A Static Memory Safety Annotation System for Safety Critical Java

Daniel Tang, Ales Plsek, Kelvin Nilsen[†], Jan Vitek
S3 lab, Purdue University † *Atego Inc.*

Abstract—**Embedded systems must be able to operate for long periods of time with limited memory. Dynamic memory allocation is often discouraged in such systems as it requires careful analysis to rule out memory-related software defects. This paper presents an annotation system that can be used to rule out memory access errors in programs written in a subset of the Java programming language which targets safety critical applications. The annotations are optional. When present, they are used by the compiler to report, ahead of time, potential memory access errors. The proposed system is part of the upcoming Safety Critical Java Specification. It has been evaluated on a number of small benchmark programs (26 KLOC of Java code) and implemented in the oSCJ real-time Java virtual machine leading to performance improvements ranging from 1.7% to 26%.**

I. Introduction

Memory is a key resource in embedded systems. Programmers must carefully apportion storage space to the different tasks that require it and, when necessary, repurpose memory that is currently not in use. This is traditionally done by a combination of static allocation and manual management. Manual memory management is the source of many software defects and requires careful analysis to prevent memory corruption. The Real-time Specification for Java (RTSJ) [2] is a high-level programming environment that offers a safer alternative with a memory management API based on regions, or *scopes*. Data objects are allocated dynamically in scopes, and entire scopes are reclaimed at once. Scopes enjoy fast allocation and constant time de-allocation. Their main drawback is that in order to prevent dangling references, i.e. references inside a scope that has been reclaimed, the RTSJ mandates dynamic checks on reference operations. These checks come at some runtime overhead and entail the possibility of an exception being thrown at runtime if the program attempts an illegal store.

The goal of this work is to prevent memory access errors at compile-time. We propose to do this with a pluggable ownership type system [5], an optional system of annotations that restrict the set of valid programs accepted by the compiler to a subset guaranteed to be free of memory access errors. As valid programs do not throw memory access exceptions, the compiler needs not emit code to check the correctness of memory operations, thus potentially speeding up execution. Our work is done in the context of the upcoming Safety Critical Java (SCJ) specification[1] which is designed to facilitate the certification of real-time Java applications under safety-critical standards. In a safety-critical setting, run-time exceptions must be shown to never occur or to be handled correctly, thus they have a significant cost in terms of verification effort. Ruling out exceptions on all reference stores will thus greatly cut down verification costs. The question whether a program will run out of memory is a separate and orthogonal issue addressed by memory usage analysis research [4].

The memory management facilities of SCJ are a special case of region-based memory management which has been used, most notably, in implementations of functional languages [13] and for memory-safe extensions of C [7]. Type systems for safe region-based allocation were formulated for ML [13], Cyclone [7] and Java [3], [17]. While our system builds on these predecessors, we cannot reuse any of them directly as they all require either (a) changes to the source language to introduce type structures describing regions, (b) changes to the semantics of certain operations such as allocation or method invocation, or (c) compiler support. The SCJ specification is designed under one hard constraint, namely no changes to the Java syntax are allowed (this is imperative to preserve the Java tool chain). We may thus not deviate from the Java source syntax nor from the Java bytecode format. Furthermore, the SCJ expert group mandated that the memory safety annotations be optional to allow users to opt in or out depending to the criticality of their project. This entails that the presence (or absence) of safety annotations should not modify the meaning of the program or its execution characteristics. We also add two design goals of our own: *conceptual simplicity* and *minimal effort*. Many of the previous proposals were rather complex, to encourage adoption we are looking for a system that is as simple as possible while remaining expressive. A related goal is that we want to minimize the programmer effort involved in adding memory safety annotations to a program.

Satisfying the above mentioned constraints and our self-imposed goals has proven to be more challenging than we expected at first. The system presented in this paper is the result of three years of work with many changes from our original design to accommodate programming idioms that are specific to SCJ. The contributions of this paper are:

- *Design of a memory safety annotation system.* The type system targets Safety Critical Java programs and leverage Java metadata annotations to express constraints on memory references without changing the Java syntax or semantics. The system is simple as it consists of only three distinct annotations (@Scope, @RunsIn, @DefineScope) on variable, method, field

[1]http://www.jcp.org/en/jsr/detail?id=302

and class definitions.

