candidate regions for localisation consists of the newly introduced regions and the regions which appear in the region constraint of the block body but do not appear in the previous set (of the regions that must escape). Based on the region constraint of the block body and the region invariants of the local variables, the function $\overline{\texttt{ors}}$ (defined in Figure 4.6) computes the set `rs` of the regions that can be safely localised. The set `rs` consists of those regions which do not outlive any region that must escape the expression block.

From this point the inference rules are different. The rule [RI−EB1] does not localise any region because the set `rs` is empty. However the substitution ρ (corresponding to equivalent regions) is applied to the result (expression block, region type and region constraint). The rule [RI−EB2] localises the regions from `rs` using a fresh region a. Thus an additional substitution ρ' is applied to the result. The region constraints that use regions from `rs` are eliminated from the region constraint of the result. This operation is denoted by $\varphi \setminus r$, where φ is a region constraint and `r` is a set of regions.

The rules [RI−EB1] and [RI−EB2] can be directly used to localize the regions inside a loop. A loop can be treated as an expression block that does not return any value (rule [RI−LOOP] of Figure 4.8). There exist programs which consume an infinite amount of space as a result of executing a loop, but a constant amount if a localization can be done for each loop iteration by introducing a `letreg` inside the loop body.

We next illustrate the region localisation with a simple code fragment in Figure 4.9(b), which constructs a cyclic structure involving two `Pair` nodes, `p1` and `p2`, as shown in Figure 4.9(a). The initial inferred program with region annotations is shown in Figure 4.9(c). After constraint simplification to coalesce equal regions together and to localise non-escaping regions, we obtain the target program in Figure 4.9(d). Because of the outlives constraint from the no-dangling requirement, every cyclic structure must be placed in the same region. Notice that `p1` and `p2` are initially placed in regions `r1` and `r2`, respectively. However, the region constraint gathered, namely $\texttt{r2}\succeq\texttt{r1b}\wedge\texttt{r1b}\succeq\texttt{r1}\wedge\texttt{r1}\succeq\texttt{r2a}\wedge\texttt{r2a}\succeq\texttt{r2}$, implies that `r1=r2=r1b=r2a`. Applying this extra constraint causes the two objects to be located in the same region. The resulting type of this block is $\texttt{Pair}\langle\texttt{r3,r3a,r3b}\rangle$ with the region constraint $\texttt{r3b}\succeq\texttt{r3}\wedge\texttt{r3a}\succeq\texttt{r3}\wedge\texttt{r3}\succeq\texttt{r2b}$. As the regions used to store the `p1` and `p2` objects do not escape the block, the [RI−EB2] rule introduces a single local region `r` to replace them. The regions `r3,r3a,r3b` are not replaced by local region `r` because they escape the expression block.

4.5 Inference for a Method

Figure 4.10 presents the inference rules [RI−METH−1] and [RI−METH−2] for a method declaration using the following judgment:

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

where Γ is the region type environment, $meth$ is the source code of the method declaration and $meth'$ is the region-annotated method declaration, while Q is the constraint abstraction corresponding to the method precondition. The constraint abstraction is useful for recursive (or mutually recursive) method declarations (see Section 4.7). The goal of the region inference for a method is to compute the method region annotation and the method precondition. The method region annotation consists of the region types of the method parameters (including the receiver) and the region type of the method result.

First the rule computes the class invariant of the receiver. The region type of the receiver `this` is taken from the type environment. Then, it computes the fresh region types and the region invariants for the method parameters and the method result. Region types of the method parameters and the receiver are used to infer the region type and the region constraint for the method body. The inferred region type must be a subtype of the expected method result type. The inference for the method body does also a region localisation at the method body level. All derived region constraints are put together to form the method region precondition. Since only the global regions $r^+, r_1^*, .., r_p^*, r_0^*$ corresponding to the region types of the parameters (including the receiver) and the result are visible outside the method, the local regions (denoted by the set rs) outliving the global regions are made equal to global regions (rule [RI−METH−2]). The algorithm is described by the definition of $nesc$. Thus $nesc$ takes as arguments a set of local regions R_l, a set of global regions R_g, a region constraint φ and returns a substitution and a constraint. The substitution maps the regions of R_l to the regions of R_g with respect to the constraint φ. When the function $nesc$ is called, it is assumed that each region of R_l outlives (or has the same lifetime as) at least one region from R_g. The function $nesc$ groups the regions of R_l into two categories denoted by R_{eq} and R_{ll}. The first category, R_{eq} contains the regions of R_l which are equal to one or more regions of R_g. The substitutions are directly generated from the equality constraints. The second category, R_{ll} contains the regions of R_l which outlive at least one region of R_g. Each region a of R_{ll} is made equal to the oldest region of R_g which is younger

$$[\underline{\textbf{RI-METH-1}}]$$

$$\frac{\begin{array}{c}
\textbf{class}\ cn\langle x^+\rangle ...\ \textbf{where}\ \hat{\varphi}\ \{...\}\in P' \quad \varphi''=[x^+\mapsto r^+]\hat{\varphi} \\
\vdash\tau_i\Rrightarrow\tau_i\langle r_i^*\rangle, \varphi_i \ \ i=0..p \\
\{\mathit{this} : cn\langle r^+\rangle,\ (v_i : \tau_i\langle r_i^*\rangle)_{i:1..p}\}\vdash e\Rrightarrow e' : \tau_0'\langle x_0^*\rangle, \varphi' \\
\vdash\tau_0'\langle x_0^*\rangle <: \tau_0\langle r_0^*\rangle\Rrightarrow\varphi_0' \quad \varphi=\varphi'\wedge\varphi_0'\wedge\varphi''\wedge\bigwedge_{i=0}^{p}\varphi_i \\
rs=\{a \mid a\in(\mathit{reg}(\varphi)\cup regs(e'))\wedge a\notin\{r^+, r_1^*, .., r_p^*, r_0^*\}\wedge \\
\wedge(\exists r'\in\{r^+, r_1^*, .., r_p^*, r_0^*\}\cdot(\varphi\Rightarrow a\succeq r' \vee \varphi\Rightarrow a=r'))\} \\
rs=\emptyset \qquad Q=\{cn.mn\langle r^+, r_1^*, .., r_p^*, r_0^*\rangle=\varphi\}
\end{array}}{\begin{array}{l}
\{\mathit{this} : cn\langle r^+\rangle\}\vdash \tau_0\ mn((\tau_i\ v_i)_{i:1..p})\ \{e\} \quad \Rrightarrow \\
\quad \tau_0\langle r_0^*\rangle\ mn\langle r^+, r_1^*, .., r_p^*, r_0^*\rangle((\tau_j\langle r_i^*\rangle\ v_i)_{i:1..p})\ \textbf{where}\ \varphi\ \{e\}, Q
\end{array}}$$

$$[\underline{\textbf{RI-METH-2}}]$$

$$\frac{\begin{array}{c}
\textbf{class}\ cn\langle x^+\rangle ...\ \textbf{where}\ \hat{\varphi}\ \{...\}\in P' \quad \varphi''=[x^+\mapsto r^+]\hat{\varphi} \\
\vdash\tau_i\Rrightarrow\tau_i\langle r_i^*\rangle, \varphi_i \ \ i=0..p \\
\{\mathit{this} : cn\langle r^+\rangle,\ (v_i : \tau_i\langle r_i^*\rangle)_{i:1..p}\}\vdash e\Rrightarrow e' : \tau_0'\langle x_0^*\rangle, \varphi' \\
\vdash\tau_0'\langle x_0^*\rangle <: \tau_0\langle r_0^*\rangle\Rrightarrow\varphi_0' \quad \varphi=\varphi'\wedge\varphi_0'\wedge\varphi''\wedge\bigwedge_{i=0}^{p}\varphi_i \\
rs=\{a \mid a\in(\mathit{reg}(\varphi)\cup regs(e'))\wedge a\notin\{r^+, r_1^*, .., r_p^*, r_0^*\}\wedge \\
\wedge(\exists r'\in\{r^+, r_1^*, .., r_p^*, r_0^*\}\cdot(\varphi\Rightarrow a\succeq r' \vee \varphi\Rightarrow a=r'))\} \\
rs\neq\emptyset \qquad \mathit{nesc}(rs, \{r^+, r_1^*, .., r_p^*, r_0^*\}, \varphi)=(\rho_e, \varphi_e) \\
Q=\{cn.mn\langle r^+, r_1^*, .., r_p^*, r_0^*\rangle=\rho_e(\varphi\wedge\varphi_e)\}
\end{array}}{\begin{array}{l}
\{\mathit{this} : cn\langle r^+\rangle\}\vdash \tau_0\ mn((\tau_i\ v_i)_{i:1..p})\ \{e\} \quad \Rrightarrow \\
\tau_0\langle r_0^*\rangle\ mn\langle r^+, r_1^*, .., r_p^*, r_0^*\rangle((\tau_j\langle r_i^*\rangle\ v_i)_{i:1..p})\ \textbf{where}\ \rho_e(\varphi\wedge\varphi_e)\ \{\rho_e e\}, Q
\end{array}}$$

$$\frac{\begin{array}{c}
R_{eq}=\{a \mid a\in R_l \wedge (\exists r\in R_g\cdot(\varphi\Rightarrow r=a))\} \quad R_{ll}=R_l-R_{eq} \\
\mathit{mkequal}(\{(a, \mathit{same}(a, R_g, \varphi)) \mid a\in R_{eq}\})=\rho_{eq} \\
\mathit{combine}(\{(a, \mathit{lower}(a, R_g, \varphi)) \mid a\in R_{ll}\})=(\rho_e, \varphi_e)
\end{array}}{\mathit{nesc}(R_l, R_g, \varphi)=_{def}(\rho_e\circ\rho_{eq}, \varphi_e)}$$

$$\mathit{lower}(a, R_g, \varphi)=_{def}\{r \mid r\in R_g \wedge (\varphi\Rightarrow a\succeq r)\wedge(\nexists r'\cdot\varphi\Rightarrow a\succeq r' \wedge \varphi\Rightarrow r'\succeq r)\}$$
$$\mathit{same}(a, R_g, \varphi)=_{def}\{r \mid r\in R_g \wedge (\varphi\Rightarrow r=a)\}$$

$$\frac{x\in s}{\mathit{sel}(s)=_{def}x} \qquad \frac{\rho_i=[a_i\mapsto \mathit{sel}(R_i)] \quad i=1..n}{\mathit{mkequal}(\{(a_1, R_1), .., (a_n, R_n)\})=_{def}\rho_1\circ..\circ\rho_n}$$

$$\frac{\rho_i=[a_i\mapsto \mathit{sel}(R_i)] \quad \varphi_i=\bigwedge_{r\in R_i}(a_i=r) \quad i=1..n}{\mathit{combine}(\{(a_1, R_1), .., (a_n, R_n)\})=_{def}(\rho_1\circ..\circ\rho_n, \varphi_1\wedge..\wedge\varphi_n)}$$

$\rho\varphi, \rho e$	region substitution on a constraint and an expression
$\mathit{reg}(\varphi)$	computes the region variables of a constraint (see Figure 4.6)
$\mathit{regs}(e)$	computes the region variables of an expression (see Definition 3.7.1.5)

Figure 4.10: Region Inference Rule for a Method

than a (see definition of *lower* in Figure 4.10). If the result of *lower* for a region a is a set (rather than a single region), all the regions of that set are made equal to a.

At the end a constraint abstraction Q is generated. With the help of the recursive constraint abstractions, our inference rules directly support region-polymorphic recursion. In the case of recursive methods, the inference rules [**RI−METH−1**] and [**RI−METH−2**] build a recursive constraint abstraction. Subsequently, a fixpoint analysis is applied to obtain a closed-form formula for that abstraction.

In Figure 4.11 we illustrate how the region annotations and the constraints are inferred for a recursive method. For simplicity, we consider a simple method without the receiver object. Figure 4.11(a) contains a method which merges two lists of objects. It is a functional merge in the sense of producing a new list as result. Because the method swaps its parameters at the recursive call, the resulting list contains alternating elements from both lists.

Our algorithm generates a set of fresh region types and collects the region constraints. We use the region inference results for the class `List` and its methods `getValue` and `getNext`, presented before in Figure 3.5. A final region annotated program is shown in Figure 4.11(b). The method precondition is the constraint abstraction `join⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩` (shown in Figure 4.11(c)) that consists of the following components:

- the class invariants, φ_i of the method parameters and the method result.
- the region constraints collected from the method body. Some of the collected constraints duplicate those from φ_i and therefore they are ignored. The only different constraint is the outlive constraint `r2`$\succeq$`r8` that corresponds to the last line of the method, where the field `value` (that is in the region `r8`) of the newly created `List` is initialized with `x` (that is in the region `r2`).
- the constraint abstractions corresponding to the recursive calls. In this case there is only one constraint abstraction because both recursive calls use the same region annotation `join⟨r4,r5, r6,r1,r2,r3,r7,r8,r9⟩`.

As the constraint abstraction is recursive, we apply a fixpoint analysis to obtain its closed-form formula. Starting with the initial version of $\mathtt{join}_0$, we progressively refine the definition of `join` until a fixpoint is reached, as highlighted in Figure 4.11(d). The initial version $\mathtt{join}_0$, that is `True` (no constraint on the method regions) denotes no more calls (the termination of the recursive calls). Based on $\mathtt{join}_0$, the first iteration $\mathtt{join}_1$ (corresponding to the last recursive

```
List join(List xs, List ys)
{if isNull(xs) then
   if isNull(ys) then null else join(ys,xs)
 else { Object x; List res;
  x=xs.getValue();xs=xs.getNext();
  res=join(ys,xs); new List(x,res) } }
```

(a) Source program

```
List⟨r7,r8,r9⟩ join⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩(List⟨r1,r2,r3⟩ xs,
     List⟨r4,r5,r6⟩ ys) where join⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩
 {if isNull⟨r1,r2,r3⟩(xs) then
   if isNull⟨r4,r5,r6⟩(ys) then null
   else join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩(ys,xs)
  else {
   Object⟨r8⟩ x; List⟨r7,r8,r9⟩ res;
   x=xs.getValue⟨r2⟩();
   xs=xs.getNext⟨r1,r2,r3⟩();
   res=join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩(ys,xs);
   new List<r7,r8,r9>(x,res) // generates the below constraint r2⪰r8
 } }
```

(b) Final region-annotated program

```
Q = {join⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩ = φi∧(r2⪰r8)∧join⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩}
   where φi=(r2⪰r1∧r3∧r1)∧(r5⪰r4∧r6⪰r4)∧(r8⪰r7∧r9⪰r7)
```

(c) Recursive Constraint Abstraction

```
join0⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩ = True
join1⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩ = r2⪰r8∧join0⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩
                                 = φi∧(r2⪰r8)
join2⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩ = r2⪰r8∧join1⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩
                                 = φi∧(r2⪰r8∧r5⪰r8)
join3⟨r1,r2,r3,r4,r5,r6,r7,r8,r9⟩ = r2⪰r8∧join2⟨r4,r5,r6,r1,r2,r3,r7,r8,r9⟩
                                 = φi∧(r2⪰r8∧r5⪰r8)
```

(d) Fixpoint analysis

Figure 4.11: Fixpoint Iteration for Recursive Method

call) is built. The iterative process ends when the result at the step i is a subset of the result of the step $i{+}1$, namely $\mathtt{join}_i \Rightarrow \mathtt{join}_{i+1}$. Fixpoint analysis always terminates for our constraint abstractions — the finite set of possible constraints is made up from a bounded set of regions.

For simplicity, in this example the local regions of the method body are made equivalent to suitable global regions (regions of the method parameters, method receiver and method result). This is not possible in all situations. Therefore the local regions may appear in the constraint abstraction, but after each fixpoint iteration they are eliminated from the iteration result. In general, only after the computation of the method precondition, the regions that outlive the global regions are made equivalent to suitable global regions, while the regions that do not outlive the global regions are localised by `letreg`.

The presented example relies on region-polymorphic recursion, whereby each recursive call may have a different region type (region parameters) from its caller. Without region-polymorphic recursion some lifetime precision may be lost or regions may be coalesced together.

4.6 Solving Method Overriding

As mentioned in Sec 3.4.3, class subtyping and method overriding must comply with their respective checks to ensure the soundness of subsumption. The class subtyping check is relatively easy to enforce. The existing inference rules[**RI−CLASS−1**] and [**RI−CLASS−2**] (shown in Figure 4.7) already accumulate the invariant from each class A to its subclass B in order to ensure:

$$\mathit{inv.B}\langle r_1..r_m..r_n\rangle \Rightarrow \mathit{inv.A}\langle r_1..r_m\rangle$$

In contrast, the method overriding check is more complex. Consider a class A, its subclass B, and a method $A.mn$ overridden by $B.mn$. For method overriding to be sound, we require the following property to be valid:

$$\mathit{inv.B}\langle r_1..r_m..r_n\rangle \wedge \mathit{pre.A.mn}\langle r_1..r_m, r'_1..r'_p\rangle \Rightarrow \mathit{pre.B.mn}\langle r_1..r_m, r_{m+1}..r_n, r'_1..r'_p\rangle$$

This property may not hold initially. To rectify this, the region inference can selectively augment the premise of each overriding check, with the following considerations:

1. We can strengthen either the premise $inv.B\langle r_1..r_m..r_n\rangle$ or the premise $pre.A.mn\langle r_1..r_m, r'_1..r'_p\rangle$ or both.
2. Strengthening $pre.A.mn\langle r_1..r_m, r'_1..r'_p\rangle$ can be problematic as some regions, namely $r_{m+1}..r_n$, are present in class B but not A.

$$\frac{I \wedge X \Rightarrow Y}{I,X,\ Y \vdash I,X}\ \textrm{[C1]} \qquad \frac{\varphi \in Y \quad \neg(I \wedge X \Rightarrow \varphi) \quad reg(\varphi) \subseteq R_X \qquad I, X \wedge \varphi,\ Y \vdash I',X'}{I,X,\ Y \vdash I',X'}\ \textrm{[C2]}$$

$$\frac{\varphi \in Y \quad \neg(I \wedge X \Rightarrow \varphi) \quad reg(\varphi) \subseteq R_B \qquad I \wedge \varphi, X,\ Y \vdash I',X'}{I,X,\ Y \vdash I',X'}\ \textrm{[C3]}$$

$$\frac{\varphi \in Y \quad \neg(I \wedge X \Rightarrow \varphi) \qquad \rho : (reg(\varphi) \cap (R_B - R_A)) \rightarrow R_A \qquad I \wedge mkconstr(\rho), X \wedge \rho\varphi,\ Y \vdash I',X'}{I,X,\ Y \vdash I',X'}\ \textrm{[C4]}$$

$$mkconstr([r_1 \mapsto r'_1, .., r_n \mapsto r'_n]) =_{def} \bigwedge_{i:1..n} (r_i = r'_i)$$

$\rho\ \varphi$	region substitution on a constraint
$reg(\varphi)$	computes the region variables of a constraint (see Figure 4.6)

Figure 4.12: Overriding Check Resolution

These two issues can be considered systematically by examining each basic constraint of $pre.B.mn\langle r_1..r_m, r_{m+1}..r_n, r'_1..r'_p\rangle$ to determine if (i) it is already valid, or (ii) it can be added to $pre.A.mn$, or (iii) it can be added to $inv.B$, or (iv) it can be split into an equality constraint for $inv.B$ and a modified constraint for $pre.A.mn$. We formalise this conflict resolution as the following inference rule:

$$I,X,\ Y \vdash I',X'$$

where I denotes the class invariant of the subclass, X denotes the precondition of the overridden method (from the superclass), while Y represents the precondition of the overriding method (from the subclass). The results I', X' are strengthened versions of I, X which satisfy the soundness of overriding. Each constraint is expressed as a conjunction of atomic constraints $\bigwedge\varphi$, where φ has the form $r_1 \succeq r_2$ or $r_1 = r_2$. Strengthening of a constraint is done by adding atomic conjuncts to that constraint.

The resolution rules are shown in Figure 4.12. Note that $R_B = \{r_1..r_m, r_{m+1}..r_n\}$, $R_X = \{r_1..r_m, r'_1..r'_p\}$ and $R_A = \{r_1..r_m\}$. Notation $\rho : R_1 \rightarrow R_2$ denotes a region substitution with $R_1(R_2)$ as its domain (co-domain). The function $mkconstr(\rho)$ transforms a substitution ρ into an equality constraint. Note that function $reg(\varphi)$ (defined earlier in Figure 4.6) returns all regions occurring in the constraint φ.

Rule [C1] corresponds to the case when the override check is valid, while the last three rules denote the cases when at least one constraint φ of Y is not implied by $I \wedge X$. Based on the region variables of the constraint φ, our algorithm strengthens either the precondition of the overridden method X (rule [C2]), or the subclass invariant I (rule [C3]), or both X and I (rule [C4]).

```
class Triple⟨r1,r2,r3,r3a⟩
  extends Pair⟨r1,r2,r3⟩ where r2⪰r1∧r3⪰r1∧r3a⪰r1 {
  Object⟨r3a⟩ thd
  Pair⟨r4,r5,r6⟩ cloneRev⟨r1,r2,r3,r3a,r4,r5,r6⟩() where r2⪰r6∧r3a⪰r5
    { Pair⟨r4,r5,r6⟩ tmp =new Pair⟨r4,r5,r6⟩(null,null);
    tmp.fst=thd; tmp.snd=fst; tmp}
```

Figure 4.13: Triple Class

To illustrate the override resolution mechanism, we define `Triple` class as a subclass of `Pair` class (defined in Figure 3.4), as shown in Figure 4.13. Two basic constraints are present in an overriding `cloneRev` method, namely `r2⪰r6` and `r3a⪰r5`. The first constraint is already satisfiable, but the second constraint cannot be directly placed in the class invariant of `Triple`, nor in the precondition of `Pair.cloneRev`. Nonetheless, we can still split it into two constraints `r3a=r3` and `r3⪰r5` that can be added to `inv.Triple` and `pre.Pair.cloneRev`, respectively. We have a choice of mapping the extra region `r3a` to either `r3` or `r2` using `[r3a↦r3]` or `[r3a↦r2]`, respectively. We choose the former since `(r3⪰r5)` exists in `pre.Pair.cloneRev` but not `(r2⪰r5)`. While multiple solutions exist, we choose a solution which minimizes the number of new constraints.

The overridden method and the overriding method must have the same signature, namely the same number of region parameters. Therefore after solving all the override checks, we change the region annotations of the methods such that the overridden methods have the same number of regions as their corresponding overriding methods. This affects only the regions of the receiver, and effectively uses the maximal regions from classes where the virtual methods are defined. Given a program, we assume that for any class of the program all the subclasses (used in that program) are known. Consider a class A$\langle r_1..r_m\rangle$, its subclass B$\langle r_1..r_m..r_n\rangle$, and a method A.mn$\langle r_1..r_m, r'_1..r'_p\rangle$ overridden by B.mn$\langle r_1..r_m..r_n, r'_1..r'_p\rangle$. The region parameters of A.mn are changed to A.mn$\langle r_1..r_m..r_n, r'_1..r'_p\rangle$.

Moreover, at each call site to a virtual method, additional regions are instantiated from either padded regions or from the first region of the current receiver, consistent with the solution adopted for downcasting (see Appendix A.4 for more details). A solution that uses the first region of the current receiver is illustrated by the following example, where the regions constraints are shown as comments:

$$\texttt{A}\langle r_1..r_m\rangle\ \texttt{a = new B}\langle p_1..p_m..p_n\rangle\texttt{(..)}//p_{m+1}{=}p_1\wedge..\wedge\ p_n{=}p_1$$

$$\cdots\ \texttt{a.mn}\langle r_1..r_m, r_{m+1}..r_n, r'_1..r'_p\rangle\texttt{(..)}\cdots\ //r_{m+1}{=}r_1\wedge..\wedge\ r_n{=}r_1$$

4.7 Dependency Graph and Mutual Dependency

Due to a fairly complex inter-dependency between classes and methods, our summary-based region analysis is required to process the classes and methods in some particular order given by a global dependency graph. We group the dependencies into two main categories:

1. *Constituent Dependencies* arising from the constituents of each class and method:

   - $cn_i \rightarrow cn_j$: the class cn_i depends on the class cn_j. It denotes that the class cn_j is the type of a field of the class cn_i or that the class cn_i is a subclass of the class cn_j.
   - $c.mn_i \rightarrow cn_j$: the method mn_i of the class c depends on the class cn_j. It denotes that the method $c.mn_i$ makes use of the class cn_j in its body.
   - $c.mn_i \rightarrow cn.mn_j$: the method mn_i of the class c depends on the method mn_j of the class cn. It denotes that the method $c.mn_i$ calls the method $cn.mn_j$.

2. *Override Dependencies* arising from the method overriding checks of the following form:

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

   where cn is a subclass of cn', $cn'.mn$ is the overridden method, while $cn.mn$ is the overriding method. The check is performed during the analysis of the overridden method, when the subclass invariant and the overriding method precondition have to be already computed. Overriding conflict resolution may strengthen the subclass invariant and/or the overridden method precondition. In order to localize the changes, the subclass and the overridden method have to be analysed together, that means they are in the same SCC node of the dependency graph. The overriding method depends on its receiver that is an instance of the subclass. Therefore in general the subclass, the overridden method, and the overriding method are in the same SCC node. For each method overriding check, the following override dependencies are generated:

   - $cn'.mn \rightarrow_o cn.mn$: overridden method $cn'.mn$ depends on overriding method $cn.mn$.
   - $cn \rightarrow_o cn.mn$: subclass cn depends on overriding method $cn.mn$. Note that a constituent dependency $cn.mn \rightarrow cn$ also exists, because $cn.mn$ is a method of cn.
   - $cn \rightarrow_o cn'.mn$: subclass cn depends on overridden method $cn'.mn$.
   - $cn'.mn \rightarrow_o cn$: overridden method $cn'.mn$ depends on subclass cn.

$$\begin{array}{cc}
\boxed{\mathbf{RI\text{-}M\text{-}CLASS}} & \boxed{\mathbf{RI\text{-}M\text{-}METH}} \\
\dfrac{\begin{array}{c}\vdash \mathit{def}_i \Rightarrow \mathit{def}'_i, Q_i \quad i:1..n \\ Q'_{i:1..n} = \mathit{fixpoint}(Q_{i:1..n})\end{array}}{\vdash \mathit{def}_{i:1..n} \Rightarrow \mathit{def}'_{i:1..n}, Q'_{i:1..n}} & \dfrac{\begin{array}{c}\Gamma_i \vdash \mathit{meth}_i \Rightarrow \mathit{meth}'_i, Q_i \quad i:1..n \\ Q'_{i:1..n} = \mathit{fixpoint}(Q_{i:1..n})\end{array}}{\Gamma_1, .., \Gamma_n \vdash \mathit{meth}_{i:1..n} \Rightarrow \mathit{meth}'_{i:1..n}, Q'_{i:1..n}}
\end{array}$$

Figure 4.14: Region Inference for Mutually Recursive Declarations

We propose a set of inference rules that attempt to collect the constituent and override dependencies from a Core-Java application (more details are given in Appendix A.3). A topological sorting algorithm orders the collected dependencies into a global dependency graph. The global dependency graph organizes the class declarations and the method declarations into a hierarchy of strongly connected components (SCCs). Each SCC consists of a set of *mutually dependent* class declarations and method declarations.

Our region inference performs a bottom-up processing of the SCCs, such that each SCC is analysed only once. In order to simplify the region inference for a SCC, inside each SCC we ignore the override dependencies and using only the constituent dependencies we topologically sort the class declarations and the method declarations into a local hierarchy. In fact this local hierarchy is still a hierarchy of SCCs, that we called Sub-SCCs to distinguish from the others. Each set of classes (or methods) in a Sub-SCC is regarded as a set of *mutually recursive* class declarations (or method declarations).

Mutually recursive class (method) declarations have to be analysed together. Figure 4.14 presents the main rules for classes and methods. Constraint abstractions are used to do the fixpoint iteration. Mutually recursive classes are handled in a similar way as the recursive fields of a class according to region monomorphic recursion principle. We illustrate the inference process in Figure 4.15. The two classes `A` and `B` are mutually recursive and therefore they have the same region annotation and invariant. The additional region `r3` is used to store the mutually recursive field `fst` of class `A`. Mutually recursive methods can be handled in a similar way as the recursive method according to the region polymorphic recursion. The fixpoint analysis is applied on a set of mutually recursive constraint abstractions (one constraint abstraction per method). The iteration is progressively done until all these constraint abstractions reach a fixpoint.

As a conclusion, the region inference for one SCC (regarded as a Sub-SCCs hierarchy that is processed bottom-up) consists of the following steps:

1. Region inference for each class declaration of the current Sub-SCC:

```
class A extends Object { B fst; }
class B extends A { Object snd; }
```
(a) Before Inference

```
class A⟨r1,r2,r3⟩ extends Object⟨r1⟩ where r3⪰r1∧r2⪰r3{
  B⟨r3,r2,r3⟩ fst; }
class B⟨r1,r2,r3⟩ extends A⟨r1,r2,r3⟩ where r3⪰r1∧r2⪰r3{
  Object⟨r2⟩ snd; }
```
(b) After Inference

Figure 4.15: Example of Mutually Recursive Classes

   (a) Set the region types for the class fields.

   (b) Set the region type for the class itself.

   (c) Two previous steps use the monomorphic recursion principle for the case of recursive fields or mutually recursive class declarations.

   (d) Compute the class invariant as a constraint abstraction.

2. Region inference for each method declaration of the current Sub-SCC:

   (a) Annotate the method with fresh region types.

   (b) Collect the region constraints from the method body and the invariants of the method parameters, method result and receiver.

   (c) Compute the method precondition as a constraint abstraction.

   (d) In the case of a recursive method (or mutually recursive methods) do the fixpoint analysis.

   (e) For an overriding method solve the method overriding. If a class invariant or a method precondition is changed re-start the analysis from the Sub-SCC that correspond to that class or method. Note that only the computation of the constraint abstractions (either preconditions or invariants) has to be re-done, the bodies of the class declarations and method declarations are not traversed again.

3. Solve the region annotations inside the body of each method of the SCC (the order is not longer important because at this step every method precondition and class invariant has reached the fixpoint). Based on the region constraint collected before from the method body, either localize the regions by *letreg* or make them equivalent to the global regions.

Treating the whole program as one big SCC may have a bad impact on the region annotation of the classes. It can lead to a bunch of unnecessary region parameters on classes, because of the monomorphic recursion principle that forces all SCC classes to have the same region annotation. The precision of the class invariant may also be affected by the no-dangling requirement imposed on the additional regions. However the method precondition computation is not affected.

4.8 Correctness of Inference Algorithm

This section is devoted to the correctness of the region inference, which is formulated in terms of soundness and completeness. We first introduce a lemma that states that any result of the region inference is correct, that means it is well typed according to the region type system and is valid according to the Definition 3.7.1.7. Note that the symbol $\Longrightarrow$ denotes logical implication, while $\Rightarrow$ denotes region constraint entailment.

Definition 4.8.0.1 (Region Annotations Erasure)**.** *The erasure of region annotations is defined as follows:*

1. *Given a region-annotated program P' such that $P' = def_1..def_n$. The function $erasure(P')$ is defined as*
$$erasure(P') \ =_{def} \ erasure(def_1)..erasure(def_n)$$
2. *Given a region-annotated class declaration def' such that def' $=$ class $cn_1\langle r^+\rangle$ extends $cn_2\langle r^+\rangle$ where φ $\{(t\,f)^*$ $meth^*\}$. The function $erasure(def')$ is defined as*
$$erasure(def') \ =_{def} \ class\ cn_1\ extends\ cn_2\ \{(erasure(t)\,f)^*\ (erasure(meth))^*\}$$
3. *Given a region-annotated method meth' such that meth' $=$ $t\ mn\langle r^*\rangle((t\ v)^*)$ where φ $\{e\}$. The function $erasure(meth')$ is defined as*
$$erasure(meth') \ =_{def} \ erasure(t)\ mn((erasure(t)\ v)^*)\ \{erasure(e)\}$$
4. *Given a region type t such that $t = \tau\langle r^*\rangle$. The function $erasure(t)$ is defined as*
$$erasure(t) \ =_{def} \ \tau$$

5. *Given a region-annotated expression e'. The function erasure(e') is defined as follows:*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{erasure}(e') =_{def} \ \mathit{case}\ e'\ \mathit{of}} \\
\{(t\ v)\ \ e\} & \rightarrow & \{(\mathit{erasure}(t)\ v)\ \ \mathit{erasure}(e)\} \\
\textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & \mathit{erasure}(e) \\
\mathit{lhs} = e & \rightarrow & \mathit{lhs} = \mathit{erasure}(e) \\
\textbf{new}\ cn\langle r^{+}\rangle(v^{*}) & \rightarrow & \textbf{new}\ cn(v^{*}) \\
\textbf{while}\ v\ e & \rightarrow & \textbf{while}\ v\ \mathit{erasure}(e) \\
\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \textbf{if}\ v\ \textbf{then}\ \mathit{erasure}(e_1)\ \textbf{else}\ \mathit{erasure}(e_2) \\
e_1\ \textbf{;}\ e_2 & \rightarrow & \mathit{erasure}(e_1)\ \textbf{;}\ \mathit{erasure}(e_2) \\
v.mn\langle r^{*}\rangle(v^{*}) & \rightarrow & v.mn(v^{*}) \\
\textbf{null} & \rightarrow & \textbf{null} \\
k & \rightarrow & k \\
\textbf{lhs} & \rightarrow & \textbf{lhs}
\end{array}
$$

Lemma 4.8.0.1. *Given any source language Core-Java program, P such that WFClasses(P) and $\forall def{\in}P \cdot$ FieldsOnce$(def)\wedge$MethodsOnce(def). Suppose $\vdash P \Rrightarrow P'$.*

1. *If $\tau \in P$, $\vdash\tau{\Rrightarrow}t, \varphi$, and $t \in P'$, then erasure$(t) = \tau$.*

2. *Given any source language Core-Java expression, e. If $\Gamma \vdash e{\Rrightarrow}e'{:}t,\ \varphi$, then erasure$(e') = e$.*

3. *Given any source language Core-Java method, $meth$ of a class $cn{\in}P$. Let be $R{=}\{r_{1..n}\}$, $\Gamma{=}\{this : cn\langle r_{1..n}\rangle\}$, and $\varphi{=}inv.cn\langle r_{1..n}\rangle$. If $\Gamma \vdash meth \Rrightarrow meth'; Q$, then erasure$(meth') = meth$.*

4. *Given any source language Core-Java class declaration, $def{\in}P$. If $\vdash def{\Rrightarrow}def'; Q$, then erasure$(def') = def$.*

Proof: Using the Definition 4.8.0.1, the proof is by induction on the inference rules.

Lemma 4.8.0.2. *Given any source language Core-Java program, P such that WFClasses(P) and $\forall def{\in}P \cdot$ FieldsOnce$(def)\wedge$MethodsOnce(def). Suppose $\vdash P \Rrightarrow P'$, $valid(P')$, WFClasses(P'), and $\forall def'{\in}P' \cdot$ FieldsOnce$(def')\wedge$MethodsOnce(def').*

1. *If $\tau \in P$, $\vdash\tau{\Rrightarrow}t, \varphi$, and $t \in P'$, then*

$$\forall R, \varphi' \cdot (reg(t){\subseteq}R \wedge \varphi'{\Rightarrow}\varphi) \Longrightarrow P'; R; \varphi' \vdash_{type} t$$

2. *Given any $t \in P'$ and $t' \in P'$. If $\vdash t{<:}t' \Rrightarrow \varphi$ and $\varphi \neq$ false, then*

$$\forall R, \varphi' \cdot (reg(\varphi) \subseteq R \wedge \varphi' \Rightarrow \varphi) \implies P'; R; \varphi' \vdash t{<:}t'$$

3. *Given any source language Core-Java expression, e. If $\Gamma \vdash e \Rrightarrow e'{:}t,\ \varphi$ and all classes that e' depends on are well-formed in P', then*

$$retvars(e') {=} \emptyset \quad and \quad retregs(e') {=} \emptyset \quad and$$

$$\forall R, \varphi' \cdot ((regs(e') \cup reg(\Gamma) \cup reg(\varphi)) \subseteq R \wedge \varphi' \Rightarrow \varphi) \implies P'; \Gamma; R; \varphi' \vdash e'{:}t$$

4. *Given any source language Core-Java method, $meth$ of a class $cn {\in} P$. Let be $R {=} \{r_{1..n}\}$, $\Gamma {=} \{this : cn\langle r_{1..n}\rangle\}$, and $\varphi {=} inv.cn\langle r_{1..n}\rangle$. If $\Gamma \vdash meth \Rrightarrow meth'; Q$ and all classes that $meth'$ depends on are well-formed in P', then*

$$valid(meth') \quad and \quad P'; \Gamma; R; \varphi \vdash_{meth} meth'$$

5. *Given any source language Core-Java class declaration, $def {\in} P$. If $\vdash def \Rrightarrow def'; Q$, all methods $meth \in def$ are inferred such that $\Gamma \vdash meth \Rrightarrow meth'; Q$ where $meth' \in def'$, for all $meth' \in def'$ the method overriding is solved with respect to the resolution rules from Figure 4.12, and all classes that def' depends on are well-formed in P', then*

$$valid(def') \ \ and \ \ P' \vdash_{def} def' \quad and \quad P' \vdash InheritanceOK(def')$$

Proof: The detailed proof is in Appendix A.2.4.

Theorem 4.8.0.3. (Correctness) *Given any well-normal-typed source program P (class declarations and their methods) in Core-Java.*

1. (Soundness)

   If $\vdash P \Rrightarrow P'$, then P' is a valid program well-typed by the region type system, i.e., $valid(P')$ and $\vdash P'$.

2. (Program Preserving)

   If $\vdash P \Rrightarrow P'$, then $erasure(P') = P$.

3. (Completeness)

   There exists a program P' in region-annotated Core-Java, such that $\vdash P \Rrightarrow P'$.

where $\vdash P \Rrightarrow P'$ denotes the inference algorithm summarized in Section 4.7.

Proof: (1) The proof is based on Lemma 4.8.0.2. (2) The proof is based on Lemma 4.8.0.1. (3) The proof is based on the fact that our fixpoint analysis always terminates [132] and that the region constraint entailment is decidable [96, 181].
Proof is detailed in Appendix A.2.5.

4.9 Field Region Subtyping

The concept of region subtyping (defined in Section 3.5) can be further extended to selected fields if they are immutable after object initialization. We assume that object initialization is done in the constructors. The fields of recursive structures are particularly important as they may involve many objects that are typically grouped into the same region. We can use an *isRecReadOnly* function to check if a class has immutable recursive fields or not. With this information, we can support a more precise region subtyping rule, as follows:

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

Note that the last region of a recursive type is used to store the recursive fields. One advantage of this field region subtyping rule is that it allows each recursive object to be placed in a region that is different (and may have a longer lifetime) from that of the prior object in the recursive chain. Such a feature is important for recursive methods that build temporary data structures during recursive invocations. An example is the following program, called *Reynolds3*, that was highlighted in [52, 18]. We use `RList` to denote a list structure with an immutable recursive field. Our current proposal places onus on the programmer to indicate the immutable fields. Compile-time techniques for checking (and/or inferring) immutable fields at definition-site can be found in [149, 16, 196], or at use-site in [103, 39]. Applying region inference with field subtyping, we are able to obtain the following program where the new lists `p1` and `p2` are created in the local regions `r1` and `r2`, respectively:

```
class RList⟨r1,r2,r3⟩ where r3⪰r1∧r2⪰r3
{  Integer⟨r2⟩ value;
   RList⟨r3,r2,r3⟩ next; ...}

class Tree⟨r1,r2,r3⟩ where r3⪰r1∧r2⪰r3
```

```
{  Integer⟨r2⟩ value;
   Tree⟨r3,r2,r3⟩ left;
   Tree⟨r3,r2,r3⟩ right; ... }

boolean search (RList⟨l1,l2,l3⟩ p, Tree⟨t1,t2,t3⟩ t)
                  where l3⪰l1∧l2⪰l3∧t2⪰l2∧t3⪰t1∧t2⪰t3 {
   if isNull(t) then return false;
   else { if member(t.value,p) then return true;
     else { boolean b1;
            letreg r1 in {
                  RList⟨r1,l2,r1⟩ p1 =new RList⟨r1,l2,r1⟩(t.value,p);
                  b1 = search(p1,t.left);
            }
            if b1 then return true;
            else { boolean b2;
                   letreg r2 in {
                   RList⟨r2,l2,r2⟩ p2 =new RList⟨r2,l2,r2⟩(t.value,p);
                   b2 = search(p2,t.right);
                   }
                   return b2; } } }
```

The memory performance of such a region-inferred program is comparable to that obtained by escape analysis [52].

In contrast, applying object region subtyping to *Reynolds3* benchmark, we are not able to localise the new lists `p1` and `p2` inside the function. They are created in the global region `l1` of input list `p`. This causes a *space leak* inside the region `l1`.

```
boolean search (RList⟨l1,l2,l3⟩ p, Tree⟨t1,t2,t3⟩ t)
                  where l3=l1∧l2⪰l3∧t2⪰l2∧t3⪰t1∧t2⪰t3 {
   if isNull(t) then return false;
   else { if member(t.value,p) then return true;
     else { boolean b1;
            RList⟨l1,l2,l1⟩ p1 =new RList⟨l1,l2,l1⟩(t.value,p);
            b1 = search(p1,t.left);
            if b1 then return true;
            else { boolean b2;
            RList⟨l1,l2,l1⟩ p2 =new RList⟨l1,l2,l1⟩(t.value,p);
            b2 = search(p2,t.right);
            return b2; } } }
```

The memory performance of such a region-inferred program is comparable to that obtained by the original region inference for ML ([18]), but worse than that obtained by escape analysis [52].

We thus advocate the use of field region subtyping where possible, to obtain better region annotations and space reuse.

4.10 Experimental Validation

4.10.1 Implementation

We have constructed a prototype of our region inference algorithm. The prototype takes as input either a Java program or a region-annotated Java program and outputs a Titanium Java program with regions.

We have used the Titanium 2.205 infrastructure (based on Java 1.4) [100] to execute region-annotated programs. The Titanium language is a Java dialect for high-performance parallel scientific computing. In addition to garbage collection, Titanium also supports memory management with explicit regions based on Gay's work [75, 76, 74]. This region-based memory management uses runtime checks to ensure that deleting a region does not create dangling references. We added instrumentation code to the Titanium infrastructure to collect dynamic memory consumption and other region statistics.

We have implemented the entire prototype using the Glasgow Haskell 6.4.1 compiler [150]. Our prototype consists of the following main modules (some of them are depicted in Figure 3.1):

- *Java to Core-Java Translator* that can translate either a Java program or a region-annotated Java program into an equivalent Core-Java or region-annotated Core-Java program. The translator can also build the dependency graph using the rules presented in Appendix A.3. A description of our translator is given in [45].

- *Region Inference Module* that takes an input Core-Java program and generates an equivalent region annotated Core-Java program. The module implements the main inference algorithm including method overriding (Section 4.6) and downcasting (Appendix A.4) and also a part of the extensions (like field region subtyping presented in Section 4.9, and statements, exceptions, static fields and methods, arrays and a simple solution for the interfaces presented in Appendix A.6).

- *Region Type Checking Module* that checks the type safety of a region-annotated Core-Java program. The region-annotated program could either be supplied by the programmer or generated by the inference module. In the latter case, the region-annotated programs always pass the type checker.
- *Core-Java with Runtime Regions Module* that implements those two analyses (described in Appendix A.5) that translate the region types into region handles used at runtime.
- *Titanium Program Generation* that can translate a Core-Java program with region handles into a Titanium Java program with regions.

We have also built a special library to solve the region lifetime constraints, φ, from Figure 3.2(b). Our constraint solver mainly performs the following three operations: *constraint simplification* to reduce the number of the gathered constraints, *constraint satisfiability* to check the possible contradictions (used only by the type checker) and *constraint entailment*, $\varphi_1 \Rightarrow \varphi_2$ to check that the constraint φ_2 is satisfied by the context φ_1. The main part of our solver consists of a simple transitive closure algorithm (based on transitivity and equality) that computes the upper and lower bounds for each region variable from a given constraint φ. The entailment is implemented as a subset check of the region bounds. Note that if a region does not occur in a constraint, by default its lower and upper bounds are $\perp$ and *heap*, respectively.

4.10.2 Experiments

The primary objective of our experiments was to validate the correctness of our region inference algorithm. In our framework, this validation can be done in two ways: either at compile time by the type checking module or at execution time by Titanium runtime checks. We run the experiments on a 3 GHz Pentium 4 machine with 2GB RAM running Linux Fedora Core 4.0. The first column of the table in Figure 4.16 presents the list of benchmark programs used in our experiments: *RegJava* benchmark programs from [41], *Java Olden* benchmarks from [34] and also two small benchmarks, *Reynolds3* (discussed in Section 4.9) from [52, 18] and our *foo-sum* that multiplies a pattern from an example described in [80] at page 6 (function *foo* from Section 3.5.2 shows that pattern). The second column of the table in Figure 4.16 shows

Programs	Size (lines)		Compile Time (sec)		Regions (maximum)		
					Class	Meth	
	Src	Ann	Infer	Check	Param	Param	Handles
RegJava benchmarks							
Sieve of Eratosthenes	80	12	0.08	0.09	2	4	2
Ackerman	67	5	0.02	0.02	1	1	0
Merge Sort	170	16	0.35	0.30	2	6	3
Mandelbrot	110	14	0.05	0.05	1	3	1
Naive Life	114	14	0.08	0.14	3	9	2
Optimized Life (array)	121	15	0.09	0.16	3	9	2
Optimized Life (dangling)	35	5	0.01	0.02	9	0	0
Optimized Life (stack)	80	10	0.04	0.05	3	3	1
Java Olden benchmarks							
BH	1191	96	7.02	12.87	26	66	9
Bisort	345	16	0.11	0.20	2	4	2
Em3d	510	37	0.26	1.10	7	11	3
Health	594	42	0.47	0.55	3	9	3
MST	494	44	1.84	5.44	10	49	5
Perimeter	750	48	0.97	1.28	4	26	2
Power	789	40	0.44	1.26	13	19	13
Treeadd	200	12	0.02	0.05	2	6	2
TSP	562	21	0.22	0.58	2	10	2
Voronoi	1058	93	7.15	13.62	9	46	3
Other benchmarks							
Reynolds3	59	12	0.11	0.18	3	7	3
foo-sum	65	10	0.11	0.15	3	6	2

Figure 4.16: Region Analysis Measurements

the number of source lines for each benchmark, while the third column denotes the number of source lines affected by region annotations (letreg statements, method preconditions, class invariants). Region annotations occur in around 12.3% of the source lines for the programs of the *RegJava* benchmarks and in around 7% of the source lines for the programs of *Java Olden*. This may represent a sizable mental effort for a programmer (with a region type checker) who manually writes the region annotations.

The second objective of our experiments was to check the scalability of our region inference system. The fourth and fifth columns of the table in Figure 4.16 show the region inference and region checking times, respectively. These times also include the region handles analysis. The region inference runs in less than one second for all of the programs of *RegJava* benchmarks, *Reynolds3* and *foo-sum* and in less than eight seconds for all programs of *Java Olden* benchmarks. These results suggest that our region inference algorithm is tractable in practice. On the other hand, the region checking times are higher than those of region inference. The region checking system requires more entailment checks than the region inference system. The table in Figure 4.16 also contains the number of region parameters (maximum number per program) inferred by our algorithm for classes (sixth column) and methods (seventh column), respectively.

Java Olden benchmarks	(1-Memory Usage/Total Allocation)*100%	
	Our system	*Jreg* system [37]
BH	91.5%	88%
Bisort	0%	0%
Em3d	0.8%	0%
Health	76%	71%
MST	0.5%	0%
Perimeter	0%	0%
Power	98.2%	97%
Treeadd	0%	0%
TSP	57.3%	56%
Voronoi	2.6%	2%

Figure 4.17: Statistics of Dynamic Memory Consumption: Part I

These data were collected before code generation (before runtime region analyses described in Appendix A.5). We also measured the number of region handles after runtime region analysis. The last column of the table in Figure 4.16 shows the number of region handles (maximum number per program) for methods. The number of region parameters of a class depends on the number of class fields (recursive and non-recursive), the class level in the class hierarchy but also on the dependencies in the global dependency graph. The number of region parameters of a method depends on the number of method arguments, but also on the number of region parameters of each argument type, including the receiver. For all benchmarks that we used, the average number of class region parameters is 3.9, the average number of method region parameters is 5.1, while the average number of method region handles is 0.4. In general, these average numbers and the maximum numbers shown in the table from Figure 4.16 suggest that our region annotations have reasonable size in practice.

The third objective of our experiments was to evaluate the quality of our automatically inferred region annotations as compared to region annotations produced by human experts. We tested our system on the *RegJava* benchmark programs. These programs have been hand-annotated for the *RegJava* region checker in [41]. We obtained the same results as those from the *RegJava* system, except for *optimized life (with dangling)* program. Our region inference produces one less local region, since our system uses the *no-dangling* policy rather than the *no-dangling-access* policy of the *RegJava* checker. This set of programs suggests that our region inference is comparable in performance to human experts.

The fourth objective of our experiments was to evaluate the ability of our region-based system to reuse memory. We have compiled our benchmarks to run on Titanium. We measured

Programs	(1-Memory Usage/Total Allocation)*100%			
	Our system			*RegJava*
	Invariant Sub	Object Sub	Field Sub	system
RegJava benchmarks				
Sieve of Eratosthenes	0%	0%	0%	0%
Ackerman	99.6%	99.6%	99.6%	99.6%
Merge Sort	82.1%	82.1%	82.1%	82.1%
Mandelbrot	99.8%	99.8%	99.8%	99.8%
Naive Life	0%	0%	0%	0%
Optimized Life (array)	80.4%	80.4%	80.4%	80.4%
Optimized Life (dangling)	0%	0%	0%	0%
Optimized Life (stack)	0%	0%	0%	0%
Other benchmarks				
Reynolds3	0%	0%	99.6%	-
foo-sum	66%	99%	99%	-

Figure 4.18: Statistics of Dynamic Memory Consumption: Part II

the total memory allocation running the programs in a setting that never reclaims memory. This setting is similar to the situation where there is only one region that is never garbage collected but is deallocated at the end of the program. The memory utilization was measured running the programs with region-based support and no garbage collector. The collected data include only the application memory size and not the memory used by the virtual machine itself. The results were measured as an average for a large set of inputs. The second column of the table in Figure 4.17 shows the statistics for the *Java Olden* benchmarks, while the fourth column of the table in Figure 4.18 shows the statistics for the *RegJava* benchmarks. These statistics represent the relative memory savings of region-based memory management with respect to the total memory allocation. They also can be regarded as a measure of how good is the inference result with respect to the trivial inference solution (put everything in one region). The results indicate that the ability of our region-based system to reuse memory depends on memory characteristics of each particular application. The memory savings fluctuate across our set of benchmarks, ranging from value 0% (denoting no reuse) to large values (which indicate a high degree of reuse). The initial results are encouraging showing that our region-based system was able to reuse significant amounts of memory for the cases where data was not live throughout the program. A closer inspection of the benchmarks for which the memory savings were less than 3% revealed that most data objects are long-lived. The current prototype implements the simple solution for downcasting from Appendix A.4.

The fifth objective of our experiments was to evaluate the performance of three kinds of region subtyping presented in the paper: *invariant (region) subtyping*, *object (region) subtyping* and *field (region) subtyping*. Columns 2-4 of the table in Figure 4.18 show the results for some of the program benchmarks. Note that our prototype uses the object region subtyping and automatically switches to the field region subtyping when the programmer provides the class fields which are immutable. In the current experiments, we provided immutability information only for *Reynolds3*. The table in Figure 4.18 indicates that the results are similar for the *RegJava* benchmarks. We also did the same experiments for the *Java Olden* benchmarks and we obtained identical results for all three kinds of region subtyping. However, two benchmarks, *foo-sum* and *Reynolds3*, require the object region subtyping and field region subtyping, respectively, in order to obtain a higher degree of reuse. Although these two benchmarks are small, they suggest that there are potential benefits for using improved region subtyping.

The sixth objective was to compare our approach with other region-based approaches. Our approach uses lexically scoped regions and forbids dangling references. First, we have compared our experimental results with those obtained using the *RegJava* system. *RegJava* uses lexically scoped regions, allows creating dangling references but prevents the program from accessing dangling references. The last column of the table in Figure 4.18 presents the results of *RegJava* as they were given in [41]. Despite an extra region localized by the *RegJava* system for *optimized life (with dangling)*, our memory reuse results are similar to those of the *RegJava* system. Second, we have compared our experimental results for the *Java Olden* benchmarks with those produced by the *Jreg* system in [37]. *Jreg* uses non-lexically scoped regions, allows creating dangling references but prevents the program from accessing dangling references. The comparison indicates that our system performed about as well as *Jreg* for all of the *Java Olden* programs.

In conclusion, the overall results for our benchmarks are encouraging, the programs based on our inferred regions were able to reuse significant amount of memory for most of the cases where data was not live throughout the execution. The experiments suggest that our results are competitive with those hand annotated by human experts, but also with the approach based on non-lexically scoped regions with no-dangling-access [37]. The experiments also suggest that our region inference analysis is scalable in terms of analysis time and the number of region parameters.

4.11 Related Work

The basic ideas of a region type system were introduced by Tofte and Talpin [191]. They proposed a region inference approach for a typed call-by-value λ-calculus, and tested their approach in a region-based implementation of Standard ML [190]. A soundness proof for region inference is presented in [192] and the inference algorithms are given in [189, 17]. In their approach, all values (including primitives and function values) are put into regions at runtime, and all points of region placement can be inferred automatically using a polymorphic effect system with effect masking inspired from [184]. Specifically, each polymorphic effect denotes the set of regions the program might access, which permits dangling references (to closures and data structures) that are never accessed. In contrast, our region type system uses outlives constraints to ensure that the program never creates dangling pointers. Our system supports region type subtyping, while Tofte and Talpin's system does not have subtyping. Their rules are based on unification of types that contain effects. Effects are sets paired with effect variables. Effect variables are used to support unification, type polymorphism (with monomorphic recursion) and higher-order functions. Although we handle function subtyping via method overriding, we have not considered type polymorphism and higher-order functions. In [94], a more permissive region type system than Tofte and Talpin's system was presented using a System F-like polymorphism in types, regions, and effects rather than the let-polymorphism of the original system.

Following Tofte and Talpin's work, Grossman, Morrisett et al. [80, 98] have developed a region-based approach for a safe dialect of C, called Cyclone. Cyclone's type system keeps track of the set of live regions (called capability as in [48]) at any program point. Whenever a pointer is dereferencing Cyclone checks at compile time whether the associated region of the pointer is in the capability. Function effects are not inferred from the function body, but rarely need to written by the programmer because of the default annotations. In order to handle type polymorphism and existential types (which can encode closures and objects) Cyclone uses a special operator on types (instead of having effect variables) to denote the region variables which occur free in a type.

Christiansen and Velschow proposed a region-based approach (similar to Tofte and Talpin's approach) to memory management in an object-oriented language like Java [41]. They call their system RegJava and use a stack of lexically scoped regions for memory management. They developed a region type system and demonstrated its soundness by linking the static semantics with the dynamic semantics. However, their system requires programmers to manually annotate programs with region annotations. In their system, each class is augmented with the full set of regions from its class hierarchy, including those from its subclasses. As a result many phantom regions may be introduced for superclasses. However, the phantom regions make downcast operations and method overriding trivially safe. As a comparison, the padded regions used by of our region-safe downcast solution are different from phantom regions. We selectively attach padded regions to superclasses only when relevant downcast operations may occur subsequently.

Researchers have recently advocated non-lexical regions to support tighter region lifetimes [75, 76, 199, 98, 185]. Some of these approaches require programmers to at least indicate when regions are to be created, allocated and released. Gay and Aiken implemented a region-based extension of C, called C@, which used reference counting on regions to safely allocate and deallocate regions with a minimum of overhead [75]. Using special region pointers and explicit deleteregion calls, they provided a means of explicitly manipulating region-allocated memory. This approach allows non-lexical regions where earlier deallocation of regions are possible, but stack implementation of regions is no longer valid. Their work indicated that region-based programming often use less memory and is faster than traditional malloc/free-based memory management. However, counting escaping references can incur noticeable overhead. One technique [7] accepts a program with lexically scoped regions, then transforms the program to allow, when possible, late creation and early deletion of these regions. This technique is complementary to our approach to region inference, as it could be used as a post-phase. With an explicit outlive relation on the lexical regions, we have also exploited the concept of region subtyping, as pioneered in [80].

Henglein *et al.* [93] developed a region type system *HMN*, where the region annotations uses an imperative language for manipulating region handles. *HMN* has its own region type system (proven sound by Niss [133]) and its inference algorithm. Makholm [119] has extended the inference algorithm to a theoretical framework applicable to multiple languages. In the case of object-oriented languages he used an approach where phantom regions are possible. However, no implementation of this inference algorithm has been provided for an object-oriented language.

The Real-Time Specification for Java (RTSJ) [19] provides a new memory management model for Java based on scoped memory areas. Scoped memory areas provide predictable allocation and deallocation performance, and ensure that real-time threads need not block while memory is being reclaimed. Our regions are similar in principle to scopes. The main difference is that scopes are first-class entities which can be entered/re-entered by multiple threads in a lexical manner. However, a scope may be reset when the last thread exits the scope. The order in which threads enter scopes induces a runtime parent relation on scopes that determines safety reference patterns. The RTSJ forbids dangling references by a rule similar to our no-dangling condition: an outer (parent) scope may not hold a reference to an object within an inner (child) scope. To avoid cycles in the scope parent relation, a scope may have at most one parent such that the nesting structure of scopes is restricted to trees. To avoid the unbounded pauses caused by the garbage collectors, hard real-time threads are prohibited from manipulating heap references, even though it is perfectly legal for heap references to be stored in any scope. The RTSJ requires runtime checks to ensure that memory accesses do not violate the safety rules.

Beebee and Rinard presented an early implementation of scoped memory for Real-Time Java in the MIT Flex compiler infrastructure [13]. They rely on both static analysis and dynamic debugging to help locate incorrect uses of scoped memory.

Higuera *et al.* [99] have studied the combination of the region-based memory management with an incremental garbage collector in the context of Real-Time Specification for Java. They proposed a solution to improve the write barrier performance of both region-based memory management and the garbage collection.

Boyapati *et al.* [23] combined region types [191, 192, 48, 41] and ownership types [43, 42, 20, 22] in a unified framework to capture object encapsulation and prevent dangling references. The static type system guarantees that scope-memory runtime checks never fail for well-typed programs. It also ensures that real-time threads do not interfere with the garbage collector. Using object encapsulation, each object and all components it *owns* are put into the same region; in order to optimize on region lifetimes. Our region type system is similar to theirs, but we separate out object encapsulation and RTSJ issues. Moreover, we infer region types automatically across procedures, whilst they have limited support through intra-procedure inference and the use of defaults for region types.

Scoped types [205] is another proposal to statically maintain the invariants that RTSJ checks dynamically. They provide a syntactic approach to map the run-time hierarchy of RTSJ scoped memory areas in the program text by using the static hierarchy of Java packages. Thus, RTSJ programs must be re-factored so that objects which are meant to live in the same memory scope have to be declared in the same Java package. The model distinguishes between two kinds of classes: scoped classes which are allocated within a particular memory scope and gate classes whose instances turn scopes into first-class entities. A thread may enter a scope only by invoking methods of gate objects. Scoped classes in a package are accessible only to the classes defined in that package and its sub-packages, while gate classes are only accessible to classes defined in their immediate parent package. Thus, classes are not allowed to access classes in inner nested sub-packages other than gates of their immediate sub-packages. These accessibility constraints are similar to our non-dangling requirement. However, for the classes which are used in several RTSJ memory scopes, it may be necessary to either modify the application logic or to duplicate code.

Deters and Cytron [51] automatically translated Java code into Real-Time Java using dynamic analysis to determine the lifetime of an object. Because the analysis is dynamic, it may not be sound — it may miss some execution paths that create and use dangling references given their extracted object lifetime information. Also, Dhurjati *et al.* [53] proposed a compiler technique, based on *type homogeneity* principle, which tolerates dangling references as long as each freed object is being overwritten by another object of the same type. Their approach requires explicit malloc/free operation to be correctly supplied by programmer. While type safety is preserved, any logical errors caused by premature deallocation of objects is not detected by their system – neither at compile time, nor at runtime. Lattner and Adve [112] have used a context-sensitive pointer analysis to segregate distinct instances of logical data structures into separate pools in the heap. However, the programmer still has to explicitly deallocate objects. Their method is therefore applicable to C programs, but not for Java-based programs. Their primary focus is on performance improvement, but not automatic memory management. They evaluate how their transformation affects memory hierarchy behavior and overall program performance.

In research performed concurrently with ours, Cherem and Rugina have developed a three-stage region inference algorithm for Java [37]. Their algorithm relies on a flow analysis to propagate unifications between regions in an interprocedural manner. Using the no-dangling-access principle, their inference produces programs that use non-lexically scoped regions different than our lexically scoped regions. While our inference system is based on a region type system where object and field subtyping could be supported, their approach is formulated directly to generate region handles, and uses points-to analysis and liveness analysis to determine when regions can be deallocated. The use of non-lexical regions could in theory achieve better precision for region lifetime, and thus improve on space usage. However experimental results for the *Java Olden* benchmarks did not confirm this.

Salagnac *et al.* [169, 170] have developed a region inference algorithm that stores all connected objects into the same region. This simplistic policy leads to regions with a high percentage of dead objects. These potential memory leaks are reported to the programmer who can modify his code to avoid the leak. In contrast, our region inference uses automatic techniques (e.g. subtyping, polymorphism) to improve regions lifetimes without modifying the original program. Recently, Alex *et al.* [178] have proposed more advanced techniques to improve the lifetime of our inferred regions.

Qian and Hendren [160] have developed a runtime region-based allocator as an alternative to the region static analyses. Their approach dynamically categorizes allocation sites as local and non-local. Local objects are stored into local regions attached as extensions to the stack frames. Non-local objects are directly allocated into a global region that is garbage collected. However their prediction scheme is coarser than our region inference algorithm. For instance the case where the same allocation site is sometimes local and sometimes non-local is categorized as non-local by their scheme.

PART II

Better Genericity

CHAPTER 5

VARIANT PARAMETRIC TYPE SYSTEM

5.1 Introduction

In object-oriented programming a large software is built by combining different small objects into a large object, thus making the software reusability (or *genericity*) one of the most important issue.

Traditionally, most mainstream object-oriented (OO) languages, such as Java, C++ and C#, have relied on class subtyping to support reuse via *subtype polymorphism* (also called *inclusion polymorphism*). Subtype polymorphism is a nominal relation, based on a class hierarchy declared by the programmer. This mechanism is convenient since it allows storage of objects via safe upcast into generic data structure. However it is not expressive enough because the converse process of retrieving objects from the generic data structure requires the programmers to insert explicit type casts for downcast testing at runtime. This results in losing the benefits of static type checking (safety at compile time) and also in incurring the runtime overheads. As an example, we consider the following program fragment that uses the class `Cell`:

```
class Cell {
 Object fst; ··· }
...
 Cell cell; Int a;
 cell.fst = a;//safe upcast
 Int b = (Int) cell.fst;//explicit cast
 String s = (String) cell.fst;//explicit cast
```

Above explicit casts, inserted by programmer, are compiled into runtime checks. As can be seen, the first cast succeeds, while the second cast fails since `Int` is not a subtype of `String`. However the type checker is unable to predict them.

To address the shortcomings of subtype polymorphism, there have been several recent proposals (amongst the Java [24, 4, 127, 136, 35] and C# [107] communities) for *parametric polymorphism* to be supported with various design and implementation schemes. Parametric polymorphism allows *parametric types* and supports *structural subtyping*. For example, each class `c` is allowed to carry a list of type parameters for its fields, e.g., $\mathtt{c}\langle t_1,..,t_n\rangle$, whereby the type of each field can either be instantiated or left as a type variable. Below are two classes whose fields have been parameterized:

```
class Cell⟨A⟩ {
 A fst; ··· }
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
 B snd; ··· }
```

With such parameterized class declarations, we may then define specialized instances, such as `Cell⟨Int⟩`, `Cell⟨Float⟩` or `Pair⟨Int,Num⟩`, which contain more specific type information for the fields of each class instance. Though parametric types can coexist with class subtyping, pointwise matching of the respective fields is required. For example, the subtyping relation $\mathtt{Pair}\langle t_1,t_2\rangle <: \mathtt{Cell}\langle t_3\rangle$ is allowed only when `Pair<:Cell` and $t_1{=}t_3$. The latter condition is for pointwise matching of the common field. Similarly, $\mathtt{Pair}\langle t_1,t_2\rangle <: \mathtt{Pair}\langle t_3,t_4\rangle$ holds, provided $t_1{=}t_3$ and $t_2{=}t_4$. Pointwise matching (invariant subtyping) is required because field reading and field writing are based on opposite flows that change the directions of subtyping. This requirement limits the reusability of programs based on parametric types. In addition, a main proposal for Java, called GJ [24], uses *raw types* in order to ensure maximum compatibility with existing non-generic code. In GJ, every parameterized class $\mathtt{c}\langle t_1,..,t_n\rangle$ provides the raw type `c` as a supertype of any parameterized type $\mathtt{c}\langle t_1,..,t_n\rangle$ for any $t_1,..,t_n$. Moreover GJ permits unsafe coercions from a raw type to a parameterized type. For example, an object of type `Cell` can be passed where `Cell⟨Int⟩` is expected. This coercion is clearly unsafe since the field `fst` of an object of type `Cell` is not necessarily of type `Int`. Thus GJ compiler accepts such unsafe operatons by signaling *unchecked warnings*. Therefore static type safety cannot be guaranteed. However dynamic safety is still guaranteed by inserting runtime downcast checks. It is the programmer's responsibility to ensure that all unchecked operations are in fact safe.

To address the shortcomings of parametric polymorphism, Igarashi and Viroli [103] developed a new variant parametric type system (or VPT, in short) to improve the subtyping of

Figure 5.1: A Rich Subtyping Hierarchy

generic structures, depending on how the fields are being accessed. Let $\mathtt{c}$ denote a class with one type parameter. Let $\mathtt{o}$ denote an object of variant parametric type $\mathtt{c}\langle\alpha_1 t_1\rangle$ while $\mathtt{v}$ denotes a location of variant parametric type $\mathtt{c}\langle\alpha_2 t_2\rangle$, into which $\mathtt{o}$ is to pass. Each variant parametric type $\mathtt{c}\langle\alpha t\rangle$ has a variance α (see Section 5.3.2) attached to its field to indicate how the field is to be accessed. A field that is subject to read-only access via reference of $\mathtt{v}$ (denoted by $\alpha_2 = \oplus$) may be supported by covariant subtyping. That is, $\mathtt{c}\langle\alpha_1 t_1\rangle\mathtt{<:}\mathtt{c}\langle\oplus t_2\rangle$ if $\alpha_1\mathtt{<:}\oplus$ and $t_1\mathtt{<:}t_2$. Conversely, a field that is subject to write-only access via reference of $\mathtt{v}$ (denoted by $\alpha_2 = \ominus$) may be supported by contravariant subtyping. That is, $\mathtt{c}\langle\alpha_1 t_1\rangle\mathtt{<:}\mathtt{c}\langle\ominus t_2\rangle$ if $\alpha_1\mathtt{<:}\ominus$ and $t_2\mathtt{<:}t_1$. Also, a field that is subject to both read and write accesses via reference of $\mathtt{v}$ (denoted by $\alpha_2 = \odot$) must be supported by invariant subtyping. That is, $\mathtt{c}\langle\alpha_1 t_1\rangle\mathtt{<:}\mathtt{c}\langle\odot t_2\rangle$ if $\alpha_1\mathtt{<:}\odot$ and $t_1\mathtt{<:}t_2 \wedge t_2\mathtt{<:}t_1$. Lastly, if a field is not accessed via reference of $\mathtt{v}$ (denoted by $\alpha_2 = \circledast$), we can use bivariant subtyping. That is, we support $\mathtt{c}\langle\alpha_1 t_1\rangle\mathtt{<:}\mathtt{c}\langle\circledast t_2\rangle$ for any $t_1\mathtt{,}t_2$.

Variant parametric types give a much richer subtyping hierarchy than parameterized types do. Figure 5.1 illustrates some variant types for `Cell` objects and their places in the subtyping hierarchy. Note that $\rightarrow$ denotes a subtyping relation in the graph. Also, $\mathtt{Cell}\langle\circledast\mathtt{t}\rangle$, $\mathtt{Cell}\langle\oplus\mathtt{Object}\rangle$ and $\mathtt{Cell}\langle\ominus\bot\rangle$ are equivalent to each other while $\mathtt{Cell}\langle\odot\mathtt{Num}\rangle$, $\mathtt{Cell}\langle\odot\mathtt{Float}\rangle$ and $\mathtt{Cell}\langle\odot\mathtt{Int}\rangle$ are unrelated. Note that $\bot$ denotes the type of *null* values which can be assigned into any class type. However, each $\mathtt{Cell}\langle\odot\mathtt{t}\rangle$ is a subtype of both $\mathtt{Cell}\langle\oplus\mathtt{t}\rangle$ and $\mathtt{Cell}\langle\ominus\mathtt{t}\rangle$. Also, types of the form $\mathtt{Cell}\langle\oplus\mathtt{t}\rangle$ and $\mathtt{Cell}\langle\ominus\mathtt{t}\rangle$ have a subtyping hierarchy based on covariance and contravariance, respectively.

The benefits of variant parametric typing have been known for some time. However, early proposals have attached access rights to the fields of each class declaration. This mechanism is known as *declaration-site variance* and is shown in the following example:

```
class DSCell⟨A⟩ {
 ⊕A fst;
 A getFst() { return fst; }
 void setFst(A x) { fst=x; } }
```

The field `fst` is declared read only using the variance $\oplus$. Consequently, the method `setFst` cannot be invoked. Using the concept of structured virtual type, Thorup and Torgersen [186] were the first to link access rights and covariant subtyping to the fields of each *use of a class* rather than the class declaration itself. This *use-site variance* mechanism is much more flexible than previous mechanisms based on *declaration-site variance*. In the following example, the access to the field `fst` is governed by the variance variable α. A reference of type `USCell⟨⊕Int⟩` allows read-only access, while a reference of type `USCell⟨⊙Int⟩` allows read-write access to the field `fst`.

```
class USCell⟨αB⟩ {
 αB fst;
 B getFst() { return fst; }
 void setFst(B x) { fst=x; } }
```

Later, Igarashi and Viroli extended this concept to support contra- and bi-variance [103]. They formalized the variant parametric type system by mapping it into a corresponding *existential type* system [33, 123, 134]. A recent proposal by Sun Microsystems for generics in Java 1.5 [193] supports *wildcard type* based on an improvement of Igarashi and Viroli's variant parametric type system, but it is still viewed as a special case of the existential type system with subtyping. However, a more general version of existential type system, called System $F_{\leq}$, has undecidable subtyping [152], while the decidability of Igarashi and Viroli's variant parametric type system, remains an open problem [104, 149].

5.1.1 Motivation and Goal

While the mechanism of variant parametric type system (VPT) has now been validated in a full-scale Java implementation, its wide-scale adoption by the programming community is likely to take some time due to the need to provide type and variance annotations manually. These annotations are non-trivial and hard to understand by the programmer. The current model of variant parametric types is based on existential types which are not so intuitive for Java regular programmers.

In this context *our goal was to develop a novel flow-based approach for variant parametric types*. The main goal of genericity is to support highly reusable software components. To allow this to happen in a type-safe way, we should strive to provide type descriptions that are concise, understandable, general and accurate. Specifically, each well-typed generic program should be accurately identified where possible. As a side benefit, the type system should be able to track type information in a precise manner, allowing redundant cast operations to be eliminated where possible. We believe that flow analysis is more easy to understand by the programmers and it can also improve the precision of typechecking. In addition type checking should be scalable to larger programs and should support separate compilation.

5.1.2 Solution and Contributions

We propose a new approach for the variant parametric type system that is based on the mechanism of flow analysis. Our flow analysis captures value flows via subtyping constraints. A major benefit of this approach is the considerable knowledge in flow analysis that has been accumulated in the recent past [142, 157, 195, 96, 97, 183, 143]. In particular, to support modular type-checking, we require non-structural subtype entailment of the form $\forall\overline{v}(C_1 \Longrightarrow \exists\overline{w}C_2)$, where C_1, C_2 are subtyping constraints while $\overline{v}, \overline{w}$ are sets of type variables. These constraints are non-structural as we use $\perp <: t <: \textit{Object}$, to support the object-oriented class inheritance mechanism. While the decidability of non-structural subtype entailment remains an open problem, there exist sound approximations that use constraint simplification and induction techniques [157, 195]. Our work is built on top of sound but practical solutions to subtyping (flow) constraints.

In summary, this chapter makes a number of technical contributions explained below:

- **Flow-based Approach**: Our framework is based on flow analysis which can concisely and intuitively capture flow of values on a per method basis (Section 5.3). We use variance annotations primarily to predict the flows of values, and not for access control. We also provide special considerations for two type values. A value of `Object` type can always flow out from any location, while a null value of $\perp$ type can always flow into any location. In contrast, the other approaches [103, 193] view variant parametric type system as a special case of the existential type system with subtyping.

- **Intersection Types**: We augment our generic type system with *intersection types* to help capture information flow more accurately. An intersection type `t1&t2` denotes a type with both the properties of `t1` and `t2`. Such types are important for languages with multiple inheritance (such as Java via its interface mechanism), and can accurately capture the flow of objects with their expected field accesses. Java 1.5 provides a restricted support for the intersection types, as they can only be used as upper bounds of method type parameters [128].

- **Modular Type Checking**: Each method is specified with a flow constraint (and variant parametric types) that is used to predict the value flows that may occur in the method's body. We verify each method separately to ensure that the predicted accesses, flow constraint and variant parametric typings are efficiently and safely checked (Section 5.5). In contrast, the previous approaches [103, 193] use a type-checking per class approach rather than a per method approach.

- **Casting and Cast Capture**: A general casting mechanism allows us to define a novel cast capture that uses a reflection technique to deal with an unknown type (Section 5.7). Cast capture has helped improve the generic implementation of several JDK 1.5 libraries. In contrast, Java 1.5 restricts the downcast mechanism to the outer type constructor [128].

- **Soundness**: We have proven the soundness of our constraint-based type checker.

- **Experimental Validation**: We have implemented a prototype of our variant parametric type checker and have run the experiments on a suite of Java libraries and some large size Java applications (Section 5.8.2). On average, we are able to eliminate 87.9% of the casts from non-generic Java 1.4 application code, that means 12.9% more casts than wildcard-generic Java 1.5 application code.

5.1.3 Outline

This chapter is mainly based on a paper published in [39]. The remainder of this chapter is organized as follows. Section 5.2 introduces the main techniques used in our approach. Section 5.3

presents the flow-based interpretation of the variance, variant parametric subtyping rules and the annotations on our object-oriented core language. Section 5.4 presents the class parameterization, the mechanism of type promotion and the class invariant. In Section 5.5 we formulate our type checking rules and the modular flow verification. Section 5.6 proves the soundness of our variant parametric type system. Section 5.7 discusses our general casting and the cast capture mechanism. Section 5.8 presents the experimental results obtained using our prototype. Section 5.9 presents other extensions, while Section 5.10 discusses related work.

5.2 Main Techniques

In this section, we examine the key aspects for which our approach based on flow analysis makes improvements over existing approaches based on existential types. Some of these improvements may not be peculiar to the flow-based approach, but they were gradually developed starting from a different view point.

5.2.1 Intersection Types

Parametric type systems use number of cast operations eliminated as a measure of accuracy [54, 72]. As it turns out, there may be competing decisions on what types to use for certain cast operations to be eliminated. The following example from [54] illustrates:

```
class B1 extends A implements I { ··· }
class B2 extends A implements I { ··· }
void foo(Boolean b) {
 Cell cb1 = new Cell(new B1());
 Cell cb2 = new Cell(new B2());
 Cell c = b ? cb1 : cb2;
 A a = (A) c.get();
 I i = (I) c.get();
 B1 b1 = (B1) cb1.get();
 B2 b2 = (B2) cb2.get();  }
```

This program contains four cast operations. With the help of parametric types, Donovan et al. [54] suggested three sets of possible types, each with a different subset of casts eliminated, as summarized below:

Types of Variables			Casts Eliminated			
cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨B1⟩	Cell⟨B2⟩	Cell			√	√
Cell⟨A⟩	Cell⟨A⟩	Cell⟨A⟩	√			
Cell⟨I⟩	Cell⟨I⟩	Cell⟨I⟩		√		

Note that `Cell` denotes a raw type where nothing is known of its components. Hence, only `Object` values are statically retrievable from it. Raw type was originally proposed in [24] for backwards compatibility, and it is the basis for generic typing through inclusion polymorphism. However, none of the three proposed solutions are able to eliminate all four casts. This indicates that parametric typing is not expressive enough to capture generic type for such programs. There are two possible improvements. First, note that the fields of `cb1, cb2` and `c` are subject to read-only accesses, and not modified in the program fragment. We can therefore provide covariant annotations to the fields of these variables, and obtain two possible outcomes, each with three casts eliminated:

cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕A⟩	√		√	√
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕I⟩		√	√	√

Second, both classes `B1` and `B2` have supertypes `A` and `I` in common. To exploit this, we can use an intersection type parameter in `Cell`⟨⊕`(A&I)`⟩ to describe the variable `c`. In a lattice of type values, an intersection type `A&I` essentially defines the greatest lower bound of `A` and `I`. With this, all four casts can now be eliminated in our new solution to genericity, as shown below:

cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕A&I⟩	√	√	√	√

Note that the above example cannot be coded in Java 1.5 syntax. Java 1.5 does not allow the use of intersection types for local variable declaration, field declaration or method argument/return types. Intersection types can be used only as upper bounds for a method type parameter.

5.2.2 Modular Flow Specification

Another important principle for better genericity is that type description should be designed in a *modular* fashion (on a per method basis). Type annotations appearing in the method header should depend only on the method body while each call site should be a specific instance of the method's type declaration. This principle is important for efficient type checking and ease of type annotation. Specifically, for each instance method, we provide the following method declaration:

t `|` t_0 `mn(`t_1 v_1`,...,`t_n v_n`) where` ψ `{...}`

A separate annotation "t `|`" is added at the beginning of each method's declaration to capture the variance of the implicit `this` parameter. This separate annotation (omitted in previous works, such as [103, 193]) allows us to capture the behaviour of each method, independent of its class declaration. Note that ψ captures the expected value flows of each method's body in terms of type of the parameters ($t_1, .., t_n$), result (t_0), and receiver (t). We support modular type checking by localizing type variables which are not present in the type of parameters, result and receiver. A previous approach [103] relies on the existential open/close mechanism for the receiver parameter to determine if the receiver parameter is of suitable variance while other parameters are checked via subtyping. In contrast, we achieve uniform treatment for all parameters.

To illustrate the modular type annotation mechanism, consider three method declarations for the `Pair` class:

```
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
 B snd;
 Pair⟨⊛,⊕Y⟩ | Y getSnd()
   {return this.snd;}
 Pair⟨⊛,⊖Y⟩ | void setSnd(Y v)
   {this.snd=v;}
 Pair⟨⊛,⊛⟩&W | Pair⟨⊙W,⊙W⟩ dup()
   {return new Pair⟨W,W⟩(this,this);} }
```

First, note that `getSnd` will read the second field while `setSnd` will write to it. Because of these effects, we may apply covariant ($\oplus$) and contravariant ($\ominus$) subtypings to the second component of the `Pair` object for `getSnd` and `setSnd`, respectively. Second, bivariant ($\circledast$) subtyping is allowed for the unaccessed component of the `Pair` object for both methods. As a shorthand, we may write $\circledast$ to denote $\circledast t$ since all bivariant types are equivalent. Note that `Y` from `getSnd` and `Y` from `setSnd` denote different type variables treated independently by our modular type checker.

The third method is an interesting application of intersection type. The method itself does not access the fields of the `this` parameter, which escapes into the two fields of the method's `Pair` result. To capture this value flow, we declare an intersection type $\texttt{Pair}\langle\circledast,\circledast\rangle\texttt{\&W}$ for the `this` parameter. The type $\texttt{Pair}\langle\circledast,\circledast\rangle$ is to acknowledge that we have a `Pair` object whose fields are not accessed by the current `dup` method. A type variable `W` helps indicate that this parameter will escape into the fields of the result with type $\texttt{Pair}\langle\odot\texttt{W},\odot\texttt{W}\rangle$. This flow allows the variant type of `W` to flow into the two fields of the output `Pair`. Hence, for a given receiver of type t, we have $t\texttt{<:Pair}\langle\circledast,\circledast\rangle$ and $t\texttt{<:W}$. Possible candidates for the type t are $\texttt{Pair}\langle\oplus\texttt{X},\oplus\texttt{Y}\rangle$ or $\texttt{Pair}\langle\oplus\texttt{X},\ominus\texttt{Y}\rangle$, etc. In contrast, if we use the following type suggested in [103]:

```
Pair⟨⊙X,⊙Y⟩ | Pair⟨⊙Pair⟨⊙X,⊙Y⟩,⊙Pair⟨⊙X,⊙Y⟩⟩ dup()
```

we require $t\texttt{=Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle$ or $t\texttt{=}\bot$, which restricts the possible uses of the method. One way to improve this situation is to duplicate the `dup` method for different scenarios, as shown below:

```
Pair⟨⊕X,⊕Y⟩ | Pair⟨⊙Pair⟨⊕X,⊕Y⟩,⊙Pair⟨⊕X,⊕Y⟩⟩ dup()
Pair⟨⊕X,⊖Y⟩ | Pair⟨⊙Pair⟨⊕X,⊖Y⟩,⊙Pair⟨⊕X,⊖Y⟩⟩ dup()
Pair⟨⊙X,⊖Y⟩ | Pair⟨⊙Pair⟨⊙X,⊖Y⟩,⊙Pair⟨⊙X,⊖Y⟩⟩ dup()
```

However, such duplications go against the goal of genericity. On the other hand, our solution with intersection types can improve genericity by allowing value flows to be accurately captured.

5.2.3 Avoiding F-Bounds where Possible

One feature that adds to the expressivity of bounded existential type is the use of F-bounds [30] which effectively capture recursive constraints of the form $T{<:}C\langle..,T,..\rangle$ where T is a type variable and C is a class name. While the designers of Java 1.5 consider this feature to be significant and useful [193], it is also a source of complication as reported recently in [120]. In particular, F-bound together with existential type is a source of undecidability for System $F_{\leq}$ which caused an earlier implementation of Java 1.5 to fail in accepting some programs with F-bounds that were actually type-safe (as first reported in [120]). Subsequent improvements in Java 1.6 have removed the reported errors, but the decidability of its type system remains an open problem.

While the flow-based approach that we advocate also supports recursive flow constraints (if the inductive mechanism of [157, 195] is used), our pragmatic philosophy is to avoid F-bounds whenever it is possible to do so.

As an example of F-bound, consider the following definition of the `Comparable` interface for Java 1.5:

```
  interface Comparable⟨T⟩ { int compareTo(T o);  }
```

Here, class parameter `T` is being used to capture the parameter of the method `compareTo`. As this parameter is required to be a subtype of `Comparable` itself, F-bound of the form `T<:Comparable⟨⊖T⟩` is usually needed when `Comparable` is used, as shown in the next example:

```
class Collections {
⟨T extends Comparable⟨? super T⟩⟩
   static T max(Collection⟨? extends T⟩ cl) {···}  }
```

In our flow-based approach, the current philosophy is to capture the value flows of each method independently. Hence, we have chosen to capture the value flow and subtyping relation directly for each method instead, as shown below for our definition of `Comparable`:

```
interface Comparable⟨A⟩ { Comparable⟨⊙T⟩ | int compareTo(T o); }
```

Based on this definition, we can write the `max` method, as follows:

```
class Collections {
  static T max(Collection⟨⊕T⟩ cl) where T<:Comparable⟨⊖T⟩ {···}}
```

This alternative is equivalent to the earlier Java 1.5 definition.

We also support a simpler way, to express `Comparable` interface, as follows:

```
interface Comparable {
  S & Comparable | int compareTo(T o) where T<:Comparable∧T<:S;}
```

The use of this definition does not require any F-bound, but it is more restrictive than Java 1.5 definition of `Comparable` interface.

Another potential use of F-bound occurs for recursive fields of class declarations. An example is the following recursive `List` class:

```
  class List⟨A,B⟩ extends Object where B<:List⟨A,B⟩ {
   A val;
   B next; ...  }
```

This solution uses an F-bound `B<:List⟨A,B⟩` that makes constraint solving more complex [157]. However, in our system we may choose to avoid the recursive constraint from the invariant of the class `List` by leaving the recursive `next` field with an incomplete variance $\oslash$, as follows:

```
  class List⟨A⟩ extends Object {
   A val;
   ⊘List⟨A⟩ next; ...  }
```

The variance of the `next` field is incomplete at its declaration site and can be promoted to either $\odot$ or $\oplus$, depending on how its underlying type parameter `List⟨A⟩` is being instantiated at the use site. This type promotion process is elaborated later in Section 5.4.1, and can be used to avoid F-bound, where possible.

5.2.4 Avoiding Existential Types Always

It has been generally acknowledged that existential type is useful for describing data types whose implementation details can be made abstract. This aspect is closely related to the use of bivariant type $\circledast t$ where the underlying type t is unknown and may be assumed to be of any type. While no-access is one way to enforce bivariant type, it is also possible to use the open/close mechanism of an existential type system to describe situations where implementation details can be made abstract. A typical example is the `copy` operation on two elements of a vector that was highlighted in [104], and reproduced below:

```
void copy(Vector⟨⊛⟩ x, int i, int j) {
   open x as [Y,y] in y.setElementAt(y.elementAt(i),j) }
```

The above code opens the bivariant type of `x` as an object bound to variable `y` with an abstract type `Y`. As all elements of each vector are of the same `Y` type, we may safely copy a value from one position of the vector into another position, without knowing the actual underlying type. The close correspondence between existential type and bivariant type is a primary reason why

Igarashi and Viroli [104] considered existential type system as the underlying model for their variant parametric type system.

However, the designers of Java 1.5 considered the open/close mechanism of existential type system to be somewhat restrictive [194]. They have therefore proposed a relaxation to open each expression as an existential type by associating it with a global type variable *without* a corresponding close operation. This use is similar to the flow-based approach where each parameter (or local variable) is regarded as a location where values may flow in and/or out. Nevertheless, in the context of existential type system, such relaxation might possibly be unsound since each existential type may in fact correspond to contradicting type values. This is possibly why correctness proof is yet to be completed (as of [194]), even though a full-scale implementation for wildcard type system has already been released for public use.

Furthermore, Java 1.5 relies on a polymorphic (generic) type system for selected methods to capture situations where invariant type appears necessary, as shown by the following example:

```
⟨T⟩ void docopy(Vector⟨T⟩ x, int i, int j) {
   T tmp = x.elementAt(i); x.setElementAt(tmp,j); }
```

Through a wildcard capture mechanism, it is possible to provide a method with bivariant parameter, as shown below:

```
  void copy(Vector⟨?⟩ x, int i, int j) { docopy(x,i,j); }
```

Note that wildcard type of `x` has been captured by the global `T` type variable. Again, the open/close mechanism is averted, even though the underlying system is still based on bounded existential type system.

Our current philosophy is to avoid existential type system altogether. To capture the effect of an unknown abstract type, we have introduced a casting mechanism that is able to capture the underlying type of an object via a fresh type variable. We refer to this as a *cast capture* technique which is elaborated in more details in Section 5.7. The same `copy` method can be re-written with a casting of the `x` parameter from bivariant type `Vector⟨⊛⟩` to an invariant type `(Vector⟨⊙T⟩)`. In the process, `T` is used to capture the unknown type, as shown below:

```
void copy(Vector⟨⊛⟩ x, int i, int j) {
  Vector⟨⊙S⟩ w;
```

```
    {w = (Vector⟨⊙T⟩) x; w.setElementAt(w.elementAt(i),j) }
}
```

While this cast capture construct may look like a syntactic sugar for the open/close mechanism, we stress that it is part of a more general mechanism that can take an arbitrary type as source (instead of a bivariant type) for casting into another arbitrary type as target (instead of an invariant type). A cast for a c_1-object into an invariant type of the form $c_2\langle(\odot t)^*\rangle$ where $c_1{<:}c_2$ is always safe since every object is built using an invariant type. Furthermore, cast-capture is a runtime mechanism while open-close is a type-related operation to expose an obtained type at compile-time. Our cast capture mechanism using reflection is more general as it can capture type values at runtime, and also support a mix of cast capture and cast testing. In our formulation of variant parametric type system, the flow-based approach with casting has therefore avoided the need for existential type systems altogether.

Some readers may contend that the casting mechanism is the prerogative of programmers and may be too burdensome to write. While this is so, we believe that there is still scope for automatic insertion of safe casts to invariant type (in a spirit similar to automatic type coercion) that is consistent with each user program.

5.3 Variance via Flow Analysis

5.3.1 An Example

A central feature of our proposed approach is the focus on flow analysis. Variance annotations are used to support the analysis of value flows to capture more accurate generic types, whereby suitable field subtypings (covariance and contravariance) are facilitated where possible. We highlight the expressiveness of variant parametric types through some more examples in Figure 5.2.

Apart from a generic `Vector⟨A⟩` class declaration, we provide a number of static methods to highlight how flow analysis may assist in the formulation of generic types. In the `copyVec` method, the elements from a first vector `Vector⟨⊕X⟩` are copied into a second vector `Vector⟨⊖Y⟩`, while a constraint `X<:Y` captures the direction of the value flow.

Method `copyNestVec` copies from a nested vector of type `Vector⟨⊕Vector⟨⊕X⟩⟩` into a second vector `Vector⟨⊖Y⟩` with flow constraint `X<:Y`. This code remains highly generic as it uses covariant and contravariant subtypings. The next example shows how we use a special type

```
class Vector⟨A⟩ extends Collection⟨A⟩ {
 Vector⟨⊛⟩ | int size() {...}
 Vector⟨⊕X⟩ | X elementAt(int i) {...}
 Vector⟨⊖X⟩ | void setElementAt(X v, int i) {...}
}

void copyVec(Vector⟨⊕X⟩ v, Vector⟨⊖Y⟩ w, int start) where X<:Y {
   for(int i=0;i<v.size()&&i+start<w.size();i++)
    w.setElementAt(v.elementAt(i),i+start);
}

void copyNestVec(Vector⟨⊕Vector⟨⊕X⟩⟩ v, Vector⟨⊖Y⟩ w)where X<:Y{
   int pos=0;
   for(int i=0; i<v.size();i++) {
    Vector⟨⊕Z⟩ s=v.elementAt(i);
    if (pos+s.size()<w.size())
     {copyVec(s,w,pos); pos +=s.size(); }
}}

void clearVec(Vector⟨⊖⊥⟩ v) {
   for(int i=0; i<v.size();i++)
    v.setElementAt(null,i);
}

Vector⟨⊙Z⟩ merge(Vector⟨⊕X⟩ v, Vector⟨⊕Y⟩ w) where X<:Z∧Y<:Z
 {...}
Vector⟨⊙Pair⟨⊙X,⊙Z⟩⟩ join(Vector⟨⊕Pair⟨⊕X,⊕Y⟩⟩ v,
   Vector⟨⊕Pair⟨⊕Y,⊕Z⟩⟩ w)
 {...}

void swap(Pair⟨⊙X,⊙Y⟩ p) where X<:Y∧Y<:X {
   T t=p.fst; p.fst=p.snd; p.snd=t;
}
```

Figure 5.2: Examples with Variant Parametric Types

$\perp$ to indicate that null values will be written into the vector. Given that `Vector`$\langle\ominus\perp\rangle$ is high up in the class hierarchy, this method is rather generic as we can supply *any* vector as its argument.