- *Implementation of static checker.* The checker uses the JSR-308 Checker Framework and is integrated in the Java compiler to validate annotated source programs. We have also modified a real-time Java virtual machine to optimize checks for annotated programs.
- *Formalism and Proof of soundness.* We have formalized key features of SCJ in a object calculus with a type system that mirrors our annotations. A proof of soundness of the type system demonstrates that well-typed programs will not experience memory access errors.
- *Evaluation.* We use a suite of seven programs (26 KLOC) written against the SCJ API to evaluate the software engineering impact of our annotations quantitatively in terms of added lines of code, and qualitatively in term of the challenges involved in refactoring existing code to abide by the restrictions imposed by our system. Lastly we evaluate the performance impact of removing dynamic memory safety checks on a LEON3 processor and a desktop x86 machine for a subset of these programs (we observe performance improvements ranging between 0.4% and 39%).

A workshop version of this paper appeared at [14]. Our implementation and benchmarks are at:

http://sss.cs.purdue.edu/projects/rtss11

II. Safety-Critical Java

The upcoming Safety Critical Java (SCJ) specification presents embedded system developers with a programming model that simplifies and streamlines that of the Real-time Specification for Java.[2] An SCJ application consists of one or more (possibly nested) *missions* which themselves are made up of a number of tasks, called *schedulable objects*. For our purposes we can restrict our focus on the memory management API of SCJ. The memory available to an application is logically segmented in a number of scoped memory areas or *scopes*. At application start up, a single shared scope, referred to as immortal memory, is available for shared persistent data. Each mission has a shared mission scope which is available to all tasks that are part of that mission or nested missions. Each schedulable object (task) has its own private scope for allocation of short-lived data, furthermore it can create nested, private, sub-scopes. Allocation is performed by new expressions which create and initialize objects in the current scope. Deallocation occurs when no schedulable is active in a given scope. At that point objects in the scope are reclaimed in a single step.

A schedulable object can change its allocation context by *entering* a scope. This gives rise to a *parenting* relation between scopes, we say that if a scope *S* has been entered by a schedulable previously in scope *S'*, then *S'* is the parent of *S*. Each schedulable has an associated stack of active scopes which is managed in a strict LIFO fashion. The parenting relation induces a restriction on object references. It is illegal for an object to hold a reference that points into a child scope. Fig. 1 illustrates the logical memory of an SCJ program consisting of one mission, two schedulables and a number of objects allocated in different scopes. An object can reference objects in the same scope (@2→@1), parent (@3→@1) and ancestor scopes (@4→@2).

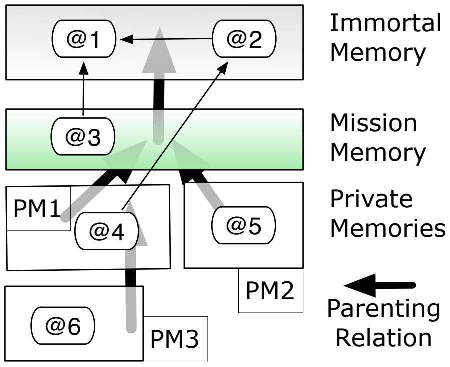

Figure 1: Memory Model.

Immortal memory is represented by a singleton instance of the ImmortalMemory class. Instances of the MissionMemory and PrivateMemory classes represent scopes of missions and tasks. ImmortalMemory is used for allocation of classes and static variables. A task's allocation context can be either one of MissionMemory or PrivateMemory. MissionMemory is the allocation context of the Mission.initialize() method. PrivateMemory is the allocation context of the handleAsyncEvent() method of the PeriodicEventHandler class. By extension, we also refer to the scope in which an object was allocated as the allocation context of this object. When handleAsyncEvent() is called, its scope stack contains immortal memory, the mission memory(-ies) and a single private memory. The private memory is cleared when the method returns. Unlike scopes in the RTSJ, private memories are guaranteed to be accessed only by a single task.