We also provide method headers for `merge` and `join`. From the type annotation of `merge`, we can tell that values from the first two vectors are retrieved, and then they flow into a new result vector. For the `join` method, we retrieve values from the two vectors `Vector`$\langle\oplus$`Pair`$\langle\oplus$`X`, $\oplus$`Y`$\rangle\rangle$ and `Vector`$\langle\oplus$`Pair`$\langle\oplus$`Y`, $\oplus$`Z`$\rangle\rangle$ before building a new vector `Vector`$\langle\odot$`Pair`$\langle\odot$`X`, $\odot$`Z`$\rangle\rangle$ that is joined on the `Y` type. The result's invariant type offers a strong post-condition with read/write capability.

For the `swap` method, the two fields of a `Pair` object are swapped. Due to both reading and writing, we require the invariant type `Pair`$\langle\odot$`X`, $\odot$`Y`$\rangle$ and the expected value flow: `X<:Y`$\wedge$`Y<:X`. Based on the flows from the three assignments of the swap body, we may obtain the following constraints: $\odot$`X<:`$\oplus$`T`, $\odot$`Y<:`$\oplus$`X` and $\odot$`T<:`$\oplus$`Y`, where `T` is a local type variable (using type rules in Section 5.5.1). These constraints are simplified to obtain the following collected flow for the method body: `X<:T`$\wedge$`Y<:X`$\wedge$`T<:Y`. The `swap` method type checks as the expected flow implies the collected flow: $\forall$`X,Y.(X<:Y`$\wedge$`Y<:X` $\Longrightarrow$ $\exists$`T.(X<:T`$\wedge$`Y<:X`$\wedge$`T<:Y))`. Note that the local type variable `T` is existentially quantified, while type variables `X,Y` from method parameters are universally quantified.

5.3.2 Improved Variant Parametric Subtyping

Variant parametric type τ consists of a variance α and a type t. Its grammar is introduced in Figure 5.4. We use variance annotations $\odot, \oplus, \ominus$ and $\circledast$, which correspond to read-write access, read-only access, write-only access, and no-access, respectively. These annotations are ordered by the following relation that is denoted by $<:_\alpha$ but abbreviated to $<:$ below:

$$\odot{<:}\oplus \quad \odot{<:}\ominus \quad \oplus{<:}\circledast \quad \ominus{<:}\circledast$$

$$\frac{\alpha_1{<:}\alpha_2 \quad \alpha_2{<:}\alpha_3}{\alpha_1{<:}\alpha_3} \qquad \alpha{<:}\alpha$$

A type t is either a type variable v_t, a variant parametric class $c\langle\tau_1,\ldots,\tau_n\rangle$, the bottom type $\perp$ or an intersection type $t\&t$. The bottom type is used to hold the `null` value.

We allow finite intersections of types through the type operator $\&$. Semantically, $t_1\&t_2$ denotes the set of objects satisfying the interface specification of both t_1 and t_2. In a lattice of type values with partial order defined by class inheritance (through `extends`) and interface

$\tau-$**subtyping**

$$\frac{\alpha_1{<:}\ominus \quad \vdash t_2{<:}t_1{\Rightarrow}\psi}{\vdash \alpha_1 t_1{<:}\ominus t_2{\Rightarrow}\psi} \qquad \frac{\alpha_1{<:}\oplus \quad \vdash t_1{<:}t_2{\Rightarrow}\psi}{\vdash \alpha_1 t_1{<:}\oplus t_2{\Rightarrow}\psi}$$

$$\frac{\vdash t_1{\equiv}t_2{\Rightarrow}\psi}{\vdash \odot t_1{<:}\odot t_2{\Rightarrow}\psi} \qquad \vdash \tau{<:}\circledast t{\Rightarrow}\textit{true}$$

$$\frac{\neg(\alpha_1{<:}\oplus)}{\vdash \alpha_1 t_1{<:}\oplus \textit{Object}{\Rightarrow}\textit{true}} \qquad \frac{\neg(\alpha_1{<:}\ominus)}{\vdash \alpha_1 t_1{<:}\ominus \bot{\Rightarrow}\textit{true}}$$

$\mathbf{t}-$**subtyping**

$$\vdash\bot{<:}t{\Rightarrow}\textit{true} \quad \vdash t{<:}\textit{Object}{\Rightarrow}\textit{true} \quad \vdash t{<:}t{\Rightarrow}\textit{true}$$

$$\frac{\vdash \tau_i{<:}\tau_i'{\Rightarrow}\psi_i,\ i=1..n}{\vdash c\langle\tau_i\rangle_{i=1}^n{<:}c\langle\tau_i'\rangle_{i=1}^n{\Rightarrow}\bigwedge_{i=1}^n \psi_i} \qquad \frac{\vdash t_1{<:}t_2{\Rightarrow}\psi_1 \ \ \vdash t_2{<:}t_3{\Rightarrow}\psi_2}{\vdash t_1{<:}t_3{\Rightarrow}\psi_1{\wedge}\psi_2}$$

$$\frac{\begin{array}{c}\textbf{class } c_1\langle V_i\rangle_{i=1}^m \textbf{ extends } ...c_2\langle\tau_i'\rangle_{i=1}^n...\ \ \rho{=}[V_i{\mapsto}\tau_i]_{i=1}^m \\ c_2\langle\rho\tau_i'\rangle_{i=1}^n{\Rightarrow}_p\ c_2\langle\rho\tau_i''\rangle_{i=1}^n\end{array}}{\vdash c_1\langle\tau_i\rangle_{i=1}^m{<:}c_2\langle\rho\tau_i''\rangle_{i=1}^n{\Rightarrow}\textit{true}}$$

$$\frac{\begin{array}{c}\textbf{class } c_1\langle V_i\rangle_{i=1}^m \ ...\textbf{implements } ...c_2\langle\tau_i'\rangle_{i=1}^n...\ \ \rho{=}[V_i{\mapsto}\tau_i]_{i=1}^m \\ c_2\langle\rho\tau_i'\rangle_{i=1}^n{\Rightarrow}_p\ c_2\langle\rho\tau_i''\rangle_{i=1}^n\end{array}}{\vdash c_1\langle\tau_i\rangle_{i=1}^m{<:}c_2\langle\rho\tau_i''\rangle_{i=1}^n{\Rightarrow}\textit{true}}$$

$$\frac{\vdash t{<:}t_1{\Rightarrow}\psi_1 \quad \vdash t{<:}t_2{\Rightarrow}\psi_2}{\vdash t{<:}(t_1\&t_2){\Rightarrow}\psi_1{\wedge}\psi_2} \qquad \frac{\vdash t_1{<:}t{\Rightarrow}\psi_1 \quad \vdash t_2{<:}t{\Rightarrow}\psi_2}{\vdash (t_1\&t_2){<:}t{\Rightarrow}\psi_1{\vee}\psi_2}$$

$$\frac{t_1{=}v_t{\vee}t_2{=}v_t}{\vdash t_1{<:}t_2{\Rightarrow}t_1{<:}t_2} \qquad \frac{\vdash t_1{<:}t_2{\Rightarrow}\psi_1 \quad \vdash t_2{<:}t_1{\Rightarrow}\psi_2}{\vdash t_1{\equiv}t_2{\Rightarrow}\psi_1{\wedge}\psi_2}$$

Figure 5.3: Variant Parametric Subtyping

mechanism (through `implements`), $t_1\&t_2$ defines the greatest lower bound of t_1 and t_2. Our intersection types are similar to the compound types proposed in [27]. Specifically, they can be of the form $[t_1\&]t_2\&...\&t_n[\&W]$, where t_1 is a class, $t_2, ..., t_n$ are interfaces, and W is a type variable.

In our system, variant parametric types are used to support flow analysis rather than access controls. As we focus on value flows at each method boundary, we apply variance annotations primarily to fields. The outermost variance of local variables is always $\odot$. For fields, variance annotations are used to support covariant or contravariant subtyping where possible.

The subtyping relations are denoted by $<:_\tau$ and $<:_t$, both abbreviated to $<:$ as follows:

$$\vdash \tau_1{<:}\tau_2{\Rightarrow}\psi \qquad \vdash t_1{<:}t_2{\Rightarrow}\psi$$

The resulting constraints ψ (see Figure 5.4 for their grammar) are kept in a disjunctive normal form. Instead of proving each subtyping directly, we collect a set of subtyping constraints ψ via $\tau{-}subtyping$ and $t{-}subtyping$ in Figure 5.3.

The first four τ-subtyping rules support contravariance, covariance, invariance and bivariance, respectively. The invariant case generates a constraint from the semantical equivalence of the two types ($t_1 \equiv t_2$). Unlike the subtyping rule of Igarashi and Viroli [103], our improved mechanism handles two special values in the subtyping hierarchy, namely $\perp$ (for type of null) and *Object* (for top of class hierarchy). These two types are special in that it is always safe to write a null (of $\perp$ type) into any location (even if it has been marked for read-only access), and it is safe to read an *Object* value from any location (even if it has been marked for write-only access). We may also cast any type τ to either $\oplus$*Object* or $\ominus\perp$ as it is always safe to read an object or write a null value. This mechanism is implemented by the last two τ-subtyping rules.

In the second part of Figure 5.3, the first two t-subtyping rules handle the bottom and top of the hierarchy. Subtyping between types of the same class is decomposed structurally by the fourth rule. The next two rules describe transitivity and the class inheritance relation. The class inheritance rule uses type promotion mechanism that is described later in Section 5.4 Intersection types satisfy the subtyping relations as in [151]. Subtyping relations that contain type variables are not simplified further and preserved in the resulting constraint. Semantic equality ($t_1{\equiv}t_2$) is given by the last t-subtyping rule. In summary, from the subtyping relations between types, we generate a set of subtyping constraints (on type variables). Note that in the following sections, we will use $\tau_1{<:}\tau_2$ as an abbreviation for ψ, where $\vdash \tau_1{<:}\tau_2{\Rightarrow}\psi$.

5.3.3 Variant Parametric Core-Java Language

We introduce a core language to ease the formulation of static and dynamic semantics. This language can be viewed as a result of translation from full Java language prior to type checking. For ease of presentation, we omit features that are related to static methods, exception handling (exceptions can not have generic types), concurrency and inner classes. (Our implementation handles all features of the Java language.)

Our core language is named Variant Parametric Core-Java, and summarised in Figure 5.4. We use the suffix notation y^* to denote a list of (zero or more) distinct syntactic terms that are suitably separated. Both class and interface declarations are supported using the same syntactic grammar term *def*. The interface definitions do not have fields, and are defined using

Programs

$$
\begin{array}{rl}
P ::= & \mathit{def}^* \\
\mathit{def} ::= & \mathbf{class}\ c\langle V^*\rangle\ \mathbf{extends}\ gc_1\ \mathbf{implements}\ gc_2..gc_n\ \mathbf{where}\ \psi_{inv} \\
& \{(\pi\, f)^*\ \mathit{meth}^*\} \\
gc ::= & c\langle \pi_1, .., \pi_n\rangle \\
\mathit{meth} ::= & t \mid t\ \mathit{mn}((t\ v)^*)\langle v_t^*\rangle\ \mathbf{where}\ \psi\ \{e\} \\
w ::= & v \mid v.f \\
e ::= & \mathbf{null} \mid w \mid w = e \mid \{t\ v = e_1;\ e_2\} \mid e_1\ \mathbf{;}\ e_2 \\
& \mid \mathbf{new}\ c\langle t^*\rangle(v^*) \mid \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
& \mid \mathbf{while}\ v\ e \mid v_0.\mathit{mn}(v^*)\langle t^*\rangle \\
& \mid (t)v \mid \{v_1 = (t)v\ ;\ e\}
\end{array}
$$

Variant Parametric Types

$$
\begin{array}{rl}
\tau ::= & \alpha t \\
t ::= & v_t \mid c\langle \tau_1, .., \tau_n\rangle \mid t\&t \mid \bot \\
\alpha ::= & \odot \mid \oplus \mid \ominus \mid \circledast
\end{array}
$$

Incomplete Variant Parametric Types

$$
\begin{array}{rl}
\pi ::= & V \mid \oslash\, s \\
s ::= & c\langle \pi_1, .., \pi_n\rangle \mid s\&s \mid \bot
\end{array}
$$

Subtyping constraints

$$
\psi ::= t_1 <: t_2 \mid \psi \wedge \psi \mid \mathit{true}
$$

Class Invariant

$$
\psi_{inv} ::= V <:_i c\langle \tau^*\rangle \mid c\langle \tau^*\rangle <:_i V \mid \psi_{inv} \wedge \psi_{inv} \mid \mathit{true}
$$

Figure 5.4: Syntax of Variant Parametric Core-Java. Primitive types are discussed in Section 5.9, while exceptions can not have generics types.

abstract methods (without body). Furthermore, while we support multiple inheritance, it is of the same restricted kind as that supported by the Java language. Each class may extend from only a single superclass but may implement multiple interfaces. In our language, the declaration **class** $c\langle V^*\rangle$ **extends** gc_1 **implements** $gc_2..gc_n$ assumes that gc_1 is a class while $gc_2..gc_n$ are interfaces. Each class declaration captures a class invariant ψ_{inv} that is expected to hold for all newly constructed objects of the class. This is being used to specify suitable class lower and/or upper bounds for type variables. Since our system is based on use-site variance, the class fields types and the arguments of class inheritance have incomplete variance at declaration-site (denoted by π and V). Section 5.4 describes the annotations of class declarations with incomplete variant parametric types.

Each method declaration *meth* contains a constraint ψ which captures the expected value flows for its type variables. It also specifies method type parameters $\langle v_t^* \rangle$ in order to support modular type checking. This set of type variables is automatically inserted by our compiler.

We use an expression-oriented language, where method body is denoted by e. Local variable declaration is supported by block structure of the form: $\{t\ v = e_1; e_2\}$, with e_2 denoting its result. Each object is always built with an invariant type $c\langle \odot t^* \rangle$ via the construct **new** $c\langle t^* \rangle(v^*)$. Our core language also supports a full casting mechanism via $(t)v$, where t can be an arbitrary variant parametric type. In addition, we support a novel cast capture mechanism via $\{v_1 = (t)v\ ;\ e\}$, where t is an invariant type with unknown type variables that may be captured at runtime and used in e. These special features will be described in more detail in Section 5.7.

For simplicity of presentation, our core language represents primitive types (such as `void`, `bool`) by their corresponding classes (such as `Void`, `Bool`). In our implementation, we handle primitive types directly, as elaborated in Section 5.9. For soundness reasons, we treat arrays in the same way as other classes (unlike Java 1.5, which assumes arrays to be covariant).

In the subtyping constraints, disjunction is supported internally as it may be generated by subtyping relation for intersection types.

5.4 Class Parameterisation and Inheritance

For class declarations, an important decision is which fields are to be parameterised and how the class inheritance mechanism is to be supported. In general, each class declaration should be written in the following manner:

class `c1`$\langle V_1 \mathinner{..} V_n \rangle$ **extends** `c2`$\langle \hat{\pi}_1 \mathinner{..} \hat{\pi}_s \rangle$ **where** ψ_{inv} $\{$
 π_1 `f1;`
 ...
 π_m `fm; ...` $\}$

where each $\{V_i\}_{i=1}^{n}$ originates either from the fields of the current class $\{\pi_i\}_{i=1}^{m}$ or from the arguments of its superclass, $\{\hat{\pi}_i\}_{i=1}^{s}$. $\{V_i\}_{i=1}^{n}$ are variables corresponding to types with variance. For instance, the following non-generic declarations of `Cell` and `Pair` classes:

```
class Cell {
 Object fst; ··· }
class Pair extends Cell {
 Object snd; ··· }
```

can be parameterized as:

```
class Cell⟨A⟩ {
  A fst; ··· }
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
  B snd; ··· }
```

The variance of the fields `fst` and `snd` is governed by the variables `A` and `B`. Given the type `Pair⟨⊕Int,⊖Int⟩`, the field `fst` is covariant and the field `snd` is contravariant.

5.4.1 Type Promotion

There are some situations where the variance of a class field cannot be specified at use site. In the following example, the variance of the field `sndP` does not have any correspondence in the class parameters `A`,`B`,`C` and remains unknown after instantiation of these parameters.

```
class Triple⟨A,B,C⟩ extends Cell⟨A⟩ {
  Pair⟨B,C⟩ sndP; ··· }
```

The compiler inserts a special variance marker $\oslash$ to represent the unknown variance of field `sndP`:

```
class Triple⟨A,B,C⟩ extends Cell⟨A⟩ {
  ⊘Pair⟨B,C⟩ sndP; ··· }
```

Note that the source program does not contain any variance markers. We use them to explain how incomplete (or unknown) variance of variant parametric types are computed to either $\oplus$ or $\odot$. This process is known as *type promotion* and can be used for incomplete variant parametric types from field declarations and arguments of class inheritance. The type promotion is defined using the following relations:

$$\rho \vdash \pi \Rightarrow_p \tau \qquad \rho \vdash s \Rightarrow_p t$$

where ρ is a substitution $[V \mapsto \tau]$ from class declaration parameters V to variant parametric types τ. The types π and s may contain unknown variance $\oslash$.

The rules are described in Figure 5.5. The second rule promotes the unknown variance $\oslash$ to either $\oplus$ or $\odot$ depending on the predicate *inv*(t) where t is the type obtained after substitution. Predicate *inv*(t) returns `true`, when all variances from `t` (if any) are $\odot$ and `false`

$$\frac{}{\rho \vdash V \Rightarrow_p \rho V} \qquad \frac{\rho \vdash s \Rightarrow_p t \quad \alpha = \texttt{if}\ inv(t)\ \texttt{then}\ \odot\ \texttt{else}\ \oplus}{\rho \vdash \oslash s \Rightarrow_p \alpha t}$$

$$\frac{\rho \vdash \pi_i \Rightarrow_p \tau_i \quad i = 1, n}{\rho \vdash c\langle \pi_1, ..\pi_n \rangle \Rightarrow_p c\langle \tau_1, ..\tau_n \rangle} \qquad \frac{\rho \vdash s_i \Rightarrow_p t_i \quad i = 1, 2}{\rho \vdash s_1 \& s_2 \Rightarrow_p t_1 \& t_2}$$

$$inv(\odot t) = true \quad \frac{\alpha = \oplus \mid \ominus \mid \circledast}{inv(\alpha t) = false} \quad inv(v_t) = true$$

$$inv(c\langle \tau_1, ..\tau_n \rangle) = \bigwedge_{i=1,n} inv(\tau_i) \qquad inv(t_1 \& t_2) = \bigwedge_{i=1,2} inv(t_i)$$

$$inv(c\langle\rangle) = true \qquad inv(\bot) = true$$

Figure 5.5: Type Promotion

otherwise. Given `Triple⟨⊕Int,⊕Int,⊕Int⟩`, the type of field `sndP` is computed as follows: $\rho \vdash \oslash$`Pair⟨B,C⟩`$\Rightarrow_p \oplus$`Pair⟨⊕Int,⊕Int⟩` where $\rho = [$`A` $\mapsto \oplus$`Int`,`B` $\mapsto \oplus$`Int`,`C` $\mapsto \oplus$`Int`$]$. As another example, given `Triple⟨⊕Int,⊙Int,⊙Int⟩`, the type of field `sndP` is computed as follows: $\rho \vdash \oslash$`Pair⟨B,C⟩`$\Rightarrow_p \odot$`Pair⟨⊙Int,⊙Int⟩` where $\rho = [$`A` $\mapsto \oplus$`Int`,`B` $\mapsto \odot$`Int`,`C` $\mapsto \odot$`Int`$]$.

Another application of type promotion is for recursive fields of a class. The recursive field `next` of the class `List` has an incomplete variance $\oslash$ as follows:

```
class List⟨A⟩ extends Object {
 A val;
 ⊘List⟨A⟩ next; ...  }
```

The variance of the field `next` is incomplete at its declaration site and can be promoted to either $\odot$ or $\oplus$, depending on how its underlying type parameter `List⟨A⟩` is being instantiated at the use site. For example, when `A` is instantiated to $\ominus$`X`, the variance of the `next` field will be promoted to $\oplus$ via $\rho \vdash \oslash$`List⟨A⟩` $\Rightarrow_p$ $\oplus$`List⟨⊖X⟩`, where $\rho = [$`A` $\mapsto \ominus$`X`$]$. On the other hand, if `A` is instantiated to $\odot$`X`, then $\rho = [$`A` $\mapsto \odot$`X`$]$ and the variance of the `next` fields is instantiated to $\odot$`X` as follows: $\rho \vdash \oslash$`List⟨A⟩` $\Rightarrow_p$ $\odot$`List⟨⊙X⟩`.

Our type promotion is a refinement of that proposed in [103]. First, we allow promotion to $\odot$ whenever possible while Igarashi and Viroli considered mainly the promotion of nested types with $\oplus$. Second, we consider type promotion for only field access and class inheritance where the outer variance is dependent on the variance of the underlying type. In contrast, Igarashi and

Viroli focused on the promotion of nested types of arguments/result for method declarations, which need not be handled in our approach as these types are fully specified in our method declarations.

5.4.2 Class Invariant

The class invariant ψ_{inv} is used to capture the lower and upper bounds for the parameterised fields of each newly created object of the class. These bounds are of the form $\bigwedge c_1\langle\tau*\rangle{<:_i}V$ $<:_i c_2\langle\tau*\rangle$. Class invariant may also support F-bounds when variable V occurs in the parameters of classes c_1 and c_2. If unspecified, the default lower and upper bounds are $\perp$ and `Object`, respectively. An upper bound invariant on a write-only field restricts the class of the object that can be written to the field to be subclasses of the bound, and a lower bound invariant on a read-only field restricts the class of the object that can be read from the field to be superclass of the bound.

We use the relation $\Rightarrow_{cinv}$ to reduce bounds from the class invariant to a constraint form: $\vdash [V_i \mapsto \tau_i]\psi_{inv}{\Rightarrow_{cinv}}\psi$, where τ_i are the current variant parametric types for the class fields. The relation $\Rightarrow_{cinv}$ is defined in Figure 5.6. Note that this relation invokes the subtyping relations defined in Figure 5.3.

$$\vdash\circledast t{<:_i}t_1{\Rightarrow_{cinv}}\mathit{true} \qquad \vdash t_1{<:_i}\circledast t{\Rightarrow_{cinv}}\mathit{true}$$

$$\frac{\vdash t{<:}t_1{\Rightarrow}\psi_1 \quad \vdash t_1{<:}t{\Rightarrow}\psi_2}{\vdash\oplus t{<:_i}t_1{\Rightarrow_{cinv}}\psi_1{\vee}\psi_2} \qquad \frac{\vdash t{<:}t_1{\Rightarrow}\psi_1 \quad \vdash t_1{<:}t{\Rightarrow}\psi_2}{\vdash t_1{<:_i}\ominus t{\Rightarrow_{cinv}}\psi_1{\vee}\psi_2}$$

$$\frac{\alpha{=}\ominus\mid\odot \quad \vdash t{<:}t_1{\Rightarrow}\psi}{\vdash\alpha t{<:_i}t_1{\Rightarrow_{cinv}}\psi} \qquad \frac{\alpha{=}\oplus\mid\odot \quad \vdash t_1{<:}t{\Rightarrow}\psi}{\vdash t_1{<:_i}\alpha t{\Rightarrow_{cinv}}\psi} \qquad \frac{\vdash\ \psi^i_{inv}{\Rightarrow_{cinv}}\psi^i}{\vdash\ \bigwedge\psi^i_{inv}{\Rightarrow_{cinv}}\bigwedge\psi^i}$$

Figure 5.6: Class Invariant

To illustrate the use of these bounded invariants, consider a class declaration for `Cell`$\langle$`X`$\rangle$ with an upper bound `X<:Num`. For declarations of the form `Cell`$\langle\ominus$`Int`$\rangle$ and `Cell`$\langle\ominus$`T`$\rangle$, the relation $\Rightarrow_{cinv}$ generates the `Int<:Num` and `T<:Num`, respectively. The first constraint reduces to `true`, while the second constraint contains a type variable and will be checked later for satisfiability. As another example, for `Cell`$\langle\ominus$`Object`$\rangle$ the relation $\Rightarrow_{cinv}$ fails as the upper bound is violated. Correspondingly, for read access, we support `Cell`$\langle\oplus$`Int`$\rangle$ and `Cell`$\langle\oplus$`Object`$\rangle$, but not `Cell`$\langle\oplus$`String`$\rangle$ since no `String` objects can be read from the `Num`-bounded field.

The class invariant is accumulated recursively from all the superclasses, as shown below:

$$\frac{\begin{array}{c}\textbf{class } c\langle V_i\rangle_{i=1}^{m}\textbf{extends}(c_k\langle\pi_{ik}\rangle_{i=1}^{n_k})_{k=1}^{s}\textbf{where } \psi_{inv}\ \{..\}{\in}P \\ \rho{=}[V_i{\mapsto}\tau_i]_{i=1}^{m} \quad \rho\vdash c_1\langle\pi_{i1}\rangle_{i=1}^{n_1}{\Rightarrow}_p t \quad \vdash \rho\psi_{inv}{\Rightarrow}_{cinv}\psi\end{array}}{\mathit{cinv}(c_1\langle\tau_i\rangle_{i=1}^{m}){=}\psi{\wedge}\mathit{cinv}(t)}\ [\underline{\textbf{CINV}}]$$

5.5 Variant Parametric Type System

Variance annotations of programs are used to support flow analysis for more accurate generic types. We verify the flow of values through the following type judgemnt:

$$\Gamma; Q \vdash e :: \alpha t, \psi$$

This judgment is for type checking, and assumes that Γ (type environment), Q (type variables in scope) and αt (type with expected variance) are given while ψ is the collected flow constraint. Figure 5.7 presents the syntax-directed type rules of the above judgment for the various language constructs.

Our type system is flow-insensitive as every location (variable, parameter and field) is given a type that never changes. In our type system, each object of type t_1 can be placed in a location of type t_2, provided $t_1{<:}t_2$. The type of a location is therefore a particular *type view* of its object. This type view of an object may be changed by upcasting (via assignment or parameter passing) or by downcast operation that is checkable at runtime.
The following rule shows how to type check an assignment expression:

$$\frac{\alpha t{=}\mathit{GetType}(\Gamma, w) \quad \alpha{<:}\ominus \quad \Gamma; Q \vdash e :: \oplus t, \psi}{\Gamma; Q \vdash w = e :: \oplus\texttt{Void}, \psi}\ [\underline{\textbf{ASSIGN}}]$$

Flow-in or write-only $\ominus$ is mandated on the left-hand side (w) while flow-out or read-only $\oplus$ is mandated on the right-hand side (e). To highlight how these flows are enforced, we present the rule for variable and field access (w stands for either v or $v.f$):

$$\frac{\tau_1{=}\mathit{GetType}(\Gamma, w) \quad \vdash \tau_1{<:}\tau{\Rightarrow}\psi}{\Gamma; Q \vdash w :: \tau, \psi}\ [\underline{\textbf{VAR-FIELD}}]$$

To retrieve the types of the variables and class fields, we use the auxiliary [`GetType`] rules from Figure 5.7. The current type τ_1 of w is retrieved from the type environment Γ. Further, the rule checks that τ_1 is a subtype of the expected variant parametric type τ. This supports a flow-out from the variable w.

For object creation, we ensure that each object is constructed with an invariant type using $c\langle\odot t_i\rangle_{i=1}^{q}$. A type is said to be *invariant* if each variance on its immediate fields is marked with $\odot$. Note that the views of nested fields, namely $t_1, .., t_q$ from $c\langle\odot t_i\rangle_{i=1}^{q}$, may still be of variant parametric types. Note that the variance of all class fields (including those which require type promotion) returned by *fields* is $\odot$.

$$\text{[\underline{NEW}]}$$
$$\frac{\begin{array}{c} \mathit{vars}\{t_i\}_{i=1}^{q}\subseteq Q \quad t_0{=}c\langle\odot t_i\rangle_{i=1}^{q} \quad (\odot t_i' \ \ f_i)_{i=1}^{p}{=}\mathit{fields}(t_0) \\ \vdash \oplus t_0{<:}\tau{\Rightarrow}\psi_0 \quad \Gamma;Q\vdash v_i :: \oplus t_i', \psi_i \quad i=1..p \end{array}}{\Gamma;Q\vdash \textbf{new}\ c\langle t_i\rangle_{i=1}^{q}(v_1,..,v_p) :: \tau, \bigwedge_{i=0}^{p}\psi_i\wedge \mathit{cinv}(t_0)}$$

For the purpose of constructing invariant types, the type variables in $\{t_i\}_{i=1}^{q}$ must be instantiated from Q. The class invariant $\mathit{cinv}(t_0)$ captures the specified upper/lower bounds on fields that must be satisfied for every object of the class. When such fields are updated, we statically ensure that their bounds are never violated. Given an instantiated class type, the rule [`FIELDS`] returns the variant parametric types of the class fields using type promotion if necessary.

Local variable declaration v is marked for read-write access via $v :: \odot t$ as shown in the rule [`LOCAL`]. The rule for method call [`Call`] collects the flow-in for receiver and arguments, flow-out for the result and the method precondition.

5.5.1 Modular Flow Verification

We design a variant parametric type system that can be verified in a modular fashion. Each method declaration is given suitable variant parametric type annotations for its parameters, result and receiver. A “may” flow constraint ψ is specified at the header of each method declaration. This *may-flow* specification captures all possible flows that may occur in the method's body e. The type checking rule for a method is formalised as follows:

$$\text{[\underline{METHOD}]}$$
$$\frac{\begin{array}{c} \mathit{chkRecv}(\texttt{cn}, t_0) \quad \Gamma{=}\{v_i{::}\oplus t_i\}_{i=1}^{p}{+}\{\texttt{this}{::}\oplus t_0\} \\ \psi_1{=}\psi\wedge \bigwedge_{i=0}^{p}\mathit{cinv}(t_i)\wedge \mathit{cinv}(t) \quad \psi_1{\neq}\mathit{false} \\ Q{=}\{V^*\} \quad \mathit{vars}(\psi){\subseteq}Q \quad \mathit{vars}(\Gamma,t){\subseteq}Q \\ \Gamma;Q\vdash e :: \oplus t, \psi_2 \quad V_I{=}\mathit{vars}(\psi_2){-}Q \quad \psi_1 \implies \exists V_I{\cdot}\psi_2 \end{array}}{\texttt{cn}\vdash_{meth}\ t_0 \mid t\ mn((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \textbf{where}\ \psi\ \{e\}}$$

We first construct an initial assumed flow constraint ψ_1 that is derived from the declared may-flow specification ψ, class invariants for each parameter, and result $\bigwedge_{i=0}^{p}\mathit{cinv}(t_i)\wedge\mathit{cinv}(t)$, The

[**NULL**]

$$\frac{}{\Gamma; Q \vdash \textbf{null} :: \tau, \oplus\bot{<:}\tau}$$

[**LOCAL**]

$$\frac{\Gamma'{=}\Gamma{+}\{v{::}\odot t\} \qquad \Gamma; Q \vdash e_1{::}\oplus t, \psi_1 \;\; \Gamma'; Q \vdash e_2{::}\tau, \psi_2}{\Gamma; Q \vdash \{t\ v{=}e_1;\ e_2\} :: \tau, \psi_1{\wedge}\psi_2}$$

[**SEQ**]

$$\frac{\Gamma; Q \vdash e_1{::}\circledast t, \psi_1 \qquad \Gamma; Q \vdash e_2{::}\tau, \psi_2}{\Gamma; Q \vdash e_1; e_2{::}\tau, \psi_1{\wedge}\psi_2}$$

[**COND**]

$$\frac{\Gamma(v){<:}\oplus\texttt{Bool} \qquad \Gamma; Q \vdash e_1 :: \tau, \psi_1 \;\; \Gamma; Q \vdash e_2 :: \tau, \psi_2}{\Gamma; Q \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 :: \tau, \psi_1{\wedge}\psi_2}$$

[**WHILE**]

$$\frac{\Gamma(v){<:}\oplus\texttt{Bool} \qquad \Gamma; Q \vdash e :: \tau, \psi}{\Gamma; Q \vdash \textbf{while}\ v\ e :: \oplus\texttt{Void}, \psi}$$

[**PROG**]

$$\frac{\vdash_{def}\ \mathit{InheritanceOK}(\mathit{def}_i),\ i = 1..n \qquad \vdash_{def}\ \mathit{def}_i,\ i = 1..n}{\vdash_{prg}\ \ \mathit{def}_{i=1..n}}$$

[**CALL**]

$$\frac{\rho = [V_j \mapsto t_j]_{j=1}^{k} \quad \tau'_i{=}\Gamma(v'_i)_{i=0}^{q} \qquad \hat{t}_0 \mid t\ \mathit{mn}((\hat{t}_i\ v_i)_{i=1}^{q})\langle V_{1..k}\rangle\ \textbf{where}\ \psi..{\in}\tau'_0 \qquad \psi_1 = \bigwedge_{i=0}^{q} \tau'_i{<:}\rho(\oplus\hat{t}_i){\wedge}\rho(\oplus t){<:}\tau}{\Gamma; Q \vdash v'_0.\mathit{mn}(v'_1, .., v'_q)\langle t_{1..k}\rangle :: \tau, \psi_1{\wedge}\rho\psi}$$

[**CLASS**]

$$\frac{c_1{\vdash_{meth}}\mathit{meth}_i, i{=}1..q \qquad \mathit{vars}\{\pi_i\}_{i=1}^{n}{\cup}(\mathit{vars}\bigcup_{k=1}^{s}\{\hat{\pi}_{ik}\}_{i=1}^{n_k}) \subseteq \{X_i\}_{i=1}^{m}}{\vdash_{def}\ \ \textbf{class}\ c_1\langle X_i\rangle_{i=1}^{m}\ \textbf{extends}\ (\hat{c}_k\langle\hat{\pi}_{ik}\rangle_{i=1}^{n_k})_{k=1}^{s}\ \textbf{where}\ \psi_{inv}\ \{(\pi_i\ f_i)_{i=1}^{n}\ \mathit{meth}_{i=1..q}\}}$$

[**INHC**]

$$\frac{\mathit{def} = \textbf{class}\ c_1\langle V_i\rangle_{i=1}^{p}\ \textbf{extends}\ c_2\langle\hat{\pi}_i\rangle_{i=1}^{q}..\textbf{where}..\{\mathit{fd}^*\ \mathit{meth}_{1..p}\} \qquad (\exists \mathit{meth}\cdot \mathit{meth} \in c_2\langle\hat{\pi}_i\rangle_{i=1}^{q}..{\wedge}\texttt{name}(\mathit{meth}){=}\texttt{name}(\mathit{meth}_i)){\Rightarrow} \vdash\mathit{OverridesOK}(\mathit{meth}_i, \mathit{meth})\ \ i{\in}1..p}{\vdash\ \mathit{InheritanceOK}(\mathit{def})}$$

[**OVERRIDE**]

$$\frac{\mathit{meth}_1 = t_0 \mid t\ \mathit{mn}((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \textbf{where}\ \psi_1\ \{e_1\} \qquad \mathit{meth}_2 = \hat{t}_0 \mid t\ \mathit{mn}((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \textbf{where}\ \psi_2\ \{e_2\} \qquad V_L{=}\mathit{vars}(\hat{t}_0){-}\mathit{vars}(t_0)\ \ \vdash t_0{<:}\hat{t}_0{\Rightarrow}\psi\ \ \exists V_L{\cdot}(\psi{\wedge}\psi_1 \implies \psi_2)}{\vdash\ \mathit{OverridesOK}(\mathit{meth}_1, \mathit{meth}_2)}$$

[**GetType1**]

$$\frac{\tau{=}\Gamma(v)}{\tau{=}\mathit{GetType}(\Gamma, v)}$$

[**GetType2**]

$$\frac{\alpha t{=}\mathit{GetType}(\Gamma, v) \quad t{=}c\langle\tau_i\rangle_{i=1}^{n} \quad (\tau\, f){\in}\mathit{fields}(c\langle\tau_i\rangle_{i=1}^{n})}{\tau{=}\mathit{GetType}(\Gamma, v.f)}$$

[**FIELDS**]

$$\frac{\textbf{class}\ c_1\langle V_i\rangle_{i=1}^{n}\ \textbf{extends}\ c_2\langle\hat{\pi}_i\rangle_{i=1}^{r}..\{(\pi'_i\ f_i)_{i=1}^{m}..\} \qquad \rho{=}[V_i{\mapsto}\tau_i]_{i=1}^{n} \quad \rho \vdash \pi'_i{\Rightarrow_p}\tau'_i, i{\in}1..m \quad \rho \vdash \hat{\pi}_i{\Rightarrow_p}\hat{\tau}'_i, i{\in}1..r}{\mathit{fields}(c_1\langle\tau_i\rangle_{i=1}^{n}) = [(\tau'_i\ f_i)]_{i=1}^{m}{+}\mathit{fields}(c_2\langle\hat{\tau}'_i\rangle_{i=1}^{r})}$$

Figure 5.7: Variant Parametric Type Rules

initial assumed flow constraint must be satisfiable, that is, $\psi_1 \neq \mathit{false}$. Furthermore, we collect the flow constraint of the method body using $\Gamma; Q \vdash e : \oplus t, \psi_2$, where ψ_2 captures all flows that may occur in the method body e. To prove the correctness of each declared flow constraint, we perform a subtype entailment on the flow constraint with V_I as local type variables using: $\psi_1 \implies \exists V_I{\cdot}\psi_2$. If this entailment holds, we have successfully verified the flow specification of a given method declaration. We also check if t_0, the given type of $this$, is compatible (no stupid cast) with the current class via the predicate $\mathit{chkRecv}(\texttt{cn}, t_0) = \texttt{cn}\langle \odot t^* \rangle <: t_0$.

Method overriding is checked by the [`Override`] rule from Figure 5.7. For safe function subtyping, we require each overriding method to have *weaker or equal* flow specification compared to the overridden method.

5.6 Soundness

The soundness of our type system can be proven by relating to dynamic evaluation semantics of the form:

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

where Π and ϖ denote runtime stack and heap, respectively. This evaluation may yield three possible runtime errors, namely **E** = **Error-Null** | **Error-Cast** | **Error-Type**. The second error is due to cast operations guarded by runtime checks inserted by the compiler. The third error is due to an object of the wrong type being written into a location with some expected static type. For well-typed programs, this last error can never happen. The progress theorem states that **Error-Type** cannot occur while the type preservation theorem shows that the type of an expression is preserved with each reduction step. We outline the two theorems below. Details of proof may be found in Appendices B.1, B.2 and B.3.

Theorem 5.1 (Progress)**.** *Let Γ be the environment mapping program variables to ground types. If $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ and $\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$, then either:*

- *e is a value, or*
- *$\langle \Pi, \varpi \rangle [e] \hookrightarrow$ **Error-Null** | **Error-Cast**, or*
- *there exist Π', ϖ', e' such that $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.*

Note that the type rules are extended to include store typing Σ. The relation $\Gamma; \Sigma; \psi \models \Pi, \varpi$ denotes a consistency relation that relates static and dynamic semantics. The following theorem states the preservation of type during dynamic evaluation.

Theorem 5.2 (Preservation)**.** *Let Γ be an environment mapping program variables to ground types. If*

$$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$$

$$\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$$

$$\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \hat{\Pi}, \hat{\varpi} \rangle [\hat{e}]$$

then there exists $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ such that

$$\Gamma - \mathit{diff}(e, \hat{e}) = \hat{\Gamma} - \mathit{diff}(\hat{e}, e)$$

$$\hat{\Sigma} \supseteq \Sigma$$

$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash \hat{e} :: \tau, \hat{\psi}$$

$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q}; \hat{\psi} \wedge \psi \models \hat{\Pi}, \hat{\varpi}.$$

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

5.7 Casting and Cast Capture

While a key goal of a generic type system is to provide precise information to eliminate unnecessary downcasts, there remains always the need for cast operations to support the class subtyping mechanism. Furthermore, the introduction of generics and variance has complicated type casting as these operations must handle type variables and nested variant parametric types. For example, cast operations may target nested types, such as $\texttt{Vector}\langle\odot\texttt{Vector}\langle\oplus\texttt{Num}\rangle\rangle$, or those with type variables, such as $\texttt{Vector}\langle\oplus X\rangle$.

However, existing solutions that support casting in Java 1.5 are restricted in that they use cast checks on the outermost type constructor only [193], and rely on unchecked warnings that may cause runtime errors (*e.g.*, when a cast to type variable occurs). The only system that supports cast operations fully (but for parametric types) was proposed by Viroli and Natali [197]. Their technique can be adapted to handle arbitrary variant parametric types.

In the presence of single inheritance, we can classify each casting relation from t_0 to t into three categories: (1) safe upcast if $\vdash t_0 <: t$, (2) downcast with runtime check if $\vdash t <: t_0$, and (3) stupid cast if $\neg(\vdash t_0 <: t \vee \vdash t <: t_0)$. However, in the presence of multiple inheritance with interfaces, a class and an interface may be unrelated but a valid downcast is still possible if the actual type is a subtype of the two. Though it is possible to identify stupid cast with a more complex test, namely $\neg(\exists X \cdot X \neq \perp \wedge X <: t \wedge X <: t_0)$, we avoid it for simplicity. Instead, we only check to ensure that the type variables used in t have come from Q. Our type rule to support a variant cast operation is given below:

$$\underline{[\textbf{CAST}]}$$

$$\frac{\alpha t_0{=}\Gamma(v) \quad \alpha{<:}\oplus \quad \textit{vars}(t){\subseteq}Q}{\Gamma;Q \vdash (t)v :: \tau, \oplus t{<:}\tau}$$

While casting is used to check a specific type for an object, we often have to deal with objects of unknown types. For example, we may have an object with a static type `List`$\langle\ominus$`A`$\rangle$, and we may be interested to know its actual invariant type `List`$\langle\odot$`T`$\rangle$, where `T` is an unknown type. To help identify the invariant type of a given object, we introduce a *cast capture* construct based on reflection mechanism: $\{v_1 = (t)v\ ;\ e\}$. The following rule shows how to type check the capture construct:

$$\underline{[\textbf{CAPTURE}]}$$

$$\frac{\begin{array}{c}\alpha c\langle\tau_i\rangle_{i=1}^{n} = \Gamma(v) \quad \alpha{<:}\oplus \quad t{=}c\langle\odot V_i\rangle_{i=1}^{n} \quad \{V_i\}_{i=1}^{n} \cap Q{=}\{\} \\ \Gamma;Q \vdash v_1 :: \ominus t, \psi_1 \quad Q_1{=}Q{\cup}\{V_i\}_{i=1}^{n} \quad \Gamma;Q_1 \vdash e :: \tau, \psi_2\end{array}}{\Gamma;Q \vdash \{v_1 = (t)v; e\} :: \tau, \psi_1{\wedge}\psi_2}$$

Note that t is an invariant type of the form $c\langle\odot V_i\rangle$: c should have the same class type as v, while the *captured type variables* V_i stand for unknown types. Each V_i can be used in the expression e with its flow captured by the collected flow ($\psi_1{\wedge}\psi_2$).

The flow of captured type variables should not cause additional restriction or generalization at the method boundary. We next present how the type system ensures the correct use of captured type variables. The actual type t obtained via reflection is guaranteed to be more precise than `v`'s static type, $\Gamma(v)$. We call this guarantee *reflection assumption*. For each method, a relation $\Gamma \vdash e \Rightarrow_C V_C, \psi_C$ collects captured type variables, V_C, and their reflection assumptions, ψ_C as follows:

$$\frac{\begin{array}{c}\alpha t_0{=}\Gamma(v) \quad \vdash t{<:}t_0{\Rightarrow}\psi \\ X{=}\textit{vars}(t) \quad \Gamma \vdash e{\Rightarrow_C}V, \psi_1\end{array}}{\Gamma \vdash \{v_1 = (t)v; e\} \Rightarrow_C V{\cup}X, \psi{\wedge}\psi_1}$$

The method judgement is modified to exclude captured type variables V_C from the local type variables V_I. Additionally, the expected flow ψ_1 is strengthened with reflection assumptions ψ_C.

$$\frac{\begin{array}{c} chkRecv(\texttt{cn}, t_0) \quad \Gamma{=}\{v_i{::}\oplus t_i\}_{i=1}^{p}{+}\{\texttt{this}{::}\oplus t_0\} \\ \Gamma \vdash e{\Rightarrow}_C V_C, \psi_C \quad \psi_1{=}\psi{\wedge}\bigwedge_{i=0}^{p} cinv(t_i){\wedge}cinv(t){\wedge}\psi_C \\ Q{=}\{V^*\} \quad vars(\psi){\subseteq}Q \quad vars(\Gamma, t){\subseteq}Q \quad \psi_1{\neq}false \\ \Gamma; Q \vdash e :: \oplus t, \psi_2 \quad V_I{=}vars(\psi_2){-}Q{-}V_C \quad \psi_1 \Longrightarrow \exists V_I{\cdot}\psi_2 \end{array}}{cn \vdash_{meth}\ t_0 \mid t\ mn((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \textbf{where}\ \psi\ \{e\}}\ \boxed{\text{METHOD-WITH-CAPTURE}}$$

The proper flow of captured type variables is then ensured by the entailment from the above rule.

5.7.1 Cast Capture Examples

The cast capture mechanism can also be viewed as a downcast to the object's invariant type. Unknown types that are captured (via reflection) may be used in the program code, as shown in the example below:

```
  void addNode(List⟨⊖A⟩ y, B z) where B<:A {
   List⟨⊙S⟩ v; List⟨⊙S⟩ w;
   {v = (List⟨⊙T⟩) y; w = new List⟨T⟩();
   w.val = z ; w.next = v.next ; v.next = w; } }
```

Though we do not know the exact type of `y`, we can use a cast capture on `(List<⊙T>)` to obtain its invariant type. Correspondingly, the reflection assumption is `A<:T`. We use the captured type `T` to build a `List⟨⊙T⟩` node, write `z` to `w.val`, and also reconstruct pointers for the linked list in a type-safe and yet generic way. For this example, the initial assumed flow is ψ_1≡`(B<:A∧A<:T)`, whereby `B<:A` is from the flow specification and `A<:T` is the reflection assumption. This initial assumed flow implies the collected flow constraint ∃`S`$\cdot\psi_2$, where ψ_2≡`(S<:T∧T<:S∧B<:S)`. Hence, modular verification holds for this example.

The same cast capture mechanism may also be used to capture an unknown invariant type, enabling a swap of elements within the same collection, without knowledge of its type. We consider:

```
void swapVec(Vector⟨⊛⟩ v,int i, int j) {
  Vector⟨⊙S⟩ w;
   {w = (Vector⟨⊙T⟩) v;
   S v1 = w.elementAt(i);
   S v2 = w.elementAt(j);
   w.setElementAt(v2,i); w.setElementAt(v1,j);} }
```

Note that input parameter `v` uses a bivariant type `Vector<⊛>`, which can be used to support an argument of an arbitrary `Vector` object. The initial assumed flow is ψ_1≡`true`, while the collected flow is $\exists$`S`$\cdot\psi_2$, where ψ_2≡`(S<:T∧T<:S)`. Hence, the entailment $\psi_1 \implies \exists$`S`$\cdot\psi_2$ holds. An example of a method that does not type check is presented below:

```
Vector⟨⊕Y⟩ foo1(Vector⟨⊛⟩ v){ Vector⟨⊙S⟩ w;{w =(Vector⟨⊙T⟩) v;w}}
```

The initial assumed flow is ψ_1≡`true` while the collected flow is ψ_2≡`T<:Y`. Note that neither `T` (captured type variable) nor `Y` (global type variable) are existentially quantified from ψ_2. The entailment $\psi_1 \implies \exists$`S`$\cdot\psi_2$ does not hold, since the captured type variable `T` introduces an additional flow at method boundary.

As another example, the following definition type checks as the collected flow from the method's body (after elimination of the local type variable `S`) is ψ_2≡`true`:

```
Vector⟨⊛⟩ foo2(Vector⟨⊛⟩ v) { Vector⟨⊙S⟩ w;{w=(Vector⟨⊙T⟩) v;w}}
```

5.8 Experimental Validation

5.8.1 Implementation

We built a prototype for our variant parametric type system and carried out initial experiments to validate its feasibility. Our system was built using the Glasgow Haskell compiler [150], with a constraint solver (for handling subtyping constraints) implemented using Constraint Handling Rules (CHR) [70].

Our constraint solver employs the following two-step algorithm to prove the non-structural subtype entailment of the form $\forall V_G\cdot(\psi_1 \implies \exists V_I\cdot\psi_2)$. Note that ψ_1, ψ_2 are conjunctions of subtyping constraints , while V_G and V_I are sets of type variables. Even though the entailment from the [`METHOD`] rule may contain disjunctions, it can be reduced to entailments of the above form.

1. We eliminate the local type variables V_I (based on their upper and lower bounds) from ψ_2 to obtain $\psi_2'=\bigwedge_{i=1}^{n} X_i{<:}Y_i$ using techniques similar to [157, 195]. To support the language's semantics a local type inference similar to [154, 137] is employed to identify appropriate instantiated types for local type variables or type parameters.

2. The resulting entailment $\forall V_G \cdot (\psi_1 \implies \bigwedge_{i=1}^{n} X_i{<:}Y_i)$ is equivalent to $\bigwedge_{i=1}^{n}(\forall V_G \cdot (\psi_1 \implies X_i{<:}Y_i))$. Each entailment can be proven by contradiction using the falsity of the formula $\forall V_G \cdot (\psi_1 \wedge \mathit{notsub}(X_i, Y_i))$, where $\mathit{notsub}(t_1, t_2)$ represents negation of subtyping relation.

Our constraint solver implements the variant subtyping rules (from Figure 5.3). Its deduction mechanism detects falsity based on pair of constraints of the form $t_1{<:}t_2$ and $\mathit{notsub}(t_1, t_2)$. Our algorithm is a sound approximation of the subtype entailment problem. The deduction mechanism can be further extended by the techniques of *case analysis* and *inductive proving*. However, from our experience working with large sets of Java library and application codes that have been annotated and checked with variant parametric types, we have yet to encounter real examples which require such extensions.

5.8.2 Experiments

To test the utility of our flow-based variant type system, we evaluated our prototype on a set of Java applications[1] as used in [54, 72]. These applications make use of library classes from package `java.util`, which we annotated with our variant parametric types. We counted each method declaration with flow specification, each class declaration with type parameters and each cast capture as a line of annotation. On average, these annotations constituted about 5.5% of the source code, which may be considered a reasonable price to pay for better reuse of type safe generic code. Due to modular type checking, the time needed to verify type-safe generic code was less than one second for each library code and less than 30 seconds for each application code. We expect that the time can be reduced by using a specialised constraint solver. Currently, our prototype is based on a meta constraint handling system written in CHR (which compiled to a Prolog program under IC-Parc's ECLiPSe system [12]).

Figures 5.8 and 5.9 show the experimental results for representative classes from the `java.util` package and application code (in terms of remaining casts). We counted the number of casts in Java 1.4 code (non-generic), Java 1.5 (annotated with wildcards) and our system (VPT - annotated with variant parametric types). The Java 1.5 compiler could not statically check some operations (especially those related to raw types and casts to type variables), and

[1]For more details: `www.junit.org`, `www.cs.princeton.edu/ ~appel/modern/java/JLex/`, `www.cs.princeton.edu/ ~appel/modern/java/CUP/`, `www.spec.org/osg/jvm98/`, `vpoker.sourceforge.net`, `telnetd.sourceforge.net`.

Library	Prog.	Java 1.4	Java 1.5		VPT	
	Lines	Casts	Casts	Warnings	Casts	Warnings
AbstractList	909	1	1	0	0	0
AbstractSet	162	1	1	0	0	0
ArrayList	623	2	8	9	1	0
HashMap	1103	7	9	20	3	0
HashSet	231	2	4	3	1	0
Hashtable	1154	10	14	31	7	0
LinkedList	814	2	4	5	2	0
Properties	925	8	8	1	0	0
Vector	1062	2	9	9	0	0
Total	6983	35	58	78	14	0

Figure 5.8: Results for Library Code

Application	Prog.	Java 1.4	Java 1.5		VPT	
	Lines	Casts	Casts	Warnings	Casts	Warnings
DB	842	19	1	0	0	0
JUnit	5886	54	30	1	15	0
VPoker	6792	36	8	0	6	0
JLex	7260	69	12	3	0	0
Jess	10639	95	34	0	12	0
TelnetD	11314	46	8	0	6	0
JavaCup	11468	543	98	2	65	0
Total	54201	862	191	6	104	0

Figure 5.9: Results for Application Code

issued unchecked warnings. Therefore, it is the programmer's responsibility to ensure that all unchecked operations are in fact safe.

To summarize, our method can eliminate a significant portion (on average 87.9%) of the casts from non-generic Java 1.4 application code and 45.5% of the casts from wildcard-generic Java 1.5 application code. We have also made improvements for library code by eliminating about 60% casts from non-generic Java 1.4 code and about 75.8% casts from the wildcard-generic Java 1.5 code. Since our system fully supports casting for variant types, we can verify the unsafe operations for which the Java 1.5 compiler generates unchecked warnings. Note that Java 1.5 libraries contain more casts than Java 1.4 libraries do, since Java 1.4 containers are based on `Object` type instead of generic types. As expected, Java 1.4 application code requires more downcasts compared to Java 1.5 code.

Figure 5.10 shows a chart that visualises the percentage of remaining casts in each benchmark written in Java 1.4, Java 1.5 and our VPT. Java 1.4 which contains the casts from the

Figure 5.10: Remaining Casts for Application Code

original application code serves as reference.

Note that the casts eliminated using our type system measure the improvement in program safety. Current Java implementation (which translates parametric programs via *type erasure*) would re-introduce casts at the bytecode level. While such re-admitted casts may cause runtime overheads, they are known to be type safe and will never fail at runtime. Obviously, a better solution is to support variant parametric type at the bytecode level and we look forward to this scenario.

5.9 Other Features

In this section, we present some features omitted in the main presentation for brevity.

The hierarchy of primitive types forms a separate lattice from reference types. Furthermore, it is *not* the case that $\bot <: p <: \mathit{Object}$ for each primitive type p. Due to such differences, primitives are excluded from use as type arguments for generic classes in Java 1.5. Furthermore, the type erasure algorithm for the parametric program will transform each parametric field into an *Object* type for backwards compatibility. This is invalid if primitive types are used as type arguments. We now show how primitive types can be used as type arguments for generic classes in our system.

First, we need to add two constraints to distinguish reference and primitive types, as shown below:

$$\psi ::= \cdots \;\mid \texttt{ref}(t) \;\mid \texttt{prim}(t)$$

As these two families of types are disjoint, we add the following CHR irrevocable rule:

$$\texttt{ref}(t) \wedge \texttt{prim}(t) \quad \Leftrightarrow \quad \mathit{false}$$

Second, we need to consider primitive types in the new variant subtyping mechanism. In the new subtyping hierarchy, $\circledast t$ denotes any type (reference or primitive) while $\oplus\mathit{Object}$ and $\ominus\bot$ denote only reference types (that are still equivalent, namely $\oplus\mathit{Object}{\equiv}\ominus\bot$). The subtyping relation is changed accordingly: $\oplus\mathit{Object}{<:}\circledast t$ still holds while $\circledast t{<:}\oplus\mathit{Object}$ does not hold anymore. Furthermore, we allow $\bot{<:}t$ and $t{<:}\mathit{Object}$ if and only if t is not a primitive type. To support these changes, we modify the main variant subtyping rules from Figure 5.3 to the following:

$$\frac{\alpha{\neq}\circledast}{\vdash \alpha t{<:}\oplus\mathit{Object}{\Rightarrow}\texttt{ref}(t)} \qquad \frac{\alpha{\neq}\circledast}{\vdash \alpha t{<:}\ominus\bot{\Rightarrow}\texttt{ref}(t)}$$

$$\frac{\alpha_1{\neq}\circledast \quad \neg(\alpha_1{<:}\oplus) \quad \vdash \mathit{Object}{<:}t_2{\Rightarrow}\psi}{\vdash \alpha_1 t_1{<:}\oplus t_2{\Rightarrow}\psi{\wedge}\texttt{ref}(t_1)} \qquad \frac{\alpha_1{\neq}\circledast \quad \neg(\alpha_1{<:}\ominus) \quad \vdash t_2{<:}\bot{\Rightarrow}\psi}{\vdash \alpha_1 t_1{<:}\ominus t_2{\Rightarrow}\psi{\wedge}\texttt{ref}(t_1)}$$

$$\vdash\bot{<:}t{\Rightarrow}\texttt{ref}(t) \qquad \vdash t{<:}\mathit{Object}{\Rightarrow}\texttt{ref}(t)$$

$$\vdash t{<:}\bot{\Rightarrow}t{<:}\bot{\wedge}\texttt{ref}(t) \qquad \vdash\mathit{Object}{<:}t{\Rightarrow}\mathit{Object}{<:}t{\wedge}\texttt{ref}(t)$$

Programmers often make use of the `instanceof` test on the runtime type of an object prior to some operations. Due to flow and path insensitivity, the type system is currently unable to take advantage of such runtime testing of types. To help eliminate more cast operations, our compiler translates each program fragment of the form:

```
if v.instanceof(t) then e1 else e2
```

to use a special program construct with fresh $\texttt{v}_0$ variable:

```
if v.instanceof(t) then let v0::t=v in [v↦v0]e1 else e2
```

This construct is part of our core intermediate language, and it is generated prior to type checking. It is valid on the proviso that any assignment into `v` is a subtype of the more specific `t`. A type rule corresponding to the new language construct is shown below:

$$\boxed{\textbf{LET-INSTANCEOF}}$$

$$\frac{\begin{array}{c} e_1 \equiv (\texttt{let}\ v_0 :: t = v\ \texttt{in}\ e) \\ \Gamma'{=}\Gamma{+}\{v_0{::}\odot t\} \quad \Gamma';Q\vdash e{::}\tau,\psi_1 \quad \Gamma;Q\vdash e_2{::}\tau,\psi_2 \end{array}}{\Gamma;Q\vdash \textbf{if}\ v.\mathit{instanceof}(t)\ \textbf{then}\ e_1\textbf{else}\ e_2 :: \tau,\psi_1{\wedge}\psi_2}$$

Flow-sensitivity may also cause some loss in type precision (such that some downcasts cannot be statically verified) when the same local variable is used for objects with different variant parametric types. To rectify this, we could use Static Single Assignment (SSA) intermediate form [49] which is known to give better flow-sensitive analysis results. Conversion of programs to SSA form can be handled in a preprocessing step, prior to type checking.

These techniques for incorporating path and flow sensitivity are quite standard, and were also explored in [200].

5.10 Related Work

Software reuse has received much research interest for its boost to software development and maintenance productivities. Recently, generic type has become a main thrust in supporting software reuse. In reusing Java code, several works have proposed for refactoring legacy Java programs into those that use generic versions of popular container classes [54, 57, 72, 198].

Other works try to achieve precise Java type inference results in the presence of parametric polymorphism and data polymorphism in order to reduce the redundant cast operations [156, 3, 200]. The precision typically comes at the price of a whole program analysis. Every time when an application code is analysed, the reachable library code must also be re-analysed.

Variant parametric types attempt to increase language expressivity and code reuse by introducing another subtyping scheme, based on the notion of variance. This idea originated from the structured virtual types proposed by Thorup and Torgersen [186]. Their work is the first to link access rights and covariant subtyping to the fields of each use of a class rather than the class itself. Igarashi and Viroli extended this concept to support contra- and bi-variance [103]. They also formalised the variant type system by mapping it into a corresponding *existential type* system [103, 104] for Featherweight Java. While Igarashi and Viroli's design faithfully models the existential type system, it has been found to be too restrictive by the designers of Java 1.5. One improvement made in Java 1.5 is to allow each wildcard type to be opened without a corresponding close operation. This provides more flexibility for writing generic code, but weakens the link to the traditional pack/unpack mechanism of the existential type system. Hence, even though a full-scale language system has been implemented, the soundness of the wildcard type system is still under development (as of [194]). Other than Java, a recently developed language

Scala [138] supports variance for parametric polymorphism. In contrast with our approach, Scala uses variance at *declaration-site*. However, an earlier version of Scala has experimented with the use-site variance mechanism that is consistent with the original system of Igarashi and Viroli but without the flexibility of the wildcard capture. This was considered to be too restrictive before the authors abandoned the approach. Recently, generic types of C# [61] were extended with *declaration-site* variance following the design adopted for the language Scala.

Theoretical foundations of the variance have also been studied in the context of typed λ-calculi, where type operators are equipped with a polarity property [31, 179, 56]. These foundations have even been extended to handle higher-order functions, but are closer in nature to declaration-site variance, and have mostly been formalised in only a theoretical setting, without practical implementations.

We have proposed a new approach based on flow analysis to support the variant parametric type system. We leverage prior knowledge that has been accumulated for flow analysis and entailment for non-structural subtyping constraints. Palsberg and O'Keefe [142] show the equivalence of flow analysis and non-structural subtyping. Subtype entailment is known to be hard even for simple subtyping constraints. Rehof and Henglein determined the complexity of structural subtype entailment: for simple types, it is coNP-complete [96] and for recursive types it is PSPACE-complete [97]. Furthermore, they showed that non-structural subtype entailment is PSPACE-hard and is conjectured PSPACE-complete [97]. Su et al. [183] show the decidability of the first-order theory of non-structural subtyping with unary function symbols. Algorithms for non-structural subtype entailment (sound, but incomplete) were developed in Pottier [157], Trifonov and Smith [195]. While the decidability of non-structural subtype entailment remains an open problem, we use sound techniques based on these previous algorithms.

PART III

Finale

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation we have designed and implemented two advanced type systems for improving the software quality in the context of a Java-like programming language. More specifically, the first type system enables an automatic safe compile-time region-based memory management, while the second type system improves software reusability (also called genericity). We have used a similar approach to develop both type systems, that consists of two main ingredients, namely a simple flow analysis and a set of partially-ordered type annotations. Flow analysis captures type annotations in a flow-insensitive manner through the program code, but summarizes a parameterized flow at each method boundary. Subtyping of annotated types provides the direction of flows. As a consequence, the type rules generate flow constraints among the annotated types. We summarize the achievements and prospects for both type systems next.

6.1 Safe Region-Based Memory Management

Our aim was to provide a fully-automatic region inference system for a core subset of Java. We achieved this by a summary-based flow insensitive analysis and by allowing classes and methods to be region-polymorphic, with region-polymorphic recursion for methods. The inferred region lifetime constraints for the classes and the methods form the classes' invariants and the methods' preconditions, respectively. We have seen how the region lifetime constraints prevent dangling references and generate appropriate region instantiations.

We have proven that the result of our region inference is correct with respect to our region type system. We have also proven that our region type system guarantees that well-typed programs are region safe and never create dangling references in the store and on the stack.

Region inference has a trivial solution by putting everything in one region. Our analysis aims for a better solution by putting objects into regions with shorter lifetimes, whenever our system is able to guarantee that it is safe to do so. In the experiments, we used the degree of memory reuse to measure the quality of our region inference results. As shown by the examples, different kinds of the region subtyping can improve the regions lifetime precision. We used a dependency graph to guide the inference process. The complex inter-dependency between classes and methods may affect the precision and the scalability of our analysis. For example, treating the whole program as one strongly connected component (SCC) of the dependency graph may have a bad impact on the precision of the inference for classes due to the monomorphic principle used for classes.

There remain a number of areas where improvements of our region inference are possible. Several directions can be taken to improve memory utilization. As an example, component objects that are *owned* by another object can be placed in the same region as the latter, since no references exist from outside the owner. This idea has been explored in [23]. Coupled with alias (including ownership) annotations that can be automatically inferred, as described in [10], we believe that ownership information can be derived to make this optimization fully automatic.

Our region type rules are flow-insensitive (within each method) but context-sensitive (across methods). The latter is due to the use of region polymorphism at method boundaries. Flow-insensitivity may cause some loss in region lifetime precision when the same local variable is used for objects with different lifetime requirements. To partially rectify this, we could use Static Single Assignment (SSA) intermediate form [49] which is known to give better flow-sensitive analysis results. Conversion of programs to SSA form can be handled in a preprocessing step. Since the region types are intended more for compilers rather than for programmers, the programmers are not required to work on the SSA form.

Other direction to further improve the memory utilization is to explore suitable liveness analysis and restructuring transformations. Effective placement of local variable declarations, object allocations and expression blocks can affect region placement and the extent to which memory is effectively reused. A promising approach is to combine the region inference with a linearity analysis that determines the objects that have become dead [113, 38]. The space of dead objects may be recycled earlier in a region. The recycling of an entire region (without deallocating it) was called region resetting and it was studied in the context of ML [190].

Another future direction is to extend our region inference system to all features of a Java-like programming language. A discussion about the possible extensions is presented in Appendix A.6. In the context of concurrency, our intention is to adapt the techniques presented in this dissertation to check and infer the scoped memory areas of scoped-based memory model proposed by the Real-Time Specification for Java [19].

6.2 Better Genericity

Our goal was to strive for type-safe object-oriented programs with better genericity via a modular flow-based approach to variant parametric type system. We have developed a novel approach that is practically driven and can give better generic types. Our flow analysis captures value flows via subtyping constraints. A major benefit of this approach is the considerable knowledge in flow analysis that has been accumulated in the recent past. In particular, to support modular type-checking, we require non-structural subtype entailment. While the decidability of non-structural subtype entailment remains an open problem, our work is built on top of sound but practical approximations. To capture information flow more accurately, we have augmented our generic type system with intersection types which support Java-like multiple (interface) inheritance. We have built a prototype system based on a set of syntax-directed type rules. This prototype is supported by a constraint-solver for variant subtyping, customized using CHR. Furthermore, our system supports full casting for variant types. Through a new cast capture mechanism, we can use reflection to handle objects with unknown types in a type-safe way. Experimental evaluation indicates that more downcasts can be eliminated by our approach, even when it is compared against the state-of-the-art type system from Java 1.5.

One future direction is to formulate and implement an inference framework. The experiments done with our flow-based type checker have confirmed the possible improvements over the current Java generics. Our flow-based approach is another step towards better generic types for Java. In addition, a good inference mechanism can support faster migration of legacy codes to variant parametric types, and can improve the productivity of writing new code.

BIBLIOGRAPHY

[1] ABADI, M. and CARDELLI, L., "An imperative object calculus (invited paper)," *Theory and Practice of Object Systems (TAPOS)*, vol. 1, no. 3, pp. 151–166, 1995.

[2] ABADI, M., CARDELLI, L., and VISWANATHAN, R., "An interpretation of objects and object types," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 396–409, 1996.

[3] AGESEN, O., "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 2–26, 1995.

[4] AGESEN, O., FREUND, S. N., and MITCHELL, J. C., "Adding type parameterization to the java language," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 49–65, 1997.

[5] AHO, A., SETHI, R., and ULLMAN, J., *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] AIKEN, A., "Introduction to set constraint-based program analysis," *Science of Computer Programming*, vol. 35, pp. 99–111, 1999.

[7] AIKEN, A., FÄHNDRICH, M., and LEVIEN, R., "Better static memory management: Improving region-based analysis of higher-order languages," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 174–185, 1995.

[8] AIKEN, A. and WIMMERS, E. L., "Type inclusion constraints and type inference," in *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 31–41, 1993.

[9] AIKEN, A., WIMMERS, E. L., and PALSBERG, J., "Optimal representations of polymorphic types with subtyping," *Higher-Order and Symbolic Computation*, vol. 12, no. 3, pp. 237–282, 1999.

[10] ALDRICH, J., KOSTADINOV, V., and CHAMBERS, C., "Alias Annotation for Program Understanding," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 311–330, 2002.

[11] AMADIO, R. M. and CARDELLI, L., "Subtyping recursive types," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 104–118, 1991.

[12] AT IMPERIAL COLLEGE, I.-P., "ECLiPSe Constraint Logic Programming." http://www.icparc.ic.ac.uk/eclipse/.

[13] BEEBEE, W. and RINARD, M., "An Implementation of Scoped Memory for Real-Time Java," in *International Workshop of Embedded Software (EMSOFT)*, pp. 289–305, 2001.

[14] BENKE, M., "Some complexity bounds for subtype inequalities," *Theorethical Computer Science*, vol. 212, no. 1-2, pp. 3–27, 1999.

[15] BIERMAN, G., PARKINSON, M., and PITTS, A., "MJ: An imperative core calculus for Java and Java with effects," Technical Report, Cambridge University, 2003.

[16] Birka, A. and Ernst, M. D., "A practical type system and language for reference immutability," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pp. 35–49, 2004.

[17] Birkedal, L. and Tofte, M., "A constraint-based region inference algorithm," *Theoretical Computer Science*, vol. 258, no. 1–2, pp. 299–392, 2001.

[18] Birkedal, L., Tofte, M., and Vejlstrup, M., "From region inference to von Neumann machines via region representation inference," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 171–183, 1996.

[19] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., and Turnbull, M., *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[20] Boyapati, C., Lee, R., and Rinard, M., "Ownership types for safe programming: Preventing data races and deadlocks," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 211–230, 2002.

[21] Boyapati, C., Lee, R., and Rinard, M., "Safe runtime downcasts with ownership types," in *ECOOP Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pp. 1–12, 2003.

[22] Boyapati, C., Liskov, B., and Shrira, L., "Ownership Types for Object Encapsulation," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 213–223, 2003.

[23] Boyapati, C., Salcianu, A., Beebee, W., and Rinard, M., "Ownership Types for Safe Region-Based Memory Management in Real-Time Java," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 324–337, 2003.

[24] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P., "Making the future safe for the past: Adding Genericity to the Java Programming Language," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 183–200, 1998.

[25] Bruce, K. B., "A paradigmatic object-oriented programming language: Design, static typing and semantics," *Journal of Functional Programming*, vol. 4, no. 2, pp. 127–206, 1994.

[26] Bruce, K., *Foundations of Object-Oriented Languages, Types and Semantics*. The MIT Press, 2002.

[27] Buchi, M. and Weck, W., "Compound types for Java," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 362–373, 1998.

[28] Calcagno, C., "Stratified operational semantics for safety and correctness of the region calculus," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 155–165, 2001.

[29] Calcagno, C., Helsen, S., and Thiemann, P., "Syntactic type soundness results for the region calculus," *Information and Computation*, vol. 173, no. 2, pp. 199–221, 2002.

[30] Canning, P. S., Cook, W. R., Hill, W. L., Olthoff, W. G., and Mitchell, J. C., "F-Bounded polymorphism for object-oriented programming," in *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 273–280, 1989.

[31] CARDELLI, L., “Notes about $F^{\omega}_{<:}$,” 1994. Available at http://research.microsoft.com/Users/luca/Notes/FwSub.ps.

[32] CARDELLI, L., “Type systems,” in *Allen B. Tucker (Ed.): The Computer Science and Engineering Handbook*, vol. 97, CRC Press, 2004.

[33] CARDELLI, L. and WEGNER, P., “On understanding types, data abstraction, and polymorphism.,” *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.

[34] CARLISLE, M. C. and ROGERS, A., “Software caching and computation migration in Olden,” in *ACM Principles and Practice of Paralle Computing*, pp. 29–38, 1993.

[35] CARTWRIGHT, R. and JR., G. L. S., “Compatible genericity with run-time types for the java programming language,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 201–215, 1998.

[36] CASTAGNA, G., “Covariance and contravariance: Conflict without a cause,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 3, pp. 431–447, 1995.

[37] CHEREM, S. and RUGINA, R., “Region analysis and transformation for Java programs,” in *International Symposium on Memory Management (ISMM)*, pp. 85–96, 2004.

[38] CHEREM, S. and RUGINA, R., “Uniqueness inference for compile-time object deallocation,” in *International Symposium on Memory Management (ISMM)*, pp. 117–128, 2007.

[39] CHIN, W.-N., CRACIUN, F., KHOO, S.-C., and POPEEA, C., “A flow-based approach for variant parametric types,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 273–290, 2006.

[40] CHIN, W.-N., CRACIUN, F., QIN, S., and RINARD, M. C., “Region inference for an object-oriented language,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 243–254, 2004.

[41] CHRISTIANSEN, M. V. and VELSCHOW, P., “Region-Based Memory Management in Java.” Master’s Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[42] CLARKE, D. G. and DROSSOPOULOU, S., “Ownership, encapsulation and disjointness of type and effect,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 292–310, 2002.

[43] CLARKE, D. G., POTTER, J. M., and NOBLE, J., “Ownership Types for Flexible Alias Protection,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 48–64, 1998.

[44] COMPAGNONI, A. B. and PIERCE, B. C., “Higher-order intersection types and multiple inheritance,” *Mathematical Structures in Computer Science*, vol. 6, no. 5, pp. 469–501, 1996.

[45] CRACIUN, F., GOH, H. Y., and CHIN, W.-N., “A framework for object-oriented program analyses via Core-Java,” in *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, (Cluj-Napoca, Romania), pp. 197–205, 2006.

[46] CRACIUN, F., GOH, H. Y., POPEEA, C., and CHIN, W.-N., “Core-java: an expression-oriented Java,” in *Companion to ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 639–640, 2006.

[47] CRACIUN, F., QIN, S., and CHIN, W.-N., "A formal soundness proof of region-based memory management for object-oriented paradigm," in *Formal Methods and Software Engineering (ICFEM)*, 2008.

[48] CRARY, K., WALKER, D., and MORRISETT, G., "Typed Memory Management in a Calculus of Capabilities," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 262–275, 1999.

[49] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., and ZADECK, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[50] DAVEY, B. A. and PRIESTLEY, H. A., *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[51] DETERS, M. and CYTRON, R., "Automated discovery of scoped memory regions for real-time Java," in *International Symposium on Memory Management (ISMM)*, pp. 132–142, 2002.

[52] DEUTSCH, A., "On the complexity of escape analysis," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 358–371, 1997.

[53] DHURJATI, D., KOWSHIK, S., ADVE, V. S., and LATTNER, C., "Memory safety without runtime checks or garbage collection," in *ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 69–80, 2003.

[54] DONOVAN, A., KIEZUN, A., TSCHANTZ, M. S., and ERNST, M. D., "Converting Java programs to use generic libraries," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 15–34, 2004.

[55] DROSSOPOULOU, S., VALKEVYCH, T., and EISENBACH, S., "Java type soundness revisited," technical report, Imperial College, 1999.

[56] DUGGAN, D. and COMPAGNONI, A., "Subtyping for Object Type Constructors," in *Foundations of Object-Oriented Languages*, pp. 1–12, 1999.

[57] DUGGAN, D., "Modular type-based reverse engineering of parameterized types in Java code.," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 97–113, 1999.

[58] DUSSART, D., HENGLEIN, F., and MOSSIN, C., "Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time," in *International Static Analysis Symposium (SAS)*, pp. 118–135, 1995.

[59] EIFRIG, J., SMITH, S. F., and TRIFONOV, V., "Type inference for recursively constrained types and its application to OOP," *Electronic Notes in Theoretical Computer Science*, vol. 1, 1995.

[60] ELSMAN, M., "Garbage collection safety for region-based memory management," in *The ACM Workshop on Types in Language Design and Implementation (TLDI)*, pp. 123–134, 2003.

[61] EMIR, B., KENNEDY, A. J., RUSSO, C., and YU, D., "Variance and generalized constraints for C# generics," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 279–303, 2006.

[62] FÄHNDRICH, M. and AIKEN, A., “Making set-constraint program analyses scale,” in *First Workshop on Set Constraints at CP’96*, 1996.

[63] FÄHNDRICH, M., FOSTER, J. S., SU, Z., and AIKEN, A., “Partial online cycle elimination in inclusion constraint graphs,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 85–96, 1998.

[64] FÄHNDRICH, M., REHOF, J., and DAS, M., “Scalable context-sensitive flow analysis using instantiation constraints,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 253–263, 2000.

[65] FÄHNDRICH, M., *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California, Berkeley, May 1999.

[66] FLANAGAN, C. and FELLEISEN, M., “Componential set-based analysis,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 235–248, 1997.

[67] FOSTER, J., *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, 2002.

[68] FOSTER, J. S., FÄHNDRICH, M., and AIKEN, A., “A theory of type qualifiers,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 192–203, 1999.

[69] FREY, A., “Satisfying subtype inequalities in polynomial space,” *Theorethical Computer Science*, vol. 277, no. 1-2, pp. 105–117, 2002.

[70] FRÜHWIRTH, T. W., “Theory and practice of constraint handling rules,” *Journal of Logic Programming*, vol. 37, no. 1-3, pp. 95–138, 1998.

[71] FUH, Y.-C. and MISHRA, P., “Polymorphic subtype inference: Closing the theory-practice gap,” in *Theory and Practice of Software Development*, vol. 2, pp. 167–183, 1989.

[72] FUHRER, R., TIP, F., KIEZUN, A., DOLBY, J., and KELLER, M., “Efficiently refactoring Java applications to use generic libraries,” in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 71–96, 2005.

[73] GAPEYEV, V., LEVIN, M. Y., and PIERCE, B. C., “Recursive subtyping revealed,” *Journal of Functional Programming*, vol. 12, no. 6, pp. 511–548, 2002.

[74] GAY, D., *Memory Management with Explicit Regions*. PhD thesis, University of California, Berkeley, 2001.

[75] GAY, D. and AIKEN, A., “Memory Management with Explicit Regions,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 313–323, 1998.

[76] GAY, D. and AIKEN, A., “Language support for regions,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 70–80, 2001.

[77] GLYNN, K., STUCKEY, P. J., SULZMANN, M., and SONDERGAARD, H., “Boolean constraints for binding-time analysis,” in *Workshop on Programs as Data Objects II (PADO)*, pp. 39–62, 2001.

[78] GOSLING, J., JOY, B., STEELE, G., and BRACHA, G., *The Java Language Specification*. Addison-Wesley, 2005.

[79] GROSSMAN, D., “Type-Safe Multithreading in Cyclone,” in *The ACM Workshop on Types in Language Design and Implementation (TLDI)*, pp. 13–25, 2003.

[80] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., and CHENEY, J., “Region-Based Memory Management in Cyclone,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 282–293, 2002.

[81] GUSTAVSSON, J. and SVENNINGSSON, J., “Constraint abstractions,” in *Workshop on Programs as Data Objects II (PADO)*, pp. 63–83, 2001.

[82] HALLENBERG, N., ELSMAN, M., and TOFTE, M., “Combining region inference and garbage collection,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 141–152, 2002.

[83] HANSON, D. R., “Fast allocation and deallocation of memory based on object lifetimes,” *Software-Practice and Experience*, vol. 20, no. 1, pp. 5–12, 1990.

[84] HEINTZE, N. and TARDIEU, O., “Ultra-fast aliasing analysis using cla: A million lines of c code in a second,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 24–34, 2001.

[85] HEINTZE, N., *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[86] HEINTZE, N., “Set-based analysis of ML programs,” in *ACM Conference on LISP and Functional Programming*, pp. 306–317, 1994.

[87] HEINTZE, N., “Control-flow analysis and type systems,” in *International Static Analysis Symposium (SAS)*, pp. 189–206, 1995.

[88] HEINTZE, N. and MCALLESTER, D. A., “Linear-time subtransitive control flow analysis,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 261–272, 1997.

[89] HEINTZE, N. and MCALLESTER, D. A., “On the cubic bottleneck in subtyping and flow analysis,” in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 342–351, 1997.

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

[91] HELSEN, S. and THIEMANN, P., “Syntactic type soundness for the region calculus,” *Electronic Notes in Theoretical Computer Science*, vol. 41, no. 3, 2000.

[92] HENGLEIN, F., “Type inference with polymorphic recursion,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 2, pp. 253–289, 1993.

[93] HENGLEIN, F., MAKHOLM, H., and NISS, H., “A direct approach to control-flow sensitive region-based memory management,” in *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pp. 175–186, 2001.

[94] HENGLEIN, F., MAKHOLM, H., and NISS, H., “Effect types and region-based memory management,” in *Advanced Topics in Types And Programming Languages*, pp. 87–136, MIT Press, 2005.

[95] HENGLEIN, F. and MOSSIN, C., “Polymorphic binding-time analysis,” in *European Symposium on Programming (ESOP)*, pp. 287–301, 1994.

[96] HENGLEIN, F. and REHOF, J., "The complexity of subtype entailment for simple types," in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 352–361, 1997.

[97] HENGLEIN, F. and REHOF, J., "Constraint automata and the complexity of recursive subtype entailment for simple type," in *International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 616–627, 1998.

[98] HICKS, M., MORRISETT, G., GROSSMAN, D., and JIM, T., "Experience with safe manual memory management in Cyclone," in *International Symposium on Memory Management (ISMM)*, pp. 73–84, 2004.

[99] HIGUERA-TOLEDANO, M. T., ISSARNY, V., BANÂTRE, M., CABILLIC, G., LESOT, J.-P., and PARAIN, F., "Region-based memory management for real-time java," in *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 387–394, 2001.

[100] HILFINGER, P., BONACHEA, D., DATTA, K., GAY, D., GRAHAM, S., LIBLIT, B., PIKE, G., SU, J., and YELICK, K., "Titanium Language Reference Manual," tech. rep., Computer Science Division (EECS), University of California, Berkeley, 2005.

[101] HOANG, M. and MITCHELL, J. C., "Lower bounds on type inference with subtypes," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 176–185, 1995.

[102] IGARASHI, A., PIERCE, B., and WADLER, P., "Featherweight Java: A Minimal Core Calculus for Java and GJ," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 132–146, 1999.

[103] IGARASHI, A. and VIROLI, M., "On variance-based subtyping for parametric types," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 441–469, 2002.

[104] IGARASHI, A. and VIROLI, M., "Variant parametric types: A flexible subtyping scheme for generics," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 5, pp. 795–847, 2006.

[105] JAGANNATHAN, S. and WRIGHT, A. K., "Effective flow analysis for avoiding run-time checks," in *International Static Analysis Symposium (SAS)*, pp. 207–224, 1995.

[106] JONES, R. E. and LINS, R. D., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

[107] KENNEDY, A. and SYME, D., "Design and implementation of generics for the .NET common language runtime.," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–12, 2001.

[108] KFOURY, A. J., TIURYN, J., and URZYCZYN, P., "Type reconstruction in the presence of polymorphic recursion," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 2, pp. 290–311, 1993.

[109] KOZEN, D., PALSBERG, J., and SCHWARTZBACH, M. I., "Efficient inference of partial types," *Journal of Computer and System Sciences*, vol. 49, no. 2, pp. 306–324, 1994.

[110] KOZEN, D., PALSBERG, J., and SCHWARTZBACH, M. I., "Efficient recursive subtyping," *Mathematical Structures in Computer Science*, vol. 5, no. 1, pp. 113–125, 1995.

[111] KUNCAK, V. and RINARD, M., "Structural subtyping of non-recursive types is decidable," in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 96–107, 2003.

[112] LATTNER, C. and ADVE, V., "Automatic pool allocation: Improving performance by controlling data structure layout in the heap," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 129–142, 2005.

[113] LEE, O., YANG, H., and YI, K., "Inserting safe memory reuse commands into ml-like programs," in *International Static Analysis Symposium (SAS)*, pp. 171–188, 2003.

[114] LINCOLN, P. and MITCHELL, J. C., "Algorithmic aspects of type inference with subtypes," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 293–304, 1992.

[115] LISKOV, B., "Keynote address - data abstraction and hierarchy," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 17–34, ACM, 1987.

[116] LISKOW, B. and WING, J. M., "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994.

[117] LIVSHITS, V. B., WHALEY, J., and LAM, M. S., "Reflection analysis for java," in *Asian Symposium on Programming Languages and Systems (APLAS)*, pp. 139–160, 2005.

[118] LOGOZZO, F., *Modular Static Analysis of Object-Oriented Languages*. PhD thesis, Ecole Polytechnique, France, 2004.

[119] MAKHOLM, H., *A language-independent framework for region inference*. PhD thesis, University of Copenhagen, 2003.

[120] MAZURAK, K. and ZDANCEWIC, S., "Type inference for Java 5: Wildcards, F-Bounds, and Undecidability," 2006. A note available at http://www.cis.upenn.edu/~stevez/note.html.

[121] MILNER, R., "A theory of type polymorphism," *Journal Computer and System Science*, pp. 348–375, 1978.

[122] MITCHELL, J. C., "Type inference with simple subtypes," *Journal of Functional Programming*, vol. 1, no. 3, pp. 245–285, 1991.

[123] MITCHELL, J. C. and PLOTKIN, G. D., "Abstract types have existential type.," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 3, pp. 470–502, 1988.

[124] MORRISETT, G., *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[125] MORRISETT, J. G., FELLEISEN, M., and HARPER, R., "Abstract Models of Memory Management," in *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 66–77, 1995.

[126] MOSSIN, C., *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, 1997.

[127] MYERS, A. C., BANK, J. A., and LISKOV, B., "Parameterized types for java," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 132–145, 1997.

[128] NAFTALIN, M. and WADLER, P., *Java Generics and Collections*. OReilly, 2006.

[129] NIEHREN, J. and PRIESNITZ, T., "Entailment of non-structural subtype constraints," in *Asian Computing Science Conference (ASIAN)*, pp. 251–265, 1999.

[130] NIEHREN, J. and PRIESNITZ, T., "Non-structural subtype entailment in automata theory," in *Theoretical Aspects of Computer Software (TACS)*, pp. 360–384, 2001.

[131] NIEHREN, J., PRIESNITZ, T., and SU, Z., "Complexity of subtype satisfiability over posets," in *European Symposium on Programming (ESOP)*, pp. 357–373, 2005.

[132] NIELSON, F., NIELSON, H., and HANKIN, C., *Principles of Program Analysis.* Springer-Verlag, 1999.

[133] NISS, H., *Regions are imperative. Unscoped regions and control-sensitive memory management.* PhD thesis, University of Copenhagen, 2002.

[134] NORDSTROM, B., PETERSSON, K., and SMITH, J. M., *Programming in Martin-Lof's Type Theory.* Oxford University Press, 1990.

[135] ODERSKY, M., SULZMANN, M., and WEHR, M., "Type inference with constrained types," *Theory and Practice of Object Systems*, vol. 5(1), pp. 35–55, 1999.

[136] ODERSKY, M. and WADLER, P., "Pizza into Java: Translating theory into practice," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 146–159, 1997.

[137] ODERSKY, M., ZENGER, C., and ZENGER, M., "Colored local type inference," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 41–53, 2001.

[138] ODERSKY, M. and ZENGER, M., "Scalable component abstractions," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 41–57, 2005.

[139] PALSBERG, J., "Closure analysis in constraint form," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 1, pp. 47–62, 1995.

[140] PALSBERG, J., "Equality-based flow analysis versus recursive types," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 6, pp. 1251–1264, 1998.

[141] PALSBERG, J., "Type-based analysis and applications," in *ACM Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pp. 20–27, 2001.

[142] PALSBERG, J. and O'KEEFE, P., "A type system equivalent to flow analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 4, pp. 576–599, 1995.

[143] PALSBERG, J. and PAVLOPOULOU, C., "From polyvariant flow information to intersection and union types," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 197–208, 1998.

[144] PALSBERG, J. and SCHWARTZBACH, M., *Object-oriented type systems.* John Wiley & Sons, New York, 1994.

[145] PALSBERG, J. and SCHWARTZBACH, M. I., "Type substitution for object-oriented programming," in *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pp. 151–160, 1990.

[146] PALSBERG, J. and SCHWARTZBACH, M. I., "Static typing for object-oriented programming," *Science of Computer Programming*, vol. 23, no. 1, pp. 19–53, 1994.

[147] PALSBERG, J. and SMITH, S. F., "Constrained types and their expressiveness," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 5, pp. 519–527, 1996.

[148] PALSBERG, J., WAND, M., and O'KEEFE, P., "Type inference with non-structural subtyping," *Formal Aspects of Computing*, vol. 9, no. 1, pp. 49–67, 1997.

[149] PALSBERG, J., ZHAO, T., and JIM, T., "Automatic discovery of covariant read-only fields," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 1, pp. 126–162, 2005.

[150] PEYTON JONES, S. and ET AL., "Glasgow Haskell Compiler." http://www.haskell.org/ghc.

[151] PIERCE, B., *Types and Programming Languages*. The MIT Press, 2002.

[152] PIERCE, B. C., "Bounded quantification is undecidable," *Information and Computation*, vol. 112, no. 1, pp. 131–165, 1994.

[153] PIERCE, B. C. and TURNER, D. N., "Simple type-theoretic foundations for object-oriented programming," *Journal of Functional Programming*, vol. 4, no. 2, pp. 207–247, 1994.

[154] PIERCE, B. C. and TURNER, D. N., "Local type inference.," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 252–265, 1998.

[155] PIZLO, F., FOX, J. M., HOLMES, D., and VITEK, J., "Real-time java scoped memory: Design patterns and semantics," in *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 101–110, 2004.

[156] PLEVYAK, J. and CHIEN, A. A., "Precise concrete type inference for object-oriented languages," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 324–340, 1994.

[157] POTTIER, F., "Simplifying subtyping constraints.," in *ACM International Conference on Functional Programming (ICFP)*, pp. 122–133, 1996.

[158] POTTIER, F., *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Universite Paris 7, 1998.

[159] PRATT, V. R. and TIURYN, J., "Satisfiability of inequalities in a poset," *Fundamenta Informaticae*, vol. 28, no. 1-2, pp. 165–182, 1996.

[160] QIAN, F. and HENDREN, L., "An adaptive, region-based allocator for java," in *International Symposium on Memory Management (ISMM)*, pp. 127 – 138, ACM Press, 2002.

[161] REHOF, J. and FÄHNDRICH, M., "Type-based flow analysis: From polymorphic subtyping to cfl reachability," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 54–66, 2001.

[162] REHOF, J., "The complexity of simple subtyping systems," Ph.D Thesis, DIKU, 1998.

[163] REHOF, J. and MOGENSEN, T., "Tractable constraints in finite semilattices," *Science of Computer Programming*, vol. 35, no. 2, pp. 191–221, 1999.

[164] REPS, T. W., HORWITZ, S., and SAGIV, S., "Precise interprocedural dataflow analysis via graph reachability," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 49–61, 1995.

[165] REYNOLDS, J. C., “Automatic computation of data set definitions,” in *International Federation for Information Processing Congress (1)*, pp. 456–461, 1968.

[166] RICE, H. G., “Classes of recursively enumerable set and their decision problems,” *Transactions of the American Mathematical Society*, no. 74, pp. 358–366, 1953.

[167] ROSS, D. T., “The AED free storage package,” *Communications of the ACM*, vol. 10, no. 8, pp. 481–492, 1967.

[168] SAGIV, S., REPS, T. W., and HORWITZ, S., “Precise interprocedural dataflow analysis with applications to constant propagation,” in *Theory and Practice of Software Development*, pp. 651–665, 1995.

[169] SALAGNAC, G., NAKHLI, C., RIPPERT, C., and YOVINE, S., “Efficient region-based memory management for resource-limited real-time embedded systems,” in *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, p. 8, IEEE Computer Society, 2006.

[170] SALAGNAC, G., RIPPERT, C., and YOVINE, S., “Semi-automatic region-based memory management for real-time java embedded systems,” in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 73–80, IEEE Computer Society, 2007.

[171] SCHWARTZ, J. T., “Optimization of very high level languages - part i and part ii,” *Computer Languages*, 1976.

[172] SESTOFT, P., *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, 1991.

[173] SESTOFT, P., “Replacing function parameters by global variables,” in *ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 39–53, 1989.

[174] SHIVERS, O., “Control flow analysis in scheme,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 164–174, 1988.

[175] SHIVERS, O., *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[176] SMITH, S. F. and WANG, T., “Polyvariant flow analysis with constrained types,” in *European Symposium on Programming (ESOP)*, pp. 382–396, 2000.

[177] SOLBERG, K. L., *Annotated Type Systems for Program Analysis*. PhD thesis, Aarhus University, Denmark, 1995.

[178] STEFAN, A., CRACIUN, F., and CHIN, W.-N., “A flow-sensitive region inference for cli,” in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[179] STEFFEN, M., *Polarized Higher-Order Subtyping*. PhD thesis, Universitat Erlangen-Nurnberg, 1997.

[180] SU, Z., FÄHNDRICH, M., and AIKEN, A., “Projection merging: Reducing redundancies in inclusion constraint graphs,” in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 81–95, 2000.

[181] SU, Z., “Algorithms for and the complexity of constraint entailment,” PhD thesis, University of California, Berkeley, 2002.