The method enterPrivateMemory() is used to create and enter a nested private scope. It takes as argument an instance of Runnable whose run() method will be called. Once that method returns, the nested scope and its contents are reclaimed. The method executeInArea(r) can be used to change temporarily the allocation context to another scope[3] and execute the run() method of the argument object r in that context. The target of the call can be an instance of ImmortalMemory, MissionMemory or PrivateMemory. However, after invoking executeInArea(), it is forbidden by a runtime check to use enterPrivateMemory().

[2]This chapter is based on SCJ v0.71 from August 2010. For an introduction to the RTSJ, we refer readers to [2].

[3]The target need not be a parent scope; a recursive sequence of call can jump around the scope tree.

A reference assignment x.f = y is valid if ax is the scope of x (obtained by MemoryArea.getMemoryArea(x) and ay is the scope of y, and either ax == ay or ay is a parent of ax. If a program attempts to perform an assignment that is not valid, an IllegalAssignmentError will be thrown. Assignments to local variables and parameters, as well as assignments to primitive types, are always valid.

III. Memory Safety Annotations

We have designed a new system of memory safety annotations which prevent the occurrence of IllegalAssignment-Errors in SCJ programs. These annotations form a static pluggable type system in the sense of Braccha [5] as they can be layered over and above the existing Java type system, and rejects programs with potentially invalid operations without changing the behavior of valid programs. The annotations are purely static and require no additional run-time type information; they can be erased from the program without impact on its execution in the same way as Java Generics [6] (Java Generics do require insertion of run-time checks, whereas we don't). The set of valid annotated programs is a strict subset of all SCJ programs. Some correct programs will be rejected by our checker – this is always the case with a static type system – but in our case, we have deliberately opted for simplicity at the cost of expressive power. For instance, we have chosen not to support a general form of parametric polymorphism for methods and class definitions. Instead, we limit polymorphism to carefully chosen patterns. This simplifies the annotation system at the cost of occasional code duplication. The starting point for this work was our previous RTSS paper [17] and its follow ups [1], [16]. The core idea in that work was to use a simple ownership type system [11] tied to syntactic properties of the program (namely package names) to describe the scope structure and place constraints on the direction of pointers. That work was considered too restrictive to be practical for SCJ as it forced a particular package structure on the code; furthermore, it modified the semantics of programs (annotated programs were rewritten by a pre-compiler) and lacked polymorphism (any class used in multiple scopes had to be duplicated). Finding a suitable set of annotations that do not have these drawbacks and that have the potential of being adopted in practice has proven surprisingly difficult. In this work we leverage Java metadata annotations and introduce the three new annotations shown in Table I to extend SCJ programs without changing the syntax or semantics of SCJ. Metadata annotations have the advantage of being supported by the entire Java tool chain, but are limited in some significant ways. It is, for example, not possible to annotate expressions which leads to somewhat less elegant code.

A static type system expresses properties of the dynamic behavior of programs as invariants that can be checked ahead of time. In our case, we must turn the run-time parenting relationship of scopes into a static property that can be used to reason about the allocation contexts of objects on both side of an assignment. We do this by asking developers to provide symbolic names for all of the scopes that will be created at run-time and to define the parenting relation between these named scopes. Object references are also annotated with scope names so that the checker can verify that assignments never create dangling references.

A. Defining the Scope Tree

The first step is to define a *static scope tree*. The static scope tree is the compile-time representation of the run-time parenting relation. For this, we introduce the @DefineScope annotation with two arguments, the symbolic name of the new scope and of its parent scope. The checker will ensure that the annotations define a well formed tree rooted at IMMORTAL, the distinguished parent of all scopes. Scopes are introduced by mission (MissionMemory) and schedulable objects (PrivateMemory). Thus we require that each declaration of a class that has an associated scope (for instance, subclasses of the MissionSequencer class which define missions, and subclasses of the PeriodicEventHandler class which hold task logic) must be annotated with a scope definition annotation. Furthermore, nested PrivateMemory scopes are created by invocation of the enterPrivateMemory() method. As Java does not allow annotation on expression, we require that the argument of the method, an instance of a subclass of SCJRunnable be annotated with a scope definition. It is notable that scopes do not have names at run-time; the only property we can rely on to connect the run-time scope structure to the static scope structure is the the distance from the root of the tree and the name of the mission classes.