[182] SU, Z. and AIKEN, A., "Entailment with conditional equality constraints," in *European Symposium on Programming (ESOP)*, pp. 170–189, 2001.

[183] SU, Z., AIKEN, A., NIEHREN, J., PRIESNITZ, T., and TREINEN, R., "The first-order theory of subtyping constraints.," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 203–216, 2002.

[184] TALPIN, J.-P. and JOUVELOT, P., "Polymorphic Type, Region and Effect Inference," *Journal of Functional Programming*, vol. 2, no. 3, pp. 245–271, 1992.

[185] TERAUCHI, T. and AIKEN, A., "Memory Management with Use-Counted Regions," technical report ucb//csd-04-1314, University of California, Berkeley, 2004.

[186] THORUP, K. K. and TORGERSEN, M., "Unifying genericity - combining the benefits of virtual types and parameterized classes.," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 186–204, 1999.

[187] TIURYN, J., "Subtype inequalities," in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 308–315, 1992.

[188] TIURYN, J. and WAND, M., "Type reconstruction with recursive types and atomic subtyping," in *Theory and Practice of Software Development*, pp. 686–701, 1993.

[189] TOFTE, M. and BIRKEDAL, L., "A region inference algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 4, pp. 734–767, 1998.

[190] TOFTE, M., BIRKEDAL, L., ELSMAN, M., HALLENBERG, N., OLESEN, T., and SESTOFT, P., *Programming with Regions in the ML Kit (for Version 4).* The IT University of Copenhagen, 2001.

[191] TOFTE, M. and TALPIN, J., "Implementing the Call-By-Value λ-calculus Using a Stack of Regions," in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 188–201, 1994.

[192] TOFTE, M. and TALPIN, J., "Region-based memory management," *Information and Computation*, vol. 132, no. 2, pp. 109–176, 1997.

[193] TORGERSEN, M., ERNST, E., HANSEN, C. P., VON DER AHE, P., BRACHA, G., and GAFTER, N., "Adding Wildcards to the Java Programming Language," *Journal of Object Technology*, vol. 3, no. 11, pp. 97–116, 2004.

[194] TORGERSEN, M., ERNST, E., and HANSEN, C. P., "Wild FJ," in *Foundations of Object-Oriented Languages*, pp. 1–12, 2005.

[195] TRIFONOV, V. and SMITH, S. F., "Subtyping constrained types.," in *International Static Analysis Symposium (SAS)*, pp. 349–365, 1996.

[196] TSCHANTZ, M. S. and ERNST, M. D., "Javari: Adding reference immutability to Java," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 211–230, 2005.

[197] VIROLI, M. and NATALI, A., "Parametric polymorphism in java: an approach to translation based on reflective features.," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 146–165, 2000.

[198] VON DINCKLAGE, D. and DIWAN, A., "Converting Java classes to use generics.," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1–14, 2004.

[199] WALKER, D. and WATKINS, K., "On regions and linear types (extended abstract)," in *ACM International Conference on Functional Programming (ICFP)*, pp. 181–192, ACM Press, 2001.

[200] WANG, T. and SMITH, S. F., "Precise constraint-based type inference for Java," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 99–117, 2001.

[201] WELLS, J. B., "Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable," in *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 176–185, 1994.

[202] WHALEY, J. and RINARD, M., "Compositional pointer and escape analysis for java programs," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 187–206, 1999.

[203] WILSON, P. R., "Uniprocessor garbage collection techniques," in *International Workshop on Memory Management (IWMM)*, pp. 1–42, 1992.

[204] WRIGHT, A. K. and FELLEISEN, M., "A Syntactic Approach to Type Soundness," *Information Computation*, vol. 115, no. 1, pp. 38–94, 1994.

[205] ZHAO, T., NOBLE, J., and VITEK, J., "Scoped Types for Real-Time Java," in *IEEE Real-Time Systems Symposium (RTSS)*, pp. 241–251, 2004.

APPENDIX A

REGION-BASED MEMORY MANAGEMENT

A.1 Dynamic Semantics of Region-Annotated Core-Java

The dynamic rules are listed in Figure A.1 and Figure A.2. Note that the evaluation rules yield two kinds of errors: `nullerr` due to a null pointer access and `danglingerr` due to a possible dangling reference creation. In the rules [D−ASSGN2], [D−ASSGN3], and [D−LOOP2] the result () denotes the singleton value of type **void**. Note that the type **void** is assumed to be isomorphic to type **unit**. In rule [D−EB], the locally declared variable is assigned, with the help of the function **init**, an initial value according to its type as follows:

$$\begin{array}{lll} \textbf{init}(t) & =_{def} & \textit{case t of} \\ \quad \textit{boolean} & \rightarrow & \textit{false} \\ \quad \textit{int} & \rightarrow & \textit{0} \\ \quad \textit{cn}\langle r_{1..n}\rangle & \rightarrow & \texttt{null} \end{array}$$

There are five rules which use runtime checks to verify a possible creation of a dangling reference: [D−ASSGN2], [D−ASSGN3], [D−NEW], [D−RETR2], and [D−INVOKE]. The corresponding rules [D−ASSGN2−DANGLERR], [D−ASSGN3−DANGLERR], [D−NEW−DANGLERR], [D−RETR2−DANGLERR], and [D−INVOKE−DANGLERR] generate a `danglingerr` error due to the failure of the runtime checks.

Rule [D−ASSGN2] checks whether a location assigned to a variable is live, namely its region is in the current store. Rule [D−ASSGN3] checks whether the region r_1 of the new value δ outlives the expected region for the object field f. The function **fieldregion**($cn\langle a^*\rangle$,f) returns the region where the object field f is expected to be stored. Rule [D−NEW] checks whether the class invariant holds, **ord**(ϖ)$\Rightarrow\varphi_{inv}$ (mainly whether the fields regions $r_{i:2..n}$ outlive the region r_1 of the object). The initial value of a field is also checked to be in a region that outlives the expected region of that field $r'_i\succeq$**fieldregion**($cn\langle r_{1..n}\rangle, f_i$). The function **fieldlist**($cn\langle r_{1..n}\rangle$) returns all fields of cn and their region types according to regions $r_{1..n}$.

Rule [D−RETR2] checks whether the region a is on the top of the store stack. Then it checks whether a reference to a does not escape neither through the value result δ, nor through the program variable environment Π, nor through the object values of the store ϖ. When a new

$$\left[\underline{\textbf{D-VAR}}\right] \quad \frac{v \in \mathit{dom}(\Pi)}{\langle\varpi,\Pi\rangle[v]\hookrightarrow\langle\varpi,\Pi\rangle[\Pi(v)]}$$

$$\left[\underline{\textbf{D-FD}}\right] \quad \frac{\Pi(v)=(r,o) \quad \varpi=\varpi_1[r\mapsto \mathit{Rgn}]\varpi_2 \quad \mathit{Rgn}(o)=\mathit{cn}\langle a^+\rangle(V)}{\langle\varpi,\Pi\rangle[v.f]\hookrightarrow\langle\varpi,\Pi\rangle[V(f)]}$$

$$\left[\underline{\textbf{D-ASSGN1}}\right] \quad \frac{\langle\varpi,\Pi\rangle[e]\hookrightarrow\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathit{lhs}=e]\hookrightarrow\langle\varpi',\Pi'\rangle[\mathit{lhs}=e']}$$

$$\left[\underline{\textbf{D-ASSGN2}}\right] \quad \frac{\begin{array}{c} v\in\mathit{dom}(\Pi) \quad \Pi'=\Pi+\{v\mapsto\delta\} \\ \delta=(r_1,o_1)\ \wedge\ r_1\in\mathit{dom}(\varpi)\end{array}}{\langle\varpi,\Pi\rangle[v=\delta]\hookrightarrow\langle\varpi,\Pi'\rangle[()]}$$

$$\left[\underline{\textbf{D-ASSGN2-DANGLERR}}\right] \quad \frac{\begin{array}{c} v\in\mathit{dom}(\Pi) \\ \delta=(r_1,o_1)\ \wedge\ r_1\notin\mathit{dom}(\varpi)\end{array}}{\langle\varpi,\Pi\rangle[v=\delta]\hookrightarrow\texttt{danglingerr}}$$

$$\left[\underline{\textbf{D-ASSGN3}}\right] \quad \frac{\begin{array}{c} \Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto \mathit{Rgn}]\varpi_2 \quad \mathit{Rgn}(o)=\mathit{cn}\langle a^+\rangle(V) \\ \mathit{Rgn}'=\mathit{Rgn}+\{o\mapsto \mathit{cn}\langle a^+\rangle(V+\{f\mapsto\delta\})\}\ \varpi'=\varpi_1[a\mapsto \mathit{Rgn}']\varpi_2 \\ \delta=(r_1,o_1)\ \wedge\ \textbf{ord}(\varpi)\Rightarrow(r_1\succeq\textbf{fieldregion}(\mathit{cn}\langle a^+\rangle,f))\end{array}}{\langle\varpi,\Pi\rangle[v.f=\delta]\hookrightarrow\langle\varpi',\Pi\rangle[()]}$$

$$\left[\underline{\textbf{D-ASSGN3-DANGLERR}}\right] \quad \frac{\begin{array}{c} \Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto \mathit{Rgn}]\varpi_2 \quad \mathit{Rgn}(o)=\mathit{cn}\langle a^+\rangle(V) \\ \delta=(r_1,o_1)\ \wedge\ \neg\,(\textbf{ord}(\varpi)\Rightarrow(r_1\succeq\textbf{fieldregion}(\mathit{cn}\langle a^+\rangle,f)))\end{array}}{\langle\varpi,\Pi\rangle[v.f=\delta]\hookrightarrow\texttt{danglingerr}}$$

$$\left[\underline{\textbf{D-NEW}}\right] \quad \frac{\begin{array}{c} \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ c\langle...\rangle\ \textbf{where}\ \varphi_{\mathit{inv}}\ \{...\}\ \in\ P \\ \textbf{ord}(\varpi)\Rightarrow\varphi_{\mathit{inv}} \\ \varpi=\varpi_1[r_1\mapsto \mathit{Rgn}]\varpi_2 \quad V=\{f_1\mapsto\Pi(v_1),...,f_p\mapsto\Pi(v_p)\} \quad \textbf{fieldlist}(\mathit{cn}\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p} \\ \mathit{if}\ \Pi(v_i)=(r'_i,o'_i)\ \mathit{then}\ \textbf{ord}(\varpi)\Rightarrow(r'_i\succeq\textbf{fieldregion}(\mathit{cn}\langle r_{1..n}\rangle,f_i)) \quad i=1..p \\ o\notin\mathit{dom}(\mathit{Rgn}) \quad \mathit{Rgn}'=\mathit{Rgn}+\{o\mapsto \mathit{cn}\langle r_{1..n}\rangle(V)\} \quad \varpi'=\varpi_1[r_1\mapsto \mathit{Rgn}']\varpi_2\end{array}}{\langle\varpi,\Pi\rangle[\textbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_{1..p})]\hookrightarrow\langle\varpi',\Pi\rangle[(r_1,o)]}$$

$$\left[\underline{\textbf{D-NEW-DANGLERR}}\right] \quad \frac{\begin{array}{c} \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ c\langle...\rangle\ \textbf{where}\ \varphi_{\mathit{inv}}\ \{...\}\ \in\ P \\ V=\{f_1\mapsto\Pi(v_1),...,f_p\mapsto\Pi(v_p)\} \quad \textbf{fieldlist}(\mathit{cn}\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p} \\ \neg(\textbf{ord}(\varpi)\Rightarrow\varphi_{\mathit{inv}})\ \vee\ (\exists i\in\{1..p\}\cdot\Pi(v_i)=(r'_i,o'_i)\ \wedge \\ \qquad \neg(\textbf{ord}(\varpi)\Rightarrow(r'_i\succeq\textbf{fieldregion}(\mathit{cn}\langle r_{1..n}\rangle,f_i)))\end{array}}{\langle\varpi,\Pi\rangle[\textbf{new}\ \mathit{cn}\langle r_{1..n}\rangle(v_{1..p})]\hookrightarrow\texttt{danglingerr}}$$

$$\left[\underline{\textbf{D-INVOKE}}\right] \quad \frac{\begin{array}{c} \{a^+,a'^+\}\subset\textbf{dom}(\varpi) \\ \Pi(v'_0)=(a_1,o) \quad \varpi(a_1)(o)=\mathit{cn}\langle a^+\rangle(V) \\ (t_0\ \mathit{mn}\langle a^+r'^+\rangle((t\,v)_{1..p})\textbf{where}\ \varphi\ \{e\})\in \mathit{cn}\langle a^+\rangle \\ n_i=\mathit{fresh}()\ \ i=0..p \quad \rho=[r'^+\mapsto a'^+] \\ \Pi'=\Pi+\{n_i\mapsto\Pi(v'_i)_{i:0..p}\} \\ e'=\texttt{ret}(n_0,..\texttt{ret}(n_p,[\mathit{this}\mapsto n_0][v_i\mapsto n_i]_{i:1}^{p}\rho e))\end{array}}{\langle\varpi,\Pi\rangle[v'_0.\mathit{mn}\langle a^+a'^+\rangle(v'_{1..p})]\hookrightarrow\langle\varpi,\Pi'\rangle[e']}$$

$$\left[\underline{\textbf{D-INVOKE-DANGLERR}}\right] \quad \frac{\neg(r^+\in\textbf{dom}(\varpi))}{\langle\varpi,\Pi\rangle[v.\mathit{mn}\langle r^+\rangle(v^*)]\hookrightarrow\texttt{danglingerr}}$$

Figure A.1: Dynamic Semantics for Region-Annotated Core-Java: Part I

[D−EB]

$$\frac{n{=}\mathit{fresh}()\ \ \Pi'{=}\Pi{+}\{(n{\mapsto}\mathbf{init}(t))\}\ \ e'{=}\mathtt{ret}(n,e)}{\langle\varpi,\Pi\rangle[\{(t\,v)\ e\}]\hookrightarrow\langle\varpi,\Pi'\rangle[e']}$$

[D−RET1]

$$\frac{\langle\varpi,\Pi\rangle[e]\hookrightarrow\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,e)]\hookrightarrow\langle\varpi',\Pi'\rangle[\mathtt{ret}(v,e')]}$$

[D−RET2]

$$\frac{}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,\delta)]\hookrightarrow\langle\varpi,\Pi{-}\{v\}\rangle[\delta]}$$

[D−LETR]

$$\frac{a{=}\mathit{fresh}()}{\langle\varpi,\Pi\rangle[\mathbf{letreg}\ r\ \mathbf{in}\ e]\hookrightarrow\langle[a{\mapsto}\emptyset]\varpi,\Pi\rangle[\mathtt{retr}(a,[r{\mapsto}a]e)]}$$

[D−RETR1]

$$\frac{\langle\varpi,\Pi\rangle[e]\hookrightarrow\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{retr}(a,e)]\hookrightarrow\langle\varpi',\Pi'\rangle[\mathtt{retr}(a,e')]}$$

[D−RETR2]

$$\frac{\begin{array}{c}(\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}\mathit{cn}\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi)\wedge\\ \forall f\in \mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a,\delta)]\hookrightarrow\langle\varpi,\Pi\rangle[\delta]}$$

[D−RETR2−DANGLERR]

$$\frac{\begin{array}{c}\neg(a{=}a_1)\vee\\ \neg((\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi)))\vee\neg((\forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))))\\ \vee\neg(\forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}\mathit{cn}\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi)\wedge\\ \forall f\in \mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi)))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a_1,\delta)]\hookrightarrow\mathtt{danglingerr}}$$

[D−IF1]

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\hookrightarrow\langle\varpi,\Pi\rangle[e_1]}$$

[D−IF2]

$$\frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\hookrightarrow\langle\varpi,\Pi\rangle[e_2]}$$

[D−LOOP1]

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]\hookrightarrow\langle\varpi,\Pi\rangle[e\,\mathbf{;}\,\mathbf{while}\ v\ e]}$$

[D−LOOP2]

$$\frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]\hookrightarrow\langle\varpi,\Pi\rangle[()]}$$

[D−SEQ1]

$$\frac{\langle\varpi,\Pi\rangle[e_1]\hookrightarrow\langle\varpi',\Pi'\rangle[e_1']}{\langle\varpi,\Pi\rangle[e_1\,\mathbf{;}\,e_2]\hookrightarrow\langle\varpi',\Pi'\rangle[e_1'\,\mathbf{;}\,e_2]}$$

[D−SEQ2]

$$\frac{}{\langle\varpi,\Pi\rangle[\delta_1\,\mathbf{;}\,e_2]\hookrightarrow\langle\varpi,\Pi\rangle[e_2]}$$

[D−NULLERR1]

$$\frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f]\hookrightarrow\mathtt{nullerr}}$$

[D−NULLERR2]

$$\frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f=\delta]\hookrightarrow\mathtt{nullerr}}$$

[D−NULLERR3]

$$\frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.mn\langle a^*\rangle(u^*)]\hookrightarrow\mathtt{nullerr}}$$

Figure A.2: Dynamic Semantics for Region-Annotated Core-Java: Part II

region is allocated, in rule [D−LETR], a fresh region name is used in order to avoid region name duplication in the store.

Rule [D−INVOKE] checks whether the method's region arguments are in the current store and then prepares the variable environment for the method's body execution.

A.2 Proof Details

A.2.1 Auxiliary Definitions and Lemmas

Lemma A.2.1.1. *Suppose* $P;\Gamma;R;\varphi;\Sigma \vdash e : t$. *If* $\rho = [(r_i \mapsto a_i)_{1..p}]$, *and for all* $i{=}1..p$, *either* $a_i \notin R$ *or* $\varphi{\Rightarrow}(r_i{=}a_i)$, *then* $P;\rho\Gamma;\rho R;\rho\varphi;\rho\Sigma \vdash \rho e : \rho t$.

Proof: By structural induction on e.

Lemma A.2.1.2. *Suppose* $\varphi_1 \Rightarrow \varphi_2$.

If $a \notin reg(\varphi_1 \wedge \varphi_2)$, *then*

1. $(r{\succeq}a \wedge \varphi_1) \Rightarrow (r{\succeq}a \wedge \varphi_2)$.

2. $(a{\succeq}r \wedge \varphi_1) \Rightarrow (a{\succeq}r \wedge \varphi_2)$.

Proof: By induction on the form of a region constraint.

Definition A.2.1.1. *Given a region constraint* $\varphi{=} \bigwedge_i (r_i{\succeq}r'_i) \wedge \bigwedge_j (r_j{=}r'_j)$ *and a set of regions* R, *we define the following notations:*

$r{\succeq}r' \in \varphi$ *iff* $\exists i$ *such that* $r{=}r_i \wedge r'{=}r'_i$

$r{=}r' \in \varphi$ *iff* $\exists j$ *such that* $r{=}r_j \wedge r'{=}r'_j$

$\mathit{TransClosure}(\varphi) =_{def} \bigwedge_{i'} (r_{i'}{\succeq}r'_{i'}) \wedge \bigwedge_{j'} (r_{j'}{=}r'_{j'})$ *such that*

$i'{\geq}i \wedge j'{\geq}j$

$\forall r_1,r_2,r_3 \cdot r_1{\succeq}r_2{\in}\mathit{TransClosure}(\varphi){\wedge}r_3{\succeq}r_1{\in}\mathit{TransClosure}(\varphi){\Rightarrow}r_3{\succeq}r_2{\in}\mathit{TransClosure}(\varphi)$

$\forall r_1,r_2,r_3 \cdot r_1{=}r_2{\in}\mathit{TransClosure}(\varphi){\wedge}r_3{=}r_1{\in}\mathit{TransClosure}(\varphi){\Rightarrow}r_3{=}r_2{\in}\mathit{TransClosure}(\varphi)$

$\varphi{-}R$ *or* $\varphi\backslash R =_{def} \varphi'$ *such that*

$\forall r{\succeq}r' \in \mathit{TransClosure}(\varphi) \wedge r{\notin}R \wedge r'{\notin}R \Rightarrow r{\succeq}r' \in \varphi'$

$\forall r{=}r' \in \mathit{TransClosure}(\varphi) \wedge r{\notin}R \wedge r'{\notin}R \Rightarrow r{=}r' \in \varphi'$

Lemma A.2.1.3. *Suppose* $\varphi_1 \Rightarrow \varphi_2$ *and a region* r.

1. $\varphi_1{-}\{r\} \Rightarrow \varphi_2{-}\{r\}$.

2. *If $r \notin reg(\varphi_2)$, then $\varphi_1 - \{r\} \Rightarrow \varphi_2$.*

Proof: By case analysis on φ_1 and φ_2.

Lemma A.2.1.4. *Suppose $P; R; \varphi \vdash_{type} t$.*

1. *If $r \notin R$, then $P; R \cup \{r\}; \varphi \vdash_{type} t$.*

2. *If $\varphi' \Rightarrow \varphi$, then $P; R; \varphi' \vdash_{type} t$.*

Proof: By structural induction on the $\vdash_{type}$ derivation.

Lemma A.2.1.5. *Suppose $P; R; \varphi \vdash t_1 <: t_2$.*

1. *If $r \notin R$, then $P; R \cup \{r\}; \varphi \vdash t_1 <: t_2$.*

2. *If $\varphi' \Rightarrow \varphi$, then $P; R; \varphi' \vdash t_1 <: t_2$.*

3. *If $r \in R$, $r \notin reg(t_1)$, and $r \notin reg(t_2)$, then $P; R - \{r\}; \varphi - \{r\} \vdash t_1 <: t_2$.*

Proof: By structural induction on the subtyping derivation using the Lemma A.2.1.4.

Lemma A.2.1.6. *Suppose $P; \Gamma; R; \varphi; \Sigma \vdash e : t$.*

1. *If $v \notin dom(\Gamma)$, then $P; \Gamma + (v : t_1); R; \varphi; \Sigma \vdash e : t$.*

2. *If $\varphi' \Rightarrow \varphi$, then $P; \Gamma; R; \varphi'; \Sigma \vdash e : t$.*

3. *If $r \notin R$, then $P; \Gamma; R \cup \{r\}; \varphi; \Sigma + r \vdash e : t$.*

4. *If $(r, o) \notin \Sigma$ and $r \in R$, then $P; \Gamma; R; \varphi; \Sigma + ((r, o) : t_1) \vdash e : t$.*

Proof: By structural induction on e.

Lemma A.2.1.7. *Suppose $P; \Gamma; R; \varphi; \Sigma \vdash e : t$.*

1. *If $v \in dom(\Gamma)$ and $v \notin vars(e)$, then $P; \Gamma - \{v\}; R; \varphi; \Sigma \vdash e : t$.*

   where $vars(e)$ is defined by Definition 3.7.1.3.

Proof: By structural induction on e.

Lemma A.2.1.8. *Suppose* $P;\Gamma;R\cup\{a\};\varphi;\Sigma \vdash e : t$.

1. *If* $a \notin reg(\Gamma)$, *and* $a \notin regs(e)$, *then* $P;\Gamma;R;\varphi{-}a;\Sigma{-}a \vdash e : t$.

   where $regs(e)$ *is defined by Definition 3.7.1.5.*

Proof: By structural induction on e.

Lemma A.2.1.9. *Suppose an expression* e.

1. *If* $retvars(e){=}\emptyset$ *and* $retregs(e){=}\emptyset$ *then* $valid(e)$ *holds.*
2. *If* $retvars(e){=}\emptyset$ *then* $lvar(e){=}\emptyset$.
3. *If* $retregs(e){=}\emptyset$ *then* $lreg(e){=}\emptyset$.

Proof: By structural induction on e.

Lemma A.2.1.10. *(Canonical Forms)*

Suppose $P;\Gamma;R;\varphi;\Sigma \vdash \delta : t$ *and* $\Gamma, R, \varphi, \Sigma \models \langle\varpi, \Pi\rangle$. *Then:*

1. *if* $t = void$ *then* $\delta = ()$.
2. *if* $t = boolean$ *then either* $\delta = true$ *or* $\delta = false$.
3. *if* $t = int$ *then* $\delta = i$ *for some integer* i.
4. *if* $t = \bot$ *then* $\delta = \texttt{null}$.
5. *if* $t = cn\langle r_{1..n}\rangle$ *then*
   - *either the value is a location,* $\delta = (r_1, o)$. *The content of that location is an object value* $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$ *that is well-typed such that* $P;\Gamma;R;\varphi;\Sigma{\vdash}cn\langle r_{1..n}\rangle(V){:}cn\langle r_{1..n}\rangle$ *(it contains the fields and the methods of the class* cn *according to the program* P*).*
   - *or the value is* $\delta = \texttt{null}$

Proof: By the definition of values and inspection of type checking rules.

Lemma A.2.1.11. *Given any source language Core-Java program, P.*

Suppose $\vdash P \Rrightarrow P'$.

1. *If* $\tau \neq$ *Object and* $\vdash \tau \Rrightarrow t, \varphi$, *then* $reg(t) \subseteq reg(\varphi)$.
2. *Given any* $t \in P'$ *and* $t' \in P'$.

   If $\vdash t <: t' \Rrightarrow \varphi$, *then* $(reg(t) \cup reg(t')) \subseteq reg(\varphi)$.
3. *Given any source language Core-Java expression,* e.

   If $\Gamma \vdash e \Rrightarrow e' : t,\ \varphi$, *then* $reg(t) \subseteq (regs(e') \cup reg(\Gamma) \cup reg(\varphi))$.

Proof:

1. Since φ is the region class invariant of a class type. By induction on the inference rules [RI−CLASS−1] and [RI−CLASS−2] we can prove that the region class invariant always contain all the regions of a region type. An exception is the region type Object$\langle r\rangle$ since its invariant is just *true*.
2. Using the case (1) we can prove the conclusion for all t and t' such that $t \neq$ Object$\langle r\rangle$ and $t' \neq$ Object$\langle r\rangle$. However the first region of a region type is always used by the region constraint φ_0 of a region subtyping relation $\vdash t <: t', \varphi_0$ (see rules of Table 3.6). Since the exceptional case Object$\langle r\rangle$ contains only one region, the conclusion is proved.
3. By structural induction on *e*. The proof is straightforward by inspection of the type inference rules.

A.2.2 Proof of Theorem 3.7.2.1 (Subject Reduction)

By structural induction on *e*.

Case: v

We let $\Sigma' = \Sigma$, $\Gamma' = \Gamma$, $R' = R$, $\varphi' = \varphi$. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the consistency relation of the hypothesis, as $\Pi(v)$ and v have the same type $\Gamma(v)$. The validity is straightforward proved.

Case: $v.f$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. From the operational semantics $\Pi(v) = (r, o)$. By the consistency relation of the hypothesis (r, o), v, and $\varpi(r)(o)$ have the same type $\Sigma(r)(o){=}cn\langle a^+\rangle$. Note that the type of a location is the type of its content (by the rule
[RC−LOCATION]). Using the hypothesis of type rule [RC−OBJ] for $\varpi(r)(o)$, we prove that the type of $V(f)$ is a subtype of the type of $v.f$. By subsumption the type judgment is proved. The validity is straightforward proved.

Case: $v = \delta$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. The type judgment is trivial as the type remains **void** as before the evaluation. We only have to prove the consistency relation for the updated variable environment Π'. By the type rule of the hypothesis the type of $\delta{=}\Pi'(v)$ is a subtype of the type of v. Using subsumption, we prove that v and $\Pi'(v)$ have the same type. The validity is straightforward proved.

Case: $v.f = \delta$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$.

The update on ϖ preserves the consistency relation except the object value $\varpi(r)(o)$. By the type rule of the hypothesis the type of value $\delta = (r_1, o_1)$ is a subtype of the type of the field $v.f$. Thus the type of $V(f)$ after updating is a subtype of the type of $v.f$. By the consistency relation of the hypothesis for the object value $\varpi(r)(o)$ before updating combined with the previous subtyping relation for the updated field $V(f)$ we can prove that object value after updating is still well typed. The type judgment is trivial as the type remains **void** as before the evaluation. The validity is straightforward proved.

Case: $\delta\,;e_2$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. By the validity from the hypothesis, $valid(\delta\,;e_2)$ we get that $retvars(e_2){=}\emptyset$ and $retregs(e_2){=}\emptyset$. Applying case (1), (2) and (3) of
Lemma A.2.1.9 we prove that $valid(e_2)$, $lvar(e_2){=}\emptyset$, and $lreg(e_2){=}\emptyset$. Note that $lloc(\delta) = \emptyset$.
Thus the validity relation of the conclusion and the conclusion's relations between Γ and

Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis.

Case: **if** v **then** e_1 **else** e_2

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. By the validity from the hypothesis, *valid*(**if** v **then** e_1 **else** e_2) we get that *retvars*(e_1)=$\emptyset$, *retregs*(e_1)=$\emptyset$, *retvars*(e_2)=$\emptyset$, and *retregs*(e_2)=$\emptyset$. Applying case (1), (2) and (3) of Lemma A.2.1.9 we prove that *valid*(e_1), *lvar*(e_1)=$\emptyset$, *lreg*(e_1)=$\emptyset$, *valid*(e_2), *lvar*(e_2)=$\emptyset$, and *lreg*(e_2)=$\emptyset$.

Note that *lloc*(**if** v **then** e_1 **else** e_2) $= \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis and the subsumption.

Case: **while** $v\ e$

We let $\Sigma' = \Sigma,\ \Gamma' = \Gamma,\ R' = R,\ \varphi' = \varphi$. By the validity from the hypothesis, *valid*(**while** $v\ e$) we get that *retvars*(e)=$\emptyset$, *retregs*(e)=$\emptyset$. Applying case (1), (2) and (3) of Lemma A.2.1.9 we prove that *valid*(e **; while** $v\ e$), *lvar*(e **; while** $v\ e$)=$\emptyset$, and *lreg*(e **; while** $v\ e$)=$\emptyset$. Note that *lloc*(**while** $v\ e$) $= \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. The consistency relation is straightforward as both the static environment and the runtime environment remain unchanged. The type judgment follows from the type judgment of the hypothesis. The second case of the loop evaluation rule (when the condition is false) is straightforward proved.

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

We let $\Sigma' = \Sigma + \{(r_1, o) : cn\langle r_{1..n}\rangle\},\ \Gamma' = \Gamma,\ \varphi' = \varphi,\ R' = R$.

Note that *lloc*(**new** $cn\langle r_{1..n}\rangle(v_{1..p})$)=$(r_1, o)$, while the functions *lvar* and *lreg* return $\emptyset$ for both **new** $cn\langle r_{1..n}\rangle(v_{1..p})$ and (r_1, o). The conclusion's type judgment is straightforward proved as the type of new location is given by the Σ'.

The store is extended with one more location (r_1, o) and

location_dom(ϖ')=*location_dom*(ϖ)$\cup\{(r_1, o)\}$=*location_dom*(Σ)$\cup\{(r_1, o)\}$=*location_dom*(Σ').

In order to prove the conclusion's consistency relation we use the case (4) of Lemma A.2.1.6 to extend the typing relations of the hypothesis and the hypothesis's consistency relation. The object value from the new location is proved to be well typed by reconstructing the hypotheses of the type rule [ObjVal] as follows: $P;R;\varphi \vdash_{type} cn\langle r_{1..n}\rangle$ is proved by the type rule [NEW] from hypothesis; by the evaluation rule $V(f_i) = \Pi(v_i)$, while from the hypothesis consistency relation $P;\Gamma;R;\varphi;\Sigma \vdash \Pi(v_i) : \Gamma(v_i)$; using the hypothesis type judgment [NEW] we get $t'_i = \Gamma(v_i)$ and $P;R;\varphi \vdash t'_i <: t_i$. The validity is straightforward proved,

Case: $\{(t\,v)\ e\}$

We let $\Sigma' = \Sigma$, $\Gamma' = \Gamma{+}(v : t)$, $\varphi' = \varphi$, and $R' = R$. By the validity from the hypothesis, $valid(\{(t\,v)\ e\})$ we get that $retvars(e){=}\emptyset$ and $retregs(e){=}\emptyset$. Applying the cases (1), (2) and (3) of Lemma A.2.1.9 we prove that $valid(\texttt{ret}(v,e))$, $lvar(\texttt{ret}(v,e)){=}\{v\}$, and $lreg(\texttt{ret}(v,e)){=}\emptyset$. Note that $lloc(\{(t\,v)\ e\}) = \emptyset$. By the hypothesis's type judgment [EB], the conclusion's type judgment [RET] is proved. We prove that $P;\Gamma;R;\varphi;\Sigma \vdash \Pi(v) : \Gamma(v)$ as follows: $\Pi(v) = \textbf{init}(t)$ and the type of $\textbf{init}(t)$ is a subtype of t, while $\Gamma(v) = t$. By the hypothesis's consistency relation we get that $reg(\Gamma){\subseteq}R$. By the hypothesis's type judgment [EB], we get that $P;R;\varphi \vdash_{type} t'$, that ensures $reg(t){\subseteq}R$. Thus $reg(\Gamma{+}(v : t)){\subseteq}R$. Then, by the consistency relation of the hypothesis and by the case (1) of Lemma A.2.1.6 the rest of the conclusion's consistency relation is proved.

Case: $\texttt{ret}(v,\delta)$

We let $\Sigma' = \Sigma$, $\varphi' = \varphi$, $R' = R$, and $\Gamma' = \Gamma{-}\{v\}$. Note that $lvar(\texttt{ret}(v,\delta)){=}\{v\}$, $lreg(\texttt{ret}(v,\delta)){=}\emptyset$, $lloc(\texttt{ret}(v,\delta)) = \emptyset$, $lvar(\delta){=}\emptyset$, and $lreg(\delta){=}\emptyset$. The valid relation $valid(\delta)$ holds. By the hypothesis's type judgment and consistency and the case (1) of Lemma A.2.1.7 (the variable v is not used neither in δ nor by object values) the type judgment and the consistency of the conclusion are straightforward proved.

Case: $\textbf{letreg}\, r\, \textbf{in}\, e$

We use Lemma A.2.1.1 for region substitutions. For simplicity, we consider that the region substitution is already done both for static and dynamic environment.

We let $R' = R \cup \{a\}$, $\Sigma' = \Sigma{+}a$, $\Gamma' = \Gamma$, and $\varphi' = (\varphi \wedge \bigwedge_{r \in R}(r \succeq a))$. By the hypothesis's valid relation $valid(\textbf{letreg}\, r\, \textbf{in}\, e)$ we get that $retvars(e){=}\emptyset$, $retregs(e){=}\emptyset$. Applying the cases

(1), (2) and (3) of Lemma A.2.1.9 we prove that $\mathit{valid}(\mathtt{retr}(a,e))$, $\mathit{lvar}(\mathtt{retr}(a,e))=\emptyset$, and $\mathit{lreg}(\mathtt{retr}(a,e))=\{a\}$. Note that $\mathit{lloc}(\textbf{letreg}\, r\, \textbf{in}\; e) = \emptyset$. Thus the validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are proved. We prove the conclusion's type judgment $P;\Gamma';R';\varphi';\Sigma'\vdash\mathtt{retr}(a,e):t$ as follows: $R_t = R$; by the hypothesis's consistency relation we get that $\mathit{reg}(\Gamma)\subseteq R$; by the hypothesis's type judgment [LETR] we prove that $\mathit{reg}(t) \subseteq R$ and $P;\Gamma;R';\varphi' \vdash e : t$; the entailment $\varphi'\Rightarrow \bigwedge_{r\in R}(r\succeq a)$ is straightforward. By the hypothesis's consistency relation, $\mathit{ord}(\varpi)\Rightarrow\varphi$ and Lemma A.2.1.2 we prove that $\mathit{ord}([a\mapsto\emptyset]\varpi)\Rightarrow\varphi'$. The remaining part of the conclusion's consistency follows directly from the hypothesis's consistency using the cases (3) and (4) of Lemma A.2.1.6.

Case: $\mathtt{retr}(a,\delta)$

We let $R' = R-\{a\}$, $\Gamma' = \Gamma$, $\varphi' = \varphi-a$, $\Sigma' = \Sigma-\{a\}$. The validity relation of the conclusion and the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. By the type judgment of the hypothesis and the case (1) of Lemma A.2.1.8 the type judgment of the conclusion is proved.

The consistency is proved as follows:

By the hypothesis's consistency and $\mathit{reg}(\Gamma)\subseteq R-\{a\}$ (from the hypothesis type judgment [RETR]) the type of each $v \in \Pi'$ does not contain the region a since that type is given by $\Gamma(v)$. But $\Pi(v)$ is either a location or a constant. Note that the type of a location always contains the region's location. Hence $\mathit{vars}(\Pi'(v))\subseteq R-\{a\}$. Thus we can apply the case (1) of Lemma A.2.1.8 to prove that Π is well typed when the region a is deallocated. Note that this means that there are not any references from the program variable environment to the deallocated region.

By the consistency and type judgment of the hypothesis, each object value of the store ϖ is well typed. By the type rule [ObjVal], an object value can have references only to regions older that the region of the current location. We can use the case (1) of Lemma A.2.1.8 to type the store ϖ. By the consistency of the hypothesis and Lemma A.2.1.3 we can prove that $\mathit{ord}(\varpi)\Rightarrow\varphi'$. As we mentioned before, the hypothesis's type judgment [RETR] ensures that $\mathit{reg}(\Gamma)\subseteq R-\{a\}$.

Case: $v'_0.mn\langle a^+\rangle(v'_{1..p})$

According to the evaluation rule ([D−INVOKE]), the actual type of the object stored at the location given by $\Pi(v'_0)$ is $cn\langle a_{1..n}\rangle$, which is a subtype of the expected type of v'_0, say $cn'\langle a_{1..m}\rangle$ at compile time.

We let $R' = R$, $\Gamma' = \Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\}$, $\varphi' = \varphi \wedge inv.cn\langle a_{1..n}\rangle$, and $\Sigma' = \Sigma$.

Since P is a valid program, the method's body e is valid. Hence we get that $valid(ret(n_{0..p}, e))$, $lreg(ret(n_{0..p}, e)) = \emptyset$ and $lvar(ret(n_{0..p}, e)) = \{n_{0..p}\}$.

Note that $lloc(ret(n_{0..p}, e)) = \emptyset$. The conclusion's consistency relation is proved as follows:

Since the hypothesis $reg(\Gamma) \subseteq R$, we get that also $reg(\Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\})$ holds.

By the consistency of the hypothesis and the cases (1) and (2) of Lemma A.2.1.6, we can prove that Π' and ϖ are well typed (type environment is extended according to the program environment extension). In order to prove that $ord(\varpi) \Rightarrow \varphi'$, we have the following two sub cases to prove: (a) $ord(\varpi) \Rightarrow \varphi$ that is true from the hypothesis's consistency and (b) $ord(\varpi) \Rightarrow inv.cn\langle a_{1..n}\rangle$ that is true because each object value of the store is well typed: class invariant is checked at object creation, in the type rule [RC−ObjVal] by the judgment $P; R; \varphi \vdash_{type} cn\langle r_{1..n}\rangle$ and from the hypothesis's consistency $R{=}dom(\varpi)$ and $ord(\varpi) \Rightarrow \varphi$.

In order to prove the type judgment of the conclusion,
$P; \Gamma'; R'; \varphi'; \Sigma' \vdash ret(n_{0..p}, e) : t$ we have to prove the assumptions of the rule [RC−METH]. By its definition Γ' is well formed. By the hypothesis's type rule [RC−INVOKE] we get that all regions that annotate the method are in R; $\varphi \Rightarrow inv.cn'$ (the invariant of the superclass), and the fact the types of the method arguments are well formed $P; R; \varphi \vdash_{type} t_j,\ j = 0..p$. By type judgment of the hypothesis φ implies the precondition of superclass $cn'\langle a_{1..m}\rangle$ method, $\varphi \Rightarrow pre.cn'.mn$. But adding the subclass invariant to both sides of the entailment we have the following entailment: $\varphi' \Rightarrow pre.cn'.mn \wedge inv.cn\langle a_{1..n}\rangle$ By the soundness of the method overriding:
$pre.cn'.mn \wedge inv.cn\langle a_{1..n}\rangle \Rightarrow pre.cn.mn$ Thus, the region constraint required by rule [RC−METH] is proved. Using Γ', R' and φ' we can typecheck the method body. Type of the method body is a subtype of the expected type, thus we use the subsumption. We also used Lemma A.2.1.1 for region substitutions.

Hence both the type judgment and the consistency relation hold.

Case: $e_1\, ;\, e_2$

By induction hypothesis for $\langle\varpi,\Pi\rangle[e_1]\hookrightarrow\langle\varpi',\Pi'\rangle[e_1']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e_1')$, $\hat{\Gamma},\hat{R},\hat{\varphi},\hat{\Sigma}\models\langle\varpi',\Pi'\rangle$, $reg(\hat{\Gamma})\subseteq\hat{R}$,

$(\hat{\Sigma}-(lreg(e_1')-lreg(e_1)))-(lloc(e_1)-lloc(e_1'))=\Sigma-(lreg(e_1)-lreg(e_1'))$,

$\hat{\Gamma}-(lvar(e_1')-lvar(e_1))=\Gamma-(lvar(e_1)-lvar(e_1'))$,

$\hat{R}-(lreg(e_1')-lreg(e_1))=R-(lreg(e_1)-lreg(e_1'))$, and

$\hat{\varphi}-(lreg(e_1')-lreg(e_1))\Rightarrow\varphi-(lreg(e_1)-lreg(e_1'))$.

We let $R'=\hat{R}$, $\Gamma'=\hat{\Gamma}$, $\varphi'=\hat{\varphi}$, and $\Sigma'=\hat{\Sigma}$. From the hypothesis's valid relation $valid(e_1;e_2)$ we get that $retregs(e_2)=\emptyset$,

$retvars(e_2)=\emptyset$, $retregs(e_1)\cap regs(e_2)=\emptyset$, $retvars(e_1)\cap vars(e_2)=\emptyset$, and $valid(e_1)$. Then, by the Lemma A.2.1.9 we get that $valid(e_2)$, $lvar(e_2)=\emptyset$, and $lreg(e_2)=\emptyset$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. In order to prove that $P;\Gamma';R';\varphi';\Sigma'\vdash e_1';e_2:t_2$, we have to prove that $P;\Gamma';R';\varphi';\Sigma'\vdash e_2:t_2$. Note that the hypothesis contains the type judgment $P;\Gamma;R;\varphi;\Sigma$ $\vdash e_2:t_2$ We also have to prove that $valid(e_1';e_2)$ holds. We use a case based analysis on the expression e_1. We discuss only the main sub cases that change either ϖ or Π (the other cases are straightforward):

- $e_1=\texttt{retr}(a,\delta)$

  $\hat{R}=R-\{a\}$, $\hat{\Gamma}=\Gamma$, $\hat{\varphi}=\varphi-a$, $\hat{\Sigma}=\Sigma-\{a\}$. From hypothesis $reg(\Gamma)\subseteq R-\{a\}$ and $a\notin regs(e_2)$. Applying Lemma A.2.1.8 on $P;\Gamma;R;\varphi;\Sigma\vdash e_2:t_2$ we prove the type judgment. The valid relation is straightforward.

- $e_1=$ **letreg** r **in** e

  $\hat{R}=R\cup\{a\}$, $\hat{\Sigma}=\Sigma+a$, $\hat{\Gamma}=\Gamma$, and $\hat{\varphi}=(\varphi\wedge\bigwedge_{r\in R}(r\succeq a))$. Note that a is a fresh region. Applying the cases (2) and (3) of Lemma A.2.1.6 on $P;\Gamma;R;\varphi;\Sigma\vdash e_2:t_2$ we prove the type judgment. The valid relation is straightforward since a is a fresh region.

- $e_1=\texttt{ret}(v,\delta)$

  $\hat{\Sigma}=\Sigma$, $\hat{\varphi}=\varphi$, $\hat{R}=R$, and $\hat{\Gamma}=\Gamma-\{v\}$.

  By the hypothesis's valid relation we get that $retvars(\texttt{ret}(v,\delta))\cap vars(e_2)=\emptyset$, hence $v\notin vars(e_2)$. Applying the case (1) of Lemma A.2.1.7 on $P;\Gamma;R;\varphi;\Sigma\vdash e_2:t_2$ we prove the type judgment. The valid relation is straightforward.

- $e_1 = \{(t\,v)\ e\}$

  $\hat{\Sigma} = \Sigma$, $\hat{\Gamma} = \Gamma{+}(v : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v is a fresh variable. Applying the case (1) of Lemma A.2.1.6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward since v is a fresh variable.

- $e_1 = \textbf{new}\ cn\langle r_{1..n}\rangle(v_{1..p})$

  $\hat{\Sigma} = \Sigma + \{(r_1, o) : cn\langle r_{1..n}\rangle\}$, $\hat{\Gamma} = \Gamma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$.

  Applying the case (4) of Lemma A.2.1.6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward.

- $e_1 = v'_0.mn\langle a^+\rangle(v'_{1..p})$

  $\hat{R} = R$, $\hat{\Gamma} = \Gamma{+}\{(n_i : \Gamma(v'_i))_{i:0..p}\}$, $\hat{\varphi} = \varphi{\wedge}inv.cn\langle a_{1..n}\rangle$, and $\hat{\Sigma} = \Sigma$. Note that all n_i variables are fresh variables. Applying the cases (1) and (2) of Lemma A.2.1.6 on $P; \Gamma; R; \varphi; \Sigma \vdash e_2 : t_2$ we prove the type judgment. The valid relation is straightforward since n_i are fresh variables and the method body is a valid block expression.

Case: $lhs = e$

By induction hypothesis for $\langle \varpi, \Pi\rangle[e] \hookrightarrow \langle \varpi', \Pi'\rangle[e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma}, \hat{R}, \hat{\varphi}, \hat{\Sigma} \models \langle \varpi', \Pi'\rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$,

$(\hat{\Sigma}{-}(lreg(e'){-}lreg(e))){-}(lloc(e){-}lloc(e')) = \Sigma{-}(lreg(e){-}lreg(e'))$,

$\hat{\Gamma}{-}(lvar(e'){-}lvar(e)) = \Gamma{-}(lvar(e){-}lvar(e'))$,

$\hat{R}{-}(lreg(e'){-}lreg(e)) = R{-}(lreg(e){-}lreg(e'))$, and

$\hat{\varphi}{-}(lreg(e'){-}lreg(e)) \Rightarrow \varphi{-}(lreg(e){-}lreg(e'))$.