B. Associating References to Scopes

The key requirement for being able to verify a program is to have a compile-time mapping of every object reference to some node in the static scope tree. With that information,

Annotation	Where	Arguments	Description
@DefineScope	Any	*Name*	Define a new scope.
@Scope	Class	*Name*	Instances are in named scope.
		CALLER	Can be instantiated anywhere.
	Field	*Name*	Object allocated in named scope.
		UNKNOWN	Allocated in unknown scope.
		THIS	Allocated enclosing class' scope.
	Method	*Name*	Returns object in named scoped.
		UNKNOWN	Returns object in unknown scope.
		CALLER	Returns object in caller's scope.
		THIS	Returns object in receiver's scope.
	Variable	*Name*	Object allocated in named scope.
		UNKNOWN	Object in an unknown scope.
		CALLER	Object in caller's scope.
		THIS	Object in receiver's scope.
@RunsIn	Method	*Name*	Method runs in named scope.
		CALLER	Runs in caller's scope.
		THIS	Runs in receiver's scope.

Table I: Annotation Summary. Default values in bold.

verification is simply a matter of checking that the right hand side of any assignment is mapped to a parent (or same) scope of the target object. This mapping is established by the @Scope annotation which takes a scope name as argument. Scope annotations can be attached to class declarations to constrain the scope in which all instances of that class are allocated. Annotating a field, local or argument declaration constrains the object referenced by that field to be in a particular scope. Lastly, annotating a method declaration constrains the value returned by that method.

C. *Limited Polymorphism*

While a general form of parametric polymorphism for scopes such as full-fledged Java generics [6] was felt to be too complex by the SCJ expert group, we introduced a limited form of polymorphism that seems to capture many common use cases. Polymorphism is obtained by adding the following scope variables: CALLER, THIS and UNKNOWN. These can be used in @Scope annotations to increase code reuse. A reference that is annotated CALLER is allocated in the same scope as the "current" or calling allocation context. References annotated THIS point to objects allocated in the same scope as the receiver (i.e. the value of this) of the current method. Lastly, UNKNOWN is used to denote unconstrained references for which no static information is available. Classes may be annotated CALLER to denote that instances of the class may be allocated in any scope.

D. *Allocation Contexts*

To determine the scope in which an allocation expression new C() is executed we need to associate methods with nodes in our static scope tree. The @RunsIn annotation does this. It takes as an argument the symbolic name of the scope in which the method will be executed. An argument of CALLER indicates that the method is scope polymorphic and that it can be invoked from any scope. In this case, the arguments, local variables, and return value are by default assumed to be CALLER. An overriding method must preserve any @RunsIn annotation inherited from the parent. THIS denotes a method which runs in the same scope as the receiver.

E. *Dynamic Guards*

Dynamic guards are our equivalent of dynamic type checks. They are used to recover the static scope information lost when a variable is cast to UNKNOWN, but they are also a way to side step the static annotation checks when these prove too constraining. We have found that having an escape hatch is often crucial in practice. A dynamic guard is a conditional statement that tests the value of one of two predefined methods, allocatedInSame() or allocatedInParent() or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be final to prevent an assignment violating the assumption.

F. *Other Issues*

Java has a number of other features that must be addressed to obtain a complete system. Reflection is luckily not part of SCJ, so we do not have to worry about reflective calls. The only exception is the newInstance() method which allocates an instance of a class passed in as a parameter in the scope of the managed memory represented by the receiver. The checker requires that the argument be a constant class name and the receiver of the invocation be annotated with a scope name specifying the target scope. Arrays are another feature that requires special treatment. By default, the allocation context of an array T[] is the same as that of its element class, T. Primitive arrays are considered to be labeled THIS. The default can be overriden by adding a @Scope annotation to an array variable declaration. The allocation context of static constructors and static fields is IMMORTAL. Thus, static variables follow the same rules as if they were explicitly annotated with IMMORTAL. Static methods are treated as being annotated CALLER. We do not leverage Java generics in the annotation system because Java generics are erased during compilation of the program and no residual information is left in the bytecode. While our current checker is working at source level, one of the requirements for our annotation system is to be enforceable on compiled bytecode class files as well as source code.