We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(lhs = e)$ we get that $retvars(e) \cap vars(lhs) = \emptyset$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. In order to prove that $P; \Gamma'; R'; \varphi'; \Sigma' \vdash lhs{=}e' : \textbf{void}$, we have to prove that $P; \Gamma'; R'; \varphi'; \Sigma' {\vdash} lhs : t$ and $P; R'; \varphi' {\vdash} t' {<:} t$, while $P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : t'$, $P; \Gamma; R; \varphi; \Sigma \vdash lhs : t$, and $P; R; \varphi \vdash t' <: t$ are given by the induction hypothesis. We also have to prove that $valid(lhs = e')$ holds. We use a case based analysis on the expression e. We discuss only the main sub cases that change either ϖ or Π (the other cases are straightforward):

- $e = \texttt{retr}(a, \delta)$

$\hat{R} = R-\{a\}$, $\hat{\Gamma} = \Gamma$, $\hat{\varphi} = \varphi-a$, $\hat{\Sigma} = \Sigma-\{a\}$. From hypothesis $reg(\Gamma) \subseteq R-\{a\}$. Since $lhs = v \mid v.f$ we get that $regs(lhs) = \emptyset$. Applying Lemma A.2.1.8 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. Since $a \notin reg(t)$ and $a \notin reg(t')$, applying the case (3) of Lemma A.2.1.5 on $P;R;\varphi \vdash t' <: t$ we prove the subtype judgment. The valid relation is straightforward.

– $e = \textbf{letreg}\, r\ \textbf{in}\ e_1$

  $\hat{R} = R \cup \{a\}$, $\hat{\Sigma} = \Sigma{+}a$, $\hat{\Gamma} = \Gamma$, and $\hat{\varphi} = (\varphi \wedge \bigwedge_{r \in R}(r \succeq a))$. Note that a is a fresh region. Applying the cases (2) and (3) of Lemma A.2.1.6 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. Applying the cases (1) and (2) of Lemma A.2.1.5 on $P;R;\varphi \vdash t' <: t$ we prove the subtype judgment. From the hypothesis valid relation $valid(\textbf{letreg}\, r\ \textbf{in}\ e_1)$ we get that $retvars(e_1) = \emptyset$. By the induction hypothesis we get that $valid(\texttt{retr}(a, e_1))$. Hence $valid(lhs = \texttt{retr}(a, e_1))$ holds.

– $e = \texttt{ret}(v, \delta)$

  $\hat{\Sigma} = \Sigma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$, and $\hat{\Gamma} = \Gamma-\{v\}$. By the hypothesis's valid relation $retvars(\texttt{ret}(v, \delta)) \cap vars(lhs) = \emptyset$, hence $v \notin vars(lhs)$. Applying the case (1) of Lemma A.2.1.7 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. The subtype judgment is the same as that of the hypothesis. The valid relation is straightforward.

– $e = \{(t\,v)\ \ e_1\}$

  $\hat{\Sigma} = \Sigma$, $\hat{\Gamma} = \Gamma{+}(v : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v is a fresh variable. By the hypothesis valid relation we get $valid((t\,v)\ \ e_1\})$ and then $retvars(e_1) = \emptyset$. Since v is a fresh variable, $valid(lhs = \texttt{ret}(v, e_1))$ holds. Applying the case (1) of Lemma A.2.1.6 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. The subtype judgment is the same as that of the hypothesis.

– $e = \textbf{new}\ cn\langle r_{1..n}\rangle(v_{1..p})$

  Applying the case (4) of Lemma A.2.1.6 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. The valid relation and the subtype judgment are straightforward.

  $\hat{\Sigma} = \Sigma + \{(r_1, o) : cn\langle r_{1..n}\rangle\}$, $\hat{\Gamma} = \Gamma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$.

– $e = v'_0.mn\langle a^+\rangle(v'_{1..p})$

  $\hat{R} = R$, $\hat{\Gamma} = \Gamma{+}\{(n_i : \Gamma(v'_i))_{i:0..p}\}$, $\hat{\varphi} = \varphi \wedge inv.cn\langle a_{1..n}\rangle$, and $\hat{\Sigma} = \Sigma$. Note that all n_i variables are fresh variables. Applying the cases (1) and (2) of Lemma A.2.1.6 on $P;\Gamma;R;\varphi;\Sigma \vdash lhs : t$ we prove the type judgment. Applying the case (2) of Lemma

A.2.1.5 on $P;R;\varphi \vdash t' <: t$ we prove the subtype judgment. The valid relation is straightforward since n_i are fresh variables and the method body is a valid block expression.

Case: $\texttt{ret}(v,e)$

By induction hypothesis for $\langle\varpi,\Pi\rangle[e]\hookrightarrow\langle\varpi',\Pi'\rangle[e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma},\hat{R},\hat{\varphi},\hat{\Sigma} \models \langle\varpi',\Pi'\rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$, $P;\hat{\Gamma};\hat{R};\hat{\varphi};\hat{\Sigma} \vdash e' : t$

$(\hat{\Sigma}-(lreg(e')-lreg(e)))-(lloc(e)-lloc(e')) = \Sigma-(lreg(e)-lreg(e'))$,

$\hat{\Gamma}-(lvar(e')-lvar(e)) = \Gamma-(lvar(e)-lvar(e'))$,

$\hat{R}-(lreg(e')-lreg(e)) = R-(lreg(e)-lreg(e'))$, and

$\hat{\varphi}-(lreg(e')-lreg(e)) \Rightarrow \varphi-(lreg(e)-lreg(e'))$.

We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(\texttt{ret}(v,e))$ we get that $v \notin retvars(e)$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. By the hypothesis's type judgment $P;\Gamma;R;\varphi;\Sigma \vdash \texttt{ret}(v,e) : t$ we get that $v \in \Gamma$. In order to prove the type judgment

$P;\Gamma';R';\varphi';\Sigma' \vdash \texttt{ret}(v,e') : t$ we only have to prove that $v \in \Gamma'$, since $P;\Gamma';R';\varphi';\Sigma' \vdash e' : t$ is given by the induction hypothesis. In order to prove the valid relation $valid(\texttt{ret}(v,e'))$ we have to prove that $v \notin retvars(e')$ since $valid(e')$ is given by the induction hypothesis. We use a case based analysis on the expression e. We discuss only the main sub cases that change Π (the other cases are straightforward):

– $e = \texttt{ret}(v',\delta)$

$\hat{\Sigma} = \Sigma$, $\hat{\varphi} = \varphi$, $\hat{R} = R$, and $\hat{\Gamma} = \Gamma-\{v'\}$. From hypothesis $v \notin retvars(e)$ holds, hence $v \neq v'$. Therefore $v \in \hat{\Gamma}$ holds since from hypothesis we have that $v \in \Gamma$.

The relation $v \notin retvars(\delta)$ is straightforward.

– $e = \{(t\,v')\;\; e_1\}$

$\hat{\Sigma} = \Sigma$, $\hat{\Gamma} = \Gamma+(v' : t)$, $\hat{\varphi} = \varphi$, and $\hat{R} = R$. Note that v' is a fresh variable. The relation $v \in \hat{\Gamma}$ holds since from hypothesis $v \in \Gamma$. From hypothesis $valid(\{(t\,v')\;\; e_1\})$ holds, therefore $retvars(e_1) = \emptyset$. Since v' is a fresh variable $v \neq v'$ holds. Hence the relation $v \notin retvars(ret(v',e_1))$ holds.

– $e = v'_0.mn\langle a^+\rangle(v'_{1..p})$

$\hat{R} = R$, $\hat{\Gamma} = \Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\}$, $\hat{\varphi} = \varphi \wedge inv.cn\langle a_{1..n}\rangle$, and $\hat{\Sigma} = \Sigma$. Note that all n_i variables are fresh variables. The relation $v \in \hat{\Gamma}$ holds since from hypothesis $v \in \Gamma$. From hypothesis the method body is a valid block expression, therefore there is not any ret in the method's body. Since n_i are fresh variables we get that $v \neq n_i\ i = 0..p$. Hence the relation $v \notin retvars(e')$ holds.

Case: $\texttt{retr}(a, e)$

By induction hypothesis for $\langle\varpi, \Pi\rangle[e] \hookrightarrow \langle\varpi', \Pi'\rangle[e']$ there exist $\hat{\Sigma}$, $\hat{\Gamma}$, $\hat{R}$, and $\hat{\varphi}$ such that $valid(e')$, $\hat{\Gamma}, \hat{R}, \hat{\varphi}, \hat{\Sigma} \vDash \langle\varpi', \Pi'\rangle$, $reg(\hat{\Gamma}) \subseteq \hat{R}$, $P; \hat{\Gamma}; \hat{R}; \hat{\varphi}; \hat{\Sigma} \vdash e' : t$
$(\hat{\Sigma} - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$,
$\hat{\Gamma} - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$,
$\hat{R} - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$, and
$\hat{\varphi} - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$.

We let $R' = \hat{R}$, $\Gamma' = \hat{\Gamma}$, $\varphi' = \hat{\varphi}$, and $\Sigma' = \hat{\Sigma}$. From the hypothesis's valid relation $valid(\texttt{retr}(a, e))$ we get that $a \notin retregs(e)$. Hence, the conclusion's relations between Γ and Γ', Σ and Σ', R and R', and φ and φ' are straightforward proved. In order to prove the valid relation $valid(\texttt{retr}(a, e'))$ we have to prove that $a \notin retregs(e')$ since $valid(e')$ is given by the induction hypothesis. In order to prove the type judgment $P; \Gamma'; R'; \varphi'; \Sigma' \vdash \texttt{retr}(a, e') : t$ we have to prove that $a \in R'$, $reg(t) \subseteq R' - lreg(e') - \{a\}$, $reg(\Gamma - lvar(e')) \subseteq R' - lreg(e') - \{a\}$, and $\varphi' \Rightarrow \bigwedge_{r \in (R' - lreg(e') - \{a\})}(r \succeq a)$, while $P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : t$ is given by the induction hypothesis. We use a case based analysis on the expression e. We discuss only the main sub cases that change $lreg$, $lvar$ and $retregs$ (the other cases are straightforward):

- $e = \texttt{retr}(r, \delta)$ then $e' = \delta$

  $R' = R - r$, $\varphi' = \varphi - r$, and $\Gamma' = \Gamma$. From hypothesis $a \notin retregs(e)$ holds, therefore $r \neq a$. The relation $a \notin retregs(e')$ is straightforward proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$.

  Note that $R' - lreg(e') - \{a\} = R - lreg(e) - \{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R - lreg(e) - \{a\}$, therefore $reg(t) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $reg(\Gamma - lvar(e)) \subseteq R - lreg(e) - \{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$. Applying the case (2) of Lemma A.2.1.3 we get that $\varphi' \Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r \succeq a)$.

– $e =$ **letreg** r **in** e_1 then $e' = \texttt{retr}(r', e_1)$ and r' is a fresh region.

  $R' = R \cup \{r'\}$, $\varphi' = (\varphi \wedge \bigwedge_{r\in R}(r \succeq r'))$, and $\Gamma' = \Gamma$. The region r' is a fresh region, therefore $r' \neq a$. From the hypothesis relation $valid(e)$, we get that $retregs(e_1) = \emptyset$. Hence, the relation $a \notin retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$.

  Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$, therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$. Since $\varphi'\Rightarrow\varphi$ we get that $\varphi'\Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r\succeq a)$.

– $e = \texttt{ret}(v', \delta)$ then $e' = \delta$

  $R' = R$, $\varphi' = \varphi$, and $\Gamma' = \Gamma-\{v'\}$. The relation $a \notin retregs(e')$ is straightforward proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $\varphi\Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r\succeq a)$, therefore $\varphi'\Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r\succeq a)$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$ therefore $reg(\Gamma' - lvar(e')) \subseteq R' - lreg(e') - \{a\}$.

– $e = \{(t\, v')\ \ e_1\}$ then $e' = \texttt{ret}(v_1, e_1)$ and v_1 is a fresh variable.

  $\Gamma' = \Gamma+(v_1 : t)$, $\varphi' = \varphi$, and $R' = R$. From the hypothesis's valid relation $valid(e)$, we get that $retregs(e_1) = \emptyset$ and $retvars(e_1) = \emptyset$. By the cases (2) and (3) of Lemma A.2.1.9 we get that $lreg(e_1) = \emptyset$ and $lvar(e_1) = \emptyset$. Hence, the relation $a \notin retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'-lreg(e')-\{a\}=R-lreg(e)-\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R-lreg(e)-\{a\}$, therefore $reg(t)\subseteq R'-lreg(e')-\{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$, therefore $\varphi' \Rightarrow \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r \succeq a)$. From the hypothesis's type judgment we get that $reg(\Gamma-lvar(e))\subseteq R-lreg(e)-\{a\}$ therefore $reg(\Gamma'-lvar(e'))\subseteq R'-lreg(e')-\{a\}$.

– $e = v'_0.mn\langle a^+\rangle(v'_{1..p})$ then $e' = \texttt{ret}(n_{1..p}, e_1)$, $n_{1..p}$ are fresh variables, and e_1 is a valid block expression such that $retvars(e_1) = \emptyset$ and $retregs(e_1) = \emptyset$.

$R' = R$, $\Gamma' = \Gamma + \{(n_i : \Gamma(v'_i))_{i:0..p}\}$, and $\varphi' = \varphi \wedge inv.cn\langle a_{1..n}\rangle$. By the cases (2) and (3) of Lemma A.2.1.9 we get that $lreg(e_1){=}\emptyset$ and $lvar(e_1){=}\emptyset$. Hence, the relation $a{\notin}retregs(e')$ is proved. From the hypothesis's type judgment we get that $a \in R$, therefore $a \in R'$. Note that $R'{-}lreg(e'){-}\{a\}{=}R{-}lreg(e){-}\{a\}$. From the hypothesis's type judgment we get that $reg(t) \subseteq R{-}lreg(e){-}\{a\}$, therefore $reg(t){\subseteq}R'{-}lreg(e'){-}\{a\}$. From the hypothesis's type judgment we get that $\varphi \Rightarrow \bigwedge_{r\in(R-lreg(e)-\{a\})}(r \succeq a)$. Since $\varphi'{\Rightarrow}\varphi$ we get $\varphi'{\Rightarrow} \bigwedge_{r\in(R'-lreg(e')-\{a\})}(r{\succeq}a)$. From the hypothesis's type judgment we get that $reg(\Gamma{-}lvar(e)){\subseteq}R{-}lreg(e){-}\{a\}$ therefore $reg(\Gamma'{-}lvar(e')){\subseteq}R'{-}lreg(e'){-}\{a\}$.

□

A.2.3 Proof of Theorem 3.7.2.2 (Progress)

By structural induction over the depth of the type derivation for expression e and using the Lemma A.2.1.10.

Cases: [RC−LOCATION, RC−ObjVal, RC−CONS1, RC−CONS2]

e is a value.

Case: [RC−VAR]

We let $\varpi' = \varpi$, $\Pi' = \Pi$, and $e' = \Pi(v)$. By hypothesis we get that $(v : t) \in \Gamma$ and $dom(\Gamma) = dom(\Pi)$, thus the check of the evaluation rule [D−VAR] does not fail.

Case: [RC−FD]

By the type judgment and the consistency of the hypothesis we get that $(v : cn\langle r_{1..n}\rangle) \in \Gamma$ and $\Pi(v) : \Gamma(v)$. According to the Lemma A.2.1.10, there are two cases for $\Pi(v)$:

1. $\Pi(v) = \texttt{null}$ then the rule [D−NULLERR1] generates an error `nullerr`.
2. $\Pi(v) = (r_1, o)$, $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$, and $P; \Gamma; R; \varphi; \Sigma{\vdash}cn\langle r_{1..n}\rangle(V){:}cn\langle r_{1..n}\rangle$.

   We let $\varpi' = \varpi$, $\Pi' = \Pi$, and $e' = V(f)$. Then rule [D−FD] is used.

Case: [RC−ASSGN]

We deal with expression $lhs = e$. From type judgment of the hypothesis we have $P;\Gamma;R;\varphi;\Sigma \vdash e : t'$. By the induction hypothesis, we have the following cases:

1. $\langle\varpi,\Pi\rangle[e]\hookrightarrow$`nullerr`,

   then the error is propagated as $\langle\varpi,\Pi\rangle[lhs{=}e]\hookrightarrow$`nullerr`

2. $\langle\varpi,\Pi\rangle[e] \hookrightarrow \langle\hat{\varpi},\hat{\Pi}\rangle[e']$.

   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $lhs = e'$. Then the evaluation rule [D−ASSGN1] is used.

3. e is a value $e = \delta$.

   There are the following two sub cases based on the form of $lhs = v \mid v.f$:

SubCase: $v = \delta$

By hypothesis's type judgment we get that $(v{:}t){\in}\Gamma$ and $dom(\Gamma){=}dom(\Pi)$, thus $v{\in}dom(\Pi)$. By the type judgment of the hypothesis, the type of δ is well-formed and is a subtype of type of v. If $\delta{=}(r_1{,}o_1)$ the type rule [RC−LOCATION] ensures that $r_1 \in R$, but from the hypothesis' consistency $dom(\varpi) = R$. Thus $r_1 \in dom(\varpi)$. We let $\varpi' = \varpi$ and $\Pi' = \Pi{+}\{v{\mapsto}\delta\}$.

The evaluation rule [D−ASSGN2] can be applied, since we proved that its run-time checks hold. Note that the rule [D−ASSGN1−DANGLERR] is never used for a well typed expression.

SubCase: $v.f = \delta$

By the type judgment and the consistency of the hypothesis we get that $(v : cn\langle a_{1..n}\rangle) \in \Gamma$ and $\Pi(v) : \Gamma(v)$. According to the Lemma A.2.1.10, there are two cases for $\Pi(v)$:

(a) $\Pi(v) =$ `null` then the rule [D−NULLERR2] generates an error `nullerr`.

(b) $\Pi(v){=}(a_1{,}o)$ and $\varpi(a_1)(o){=}cn\langle a_{1..n}\rangle(V)$.

By the hypothesis type rule [RC−ASSGN] we get that the type of δ is well-formed and is a subtype of type of $v.f$. If $\delta{=}(r_1,o_1)$, the subtyping rule of the type rule [RC−ASSGN] and the subtyping judgment [ObjRegSub] ensures that the first region of type of δ, r_1 outlives the first region of type of $v.f$

according to the region constraint φ. But from the consistency of hypothesis $ord(\varpi){\Rightarrow}\varphi$ holds. Thus the rule [D−ASSGN3] can be applied, since we proved that its runtime checks hold.

Note that the rule [D−ASSGN3−DANGLERR] is never used for a well typed expression.

Case: [RC−NEW]

By the consistency of the hypothesis we get that $ord(\varpi){\Rightarrow}\varphi$ and $R = dom(\varpi)$. By the type judgment $P; R; \varphi \vdash_{type} cn\langle r_{1..n}\rangle$ of the hypothesis we get that $\varphi{\Rightarrow}\varphi_{inv}$. Thus the first runtime check $ord(\varpi){\Rightarrow}\varphi_{inv}$ holds. By the hypothesis's type judgment the type of each variable v_i is a subtype of the corresponding class field f_i type. By the subtyping judgment [ObjRegSub], the first region of the region type of v_i outlives the first region of the region type of f_i according to the region constraint φ. Thus the runtime check $\mathbf{ord}(\varpi){\Rightarrow}(r'_i{\succeq}\mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i))$ holds for each object field f_i. Thus the rule [D−NEW] can be applied, since we proved that its runtime checks hold.

Note that the rule [D−NEW−DANGLERR] is never used for a well typed expression.

Case: [RC−EB]

We let $\varpi' = \varpi$, $\Pi' = \Pi{+}\{n{\mapsto}\mathbf{init}(t)\}$, and $e'{=}\texttt{ret}(n, e)$ where n is a fresh variable. Then the evaluation rule [D−EB] can be applied.

Case: [RC−RET]

We deal with $\texttt{ret}(v, e)$. Based on the expression e, there are two cases:

1. e is a value and then the rule [D−RET2] can be applied.
2. e is not a value. By the induction hypothesis, there are two sub cases:
   (a) $\langle\varpi, \Pi\rangle[e] \hookrightarrow \langle\hat{\varpi}, \hat{\Pi}\rangle[e']$.

   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $\texttt{ret}(v, e')$. The evaluation rule [D−RET1] can be applied.

   (b) $\langle\varpi, \Pi\rangle[e]{\hookrightarrow}\texttt{nullerr}$

   The error is propagated as $\langle\varpi, \Pi\rangle[\texttt{ret}(v, e)]{\hookrightarrow}\texttt{nullerr}$.

Case: [RC−LETR]

We let $\varpi'{=}[a{\mapsto}\emptyset]\varpi$ and $\Pi' = \Pi$. The evaluation rule [D−LETR] can be applied.

Case: [RC−RETR]

We deal with `retr`(a,e). Based on the expression e, there are two sub cases:

1. e is a value such that $e{=}\delta$.

   In order to apply the rule [D−RETR2], we have to prove that its runtime checks are redundant. Note that $lreg(\delta)=\emptyset$ and $lvar(\delta)=\emptyset$. By the type judgment of the hypothesis we get that $a\in R$, but from the hypothesis's consistency we get that $R=dom(\varpi)$. Thus $a\in dom(\varpi)$. In addition, by the hypothesis's type judgment we get that $\varphi\Rightarrow\bigwedge_{r\in R_t}(r{\succeq}a)$ where $R_t{=}R{-}\{a\}$. By the hypothesis consistency we get that $ord(\varpi){\Rightarrow}\varphi$. Thus, we proved that a is the region on the top of the stack ϖ such that $\varpi{=}[a{\mapsto}Rgn]\varpi'$. If $\delta=(r,o)$ then its type t contains the region r, but from the hypothesis's type judgment we get that $reg(t){\subseteq}R{-}a$. Thus we proved that $r\in dom(\varpi')$. By the hypothesis's type judgment we get that $reg(\Gamma){\subseteq}R{-}a$, but from the consistency relation $dom(\Gamma)=dom(\Pi)$ holds and for each $v\in\Pi$ the type of $\Pi(v)$ is $\Gamma(v)$. Since the type of a location contains the location's region, we proved that $\forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}dom(\varpi'))$. By the hypothesis's consistency relation and the type judgment for an object value [RC−ObjVal] we get the following relations for each location $(r_1,o){\in}dom(\varpi')$ with $\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)$: $r_i\succeq r_1, i=2..n$ and for each field $f\in dom(V)$ its type is a subtype of the expected type given by $fieldlist(cn\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p}$, (that means the regions of its type are older than $r_1,..,r_n$). Since $r_1\succeq a$ holds, we proved the last check (about ϖ') of the rule [D−RETR2]. Thus, we can apply the rule [D−RETR2], while the rule [D−RETR2−DANGLERR] is never used for a well typed program.

2. e is not a value. By the induction hypothesis, there are two sub cases:

   (a) $\langle\varpi,\Pi\rangle[e]\hookrightarrow\langle\hat{\varpi},\hat{\Pi}\rangle[e']$.

   We let $\varpi'=\hat{\varpi}$, $\Pi'=\hat{\Pi}$, and the new expression is `retr`(a,e'). The evaluation rule [D−RETR1] can be applied.

   (b) $\langle\varpi,\Pi\rangle[e]{\hookrightarrow}$`nullerr`

   The error is propagated as $\langle\varpi,\Pi\rangle[$`retr`$(a,e)]{\hookrightarrow}$`nullerr`.

Case: [RC−IF]

By the hypothesis's type judgment the type of v is $boolean$. According to the

Lemma A.2.1.10, there are two cases: either *v* is *true* and the rule [D−IF1] is applied, or *v* is *false* and the rule [D−IF2] is applied.

Case: [RC−LOOP]

By the hypothesis's type judgment the type of v is $boolean$. According to the Lemma A.2.1.10, there are two cases: either *v* is *true* and the rule [D−LOOP1] is applied, or *v* is *false* and the rule [D−LOOP2] is applied.

Case: [RC−SEQ]

We deal with $e_1; e_2$. Based on the expression e_1, there are two cases:

1. e_1 is a value and then the rule [D−SEQ2] can be applied.
2. e_1 is not a value. By the induction hypothesis, there are two sub cases:
   (a) $\langle\varpi,\Pi\rangle[e_1] \hookrightarrow \langle\hat{\varpi},\hat{\Pi}\rangle[e_1']$.
   We let $\varpi' = \hat{\varpi}$, $\Pi' = \hat{\Pi}$, and the new expression is $e_1'; e_2$. The evaluation rule [D−SEQ1] can be applied.
   (b) $\langle\varpi,\Pi\rangle[e_1]\hookrightarrow\texttt{nullerr}$
   The error is propagated as $\langle\varpi,\Pi\rangle[e_1; e_2]\hookrightarrow\texttt{nullerr}$.

Case: [RC−INVOKE]

We deal with $v_0'.mn\langle a^+a'^+\rangle(v_{1..p}')$. By the hypothesis's type judgment [RC−INVOKE] we get that the regions $\{a^+a'^+\} \subset R$. By the hypothesis's consistency relation we get that $dom(\varpi) = R$. Thus the runtime check of [RC−INVOKE] is proved and the rule [D−INVOKE−DANGLERR] is never used by a well typed program. By the hypothesis's type judgment the type of v_0' is $cn\langle a^+\rangle$. By the hypothesis's consistency $dom(\Gamma) = dom(\Pi)$, thus $v_0' \in dom(\Pi)$. According to the Lemma A.2.1.10, there are two cases:

1. $\Pi(v_0') = \texttt{null}$. Then the rule [D−NULLERR3] generates an error nullerr.
2. $\Pi(v_0') = (a_1, o)$ that is well-typed. Thus the rule [D−INVOKE] can be applied.

□

A.2.4 Proof of Lemma 4.8.0.2 (Correctness)

1. Based on the form of τ we apply either [RI−CT], or [RI−OBJ], or [RI−PRIM] and we obtain the region type t and region class invariant φ corresponding to τ. We let $R = reg(t)$. The

judgment $P';R;\varphi \vdash_{type} t$ is directly verified, namely its two checks referring to the regions of type t and the class invariant of t. By using the Lemma A.2.1.4 we prove that for all R' and φ' such that $R \subseteq R'$ and $\varphi' \Rightarrow \varphi$ the judgment $P';R';\varphi' \vdash_{type} t$ holds.

2. By the induction hypothesis using the previous case and the inference rule [RI−SUBTYPE], we prove that $\forall R_1, \varphi_1 \cdot (reg(t) \subseteq R_1 \wedge \varphi_1 \Rightarrow \varphi) \implies P';R_1;\varphi_1 \vdash_{type} t$ and

   $\forall R_1', \varphi_1' \cdot (reg(t') \subseteq R_1' \wedge \varphi_1' \Rightarrow \varphi') \implies P';R_1';\varphi_1' \vdash_{type} t'$. By the case (2) of Lemma A.2.1.11 and the inference rule [RI−SUBTYPE] we get that $reg(t) \cup reg(t') \subseteq reg(\varphi_0 \wedge \varphi \wedge \varphi')$. The third check of $P';R;\varphi \vdash t <: t'$ verifies the region subtyping constraint $\vdash t <: t', \varphi_0$. But from the inference rule [RI−SUBTYPE] we get that $(\varphi_0 \wedge \varphi \wedge \varphi') \Rightarrow \varphi_0$. Thus the third check is proved.

   Applying the Lemma A.2.1.5 we prove the conclusion for all R'' and φ'' such that $R'' \supseteq reg(\varphi_0 \wedge \varphi \wedge \varphi')$ and $\varphi'' \Rightarrow (\varphi_0 \wedge \varphi \wedge \varphi')$.

3. If e is a source language Core-Java expression, then it does not contain any intermediate expression. Note that the inference algorithm is not defined for the intermediate expressions and it does not produce any intermediate expression. Therefore $retvars(e') = \emptyset$ and $retregs(e') = \emptyset$ are straightforward proved. We prove the type checking relation by a structural induction on e as follows:

Cases: $k \mid$ `null` $\mid v \mid v.f$

These cases are straightforward proved.

Case: $lhs = e_1$

Note that $e' = (lhs = e_1')$. By induction on the inference rule [RI−ASSGN] hypotheses and the case (2) of the current lemma, we get that

$\forall R_1, \varphi_1 \cdot (R_1 \supseteq (reg(\Gamma) \cup regs(lhs))) \implies P';\Gamma;R_1;\varphi_1 \vdash lhs{:}t$,

$\forall R_2, \varphi_2 \cdot (R_2 \supseteq (reg(\Gamma) \cup reg(\varphi') \cup regs(e_1')) \wedge \varphi_2 \Rightarrow \varphi') \implies P';\Gamma;R_2;\varphi_2 \vdash e'{:}t'$, and

$\forall R_3, \varphi_3 \cdot (R_3 \supseteq reg(\varphi) \wedge \varphi_3 \Rightarrow \varphi) \implies P';R_3;\varphi_3 \vdash t <: t'$.

Applying the cases (2) and (3) of the Lemma A.2.1.6 and the cases (1) and (2) of the Lemma A.2.1.5 we prove that the above relations hold for all R and φ'' such that $R \supseteq (reg(\Gamma) \cup reg(\varphi \wedge \varphi') \cup regs(e'))$ and $\varphi'' \Rightarrow (\varphi \wedge \varphi')$. Hence, the type checking rule [RC−ASSGN] is proved.

Case: **new** $cn(v_{i:1..p})$

Note that $e' = \textbf{new}\,cn\langle x^+\rangle(v_{i:1..p})$. By induction on the inference rule [RI−NEW] hypotheses and the cases (1) and (2) of the current lemma, we get that

$\forall R_0', \varphi_0' \cdot (R_0' \supseteq \{x^+\} \wedge \varphi_0' \Rightarrow \varphi_0) \implies P'; R_0'; \varphi_0' \vdash_{type} cn\langle x^+\rangle$, and

$\forall i:1..p \;\; \forall R_i', \varphi_i' \cdot R_i' \supseteq reg(\varphi_i) \wedge \varphi_i' \Rightarrow \varphi_i \wedge P'; R_i'; \varphi_i' \vdash t_i' <: t_i$. By the definition of $regs$, we get that $\{x^+\} \subseteq regs(\textbf{new}\,cn\langle x^+\rangle(v_{i:1..p}))$. Applying the cases (1) and (2) of the Lemma A.2.1.5, the cases (1) and (2) of the Lemma A.2.1.4 we prove that the above relations hold for all R and φ' such that $R \supseteq (reg(\Gamma) \cup reg(\bigwedge_{i:0..p} \varphi_i) \cup regs(e'))$ and $\varphi' \Rightarrow (\bigwedge_{i:1..p} \varphi_i)$. Hence, the type checking rule [RC−NEW] is proved.

Case: $e = \{(\tau_1\, v_1)\;\; e_1\}$

There are two sub cases based on the inference rules:

SubCase: inference rule [RI−EB1]

By induction on the inference rule [RI−EB1] hypotheses and the case (1) of the current lemma, we get that

$\forall R_1', \varphi_1' \cdot (reg(\tau_1\langle x_1^*\rangle) \subseteq R_1' \wedge \varphi_1' \Rightarrow \varphi_1) \implies P'; R_1'; \varphi_1' \vdash_{type} \tau_1\langle x_1^*\rangle$ and

$\forall R', \varphi' \cdot ((regs(e_1') \cup reg(\Gamma, \{v_1 : \tau_1\langle x_1^*\rangle\}) \cup reg(\varphi)) \subseteq R' \wedge \varphi' \Rightarrow \varphi) \implies$

$P'; \Gamma; R; \varphi' \vdash e_1' : \tau\langle r^*\rangle$.

By the case (1) of the Lemma A.2.1.11, we get that $reg(\tau_1\langle x_1^*\rangle) \subseteq reg()\varphi_1$. Applying the cases (2) and (3) of the Lemma A.2.1.6 and the cases (1) and (2) of the Lemma A.2.1.4, we prove that the above relations hold for all R and φ'' such that $R \supseteq (reg(\Gamma) \cup reg(\varphi \wedge \varphi_1) \cup regs(\{(\tau_1\langle x_1^*\rangle\, v_1)\;\; e_1'\}))$ and $\varphi'' \Rightarrow (\varphi \wedge \varphi_1)$. Applying the Lemma A.2.1.1 for the substitution ρ we prove the type checking rule [RC−EB].

SubCase: inference rule [RI−EB2]

In order to prove the type checking rule [RC−LETR], we have to prove that:

$\forall R, \varphi' \cdot ((regs(\textbf{letreg}\; a\; \textbf{in}\; \rho'\rho\{(\tau_1\langle x_1^*\rangle\; v_1)\; e'\}) \cup reg(\Gamma) \cup reg(\rho((\varphi \wedge \varphi_1) \backslash rs))) \subseteq R$

$\wedge \varphi' \Rightarrow (\rho((\varphi \wedge \varphi_1) \backslash rs))) \implies P'; \Gamma; R; \varphi' \vdash \textbf{letreg}\; a\; \textbf{in}\; \rho'\rho\{(\tau_1\langle x_1^*\rangle\; v_1)\; e'\} : \tau\langle \rho r^*\rangle$.

Note that by induction on the expression block using the previous subcase for the inference rule [RI−EB1] and type checking rule [RC−EB], we obtain the following: $\Gamma \vdash \{(\tau_1\; v_1)\; e\} \Rrightarrow \rho\,\{(\tau_1\langle x_1^*\rangle\; v_1)\; e'\} : \tau\langle \rho\, r^*\rangle, \rho(\varphi \wedge \varphi_1)$ and

$\forall R_b, \varphi_b \cdot ((regs(\rho\,\{(\tau_1\langle x_1^*\rangle\; v_1)\; e'\}) \cup reg(\Gamma) \cup reg(\rho(\varphi \wedge \varphi_1))) \subseteq R_b \wedge \varphi_b \Rightarrow (\rho(\varphi \wedge \varphi_1)))$

$\implies P'; \Gamma; R_b; \varphi_b \vdash \rho\,\{(\tau_1\langle x_1^*\rangle\; v_1)\; e'\} : \tau\langle \rho\, r^*\rangle$.

Based on the checks of the type checking rule [**RC−LETR**], the proof consists of three parts:

(a) to prove that the fresh region introduced by the checking rule is not in the current set of the regions R. This is ensured by the fact that the region is fresh.

(b) to prove that the check $reg(\tau\langle\rho r^*\rangle) \subseteq R$ holds, namely we have to prove that $reg(\tau\langle\rho r^*\rangle) \subseteq (regs(\textbf{letreg}\ a\ \textbf{in}\ \rho'\rho\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\}) \cup reg(\Gamma)$ $\cup reg(\rho((\varphi\wedge\varphi_1)\backslash rs)))$. Applying the case (3) of the Lemma A.2.1.11 on the result of the induction on the expression block, we get that $reg(\tau\langle\rho r^*\rangle) \subseteq (regs(\rho\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\}) \cup reg(\Gamma)\ \cup reg(\rho(\varphi\wedge\varphi_1)))$. Note that the set of regions rs is computed by $\overline{ors}$. By the definition of $ors(\varphi, s_1, s_2)$, it is straightforward to prove that $\forall\varphi \cdot s_1 \cap s_2 \subseteq ors(\varphi, s_1, s_2)$ $\wedge \forall r \in (s_1 \cap s_2) \cdot r \notin \overline{ors}(\varphi, s_1, s_2)$. Hence, by the definition of rs from the inference rule [**RI−EB2**] we get that both $reg(\tau\langle\rho r^*\rangle) \notin rs$ and $reg(\Gamma) \notin rs$ hold. Thus we proved that $reg(\tau\langle\rho r^*\rangle) \subseteq R$ holds.

(c) to prove that $\forall R, \varphi' \cdot ((\{a\} \cup (regs(\textbf{letreg}\ a\ \textbf{in}\ \rho'\rho\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\}) \cup reg(\Gamma)$ $\cup reg(\rho((\varphi\wedge\varphi_1)\backslash rs))) \subseteq R) \wedge \varphi' \Rightarrow (\rho((\varphi\wedge\varphi_1)\backslash rs) \wedge \bigwedge_{r'\in R}(r' \succeq a)))$ $\Longrightarrow P'; \Gamma; R; \varphi' \vdash \rho'\rho\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\} : \tau\langle\rho r^*\rangle$. Note that the substitution ρ' maps all regions from the set rs to the region a. We do the proof by starting from the result of the induction on the hypothesis: $\forall R_b, \varphi_b \cdot (((regs(\rho\,\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\}) \cup reg(\Gamma) \cup reg(\rho(\varphi\wedge\varphi_1)))) \subseteq R_b$ $\wedge \varphi_b \Rightarrow (\rho(\varphi\wedge\varphi_1))) \Longrightarrow P'; \Gamma; R_b; \varphi_b \vdash \rho\,\{(\tau_1\langle x_1^*\rangle\ v_1)\ e'\} : \tau\langle\rho\, r^*\rangle$. Since equality is the strongest constraint, we get that $\bigwedge_{r\in rs}(r{=}a) \wedge (\rho(\varphi\wedge\varphi_1)) \Rightarrow (\rho(\varphi\wedge\varphi_1))$. In addition by the definition of $rs = \overline{ors}(\varphi, s_1, s_2)$, the regions of rs do not have longer lifetime than any of the regions $(s_1 \cup s_2)\backslash rs$. By the instantiation of s_1 and s_2 in the inference rule [**RI−EB2**], we get that the set $s_1 \cup s_2$ denotes all the regions.

Thus we can prove that $\bigwedge_{r'\in R}(r' \succeq a)) \wedge \bigwedge_{r\in rs}(r{=}a) \wedge (\rho(\varphi\wedge\varphi_1)) \Rightarrow \bigwedge_{r\in rs}(r{=}a) \wedge (\rho(\varphi\wedge\varphi_1))$. Since the regions of rs are younger than all other regions and also the regions of rs are equal between them we can prove that:

$\bigwedge_{r'\in R}(r'\succeq a))\wedge(\rho((\varphi\wedge\varphi_1)\backslash rs))\Rightarrow(\rho(\varphi\wedge\varphi_1))$. Applying the case (2) of the Lemma A.2.1.6 to strengthen φ_b, the Lemma A.2.1.1 to apply the substitution ρ' corresponding to the regions of rs, and taking into account the definition of $regs$ for $letreg$ we proved the type checking rule.

Case: $v'_1.mn(v'_{2..p})$

Note that $e' = v'_1.mn\langle y^+\rangle(v'_{2..p})$.

By induction on the inference rule [RI−INVOKE] hypotheses and the case (2) of the current lemma, we get that for each $j = 2..p$

$\forall R_j, \varphi'_j \cdot (R_j \supseteq reg(\varphi_j) \wedge \varphi'_j \Rightarrow \varphi_j) \implies P'; R_j; \varphi'_j \vdash \tau'_j\langle x'^*_j\rangle <: \tau_j\langle x^*_j\rangle$.

Applying the cases (1) and (2) of the Lemma A.2.1.5 we prove that the above relations hold for all R and φ'' such that $R \supseteq (reg(\Gamma) \cup reg(\hat{\varphi} \wedge \bigwedge_{j=1}^{p} \varphi_j) \cup \{y^+\})$ and $\varphi'' \Rightarrow (\hat{\varphi} \wedge \bigwedge_{j=1}^{p} \varphi_j)$. Since $\{x'^+_1\} \subseteq \{y^+\}$ and $\varphi'' \Rightarrow \varphi_1$, we use the same proof as for the case (1) of the current lemma to prove that $\forall R, \varphi'' \cdot P'; R; \varphi'' \vdash_{type} cn\langle x'^+_1\rangle$ Since $\{y^+\} \subseteq R$ and $\varphi'' \Rightarrow \hat{\varphi}$ we proved the type checking rule [RC−INVOKE].

Case: e_1 **;** e_2

Note that $e' = e'_1$ **;** e'_2. By induction on the inference rule [RI−SEQ] hypotheses we get that $\forall R_1, \varphi'_1 \cdot ((regs(e'_1) \cup reg(\Gamma) \cup reg(\varphi_1)) \subseteq R_1 \wedge \varphi'_1 \Rightarrow \varphi_1) \implies P'; \Gamma; R_1; \varphi'_1 \vdash e'_1 : t$ and $\forall R_2, \varphi'_2 \cdot ((regs(e'_2) \cup reg(\Gamma) \cup reg(\varphi_2)) \subseteq R_2 \wedge \varphi'_2 \Rightarrow \varphi_2) \implies P'; \Gamma; R_2; \varphi'_2 \vdash e'_2 : t$. Applying the cases (1) and (2) of the Lemma A.2.1.6 we prove that the above relations hold for all R and φ'' such that $R \supseteq (reg(\Gamma) \cup reg(\varphi_1 \wedge \varphi_2) \cup regs(e'_1 \,;\, e'_2))$ and $\varphi'' \Rightarrow (\varphi_1 \wedge \varphi_2)$. Thus we proved the type checking rule [RC−SEQ].

Case: **while** v e

The proof is straightforward by induction on the inference rule [RC−LOOP] hypothesis.

Case: **if** v **then** e_1 **else** e_2

Note that $e' =$ **if** v **then** e'_1 **else** e'_2.

By induction on the inference rule [RI−IF] hypotheses and the case (2) of the current lemma we get that

$\forall R_1, \varphi''_1 \cdot ((regs(e'_1) \cup reg(\Gamma) \cup reg(\varphi'_1)) \subseteq R_1 \wedge \varphi''_1 \Rightarrow \varphi'_1) \implies P'; \Gamma; R_1; \varphi''_1 \vdash e'_1 : t_1$,

$\forall R_2, \varphi''_2 \cdot ((regs(e'_2) \cup reg(\Gamma) \cup reg(\varphi'_2)) \subseteq R_2 \wedge \varphi''_2 \Rightarrow \varphi'_2) \implies P'; \Gamma; R_2; \varphi''_2 \vdash e'_2 : t_2$,

$\forall R_3, \varphi_3'' \cdot (R_3 \supseteq reg(\varphi_1) \wedge \varphi_3'' \Rightarrow \varphi_1) \Longrightarrow P'; R_3; \varphi_3'' \vdash t_1 <: t$, and

$\forall R_4, \varphi_4'' \cdot (R_4 \supseteq reg(\varphi_2) \wedge \varphi_4'' \Rightarrow \varphi_2) \Longrightarrow P'; R_4; \varphi_4'' \vdash t_2 <: t$.

Applying the cases (1) and (2) of the Lemma A.2.1.6 and the cases (1) and (2) of the Lemma A.2.1.5, we prove that the above relations hold for all R and φ'' such that $R \supseteq (reg(\Gamma) \cup reg(\varphi_1 \wedge \varphi_2 \wedge \varphi_1' \wedge \varphi_2') \cup regs(\textbf{if}\ v\ \textbf{then}\ e_1'\ \textbf{else}\ e_2'))$ and $\varphi'' \Rightarrow (\varphi_1 \wedge \varphi_2 \wedge \varphi_1' \wedge \varphi_2')$. Thus we proved the type checking rule [RC−IF].

4. Note that $meth' = \tau_0\langle r_0^*\rangle\ mn\langle r^+, r_1^*, .., r_p^*, r_0^*\rangle((\tau_j\langle r_i^*\rangle\ v_i)_{i:1..p})$ **where** $\varphi\ \{e'\}$

The method $meth'$ is valid since its body does not contain any intermediate expression. As can be seen, the inference rules do not introduce any intermediate expression.

By induction on the inference rules [RI−METH−1] and [RI−METH−2] hypotheses and the case (1), (2) and (3) of the current lemma we get that

for each $i = 0..p\ \ \forall R_i, \varphi_i'' \cdot (\{r_i^*\} \subseteq R_i \wedge \varphi_i'' \Rightarrow \varphi_i) \Longrightarrow P'; R_i; \varphi_i'' \vdash_{type} \tau_i\langle r_i^*\rangle$,

$\forall R_s, \varphi_s \cdot (reg(\varphi_0') \subseteq R_s \wedge \varphi_s \Rightarrow \varphi_0') \Longrightarrow P'; R_s; \varphi_s \vdash \tau_0'\langle x_0^*\rangle <: \tau_0\langle r_0^*\rangle$

$\forall R_b, \varphi_b \cdot ((regs(e') \cup \{r^+, r_1^*, .., r_p^*\} \cup reg(\varphi')) \subseteq R_b \wedge \varphi_b \Rightarrow \varphi') \Longrightarrow P'; \Gamma; R_b; \varphi_b \vdash e' : \tau_0'\langle x_0^*\rangle$

We let $R_g = \{r^+, r_1^*, .., r_p^*, r_0^*\}$ and $\varphi_g = \varphi \wedge \varphi_e$. As R and φ' of the type checking rule [RC−METH], R_g contains only the region parameters of the method, while φ_g consists only of the receiver class invariant and the method precondition. By the method inference rules we directly prove that (a) for each $i = 0..p\ \ \{r_i^*\} \subseteq R_g$ and $varphi_g \Rightarrow \varphi_i$; (b) $\varphi_g \Rightarrow \varphi_0'$; and (c) $\varphi_g \Rightarrow \varphi'$. The first inference rule [RI−METH−1] ensures that there are not regions that outlive the method region parameters as $((reg(\varphi) \cup regs(e')) \cap \{r^+, r_1^*, .., r_p^*, r_0^*\}) = \emptyset$, while the second rule [RI−METH−2] ensures by the algorithm to compute $nesc$ that each region that outlives one of the method region parameters is made equivalent to one or more suitable method region parameters. Both region inference rules assume that a region localization was done before for the method body. Using the above considerations we prove that $reg(\varphi_0') \subseteq R_g$ and $(regs(e') \cup \{r^+, r_1^*, .., r_p^*\} \cup reg(\varphi')) \subseteq R_g$. The fix point analysis always adds more constraints to the collected region constraint, strengthening φ_g. Thus R_g and φ_g can be used to instantiate R_i and φ_i, R_s and φ_s and R_b and φ_b in the relations derived by the induction. Thus the type checking rule [RC−METH] is proved.

5. The validity is directly proved using the previous case of the current lemma that states that each inferred method is a valid method.

We prove $P' \vdash def'$ as follows:

The first check of the type checking rule [RC−CLASS], $r_1 \notin \bigcup_{i=1}^{p} reg(field_i)$ holds, since both inference rules [RI−CLASS−1] and [RI−CLASS−2] generate fresh regions (including the special region for the recursive fields) to annotate the fields. The second check $\varphi \Rightarrow r_i \succeq r_1 \;\; i = 2..n$ holds because (a) both inference rules ensure that the regions of the non-recursive fields outlive the first region; (b) the second inference rule ensures that the special region for the recursive fields outlives the first region; and (c) an induction on the class parent proves that the check also holds for the class parent. The third check about the methods is proved by applying the case (4) of the current lemma on the hypotheses of the current case. The fourth check (about the fields) for non-recursive fields is proved by applying the case (1) of the current lemma on the inference rules hypotheses $\vdash \tau_i \Rrightarrow \tau_i\langle r_i^* \rangle, \varphi_i$. The proof of the fourth check for recursive fields is similar to the proof of the case (1) of the current lemma. The class invariant of a recursive field is obtained from the class invariant by replacing the first region with the special region for recursive fields. Thus the class invariant entails the recursive field class invariant. Thus we proved the type checking rule [RC−CLASS]. The fixpoint analysis for the mutually recursive classes do not affect the proof since the fixpoint is strengthening the class invariant and is increasing the number of the class regions.

We prove $P' \vdash InheritanceOK(def')$ as follows:

The first check about the relation between the subclass regions and the superclass regions is validated by both inference rules. The second check about the subclass invariant and the superclass invariant is directly proved since the superclass invariant is part of the subclass invariant in both inference rules. The overriding check resolution rules from Figure 4.12 directly ensure the method overriding check.

□

A.2.5 Proof of Theorem 4.8.0.3 (Soundness and Completeness)

1. *(Soundness)*:

Based on the order in which region inference proceeds (that is described in Section 4.7), we re-organize the declarations in $P = def^*$ as the following:

$$\{cn_1^*,\ (cn.mn)_1^*\},\ ...,\ \{cn_p^*,\ (cn.mn)_p^*\}$$

Each $\{cn_i^*,\ (cn.mn)_i^*\}$ forms a strongly connected component (SCC) in the global dependency graph. By applying the algorithm described at the end of Section 4.7 and Lemma 4.8.0.2 to these SCCs in accordance with the above order, we get the well-formedness for all the class and method definitions and well-typedness of the method bodies. Mutual dependency does not pose a problem as we can provide type signature (including constraint abstraction) ahead of time before the inference is applied simultaneously to each class and method in the SCC. Notice that our override conflict resolution can only strengthen the constraint, which preserves the well-typedness due to Lemma A.2.1.5 and Lemma A.2.1.6. Using the type checking rule for a program, we can conclude that the region inference result P' is well-typed with respect to our region type system.

2. *(Program Preserving)*:

   The proof is based on Lemma 4.8.0.1.

3. *(Completeness)*:

   Applying the inference algorithm according to the order given by a dependency graph, as it was summarized in Section 4.7 we can obtain a region-annotated program P' (with region constraints) for any (well-normal-typed) source program P. The termination of the inference algorithm can be directly proved, however there are some special situations as follows:

   - *fixpoint analysis required by the (mutually-)recursive methods*: Fixpoint analysis always terminates because the finite set of possible constraints is made up from a bounded set of regions [132].
   - *overriding check resolution*: There is always a solution for those rules, in the worst case all regions are equal.
   - *constraint entailment used by region localization rule*: The entailment used in our algorithm is known to be decidable [96, 181].

   Moreover, final region constraints collected during the inference process are always satisfiable due to the fact that we only use outlives relation($\succeq$). Therefore there is always a trivial solution where all regions are equal. □

$$\frac{se = k \mid v \mid v.f \mid \textbf{null}}{\Gamma \vdash se \leadsto_d \{\},\{\}} \qquad \frac{\Gamma \vdash e \leadsto_d C,M}{\Gamma \vdash lhs = e \leadsto_d C,M} \qquad \frac{}{\Gamma \vdash \textbf{new}\ cn(v^*) \leadsto_d \{cn\},\{\}}$$

$$\frac{\begin{array}{c}\Gamma,\{(v:\tau)\} \vdash e \leadsto_d C,M \\ C' = C \cup \{\tau \mid isClassType(\tau)\}\end{array}}{\Gamma \vdash \{(\tau\ v)\ \ e\} \leadsto_d C',M} \qquad \frac{(v_0 : cn) \in \Gamma}{\Gamma \vdash v_0.mn(v_1,..,v_n) \leadsto_d \{\},\{cn.mn\}}$$

$$\frac{\Gamma \vdash e_i \leadsto_d C_i,M_i,\ i=1,2}{\Gamma \vdash e_1\ \textbf{;}\ e_2 \leadsto_d C_1 \cup C_2, M_1 \cup M_2} \qquad \frac{\Gamma \vdash e \leadsto_d C,M}{\Gamma \vdash \textbf{while}\ c\ e \leadsto_d C,M}$$

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

Figure A.3: Constituent Dependencies Inference for Expressions

A.3 Inference Rules for Dependencies

Given a Core-Java program P, the constituent dependencies, $D_i{'}\,s$ and the override dependencies, $O_i{'}\,s$ can be systematically gathered by the following rule:

$$\frac{P = def_{1..n} \quad P \vdash def_i \leadsto_o O_i \quad \vdash def_i \leadsto_d D_i \quad i = 1..n}{\vdash P \leadsto \bigcup_{i:1..n}(D_i \cup O_i)}$$

A.3.1 Inference for Constituent Dependencies

Constituent dependencies are systematically gathered for each class:

$$\frac{\begin{array}{c} def = \textbf{class}\ cn\ \textbf{extends}\ cn'\ \{(\tau_i\, f_i)_{i:1..n}\ meth_{1..m}\} \\ \{this : cn\} \vdash meth_i \leadsto_d D_i,\ i = 1..m \\ D = \{(cn \to \tau_i) \mid i \in \{1..n\}\ \wedge\ isClassType(\tau_i)\} \\ D' = \{(cn.mn_i \to cn) \mid i \in \{1..m\}\ \wedge\ mn_i = name(meth_i)\} \end{array}}{\vdash def \leadsto_d \bigcup_{i:1..m} D_i \cup \{cn \to cn'\} \cup D \cup D'}$$

The class rule collects the dependencies of the fields, methods and parent class. We define a method rule to analyse the dependencies from a method body, as follows:

$$\Gamma \vdash mn \leadsto_d C$$

where Γ is the type environment:

$$\frac{\begin{array}{c}(this : cn) \in \Gamma \qquad \Gamma,\{(v_i : \tau_i)_{i:1..n}\} \vdash e \leadsto_d C,M \\ C' = C \cup \{\tau_i \mid i \in \{0..n\}\ \wedge\ isClassType(\tau_i)\}\end{array}}{\Gamma \vdash \tau_0\ mn((\tau_i\ v_i)_{i:1..n})\ \{e\} \leadsto_d \{cn.mn \to p \mid p \in C' \cup M\}}$$

For expressions, we attempt to gather all types and methods that are being used. Given an

$$\frac{\begin{array}{c}(\textbf{class}\ cn'\ extends\ cn''\ \{...meth_{i:1..p}...\}) \in P \quad (\exists i \in 1..p\ \ meth_i = meth) \\ meth = \tau_0\, mn((\tau_i\, v_i)_{i:1..q})... \\ O = \{cn'.mn \rightarrow_o cn.mn, cn \rightarrow_o cn.mn, cn'.mn \rightarrow_o cn, cn \rightarrow_o cn'.mn\}\end{array}}{P, cn, cn' \vdash meth \rightsquigarrow_o O}$$

$$\frac{\begin{array}{c}(\textbf{class}\ cn'\ extends\ cn''\ \{...meth_{i:1..p}...\}) \in P \quad (\forall i \in 1..p\ \ meth_i \neq meth) \\ P, cn, cn'' \vdash meth \rightsquigarrow_o O\end{array}}{P, cn, cn' \vdash meth \rightsquigarrow_o O}$$

$$\frac{(\textbf{class}\ Object\ \{...meth_{i:1..p}...\}) \in P \quad (\forall i \in 1..p\ \ meth_i \neq meth)}{P, cn, Object \vdash meth \rightsquigarrow_o \{\}}$$

Figure A.4: Override Checks for a Method

expression e and a type environment Γ, we gather a set of used class types C, and a set of invoked methods M, as follows:

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

The syntax-directed inference rules for expressions are detailed in Figure A.3.

A.3.2 Inference for Override Dependencies

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

$$\frac{\begin{array}{c}(def = \textbf{class}\ cn\ extends\ cn'\ \{field^*\ meth_{1..p}\}) \in P \\ P, cn, cn' \vdash meth_i \rightsquigarrow_o O_i \quad i = 1..p\end{array}}{P \vdash def \rightsquigarrow_o \bigcup_{i:1..p} O_i}$$

For each class, we check each method to see if it overrides a corresponding method in one of its superclasses. If so, we gather the dependencies of the method overriding check. The rules for a method are shown in Figure A.4. The search is stopped either when the overridden method is found or when the top of the class hierarchy is reached. The rules use the predicate $meth_i{=}meth$ to verify if the methods have the same signature (name, result type, and parameters' types).

A.4 Handling Downcast

One important feature that is still missing from Core-Java is the *downcast* operation. In general, this operation may be type unsafe if the object in question is not the subtype that was expected. In case of region types a downcast may also be unsafe due to the region parameters of the region types. In [21], a type-passing approach was extended to carry ownership information to allow this property to be checked at runtime. If a region error is detected at runtime, the blame

```
class A⟨r1,r2⟩ ...;
class B⟨r1,r2,r3⟩ extends A⟨r1,r2⟩ ...;
class C⟨r1,r2,r3⟩ extends A⟨r1,r2⟩ ...;
class D⟨r1,r2,r3,r4⟩ extends C⟨r1,r2,r3⟩ ...;
class E⟨r1,r2,r3,r4,r5⟩ extends A⟨r1,r2⟩ ...;
   :
  A⟨r1,r2⟩ a; A⟨r1',r2'⟩ a2;
  if ..  then
    a = lb:new B⟨lb1,lb2,lb3⟩(..);//B upcast to A
  else ..
    a = lc:new C⟨lc1,lb2,lc3⟩(..);//C upcast to A
  else ..
    a = le:new E⟨le1,le2,le3,le4,le5⟩(..);//E upcast to A
  B⟨rb1',rb2',rb3'⟩ b = lbb:(B⟨rb1,rb2,rb3⟩)a;//downcast to B
  C⟨rc1',rc2',rc3'⟩ c = lcc:(C⟨rc1,rc2,rc3⟩)a;//downcast to C
  D⟨rd1',rd2',rd3',rd4'⟩ d = ldd:(D⟨rd1,rd2,rd3,rd4⟩)c;//downcast to D
```

Figure A.5: Program Fragment with Downcasts

can still be pinned on the programmer for a wrong region annotation. With automatic region inference, the onus will be on the type inference system to prevent such a situation; moreover at compile-time. In this section we elaborate how this problem can be solved.

Core-Java is extended with a new construction, that allows downcasting only on variables, as follows:

$$e ::= ... \mid (cn)\, v$$

Downcast and upcast represent opposite operations. In our present formulation, regions may be lost during upcast operations. As a consequence, we are unable to carry out region-safe downcast, as the lost regions cannot be recovered. To illustrate the problem, consider a program fragment with the class hierarchy in Figure A.5. For exposition purposes, the `new` expressions and the cast expressions are labeled with unique program points. During the upcast operations, regions `lb3,lc3,le3,le4,le5` are lost. These lost regions cannot be recovered when subsequent downcast operations are performed, leading to unknown regions `rb3,rc3,rd4`.

To support region-safe downcasting, a key technique is to preserve the regions that were supposedly lost during upcasting. We propose two solutions. The first solution is modular but imprecise, while the second solution is more precise but it requires a nonlocal flow analysis. Both our solutions are more precise than a solution based on phantom regions from *RegJava* [41] (more details are given in Appendix A.7).

The first solution preserves lost regions during upcasting by equating them with the object's first region. In this way, downcasting can always be achieved through the object's first region.

In the following example, the upcast operation forces the region `p3` to be equivalent to `p1`. As a consequence, the lost region can easily be recovered during a downcast operation from the first region `p3'=p1'`, as follows:

```
        A⟨ra1,ra2⟩ a = new B⟨p1,p2,p3⟩(..)//p3=p1∧p1⪰ra1∧p2=ra2
        ···(B⟨p1',p2',p3'⟩) a···//p3'=p1'∧p1'=ra1∧p2'=ra2
```

Applying this technique to the program fragment of Figure A.5 results in the following program (the region constraints are shown as comments):

```
A⟨r1,r2⟩ a; A⟨r1',r2'⟩ a2;
if ..  then
        a = new B⟨lb1,lb2,lb3⟩(..);//lb3=lb1∧lb1⪰r1∧lb2=r2
else ..
        a = new C⟨lc1,lc2,lc3⟩(..);//lc3=lc1∧lc1⪰r1∧lc2=r2
else ..
        a = new E⟨le1,le2,le3,le4,le5⟩(..);
        //le3=le4=le5=le1∧le1⪰r1∧le2=r2
B⟨rb1',rb2',rb3'⟩ b = (B⟨rb1,rb2,rb3⟩)a;
        //rb3=rb1∧rb1=r1∧rb2=r2∧rb1⪰rb1'∧rb2'=rb2∧rb3'=rb3
C⟨rc1',rc2',rc3'⟩ c = (C⟨rc1,rc2,rc3⟩)a;
        //rc3=rc1∧rc1=r1∧rc2=r2∧rc1⪰rc1'∧rc2'=rc2∧rc3'=rc3
D⟨rd1',rd2',rd3',rd4'⟩ d = (D⟨rd1,rd2,rd3,rd4⟩)c;
        //rd4=rd1∧rd1=rc1∧rd2=rc2∧rd3=rc3∧rd1⪰rd1'∧
        //rd2'=rd2∧rd3'=rd3∧rd4'=rd4
```

While this solution is simple to implement, some lifetime precision is lost due to the region equality constraints imposed during upcasting. These equality constraints force the additional fields to be stored in the same region as the object itself. Therefore larger regions than necessary could be generated.

The second solution maintains extra regions during upcasting, if they may be downcast subsequently. Specifically, all variables to objects that may be downcast (to some subclasses) must be padded in advance with a sufficient number of extra regions to support region-safe downcasting later. A flow-based analysis is required to determine the scope in which each object may be downcast. For each such object, we pad its region-type with a sufficient number of extra regions to support downcast operations later. In the following example, the region type of `a` is padded with the extra region `r` since `a` may be downcast to `B` later. During the upcast operation the region `p3` is equated with the padded region `r`. As a consequence, the region `p3` is not lost and the downcast operation can recover it from the padded region, `p3′=r`, as follows:

```
        A⟨ra1,ra2⟩[r] a = new B⟨p1,p2,p3⟩(..)//p3=r∧p1⪰ra1∧p2=ra2
        ···(B⟨p1′,p2′,p3′⟩)a···//p3′=r∧p1′=ra1∧p2′=ra2
```

Since extra regions of a padded region type correspond to fields, they satisfy the no-dangling requirement and outlive the first region (e.g. `r⪰ra1`). For the program fragment of Figure A.5, the flow analysis can determine that the object `a` may be downcast to `B,C,D`, while the object `c` and the cast expression labeled with `lcc` may be downcast to `D`. As the subclass `D` has the maximum number of regions, their region types are padded with upto four regions, namely `A⟨r1,r2⟩[r3,r4]` for `a`, `C⟨rc1′,rc2′,rc3′⟩[rc4′]` for `c`, and `C⟨rc1,rc2,rc3⟩[rc4]` for `lcc` to support region-safe downcast to either `B,C` or `D`. Note that `[r3,r4]`, `[rc4]` and `[rc4′]` denote the padded regions for `a`, `lcc` and `c`, respectively. In contrast, object `a2` and `b` are never downcast, hence we do not impose any extra regions on their region types. The program fragment of Figure A.5 can now be transformed to the following program (the region constraints are shown as comments):

```
A⟨r1,r2⟩[r3,r4] a; A⟨r1′,r2′⟩ a2;
if ..  then
        a = new B⟨lb1,lb2,lb3⟩(..);//lb3=r3∧lb1⪰r1∧lb2=r2
else ..
        a = new C⟨lc1,lc2,lc3⟩(..);//lc3=r3∧lc1⪰r1∧lc2=r2
else ..
        a = new E⟨le1,le2,le3,le4,le5⟩(..);
         //le3=r3∧le4=r4∧le5=le1∧le1⪰r1∧le2=r2
B⟨rb1′,rb2′,rb3′⟩ b = (B⟨rb1,rb2,rb3⟩)a;
//rb3=r3∧rb1=r1∧rb2=r2∧rb1⪰rb1′∧rb2′=rb2∧rb3′=rb3
C⟨rc1′,rc2′,rc3′⟩[rc4′] c = (C⟨rc1,rc2,rc3⟩[rc4])a;
//rc3=r3∧rc4=r4∧rc1=r1∧rc2=r2∧rc1⪰rc1′∧rc2′=rc2∧rc3′=rc3∧rc4′=rc4
D⟨rd1′,rd2′,rd3′,rd4′⟩ d = (D⟨rd1,rd2,rd3,rd4⟩)c;
//rd4=rc4′∧rd1=rc1′∧rd2=rc2′∧rd3=rc3′∧rd1⪰rd1′
//∧rd2′=rd2∧rd3′=rd3∧rd4′=rd4
```

Take note that `a` does not have enough padded regions to save the extra regions `le3,le4,le5` of `E`. Hence, the extra region `le5` of `E` is saved into the first region `le1` (as the first solution does). The reason is that the `E` class is not in the set to which object `a` may be downcast.

However this padded region solution requires alias analysis for downcasting on the object fields. Moreover, the region types of the object fields also need to be padded in advance with extra regions. The fields' padded regions may lead to a tree-like structure of the types' region annotations. In order to avoid these problems and to keep the region annotations simpler (list-like structure), we propose the following mixed solution: the first solution (that uses first region) for downcasting on object fields and the second solution (that uses padded regions) for downcasting on objects. We illustrate our mixed solution on the program fragment of Figure A.6. Our flow analysis attempts to determine the possible downcasts for each object from the program. In order to avoid the complexity of alias analysis, our flow analysis does not trace the downcasts for the fields and through the fields. For the previous program fragment, the analysis can determine that object `v3` may be downcast to `Pair,Cell`, while the object `v1` may be downcast to `Cell`. The region types of these objects are padded with extra regions, `v3` upto three regions and `v1` upto two regions, respectively. The result is `Object⟨rv3⟩[p1,p2]` for `v3` and `Object⟨rv1⟩[p3]` for `v1`. Note that the downcasts for the fields `a.fst` and `b.fst` are not captured. Since `v1` flows into `v3` through the field `fst`, the downcast of `v1` to `Pair` also cannot be determined. Downcasts that are not captured by the flow analysis are resolved using the first region. The program fragment of Figure A.6 can be transformed into the following program (the region constraints and the region subtyping rules of Figure A.7 are shown as comments):

```
Pair⟨ra1,ra2,ra3⟩ a;
Pair⟨rb1,rb2,rb3⟩ b;
Object⟨rv1⟩[p3] v1;
if ..  then
   v1=new Pair⟨l1,l2,l3⟩(..);//l1⪰rv1∧l2=p3∧l3=l1(Rule[PadSubClass-3])
else ..
   v1 = new Cell⟨l1′,l2′⟩(..);//l1′⪰rv1∧l2′=p3 (Rule [PadSubClass-1])
a=b;//rb1⪰ra1∧rb2=ra2∧rb3=ra3
b.fst = v1;//rv1⪰rb2∧p3=rv1 (Rule [PadRegSub-2])
Object⟨rv3⟩[p1,p2] v3 = a.fst;//ra2⪰rv3 (Rule [PadRegSub-3])
Pair⟨rc1′,rc2′,rc3′⟩ c = (Pair⟨rc1,rc2,rc3⟩) v3; //(Rule [RegEqual-2])
//rc1=rv3∧rc2=p1∧rc3=p2∧rc1⪰rc1′∧rc2=rc2′∧rc3=rc3′
```

```
class Cell⟨r1,r2⟩ extends Object⟨r1⟩ ...;
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩ {
        Object⟨r2⟩ fst;
        Object⟨r3⟩ snd; ...  }
:
Pair⟨ra1,ra2,ra3⟩ a;
Pair⟨rb1,rb2,rb3⟩ b;
Object⟨rv1⟩ v1;
if ..  then
        v1 = lp:new Pair⟨l1,l2,l3⟩(..);//Pair upcast to Object
else ..
        v1 = lc:new Cell⟨l1',l2'⟩(..);//Cell upcast to Object
a=b;//two aliases are created
b.fst = v1;// flow into a field
Object⟨rv3⟩ v3 = a.fst;//flow from a field
Pair⟨rc1',rc2',rc3'⟩ c = lcc:(Pair⟨rc1,rc2,rc3⟩) v3;//downcast to Pair
v1=v3;//two aliases are created
Cell⟨rd1',rd2'⟩ d = ldd:(Cell⟨rd1,rd2⟩) v1;//downcast to Cell
```

Figure A.6: Program Fragment with Downcasts

```
v1=v3;//rv3⪰rv1∧p3=p1∧p2=rv3 (Rule [PadRegSub-1])
Cell⟨rd1',rd2'⟩ d = (Cell⟨rd1,rd2⟩) v1; //(Rule [RegEqual-2])
//rd1=rv1∧rd2=p3∧rd1⪰rd1'∧rd2=rd2'
```

Figure A.7 contains the additional region subtyping rules that support downcasting. The first three rules [**PadSubClass−1**], [**PadSubClass−2**], and [**PadSubClass−3**] extend to padded region types the class subtyping rule [**SubClass**] of Figure 3.6. The extra regions of the subtype are saved either into the padded regions of the supertype or into the first region of the subtype. These rules can be used by the mixed solution. When the first solution is applied alone (all padded region types have zero extra regions), we use only the rule [**PadSubClass−2**] by making n equal to zero. Next three rules [**PadRegSub−1**], [**PadRegSub−2**], and [**PadRegSub−3**] define the region subtyping for the padded region types. When both types are padded region types, the supertype cannot have more padded regions than the subtype due to the downcast flow propagation rules.

The last two rules of Figure A.7 are used for the downcast expressions. They define the equality relation $\cong_r$ between the regions of the target type (written on the left hand side) and the regions of the type to be cast (written on the right hand side). The first solution ([**RegEqual1**]) equates the additional regions of the target type with its first region. In case of the mixed solution, the target type and the type to be cast have the same number of regions ([**RegEqual2**]).

The backward flow analysis that computes in advance the number of the padded regions is formulated as a global analysis in the next section.

$$[\textbf{PadSubClass}-\textbf{1}]$$

$$\frac{\begin{array}{c} 0<p\ \ 0\leq q\ \ 0\leq k\ \ 0\leq n\quad p+q+k\leq p+q+k+n \\ \textbf{class}\ cn\langle x_{1..(p+q+k)}\rangle\ \textbf{extends}\ cn'\langle x_{1..(p+q)}\rangle ... \in P' \\ \varphi' = \bigwedge_{i=1}^{k}(r_{(p+q+i)} = r'_{(p+q+i)}) \\ \vdash cn'\langle r_{1..(p+q)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+q)}], \varphi \end{array}}{\vdash cn\langle r_{1..(p+q+k)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+q+k+n)}], \varphi\wedge\varphi'}$$

$$[\textbf{PadSubClass}-\textbf{2}]$$

$$\frac{\begin{array}{c} 0<p\ \ 0\leq q\ \ 0\leq k\ \ 0\leq n\leq q\quad p+q>p+n \\ \textbf{class}\ cn\langle x_{1..(p+q+k)}\rangle\ \textbf{extends}\ cn'\langle x_{1..(p+q)}\rangle ... \in P' \\ \varphi' = \bigwedge_{i=1}^{k}(r_{(p+q+i)} = r_1) \\ \vdash cn'\langle r_{1..(p+q)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+n)}], \varphi \end{array}}{\vdash cn\langle r_{1..(p+q+k)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+n)}], \varphi\wedge\varphi'}$$

$$[\textbf{PadSubClass}-\textbf{3}]$$

$$\frac{\begin{array}{c} 0<p\ \ 0\leq q\ \ 0\leq k\ \ 0<n<k\quad p+q+k>p+q+n \\ \textbf{class}\ cn\langle x_{1..(p+q+k)}\rangle\ \textbf{extends}\ cn'\langle x_{1..(p+q)}\rangle ... \in P' \\ \varphi' = \bigwedge_{i=1}^{n}(r_{(p+q+i)} = r'_{(p+q+i)}) \wedge \bigwedge_{i=1}^{k-n}(r_{(p+q+n+i)} = r_1) \\ \vdash cn'\langle r_{1..(p+q)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+q)}], \varphi \end{array}}{\vdash cn\langle r_{1..(p+q+k)}\rangle <: cn''\langle r'_{1..p}\rangle[r'_{(p+1)..(p+q+n)}], \varphi\wedge\varphi'}$$

$$[\textbf{PadRegSub}-\textbf{1}]$$

$$\frac{\begin{array}{c} 0<m\ \ 0\leq k\ \ 0\leq p\ \ 0\leq q\quad 0<p+q\quad m+k+p+q\geq m+k+p \\ \varphi' = \bigwedge_{i=1}^{q}(r_{(m+k+p+i)} = r_1) \wedge \bigwedge_{i=1}^{p}(r_{(m+k+i)} = r'_{m+k+i}) \\ \vdash cn\langle r_{1..(m+k)}\rangle <: cn'\langle r'_{1..m}\rangle[r'_{(m+1)..(m+k)}], \varphi \end{array}}{\vdash cn\langle r_{1..(m+k)}\rangle[r_{(m+k+1)..(m+k+p+q)}] <: cn'\langle r'_{1..m}\rangle[r'_{(m+1)..(m+k+p)}], \varphi\wedge\varphi'}$$

$$[\textbf{PadRegSub}-\textbf{2}]$$

$$\frac{\begin{array}{c} 0<m\ \ 0\leq k\ \ 0<p\ \ 0\leq n\leq k\quad m+k+p>m+n \\ \varphi' = \bigwedge_{i=1}^{p}(r_{(m+k+i)} = r_1) \\ \vdash cn\langle r_{1..(m+k)}\rangle <: cn'\langle r'_{1..m}\rangle[r'_{(m+1)..(m+n)}], \varphi \end{array}}{\vdash cn\langle r_{1..(m+k)}\rangle[r_{(m+k+1)..(m+k+p)}] <: cn'\langle r'_{1..m}\rangle[r'_{(m+1)..(m+n)}], \varphi\wedge\varphi'}$$

$$[\textbf{PadRegSub}-\textbf{3}] \qquad\qquad [\textbf{RegEqual}-\textbf{1}]$$

$$\frac{\vdash cn\langle r_{1..m}\rangle <: cn\langle r'_{1..m}\rangle, \varphi}{\vdash cn\langle r_{1..m}\rangle <: cn\langle r'_{1..m}\rangle[r'_{(m+1)..(m+p)}], \varphi} \qquad \frac{\varphi = \bigwedge_{i=1}^{m}(r_i = r'_i) \wedge \bigwedge_{i=1}^{p}(r_{(m+i)} = r_1)}{\vdash cn\langle r_{1..(m+p)}\rangle \cong_r cn'\langle r'_{1..m}\rangle, \varphi}$$

$$[\textbf{RegEqual}-\textbf{2}]$$

$$\frac{\varphi = \bigwedge_{i=1}^{(m+p+q)}(r_i = r'_i)}{\vdash cn\langle r_{1..(m+p)}\rangle[r_{(m+p+1)..(m+p+q)}] \cong_r cn'\langle r'_{1..m}\rangle[r'_{(m+1)..(m+p+q)}], \varphi}$$

Figure A.7: Region Subtyping Rules for Downcast

A.4.1 Backward Flow Analysis

We introduce a backward flow analysis that attempts to analyse downcast operations which may be subsequently performed for the objects of a given program. In the interest of simplicity, our analysis is flow-insensitive and context-insensitive. It also does not trace the downcasts for the object fields and through the object fields.

Moreover it is a global analysis for which each program point is represented uniquely. The parameters `v`* and the receiver of each instance method `mn` of a class `cn` are represented by `cn.mn.v`* and `cn.mn.this`, while its result is represented by `cn.mn`. Every `new` expression is labeled with a unique program point using `l:new cn(..)`, and similarly for each block `l:{(t v) e}`. The purpose of labeling the former is to identify its source location, while the label for block is to allow local variables to be uniquely renamed. Specifically, we rename `v` by `l.v` in `l:{(t v) e}` to make all variable declarations unique. We also attach a label to cast expression `l:(t)v` in order to determine the possible subsequently downcasts of the intermediate value of type `t`.

The purpose of this flow analysis is to identify the set of classes to which the object at a program point may be downcast later. The algorithm consists of two steps: first it gathers the flows to build the flow graph and then propagates the backward flows through the graph until a fixpoint is reached.

The new inference rule to gather the set of backward flows is defined as:

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

where x is a *receiver* that may capture the result of e under the type environment Γ. The receiver can be a variable, a method parameter, a method receiver or a method result. The output C denotes the set of backward flows that occur in e and its receiver x. For convenience, we omit Γ in the rules and assume that it is properly maintained with the set of live variables and their types at each scope. Each backward flow is represented using either $v_1 \dashrightarrow v_2$ or $v_1 \xrightarrow{D} v_2$, where the arrows indicate that v_1 captures a value from v_2. In addition, the second arrow is annotated with a D-class to indicate that its source, namely v_2, may be subjected to a downcast-to-D operation. The rule for downcast operation itself is defined as follows:

$$x \vdash l : (D)\, v, \{x \xrightarrow{D} v, x \dashrightarrow l\}$$

where the backward flow $x \xrightarrow{D} v$ denotes that the value of v may flow into its outer receiver x and be subjected to a downcast operation, while the backward flow $x \dashrightarrow l$ denotes that the intermediate value (of type `D`) of cast expression may flow into its outer receiver x.

The next few rules show the backward flows of variables, object fields and primitives:

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

Take note that we only capture the flows of object values and ignore those from object fields and primitive values, including `null`.

The next few rules are due to various expressions. Note that the flow into fields is ignored. Both branches of a conditional flow into its outer receiver. For expression block, the local variable is made unique with the help of label *l*. In the case of the `new c(v`$_{1..p}$`)` constructor, the created object denoted by *l* flows into the outer receiver.

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

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

To gather the set of backward flows that occur within each method *mn* of the class *cn*, we introduce the relation: $cn \vdash \mathit{meth}, R; W; C$, where *R* and *W* denote the sets of parameters that are being used as source (covariant via read) and destination (contravariant via write), respectively:

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

Some parameters are used as both read and write, and they appear twice. Others may not appear in either set, if they are of primitive types or only their fields are used in the method body. The following functions classify the parameters based on the directions of value flow, as follows:

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

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

The function *link*, connects up the current arguments $w_{1..n}$ of a method with its formal parameters using the sets, R and W, for reading and writing.

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

In the case of a method invocation, the result of the method flows into outer receiver *x*. Due to method overriding and inheritance, each method invocation may have to deal with a set of methods at compile-time. To handle this, we provide bidirectional flows for each of the non-primitive parameters, as shown below:

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

The backward flows for the entire program are captured by the next two rules:

$$\frac{\vdash \mathit{def}_i, C_i \quad i = 1, .., n}{\vdash \mathit{def}_{1..n},\ \bigcup_{1..n} C_i}$$

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

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

A.4.1.1 Transitive Closure of Flows

After the entire set of backward flows is generated, we can proceed to perform a transitive closure to gather all program points that could be downcast. The goal of our analysis is to find a set of classes that could be subsequently downcast for each object at a given program point.

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

$$v \xrightarrow{D} w\ \wedge\ w[S]\ \Rightarrow\ v \dashrightarrow w\ \wedge\ w[S \cup \{D\}]$$

Once this rule is applied to all arrows with downcasts, we can proceed to perform the *closure of downcast flow analysis* using the following rule:

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

This rule is repeatedly applied until a fixpoint is reached. Closure of downcast terminates as there is a finite number of classes for each program. Consider the earlier program fragment of

Figure A.5, the gathered set of flows is:

$$\{a \dashrightarrow lb,\ a \dashrightarrow lc,\ a \dashrightarrow le,\ b \xrightarrow{B} a,\ b \dashrightarrow lbb,\ c \xrightarrow{C} a,\ c \dashrightarrow lcc,\ d \xrightarrow{D} c,\ d \dashrightarrow ldd\}$$

The arrows with downcasts are initially converted as follows:

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

Applying the downcast closure rule we obtain the following result:

$$a[B,C,D],\ c[D],\ lcc[D],\ lb[B,C,D],\ lc[B,C,D],\ le[B,C,D]$$

This outcome guides the padding of extra regions for each variable declaration, object creation site or target type of a cast expression.

Considering the program fragment of Figure A.6, the gathered set of flows does not contain the flow from/to fields:

$$\{v1 \dashrightarrow lp,\ v1 \dashrightarrow lc,\ a \dashrightarrow b,\ c \xrightarrow{Pair} v3,\ c \dashrightarrow lcc,\ v1 \dashrightarrow v3,\ d \xrightarrow{Cell} v1,\ d \dashrightarrow ldd\}$$

Converting the arrows with downcasts and then applying the downcast closure rule we obtain the following result:

$$v1[Cell],\ v3[Cell,Pair],\ lp[Cell],\ lc[Cell]$$

A.5 Runtime regions

In this section we present two analyses on region-annotated programs which are done before code generation. The first analysis, called region coalescing, minimizes (where possible) the number of method's region parameters which are used in the method's body. Using the result of the region coalescing, the second analysis generates the region handles which may need to be passed to each method at its call sites at runtime.

A.5.1 Region Coalescing

Each method takes a set of region parameters. In the context of the method's precondition, some of the method region parameters may be equivalent denoting regions with the same lifetime. The equivalent method's region parameters may be coalesced together to minimize the number of region parameters which are used in the method's body. This simplification helps the next region handles analysis to reduce the number of the region parameters that have to be passed to the methods' call sites at runtime. For instance, given the following method with three region parameters and the following precondition:

$$\boxed{\textbf{RCO-Class}}$$

$$\frac{\varphi,\ \{\mathit{this} : \mathit{cn}\langle r_{1..n}\rangle\} \vdash \mathit{meth}_i \hookrightarrow \mathit{meth}'_i, i = 1..q}{\begin{array}{c}\vdash \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..l}\rangle \textbf{where}\ \varphi\ \{\mathit{fd}_{i:1..p}\ \mathit{meth}_{i:1..q}\}\\ \hookrightarrow \textbf{class}\ \mathit{cn}\langle r_{1..n}\rangle\ \textbf{extends}\ \mathit{cn}'\langle r_{1..l}\rangle \textbf{where}\ \varphi\ \{\mathit{fd}_{i:1..p}\ \mathit{meth}'_{i:1..q}\}\end{array}}$$

$$\boxed{\textbf{RCO-Meth}}$$

$$\frac{R{=}\mathit{reg}(\Gamma){\cup}\{r^+\} \quad \rho{=}\mathit{buildeq}(\{\}, R, \varphi{\wedge}\varphi') \quad e' = \rho\, e}{\begin{array}{c}\varphi,\ \Gamma\ \vdash \tau_0\langle x_0^*\rangle\ \mathit{mn}\langle r^+\rangle((\tau_j\langle x_j^*\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi'\ \{e\}\\ \hookrightarrow \tau_0\langle x_0^*\rangle\ \mathit{mn}\langle r^+\rangle((\tau_j\langle x_j^*\rangle\ v_j)_{j:2..p})\ \textbf{where}\ \varphi'\ \{e'\}\end{array}}$$

$$\frac{}{\mathit{buildeq}(S, [\,], \varphi) =_{def} [\,]} \qquad \frac{\neg(\exists s \in S \cdot \varphi \Rightarrow (r{=}s))}{\mathit{buildeq}(S, \{r\}{\cup}R, \varphi) =_{def} \mathit{buildeq}(S{\cup}\{r\}, R, \varphi)}$$

$$\frac{\exists s \in S \cdot \varphi \Rightarrow (r{=}s)}{\mathit{buildeq}(S, \{r\}{\cup}R, \varphi) =_{def} [r{\mapsto}s]{\cup}\mathit{buildeq}(S, R, \varphi)}$$

$\rho\, e$	region substitution on an expression
$\mathit{reg}(\Gamma)$	computes the region variables of a type environment types (see Figure 4.6)

Figure A.8: Region Coalescing Analysis

τ_0 `mn⟨r1,r2,r3⟩(`τ_1`,..,`τ_k`) where r2=r3 {e}`

The method's body can use either the region `r2` or `r3`, as both are equivalent. Thus the substitution ρ=`[r2`$\mapsto$`r3]` can be applied on the method's body as follows:

τ_0 `mn⟨r1,r2,r3⟩(`τ_1`,..,`τ_k`) where r2=r3 {`ρ`e}`

Region coalescing rules are shown in Figure A.8. The first two rules collect the receiver's class invariant and the method's precondition for each method. The last three rules describe the function *buildeq*, that derives a substitution to map equivalent regions together. The body of each method is then subjected to its corresponding region identity substitution.

A.5.2 Region Handles

In a region-annotated program, all regions of parameters (including receiver) and results are passed into each method. These regions may be accessed for reading, updating or creation of the objects. However, in the final code, we are only required to have access to the handles of regions that may be allocated with new objects. Therefore we perform a transformation to generate a corresponding program with such region handles. Our analysis is isomorphic to *get-region dropping* from [18]. For each expression, we generate a corresponding expression with a minimal set of required region handles, namely those that may be allocated with new objects. We use the following inference rule:

$$\Gamma \vdash e \hookrightarrow e' \ \#\ W$$

$$\frac{se = k \mid v \mid v.f \mid \textbf{null}}{\Gamma \vdash se \hookrightarrow se \;\#\; \{\}} \qquad \frac{\Gamma \vdash e \hookrightarrow e' \;\#\; W}{\Gamma \vdash lhs{=}e \hookrightarrow lhs{=}e' \;\#\; W} \qquad \frac{\Gamma \vdash e_i \hookrightarrow e'_i \;\#\; W_i \quad i{=}1,2}{\Gamma \vdash e_1;e_2 \hookrightarrow e'_1\ ;\ e'_2 \;\#\; W_1 \cup W_2}$$

$$\Gamma \vdash \textbf{new}\ cn\langle r_{1..n}\rangle(v_1..v_m) \hookrightarrow \textbf{new}\ cn(v_1..v_m)@r_1 \;\#\; \{r_1\}$$

$$\frac{\Gamma \vdash e_i \hookrightarrow e'_i \;\#\; W_i \quad i{=}1,2}{\Gamma \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \hookrightarrow \textbf{if}\ v\ \textbf{then}\ e'_1\ \textbf{else}\ e'_2 \;\#\; W_1 \cup W_2}$$

$$\frac{\Gamma, \{(v : t)\} \vdash e \hookrightarrow e' \;\#\; W \qquad \tau'{=}erase(t)}{\Gamma \vdash \{(t\ v)\ e\} \hookrightarrow \{(\tau'\ v)\ e'\} \;\#\; W}$$

$$\frac{\Gamma \vdash e \hookrightarrow e' \;\#\; W \qquad r \in W}{\Gamma \vdash \textbf{letreg}\ r\ \textbf{in}\ e \hookrightarrow \textbf{letreg}\ r\ \textbf{in}\ e' \;\#\; W \setminus r} \quad \frac{\Gamma \vdash e \hookrightarrow e' \;\#\; W \qquad r \notin W}{\Gamma \vdash \textbf{letreg}\ r\ \textbf{in}\ e \hookrightarrow e' \;\#\; W}$$

$$\frac{(v'_1 : cn\langle x'^{+}_1\rangle) {\in} \Gamma \quad (\tau_0\ mn\langle x'^{+}_1, y^* \;\#\; W\rangle((t_j\ v_j)_{j:2..p})\ \{e\}) {\in} cn\langle x'^{+}_1\rangle \quad W'{=}[y^* {\mapsto} y'^*]W}{\Gamma\ \vdash v'_1.mn\langle x'^{+}_1, y'^*\rangle(v'_2, ..., v'_p) \hookrightarrow v'_1.mn\langle W'\rangle(v'_2, ..., v'_p) \;\#\; W'}$$

Figure A.9: Region Handles Analysis for Expressions

where Γ is the region type environment, e is a well-typed region annotated expression, e' is the expression with region handles and W is the set of regions that may be allocated with new objects. Note that region types are erased in the target program. The syntax-directed inference rules for expressions are detailed in Figure A.9.

$$\frac{\Gamma, \{(v_j : \tau_j\langle x^*_j\rangle)\}_{j:1..p} \vdash e \hookrightarrow e' \;\#\; W \quad (this : cn\langle r^+\rangle) \in \Gamma \quad \{y^+\}{=}\{r^+\}{\cup}\{z^*\}}{\begin{array}{c}\Gamma\ \vdash (\tau_0\langle x^*_0\rangle\ mn\langle y^+\rangle((\tau_j\langle x^*_j\rangle\ v_j)_{j:1..p})\ \textbf{where}\ \varphi\ \{e\}) \\ \hookrightarrow (\tau_0\ mn\langle r^+, z^* \;\#\; W\rangle((\tau_j\ v_j)_{j:1..p})\ \{e'\})\end{array}}$$

$$\frac{(\{this : cn\langle r_{1..n}\rangle\} \vdash meth_i \hookrightarrow meth'_i)_{i:1..q} \qquad (fd'_i = erase(fd_i))_{i:1..p}}{\begin{array}{c}\vdash \textbf{class}\ cn\langle r_{1..l}, r_{l+1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..l}\rangle \textbf{where}\ \varphi\ \{fd_{i:1..p}\ meth_{i:1..q}\} \\ \hookrightarrow \textbf{class}\ cn\ \textbf{extends}\ cn'\ \{fd'_{i:1..p}\ meth'_{i:1..q}\}\end{array}}$$

$$\frac{\begin{array}{c} meth_1{=}t_0\ mn\langle r_{1..p}, z_{1..n} \# W_1\rangle((\tau\ v)_{1..m})\{e_1\} \\ meth_2{=}t_0\ mn\langle r'_{1..p}, z'_{1..n} \# W_2\rangle((\tau\ v)_{1..m})\{e_2\} \\ \rho = [r_{1..p} \mapsto r'_{1..p}][z_{1..n} \mapsto z'_{1..n}] \quad W{=}(\rho W_1){\cup}W_2 \\ meth'_1{=}t_0\ mn\langle r'_{1..p}, z'_{1..n} \# W\rangle((\tau\ v)_{1..m})\{\rho e_1\} \\ meth'_2{=}t_0\ mn\langle r'_{1..p}, z'_{1..n} \# W\rangle((\tau\ v)_{1..m})\{e_2\} \end{array}}{P \vdash Overrides(meth_1, meth_2) \hookrightarrow Overrides(meth'_1, meth'_2)}$$

Figure A.10: Region Handles Analysis for Methods

The transformation rules for each method and class are shown in Figure A.10. The rule for the method declaration decomposes the list of the region parameters into three components: the receiver's regions r^+, the regions of the method's parameters and result z^*, and the regions W that may be allocated in the method body. Only the regions of W are required at runtime. The other two annotations, r^+ and z^* are used by the current analysis for the method call expression and to solve the overriding. $Overrides(meth_1, meth_2)$ relation computes the minimal union set of all the regions that may be allocated either by the overridden method or by the overriding method, or by both methods.

A.6 Discussion of Other Java Features

In this section we discuss other features of Java that may be handled in our approach to region inference.

Arithmetic, Logic, and Comparison Operations

In order to simplify the semantics and the inference rules, our translator translates all Java (arithmetic, logic, and comparison) operations into corresponding methods of a special class.

Pass-by-reference mechanism

Our Core-Java language can be extended to support in-out parameters for methods. The region subtyping rule [**InvRegSub**] from Figure 3.6 can be used to support those flows that are based on pass-by-reference semantics.

Statements

In order to support statements and exceptions we can extend Core-Java with new language constructs, as follows:

$$e ::= ... \mid \textbf{return}\ [v] \mid \textbf{throw}\ [v] \mid \textbf{try}\ e\ [\textbf{catch}\ (\tau\ v\ e)]^* \mid \textbf{break} \mid \textbf{continue}$$

Unstructured control flow statements of Java are translated in Core-Java using conditional statements. Translation of Java to Core-Java is done automatically by our translator. Although our region inference is a flow-insensitive analysis, it has to take into account the different ways to escape from the method body. The statements break and continue can be directly handled by our region inference rules, because they can only change the local flow. However for return and throw statements, we propose a new inference judgment, based on the *(normal type, return type, exception type) tuple* from [55]:

$$\Gamma \vdash e \Rrightarrow_e e' : t\#t_r\#\{t_{i:1..m}\}, \varphi$$

where t is the *normal type* that characterizes normal execution of the expression without any return or throw statement, t_r is the *return type* that denotes the return type of the current method, and $\{t_{i:1..m}\}$ is the *set of exception types* denoting the types of all uncaught exceptions thrown during the execution of e. The types of the return statements from expression e are subtypes of t_r and the generated constraints are collected into φ. Note that the collected constraint φ consists of all region constraints referring to all three kinds of types. The above judgment allows the localization into regions of the exceptions which are caught by the catch clauses of *try-catch* mechanism.

Exception mechanism

The treatment of exceptions raises two issues in the context of region inference.

First, when an exception occurs, the exception object being thrown may escape non-lexically through the run-time call stack before it is caught by an exception handler. As a result, exceptions themselves are harder to be placed into regions. One simple solution is to place them into a special region that lives forever (like heap). A more precise solution is to localize exceptions using the new inference judgment $\Rrightarrow_e$ proposed above.

Second, if a method terminates abruptly with an exception, the program must reclaim all of the local regions that are still live. The number of regions to deallocate is not known at compile time. A simple solution relies on run-time support. For example, Cyclone [80] stores region handles and exception handlers in an integrated list that operates in a last-in-first-out manner. When an exception is thrown, the list is traversed deallocating regions until an exception handler is met. In this fashion, a region is always deallocated when control returns. We propose a similar solution. The $letreg$ expression is compiled as *try-finally* mechanism that deallocates the current region in the *finally* clause.

Static fields and methods

Static class fields may also be added to Core-Java. As they must persist throughout the entire program execution, objects created here must be placed in a special region that lives forever (like heap). Static methods are treated in a similar fashion as instance methods, except that they cannot be overwritten and do not have a receiver.

Interfaces

An interface produces a completely abstract class without any method definition nor fields. A major unknown is the number of region parameters that we should allocate for each interface in order to support region-safe upcast and downcast operations (Appendix A.4). A simple solution (similar to the first solution for downcast) is to automatically provide each interface `I` with three regions, for example `I`$\langle$`ri1,ri2,ri3`$\rangle$, where `ri1` is the region for the instance object, `ri2` is the region for non-recursive fields and `ri3` is the region for recursive fields. With this, any upcast (or downcast) from a class into its interface would map its non-recursive fields into the second region and the recursive fields into the third region of the interface. The classes themselves can have as any many region parameters in accordance with inference technique of Section 4.2 for classes. The mapping of regions only occur during upcast and downcast operations. The following program shows how regions of classes with interfaces are inferred:

```
interface I⟨ri1,ri2,ri3⟩ {...}
class A⟨ra1,ra2⟩...implements I
      {...A⟨ra2,ra2⟩ fld;...}//one recursive field
class B⟨rb1,rb2,rb3⟩...implements I {...} //no recursive fields
:
I⟨r1,r2,r3⟩ i;
Pair⟨r4,r5,r6⟩ pp;
if ...  then ...
     i= new A⟨r7,r8⟩(...);//r7⪰r1∧r8=r3
else ...
     i= new B⟨r9,r10,r11⟩(...);//r9⪰r1∧r10=r11=r2
   :
   ...(A⟨r12,r13⟩)i...//r12=r1∧r13=r3
```

Since an interface is a superclass of all classes that implement it, the interface class invariant has to satisfy the class subtyping check and each interface method has to satisfy the method overriding check. The interface class invariant is always *true* such that the class subtyping check is enforced by default. The precondition of an interface method has to be stronger than all the preconditions of the methods which implement that interface method. We start with the interface method precondition set to *true* and then for each method that implements the interface method we strengthen the interface method precondition according to the method overriding resolution from Section 4.6. After each step the current interface method precondition is a strengthening of the previous precondition, therefore the previous overriding checks still hold.

Arrays

We separate the array and its components in different regions. Thus, the region type of an array consists of two parts: the first region is for the array itself and the rest of the regions correspond to the region type of the array's components. Due to no-dangling requirement all regions of the array's components outlive the region where the array itself is stored.

Multi-threading

Multi-threading in Java is used to support concurrency. Each thread may have its own execution lifetime that is synchronized by access to shared objects. Correspondingly, regions may also be shared amongst a group of threads. The lifetime of regions must outlive the last use amongst its thread clients. This can be guaranteed via a reference counting mechanism on the shared region, but stack-like behavior is partially lost, so each region may only be deleted after all processes which use it have released access. SafeJava [23] and an extension of Cyclone [79] extend region types to multithreaded programs by allowing explicit memory management for objects shared between threads. They allow threads to communicate through objects in shared regions in addition to the heap. A shared region is deleted when all threads exit the region. A similar solution may also be adopted though such regions may have longer lifetimes due to the need to wait for concurrent processes to release the shared region. There are a couple of solutions to avoid the potential for memory leaks. One solution proposed by SafeJava [23] is to use *subregions* that can be recycled and reallocated, while its parent region remains live. Another solution is to use linearity analysis to determine objects that have become dead and thus may have their space recycled [113]. We prefer the latter solution as it is closer to the approach of automatic region inference.

Generic types

At present, although each class in Core-Java is region polymorphic, the base type is still monomorphic. There have been several recent proposals [24, 193] to add generic types to Java. Such extensions can help reduce the number of downcast operations, and could be used to improve on the lifetimes of the regions. To support genericity, we have to modify the region type system to support polymorphic region variables. A polymorphic region variable denotes a set of regions like the original work in ML [191] and can be instantiated, similar to techniques proposed in [77].

Reflection

Reflection mechanism in Java allows the programmer to perform runtime actions given the descriptions of the objects involved: one can create objects given their class names, access objects fields given their name, and call methods by their name. A common usage pattern for object creation using the reflection APIs is shown below:

```
String className = ...;//class name is provided at run time
Class c = Class.forName(className);//returns a class given its name
Object o = c.newInstance();//creates an instance of the class c
A t = (A) o; //cast to an appropriate type A
```

Using the class name, `Class.forName` creates a `Class` object and then `newInstance` creates a new instance of that class. The new instance is upcast to `Object` and then is downcast to an appropriate type. The appropriate type is either the same or a superclass of the class whose name was given by the string `className`. Based on the explicit cast operation, our region inference algorithm can use the same approaches as those used for downcasting in Appendix A.4. For instance, we assume that the region type that corresponds to class `A` is `A⟨r1,r2⟩`. Using the modular solution based on the first region we obtain the following result. Note that a region handle is passed to `newInstance` such that the new instance is allocated in that region.

```
Object⟨r1⟩ obj = c.newInstance⟨r1⟩();
A⟨r1,r1⟩ t = (A⟨r1,r1⟩) obj;
```

In case of the dynamic loading we can use techniques from [117] to estimate all possible classes for which `newInstance` may create a new instance.

A.7 Our Approach vs. Phantom Region Based Approach

In RegJava [41], all classes within the same class hierarchy have the same set of region parameters. As a result many phantom regions may be introduced for superclasses. The main advantage of this approach is that it can provide immediate support to both method overriding and downcast. However, phantom regions may pose a number of problems for region inference. For example, if we have to patch up the `Object` class with the two extra regions `r2,r3` from `Pair`, we have several more issues to consider, including:

- Is the outlive relation, namely $r2 \succeq r1 \wedge r3 \succeq r1$, required on the phantom regions for the `Object` class?
- What specific regions should be used for the `Object` fields that are found in other classes, such as the `Pair` class itself?
- How should the extra regions from the other sub-classes of `Object` be handled? Must phantom regions be propagated mutually across the sub-classes, via their common superclass, as done in [41]?

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

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

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

This inclusion of phantom regions forces all connected `Pair` objects to have the same region for their `fst` field, and another region for `snd`. For instance, we consider a simple program, as follows:

```
Pair⟨ra1,ra2,ra3⟩ pa;//ra2⪰ra1 ∧ ra3⪰ra1
Pair⟨rb1,rb2,rb3⟩ pb;//rb2⪰rb1 ∧ rb3⪰rb1
Pair⟨rc1,rc2,rc3⟩ pc;//rc2⪰rc1 ∧ rc3⪰rc1
pa.fst = pb; //rb1⪰ra2 ∧ rb2=ra2 ∧ rb3=ra3
pa.snd = pc; //rc1⪰ra3 ∧ rc2=ra2 ∧ rc3=ra3
```

Solving the constraints we obtain the following types:

```
Pair⟨ra1,ra2,ra3⟩ pa;
Pair⟨ra2,ra2,ra3⟩ pb;
Pair⟨ra3,ra2,ra3⟩ pc;
```

In the case of the example of Figure 4.1, the phantom regions force objects `p2,p4` to be in the same region, and similarly for objects `p3,p4`. A consequence is that `p2,p3,p4` are now in one region, while `p1` is in a separate region. This is undesirable as the `p3` object cannot be freed earlier, as it is in the same region as `p2` and `p4`, even though `p3` is already dead after this code fragment. This example shows that phantom regions can add *extra* region constraints that cause loss in lifetime precision.

As a comparison, the padded regions used by of our region-safe downcast solution are different from phantom regions. As shown in the examples of Appendix A.4, we selectively attach padded regions to superclasses only when relevant downcast operations may occur subsequently.

APPENDIX B

BETTER GENERICITY

B.1 Dynamic Semantics of Variant Parametric Core-Java

The operational semantics of Variant Parametric Core-Java is described in small steps. Notations used are defined as follows.

Locations :	ι	$\in$	*Location*
Primitives :	k	$\in$	*prim* $=$ `int` $\uplus$ `bool` $\uplus$ `float` $\uplus$ `null` $\uplus$ `void`
Values :	δ, ν	$\in$	*Value* $= ($*TyPrim* $\times$ *prim*$) \uplus$ *Location*
Subs :	μ, ρ	$\in$	*Subs* $=$ *TVar* $\rightharpoonup_{\text{fin}}$ *Type*
Store :	ϖ	$\in$	*Store* $=$ *Location* $\rightharpoonup_{\text{fin}}$ *ObjVal*
Variable Env :	Π	$\in$	*VEnv* $=$ *Var* $\rightharpoonup_{\text{fin}}$ *Value*
Object values :	η	$\in$	*ObjVal* $=$ *Type* $\times$ $($*Fd* $\rightharpoonup_{\text{fin}}$ *Value*$)$
Type :	t	$\in$	*Type*

TyPrim consists of primitive types. A type t maintained at run-time does not contain any variant information. If need be, it will be treated as one with invariant annotation $\odot$. A runtime environment Π is a finite map from program variables to their associated values. A value can be a location referencing an object or a pair containing a primitive value and a primitive type.

A runtime store ϖ is a finite map from locations to object values. An object value is comprised of its type and its field values. We write $\eta.f$ to denote the value of the field f of an object η. When the object is referred by its location ι, we also write $\iota.f$ to refer to the value of its field f.

We overload the function *type* to accept (1) primitive value and return the primitive type; (2) location and return the type of the dereferenced object; (3) object and return the object type; and (4) object field and return the field type.

The variable environment Π is such a stackable mapping. We write $\Pi[\nu/v]$ to denote an update of the value of the latest variable v in Π to ν. We write $\Pi + \{v \mapsto \nu\}$ to denote an extension of Π to include a binding of ν to v, while $\Pi - \{v^*\}$ removes a subset of the mappings.

$$\lfloor\textbf{D-Const}\rceil\quad\frac{k\text{ has type }t}{\langle\Pi,\varpi\rangle[k]\hookrightarrow\langle\Pi,\varpi\rangle[(t,k)]}\qquad\lfloor\textbf{D-Var-FD}\rceil\quad\frac{w=v|v.f\quad \nu=\mathit{read}(\Pi,\varpi,w)}{\langle\Pi,\varpi\rangle[w]\hookrightarrow\langle\Pi,\varpi\rangle[\nu]}$$

$$\lfloor\textbf{D-Assign-1}\rceil\quad\frac{\langle\Pi,\varpi\rangle[e]\hookrightarrow\langle\Pi',\varpi'\rangle[e']}{\langle\Pi,\varpi\rangle[w=e]\hookrightarrow\langle\Pi',\varpi'\rangle[w=e']}\qquad\lfloor\textbf{D-Assign-2}\rceil\quad\frac{(\Pi',\varpi')=\mathit{upd}(\Pi,\varpi,w,\nu)}{\langle\Pi,\varpi\rangle[w=\nu]\hookrightarrow\langle\Pi',\varpi'\rangle[(\texttt{void},())]}$$

$$\lfloor\textbf{D-If-false}\rceil\quad\frac{\Pi(v)=(\texttt{Bool},\texttt{false})}{\langle\Pi,\varpi\rangle[\texttt{if }v\texttt{ then }e_1\texttt{ else }e_2]\hookrightarrow\langle\Pi,\varpi\rangle[e_2]}\qquad\lfloor\textbf{D-If-true}\rceil\quad\frac{\Pi(v)=(\texttt{Bool},\texttt{true})}{\langle\Pi,\varpi\rangle[\texttt{if }v\texttt{ then }e_1\texttt{ else }e_2]\hookrightarrow\langle\Pi,\varpi\rangle[e_1]}$$

$$\lfloor\textbf{D-Blk-1}\rceil\quad\frac{\langle\Pi,\varpi\rangle[e_1]\hookrightarrow\langle\Pi',\varpi'\rangle[e_1']}{\langle\Pi,\varpi\rangle[\{t\ v{=}e_1;\ e_2\}]\hookrightarrow\langle\Pi',\varpi'\rangle[\{t\ v{=}e_1';\ e_2\}]}$$

$$\lfloor\textbf{D-Blk-2}\rceil\quad\frac{\mathit{subType}(\mathit{type}(\nu),t)\qquad\Pi'=\Pi+\{v\mapsto\nu\}}{\langle\Pi,\varpi\rangle[\{t\ v=\nu;\ e_2\}]\hookrightarrow\langle\Pi',\varpi\rangle[\texttt{ret}_\texttt{d}(v,e_2)]}$$

$$\lfloor\textbf{D-While-true}\rceil\quad\frac{\Pi(v)=(\texttt{Bool},\texttt{true})}{\langle\Pi,\varpi\rangle[\texttt{while }v\texttt{ do }e]\hookrightarrow\langle\Pi,\varpi\rangle[e\ ;\texttt{while }v\texttt{ do }e]}$$

$$\lfloor\textbf{D-While-false}\rceil\quad\frac{\Pi(v)=(\texttt{Bool},\texttt{false})}{\langle\Pi,\varpi\rangle[\texttt{while }v\texttt{ do }e]\hookrightarrow\langle\Pi,\varpi\rangle[(\texttt{void},())]}$$

$$\lfloor\textbf{D-Ret-d-1}\rceil\quad\frac{\langle\Pi,\varpi\rangle[e]\hookrightarrow\langle\Pi',\varpi'\rangle[e']}{\langle\Pi,\varpi\rangle[\texttt{ret}_\texttt{d}(v^*,e)]\hookrightarrow\langle\Pi',\varpi'\rangle[\texttt{ret}_\texttt{d}(v^*,e')]}\qquad\lfloor\textbf{D-Ret-d-2}\rceil\quad\frac{\Pi'=\Pi-(v^*)}{\langle\Pi,\varpi\rangle[\texttt{ret}_\texttt{d}(v^*,\nu)]\hookrightarrow\langle\Pi',\varpi\rangle[\nu]}$$

$$\lfloor\textbf{D-Ret-m-1}\rceil\quad\frac{\langle\Pi,\varpi\rangle[e]\hookrightarrow\langle\Pi',\varpi'\rangle[e']}{\langle\Pi,\varpi\rangle[\texttt{ret}_\texttt{m}(Q,v^*,t,e)]\hookrightarrow\langle\Pi',\varpi'\rangle[\texttt{ret}_\texttt{m}(Q,v^*,t,e')]}\qquad\lfloor\textbf{D-Ret-m-2}\rceil\quad\frac{\mathit{subType}(\mathit{type}(\nu),t)\ \ \Pi'=\Pi-(v^*)}{\langle\Pi,\varpi\rangle[\texttt{ret}_\texttt{m}(Q,v^*,t,\nu)]\hookrightarrow\langle\Pi',\varpi\rangle[\nu]}$$

$$\lfloor\textbf{D-Seq-1}\rceil\quad\frac{\langle\Pi,\varpi\rangle[e_1]\hookrightarrow\langle\Pi',\varpi'\rangle[e_1']}{\langle\Pi,\varpi\rangle[e_1;e_2]\hookrightarrow\langle\Pi',\varpi'\rangle[e_1';e_2]}\qquad\lfloor\textbf{D-Seq-2}\rceil\quad\frac{}{\langle\Pi,\varpi\rangle[\delta;e_2]\hookrightarrow\langle\Pi,\varpi\rangle[e_2]}\qquad\lfloor\textbf{D-Cast}\rceil\quad\frac{\begin{array}{c}\langle\Pi,\varpi\rangle[v]\hookrightarrow\langle\Pi,\varpi\rangle[\nu]\\ \mathit{chkCast}(\mathit{type}(\nu),t)\end{array}}{\langle\Pi,\varpi\rangle[(t)\ v]\hookrightarrow\langle\Pi,\varpi\rangle[\nu]}$$

$$\lfloor\textbf{D-Capture}\rceil\quad\frac{\begin{array}{c}\langle\Pi,\varpi\rangle[v]\hookrightarrow\langle\Pi,\varpi\rangle[\nu]\ \ t_0=\mathit{type}(\nu)\\ \rho{=}\mathit{match}(t,t_0)\ \ (\Pi',\varpi'){=}\mathit{upd}(\Pi,\varpi,v_1,\nu)\end{array}}{\langle\Pi,\varpi\rangle[\{v_1=(t)\ v;e\}]\hookrightarrow\langle\Pi',\varpi'\rangle[\rho(e)]}$$

$$\lfloor\textbf{D-New}\rceil\quad\frac{\begin{array}{c}\texttt{class }c\langle X_i\rangle_{i=1}^{q}..\texttt{where }\psi\ \{..\}{\in}P\ \ \iota{=}\mathit{fresh}()\\ \mu{=}[t_i/X_i]_{i=1}^{q}\ \ \nu_i=\mathit{read}(\Pi,\varpi,v_i)\ \forall i\in\{1..p\}\\ \mathit{chk}(\mu(\psi))\quad t_i'=\mathit{type}(\nu_i)\quad\forall i\in\{1..p\}\\ \mathit{subType}(c\langle t_i'\rangle_{i=1}^{q},c\langle t_i\rangle_{i=1}^{q})\\ \eta{=}(c\langle t_i\rangle_{i=1}^{q},\{f_i{\mapsto}\nu_i\}_{i=1}^{p})\ \ \varpi'{=}\varpi{+}\{\iota{\mapsto}\eta\}\end{array}}{\langle\Pi,\varpi\rangle[\texttt{new }c\langle t_i\rangle_{i=1}^{q}(v_{1..p})]\hookrightarrow\langle\Pi,\varpi'\rangle[\iota]}$$

$$\lfloor\textbf{D-Call}\rceil\quad\frac{\begin{array}{c}\nu_i=\Pi(v_i')\quad\forall i\in\{0..q\}\quad c\langle t_i'\rangle_{i=1}^{m}=\mathit{type}(\nu_0)\\ t_0\mid t\ \mathit{mn}((t_i\ v_i)_{i=1..q})\langle V^*\rangle\ \texttt{where }\psi\ eb\ \in\ \mathit{mtds}(c)\\ \mu=[t^*/V^*]\quad\mathit{chk}(\mu(\psi))\quad\Pi'=\Pi+[\nu_0/\texttt{this}][\nu_i/v_i]_{i=1}^{q}\\ \mathit{subType}(\mathit{type}(\nu_i),\mu(t_i))\quad\forall i\in\{0..q\}\\ V'=\{\texttt{this}\}\cup\{v_i\}_{i=1}^{q}\quad e=\texttt{ret}_\texttt{m}(V^*,V',\mu(t),\mu(eb))\end{array}}{\langle\Pi,\varpi\rangle[v_0'.\mathit{mn}(v_1',..,v_q')\langle t^*\rangle]\hookrightarrow\langle\Pi',\varpi\rangle[e]}$$

Figure B.1: Dynamic Semantics for Variant Parametric Core-Java: Part I

Similar notations are used for the update and enhancement of object values and stores. In the case of store, we also provide an abbreviated notation $\varpi[\nu/\iota.f] =_{def}$ `let` $(t,\xi) = \varpi(\iota)$ `in` $\varpi[(t,\xi[\nu/f])/\iota]$. Given an object value, $\eta = (t,\xi)$, we have $\mathit{Flds}(\eta) =_{def} \xi$.

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

$$e ::= \cdots \mid \eta \mid \iota \mid \nu \mid \texttt{ret}_{\texttt{d}}(v^*, e) \mid \texttt{ret}_{\texttt{m}}(Q, v^*, \tau, e)$$

The expression $\texttt{ret}_{\texttt{d}}(v^*, e)$ is used to capture the result of evaluating a local block, and $\texttt{ret}_{\texttt{m}}(Q, v^*, \tau, e)$ captures the result of method invocation. The set of variables v^* occurring in both result structures contain the local names and method parameters when entering local body and method body respectively. They are dropped at the end of the local/method body's evaluation. The type τ captures the type of the result of method invocation, whereas Q captures the set of type variables declared in the method header. Q is an instrument used to facilitate our soundness proof.
The dynamic evaluation rules are of the following form:

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

The rules are formulated using an exception-style semantics with three possible errors, namely

E = **Error-Null** | **Error-Cast** | **Error-Type**.

Whenever one such error is raised, the evaluation aborts. This error occurrence can be stated using $\langle \Pi, \varpi \rangle \, [e] \hookrightarrow$ **E**. The small-step dynamic call-by-name semantics is formalised in Figure B.1, together with some auxiliary functions in Figure B.2.

B.2 Soundness of Variant Type System

Before formulating the soundness, we extend the static semantics of the language to include those intermediate expressions introduced in Appendix B.1. In the process, we require introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with that captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics. In our case, it is denoted by:

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

Judgements in the static semantics will be extended with store typing, as follows:
$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$.

```
$read(\Pi, \varpi, v) = \Pi(v);$
$read(\Pi, \varpi, v.f) =$
    $\iota = \Pi(v);$
    $if\ \varpi(\iota) = null\ throw$ **Error-Null**;
    $\varpi(\iota).f;$

$chk(\phi) =$
    $if\ \neg(\vdash \phi)\ throw$ **Error-Type**;
    $true;$

$chkCast(t_1, t_2) =$
    $if\ \neg(\vdash t_1 <: t_2)\ throw$ **Error-Cast**;
    $true;$

$upd(\Pi, \varpi, v, \nu_s) =$
    $\nu = \Pi(v);$
    $if\ \neg(\vdash type(\nu_s) <: type(\nu))$
            $throw$ **Error-Type**;
    $(\Pi[\nu_s/v], \varpi);$

$upd(\Pi, \varpi, v.f, \nu_s) =$
    $\iota = \Pi(v);$
    $if\ \varpi(\iota) = null\ \ throw$ **Error-Null**;
    $\nu_f = \varpi(\iota).f;$
    $if\ \neg(\vdash type(\nu_s) <: type(\nu_f))\ \ throw$ **Error-Type**;
    $(\Pi, \varpi[\nu_s/\varpi(\iota).f]);$

$subType(t_1, t_2) =$
    $if\ \neg(\vdash t_1 <: t_2)\ \ throw$ **Error-Type**;
    $true;$

$match(t_v, t) = [t/t_v];$
$match(c\langle t_v^* \rangle, c\langle t^* \rangle) = [t^*/t_v^*];$
$match(t', t) = throw$ **Error-Type**;
```

Figure B.2: Dynamic Semantics for Variant Parametric Core-Java: Part II

The static semantics for these intermediate expressions is shown in Figure B.3.

The soundness of our static semantics relies on the following consistency relationship between the static and dynamic semantics, defined as follows:

$$\frac{\begin{array}{c} dom(\Pi) = dom(\Gamma) \quad dom(\varpi) = dom(\Sigma) \quad V_L = vars(\psi) - Q \\ \forall v \in dom(\Pi) \cdot \forall \rho_1 \in Subs \cdot \exists \rho_L \in Subs \cdot (dom(\rho_L) = V_L \wedge \rho = \rho_1 \circ \rho_L \wedge (\rho(\psi) \Rightarrow \\ (\Pi(v) \in prim \Rightarrow type(\Pi(v)) <: \rho(\Gamma(v))) \wedge (\Pi(v) \in Location \Rightarrow type(\varpi(\Pi(v))) <: \rho(\Gamma(v))))) \end{array}}{\Gamma; \Sigma; Q; \psi \models \Pi, \varpi}$$

In the above relation, ρ_L is a ground substitution of local type variables occurring in the constraint ψ, and the composition of substitutions is recursively defined as: $(\rho_1 \circ \rho_2)(v) = if\ (v \in dom)$ $then\ \rho_2(v)\ else\ \rho_1(v)$.

The following theorem states the progress of well-typed expressions:

THEOREM 5.1 (PROGRESS) *Let Γ be an environment mapping program variables to ground types. If $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ and $\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$, then either*

- *e is a value, or*
- *$\langle \Pi, \varpi \rangle [e] \hookrightarrow$ **Error-Null** | **Error-Cast**, or*

$$\underline{[\mathbf{ELF_m}]}$$
$$\frac{\begin{array}{c} v^* \subseteq \mathit{dom}(\Gamma) \quad Q' \subseteq Q \\ \Gamma;\Sigma;Q \vdash e :: \tau,\psi \quad \vdash \tau{<:}\tau_1{\Rightarrow}\psi_1 \end{array}}{\Gamma;\Sigma;Q \vdash \mathtt{ret_m}(Q',v^*,\tau,e) :: \tau_1, \psi \wedge \psi_1}$$

$$\underline{[\mathbf{ELF_d}]} \qquad \frac{\Gamma;\Sigma;Q \vdash e :: \tau,\psi}{\Gamma;\Sigma;Q \vdash \mathtt{ret_d}(v^*,e) :: \tau,\psi} \qquad\qquad \underline{[\mathbf{LOC}]} \qquad \frac{\tau = \Sigma(\iota) \quad \vdash \tau{<:}\tau_1{\Rightarrow}\psi}{\Gamma;\Sigma;Q \vdash \iota :: \tau_1,\psi}$$

$$\underline{[\mathbf{OBJ}]} \qquad \frac{(t,\xi) = \eta \quad \vdash \odot\, t{<:}\tau{\Rightarrow}\psi}{\Gamma;\Sigma;Q \vdash \eta :: \tau,\psi} \qquad\qquad \underline{[\mathbf{VALUE}]} \qquad \frac{\vdash \odot\, t{<:}\tau{\Rightarrow}\psi}{\Gamma;\Sigma;Q \vdash (t,\delta) :: \tau,\psi}$$

Figure B.3: Type Rules for Intermediates

- *there exist* Π', ϖ', e' *such that* $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.

A proof of Theorem 5.1 can be found in Appendix B.3.1.

The next theorem states that each well-typed expression preserves its type under reduction with a runtime environment and a store which are consistent with the compile-time counterparts:

THEOREM 5.2 (PRESERVATION) *Let* Γ *be an environment mapping program variables to ground types. If*

$$\Gamma;\Sigma;Q \vdash e :: \tau,\psi$$

$$\Gamma;\Sigma;Q;\psi \models \Pi,\varpi$$

$$\langle \Pi,\varpi \rangle [e] \hookrightarrow \langle \hat{\Pi},\hat{\varpi} \rangle [\hat{e}]$$

then there exists $\hat{\Gamma}$, $\hat{\Sigma}$ *and* $\hat{Q}$ *such that*

$$\Gamma - \mathit{diff}(e,\hat{e}) = \hat{\Gamma} - \mathit{diff}(\hat{e},e)$$

$$\hat{\Sigma} \supseteq \Sigma$$

$$\hat{\Gamma};\hat{\Sigma};\hat{Q} \vdash \hat{e} :: \tau,\hat{\psi}$$

$$\hat{\Gamma};\hat{\Sigma};\hat{Q};\hat{\psi} \wedge \psi \models \hat{\Pi},\hat{\varpi}.$$

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

$$\begin{array}{lllll}
\mathit{diff}(e,e') & =_{def} & \mathit{let} & \mathit{lst} & = \quad \mathit{local}(e) \\
 & & & \mathit{lst1} & = \quad \mathit{local}(e') \\
 & & & n & = \quad \mathit{length}(\mathit{lst}) - \mathit{length}(\mathit{lst1}) \\
 & & \mathit{in} & \multicolumn{2}{l}{(\mathit{take}(n,\mathit{lst}) \lhd n \geq 0 \rhd [\,])} \\
\mathit{take}(n,\mathit{lst}) & =_{def} & \multicolumn{3}{l}{([\,] \lhd n \leq 0 \rhd [\mathit{head}(\mathit{lst})] {+\!\!+} \mathit{take}(n-1,\mathit{tail}(\mathit{lst})))} \\
x \lhd b \rhd y & =_{def} & \multicolumn{3}{l}{\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y}
\end{array}$$

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

$$\begin{array}{lll} \textit{local}(e) =_{def} \textit{case } e \textit{ of} & & \\ \quad \texttt{ret}_{\texttt{m}}(Q, v^*, \tau, e) & \rightarrow & \textit{local}(e) +\!\!+ [\{v^*\}] \\ \quad \texttt{ret}_{\texttt{d}}(v^*, e) & \rightarrow & \textit{local}(e) +\!\!+ [\{v^*\}] \\ \quad w = e & \rightarrow & \textit{local}(e) \\ \quad (t\ v = e_1; e_2) & \rightarrow & \textit{local}(e_1) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

Note that $\Gamma - [\,] =_{def} \Gamma$, $\Gamma - ([s] +\!\!+ S) =_{def} (\Gamma - s) - S$. A proof of Theorem 5.2 can be found in Appendix B.3.2.

B.3 Proofs of Theorems

B.3.1 Proof of Theorem 5.1 (Progress)

By induction over the depth of type derivation for expression e.

Cases $[\mathbf{NULL}, \mathbf{VOID}, \mathbf{VALUE}, \mathbf{LOC}, \mathbf{OBJ}]$. Trivial.

Case $[\mathbf{VAR{-}FIELD}]$. We deal with expression w. As $w = v \mid v.f$ is well-typed, the evalution rule $[\mathbf{D{-}Var{-}FD}]$ is followed, the evaluation either reports an **Error-Null** or advances one step yielding a value.

Case $[\mathbf{ASSIGN}]$. We deal with expression $w = e$. From type rule, we have $\Gamma; \Sigma; Q \vdash e :: \oplus\, t, \psi$. By induction hypothesis, either (i) e is a value ν, or (ii) $\langle \Pi, \varpi \rangle\, [e] \hookrightarrow \textit{Error}$, or (iii) $\langle \Pi, \varpi \rangle\, [e] \hookrightarrow \langle \Pi', \varpi' \rangle\, [e']$.

Subcase (i): Let the runtime type of ν be $\hat{t}$, and that of w be t_1. Then, we have $\odot\, \hat{t} <: \oplus\ t$ and $\odot\, t_1 <: \ominus\ t$, which implies $\hat{t} <: t <: t_1$. Hence, the *upd* function at $[\mathbf{D{-}Assign{-}2}]$ will not throw **Error-Type** exception, and proceed to update the runtime environment Π or the runtime store, as described in $[\mathbf{D{-}Assign{-}2}]$.

Subcase (ii): This will result in the execution of $\langle \Pi, \varpi \rangle\, [w = e]$ aborted with *Error*.

Subcase (iii): This will result in the execution of the assignment to reach $\langle \Pi, \varpi \rangle\, [w = e']$, via $[\mathbf{D{-}Assign{-}1}]$.

Case $[\mathbf{SEQ}]$. We have $\Gamma; \Sigma; Q \vdash e_1 :: \circledast\, t, \psi$. By induction hypothesis, either (i) e_1 is a value ν, or (ii) $\langle \Pi, \varpi \rangle\, [e_1] \hookrightarrow \textit{Error}$, or (iii) $\langle \Pi, \varpi \rangle\, [e_1] \hookrightarrow \langle \Pi', \varpi' \rangle\, [e_1']$.

Subcase (i): The execution proceeds to reach $\langle\Pi,\varpi\rangle\,[e_2]$ unconditionally, according to $[\mathbf{D{-}Seq{-}2}]$.

Subcase (ii): The execution will be aborted with *Error* exception.

Subcase (iii): The execution proceeds to reach $\langle\Pi',\varpi'\rangle\,[e_1';e_2]$, according to $[\mathbf{D{-}Seq{-}1}]$.

Case $[\textbf{LOCAL}]$. Given that $\Gamma;\Sigma;Q\vdash\{t\ v\ =\ e_1\ ;\ e_2\}::\tau,\psi_1\wedge\psi_2$. We have $\Gamma;\Sigma;Q\vdash e_1::\oplus\,t,\psi_1$. By induction hypothesis, either (i) e_1 is a value ν, or (ii) $\langle\Pi,\varpi\rangle\,[e_1]\hookrightarrow$ *Error*, or (iii) $\langle\Pi,\varpi\rangle\,[e_1]\hookrightarrow\langle\Pi',\varpi'\rangle\,[e_1']$.

Subcase (i): Let the runtime type of ν be $\hat{t}_0$ and the runtime type of v be $\hat{t}$. As the consistency relation holds between the static and the dynamic semantics, we have for all ground substitution ρ, $\vdash\rho(\psi_1)\Rightarrow\hat{t}=\rho(t)$. Since $\vdash\rho(\psi_1)\Rightarrow\odot\hat{t}_0{<:}\oplus\ t$, $\mathit{subType}(\mathit{type}(\nu),\hat{t})$ = $\mathit{subType}(\hat{t}_0,\hat{t})$ = *true*. Hence, the execution will proceed to the state $\langle\Pi',\varpi\rangle\,[\mathtt{ret_d}(v,e_2)]$ according to $[\mathbf{D{-}Blk{-}2}]$.

Subcase (ii). The execution will throw the corresponding *Error* exception.

Subcase (iii). The execution will proceed to $\langle\Pi',\varpi'\rangle\,[\{t\ v\ =\ e_1';e_2\}]$ according to $[\mathbf{D{-}Blk{-}1}]$.

Case $[\textbf{NEW}]$. Given $\Gamma;\Sigma;Q\vdash\mathtt{new}\ c\langle t_i\rangle_{i=1}^{q}(v_1,..,v_p)::\tau,\psi$, let $\hat{t}_i$ (for all $i=1..q$) and $\hat{t}_{v_i}$ (for all $i=1..p$) be the runtime types of type arguments and value arguments to `new`. Then we have, for all ground substitution ρ, $\vdash\rho(\psi)\Rightarrow\wedge_{i=1}^{q}(\hat{t_i}{=}\rho(t_i))$ and $\vdash\rho(\psi)\Rightarrow\wedge_{i=1}^{p}(\hat{t}_{v_i}{<:}\rho(\Gamma(v_i)))$. Furthermore, $\vdash\rho(\psi)\Rightarrow\rho(\Gamma(v_i)){<:}t_i'$, for all i. Hence, both calls to *chk* and *subType* at runtime do not fail, and the execution proceeds to the state $\langle\Pi,\varpi'\rangle\,[\iota]$, where ι is the location referencing the new object.

Case $[\textbf{COND}]$. Given $\Gamma;\Sigma;Q\vdash\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2::\tau,\psi$ and $\Gamma(v){<:}\oplus\ \mathtt{Bool}$, the runtime value of v will either be *true*, *false*, or null $()$. In the first two subcases, the execution proceeds to next state according to the rules $[\mathbf{D{-}If{-}true}]$ and $[\mathbf{D{-}If{-}false}]$ respectively. In the last subcase, there is no corresponding dynamic rule, and exception **Error-Null** will be thrown.

Case $[\textbf{WHILE}]$. Given $\Gamma;\Sigma;Q\vdash\mathtt{while}\ v\ \mathtt{do}\ e\ ::\tau,\psi$ and $\Gamma(v){<:}\oplus\ \mathtt{Bool}$, the runtime value of v will either be *true*, *false*, or null $()$. In the first two subcases, the execution proceeds to next state according to the rules $[\mathbf{D{-}While{-}true}]$ and $[\mathbf{D{-}While{-}false}]$ respectively. In the last subcase, there is no corresponding dynamic rule, and exception **Error-Null** will be thrown.

Case $[\mathbf{ELF_d}, \mathbf{ELF_m}]$. We have $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ as the premise of the static semantics. By induction hypothesis, either (i) e is a value ν, or (ii) $\langle \Pi, \varpi \rangle [e]$ produces *Error*, or (iii) $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.

Subcase (i): Let the runtime type of ν be $\hat{t}_\nu$ and that of return value be $\hat{t}$ then for all ground substitution ρ we have $\vdash \rho(\psi) \Rightarrow \rho(\tau) = \odot\hat{t}$. Also, we have $\vdash \rho(\psi) \Rightarrow \hat{t}_\nu {<:} \rho(\tau)$. Hence, the call to *subType* in the rule $[\mathbf{D{-}Ret{-}2}]$ returns *true*, and the execution proceeds to $\langle \Pi', \varpi \rangle [\nu]$ accordingly.

Subcase (ii): The execution will throw the corresponding *Error* exception, as no rule applies.

Subcase (iii): The execution step to the new state following rule $[\mathbf{D{-}Ret{-}1}]$.

Case $[\textsc{cast}]$. Any type mismatch during cast will be captured by *chkCast* and **Error-Cast** exception will be thrown. Otherwise, casting will succeeds and the execution proceeds to the next state $\langle \Pi, \varpi \rangle [(t, \iota)]$.

Case $[\textsc{capture}]$. We have $\Gamma; \Sigma; Q \vdash \{v_1 = (t)v; e\} :: \tau, \psi_1 \wedge \psi_2$. From its premise, we have $t = c\langle \odot\, V_i \rangle_{i=1}^{n}$. Executing the expression v either yields an *Error* exception or returns a value ν. We consider the case where ν is returned. Let t_0 be the type of ν as declared in the runtime environment. The use of flow symbol $\odot$ in t implies that $match(t, t_0)$ succeeds and produces ρ only when $\rho(t) = t_0$. Hence, by rule $[\mathbf{D{-}Capture}]$, the execution proceeds to the state $\langle \Pi', \varpi' \rangle [\rho e]$. Updating of v_1 does not fail, similar with $[\textsc{assign}]$.

Case $[\textsc{call}]$. Given $\Gamma; \Sigma; Q \vdash v_0'.mn(v_1', .., v_q')\langle t^* \rangle : \tau, \psi$. Let the runtime type arguments be $\langle \hat{t}^* \rangle$ and the value arguments have type $\hat{t}_{v_i'}$ for $i = 0..q$. Also, the ground substitution μ in $[\mathbf{D{-}Call}]$ is an instance of ρ in $[\textsc{call}]$, which makes ψ true. Thus, we have, $\vdash \mu(\psi) \Rightarrow \hat{t}_{v_i'} {<:} \mu(\tau_i')$, $i = 0..q$, and $\vdash \hat{t}_0 {<:} \mu(t_0)$. Hence, the call to *subType* in $[\mathbf{D{-}Call}]$ yields true, and the execution proceeds to the state $\langle \Pi, \varpi \rangle [e]$ according to $[\mathbf{D{-}Call}]$. □

B.3.2 Proof of Theorem 5.2 (Preservation)

The proof for Theorem 5.2 requires several lemmas.

Lemma B.1 (Type Substitution). *If $\Gamma;\Sigma;Q \vdash e :: \tau,\psi$, then for all substitution ρ such that $\vdash \rho(\psi)$, we have $\rho(\Gamma);\rho(\Sigma);Q \vdash \rho(e) :: \rho(\tau),\rho(\psi)$.*

The proof is by induction on a derivation of $\Gamma;\Sigma;Q \vdash e :: \tau,\psi$.

The next lemma, called *assumption weakening lemma*, states that the static judgment remains valid despite a variation of its assumption. This assumes the store type Σ to have unbounded mapping of locations to types. However, the type environment Γ takes the form of stackable mapping between variables and types, and is allowed to grow (by pushing in new mappings) and shrink (by popping out mappings from stack). The lemma states that such change to type environment preserves the type judgment, if the change are properly constrained.

Lemma B.2 (Assumption Weakening). *Given that the following judgment holds:*

$$\Gamma;\Sigma;Q \vdash e :: \tau,\psi$$

Let $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ be such that:

$$\begin{gathered} \mathit{vars}(e) \subseteq \mathit{dom}(\Gamma) \cap \mathit{dom}(\hat{\Gamma}) \\ Q \subseteq \hat{Q} \vee \hat{Q} \subseteq Q \\ \mathit{vars}(\psi) - Q = \mathit{vars}(\psi) - \hat{Q} \\ \exists v^* \cdot (\Gamma - \{v^*\} = \hat{\Gamma}) \vee (\hat{\Gamma} - \{v^*\} = \Gamma) \\ \hat{\Sigma} \supseteq \Sigma \end{gathered}$$

Then, there exists $\hat{\psi}$ such that $\vdash \hat{\psi} \Leftrightarrow \psi$ and

$$\hat{\Gamma};\hat{\Sigma};\hat{Q} \vdash e :: \tau,\hat{\psi}$$

The call $\mathit{vars}(e)$ returns all program variables occurring in e, whereas $\mathit{vars}(\psi)$ returns all (type) variables occurring in ψ.

Proof of Lemma B.2: By structural induction on the static semantics of the form $\Gamma;\Sigma;Q \vdash e :: \tau,\psi$. For any $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$, we say that they satisfy the premises of the Lemma if the following holds:

$$\begin{gathered} \mathit{vars}(e) \subseteq \mathit{dom}(\Gamma) \cap \mathit{dom}(\hat{\Gamma}) \\ Q \subseteq \hat{Q} \vee \hat{Q} \subseteq Q \\ \mathit{vars}(\psi) - Q = \mathit{vars}(\psi) - \hat{Q} \\ \exists v^* \cdot (\Gamma - \{v^*\} = \hat{\Gamma}) \vee (\hat{\Gamma} - \{v^*\} = \Gamma) \\ \hat{\Sigma} \supseteq \Sigma \end{gathered}$$

Cases $[\textbf{NULL}, \textbf{VOID}, \textbf{LOC}, \textbf{OBJ}, \textbf{VALUE}]$. Trivial.

Case $[\mathbf{VAR{-}FIELD}]$. We deal with expression w, where $w = v \mid v.f$. For any $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ satisfying the premise of the lemma, we see that $\Gamma(v) = \hat{\Gamma}(v)$. Hence, $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash w : \tau, \psi$.

Case $[\mathbf{ASSIGN}]$. We deal with expression $w = e$. We have $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash e :: \oplus t, \hat{\psi}$ for αt=*GetType*$(\hat{\Gamma}, w)$=*GetType*(Γ, w). The desired result is then derived by induction hypothesis.

Cases $[\mathbf{LOCAL}, \mathbf{SEQ}, \mathbf{COND}, \mathbf{WHILE}, \mathbf{CAST}, \mathbf{CAPTURE}, \mathbf{ELF_d}, \mathbf{ELF_m}]$. By induction hypothesis.

Case $[\mathbf{NEW}]$. The result holds because $\Gamma(v_i) = \hat{\Gamma}(v_i)$, for all $i = 1..p$.

Cases $[\mathbf{CALL}]$. The result holds because $\Gamma(v_i') = \hat{\Gamma}(v_i')$ for all $i = 1..q$. □

Proof of Theorem 5.2: This can be proven by induction over the depth of type derivation of expression e.

Cases $[\mathbf{NULL}, \mathbf{VOID}, \mathbf{LOC}, \mathbf{OBJ}, \mathbf{VALUE}]$. Vacuously true.

Case $[\mathbf{VAR{-}FD}]$. This can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case $[\mathbf{ASSIGN}]$. There are two rules by which one-step derivation can be applied:

Subcase $[\mathbf{D{-}Assign{-}1}]$: By induction hypothesis, there exists Γ', Σ' and Q' such that $\Gamma'; \Sigma'; Q' \vdash e' :: \oplus\, t', \psi'$ and which satisfies the premise of the theorem. Since $\oplus\, t' <: \oplus\, t$, we thus have $\Gamma'; \Sigma'; Q' \vdash e' :: \oplus\, t, \psi' \wedge \psi''$, where $\vdash \oplus\, t' <: \oplus\ t \Rightarrow \psi''$. The desired result can then be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ', Σ' and Q' respectively.

Subcase $[\mathbf{D{-}Assign{-}2}]$: Consider the assignment to a variable v. Given that *upd*(Π, ϖ, w, ν) returns successfully (Π', ϖ'), it must be the case that *type*(ν) $<:$*type*$(\Pi'(v))$. The desired result can then be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively. Similar argument applies to the assignment to a field.

Case $[\mathbf{SEQ}]$. There are two rules by which one-step derivation can be applied:

Subcase $[\mathbf{D{-}Seq{-}1}]$: By induction hypothesis, there exists Γ', Σ' and Q' that establishes the consistency relation at the hypothesis. We elect $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to be Γ', Σ' and Q' respectively to obtain the desired result.

Subcase $[\mathbf{D{-}Seq{-}2}]$: By setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to be Γ, Σ and Q respectively.

Case $[\textbf{COND}]$. There are two rules by which one-step derivation can be applied: $[\textbf{D-If-True}]$, $[\textbf{D-If-False}]$. Both can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case $[\textbf{WHILE}]$. Similar as the argument for case $[\textbf{COND}]$.

Case $[\textbf{LOCAL}]$. There are two rules to consider:

Subcase $[\textbf{D-Blk-1}]$: By induction hypothesis.

Subcase $[\textbf{D-Blk-2}]$: Since *subType*(*type*(ν), t), Γ' and Σ used in $[\textbf{LOCAL}]$ remain consistent with Π' and ϖ in this subcase. We let $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ', Σ and Q respectively.

Case $[\textbf{CAST}]$. This can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case $[\textbf{CAPTURE}]$. The argument for one-step derivation $[\textbf{D-Capture}]$ is similar to that for case $[\textbf{D-Assign-2}]$, except for the assignment of runtime type information of ν to the type variables occurring in t. This assignment proceeds successfully because of the premise of $[\textbf{CAPTURE}]$. We let $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively. It suffices to show that $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash \rho(e) :: \tau, \hat{\psi}$.This is true by applying Type Substitution Lemma to the following premise of

$[\textbf{CAPTURE}]$: $\Gamma; \Sigma; Q \vdash e :: \tau, \psi_2$.

Case $[\textbf{NEW}]$. We let $\hat{\Gamma} = \Gamma$, $\hat{\Sigma} = \Sigma + \{\iota \mapsto \odot\, c\langle \odot\, t_i \rangle_{i=1}^{q}\}$ and $\hat{Q} = Q$.

Case $[\textbf{CALL}]$. The fact that $\hat{\tau}$, as obtained from $[\textbf{ELF}_\textbf{m}]$, is a subtype of τ obtained from $[\textbf{CALL}]$, is established from the result of $[\textbf{ELF}_\textbf{m}]$ and the constraint $\rho(\oplus\, t) <: \tau$ occurred in ψ in the premise of $[\textbf{CALL}]$. Finally, by assumption weakening rule, we let

$\hat{\Gamma} = \Gamma + \{v_i :: \oplus \hat{t}_i\}_{i=1}^{q} + \{\texttt{this} :: \oplus \hat{t}_0$, $\hat{\Sigma} = \Sigma$, $\hat{Q} = Q \cup \{V^*\}$.

Case $[\textbf{ELF}_\textbf{d}, \textbf{ELF}_\textbf{m}]$. There are two subcases for consideration:

Subcase $[\textbf{D-Ret-d-1}, \textbf{D-Ret-m-1}]$: By induction hypothesis.

Subcase $[\textbf{D-Ret-d-2}, \textbf{D-Ret-m-2}]$: By induction hypothesis and the Assumption Weakening Lemma. $\square$