G. *Reducing Annotation Overhead*

To reduce the annotation burden for programmers, annotations that have default values can be omitted from the program source. For class declarations, the default value is CALLER. This is also the annotation on Object. This means that when annotations are omitted classes can be allocated in any context (and thus are not tied to a particular scope). Local variables and arguments default to CALLER as well. For fields, we assume by default that they infer to the same scope as the object that holds them, i.e. their default is THIS. Instance methods have a default @RunsIn(THIS) annotation.

H. *Memory Safety Rules*

We will now review the constraints imposed by the checker. Subclasses must preserve annotations. A subclass of a class annotated with a named scope must retain the exact same scope name. A subclass of a class annotated CALLER may override this with a named scope. Method annotations must be retained in subclasses to avoid upcasting an object to the supertype and executing the method in a different scope. The value of polymorphic annotations such as THIS and CALLER can be inferred from the context in certain cases. A concretization function translates THIS or CALLER to a named scope. For instance a variable annotated THIS takes the scope of the enclosing class (which can be CALLER or a named scope). An object returned from a method annotated CALLER is concretized to the value of the calling method's @RunsIn which, if it is THIS, can be concretized to the enclosing class' scope. We say that two scopes are the same if they are identical after concretization. *A method invocation* z=x.m(...,y,...) is valid (1) if its @RunsIn is the same as the current scope or it is annotated @RunsIn(CALLER), (2) if the scope of every argument y is the same as the corresponding argument declaration or if argument is UNKNOWN, (3) if the scope of the return value is the same as z. *An assignment expression* x.f=y is valid if one of the following holds: (1) x.f and y have the same scope and are not UNKNOWN or THIS, (2) x.f has scope THIS and x and y has the same, non-UNKNOWN scope, or (3) x.f is THIS, f is UNKNOWN and the expression is protected by a dynamic guard. *A cast expression* (C) exp may refine the scope of an expression from an object annotated with CALLER, THIS, or UNKNOWN to a named scope. For example, casting a variable declared @Scope(UNKNOWN)Object to C entails that the scope of expression will be that of C. Casts are restricted so that no scope information is lost. *An allocation expression* new C() is valid if the current allocation context is the same as that of the class C. A variable or field declaration, C x, is valid if the current allocation context is the same or a child of the allocation context of C. Consequently, classes with no explicit @Scope annotation cannot reference classes which are bound to named scopes, since THIS may represent a parent scope.

IV. Formal Account of Memory Safety

As with any non-trivial type system it is not immediately obvious that the proposed rules actually guarantee memory safety. In fact, as we experimented with different rules and annotations, we repeatedly found unexpected corner cases that we had not anticipated. In order to gain assurance we propose to formalize the annotation system and prove a soundness theorem that entails memory safety. The approach we choose is to boil down SCJ and Java to a core calculus with only the features that are relevant to showing memory safety, namely, objects, dynamic allocation, scopes, references and subtyping. We arrange the SCJ annotation system as an extension of the Java type system so that any type becomes a pair, consisting of a scope S and a class C. The proof of soundness result is obtained by showing that a well typed program under the extended type system will never lead to an illegal memory access. The formalism is based on the calculus presented in [15] which is an imperative version of Featherweight Java [8]. The resulting language is fairly minimal as shown by Fig. 2 which gives its complete syntax. A program p is composed of a sequence of classes. Each class C has a scope S, a parent class D, some fields $\overline{fd}$ and methods $\overline{md}$. A field declaration consists of a type τ and a field name f. A method declaration is made up of a return type τ, a runs-in scope S, a sequence of arguments $\overline{\tau \mathsf{x}}$ and a body composed of statement s; return y. Statements include composition s; s, assignment to a final local variable τ x = y.f, assignment to a field $\mathsf{x}.f = \mathsf{y}$, upcast $\tau\ \mathsf{x} = (C)\mathsf{y}$, allocation $\tau\ x =$ new C, method invocation $\tau\ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})$, dynamic guard $(\mathsf{x} \overset{@}{=} \mathsf{y})\, \mathsf{x.f} = y$, and return return x. We use a "named form," where expressions do not nest and must be immediately stored in a final variable as this simplifies the syntax. Another simplification is the elimination of implicit upcasts for arguments, return values, and assignments. All casts are performed explicitly by cast statements which simplifies the other rules as they can assume type equality. These simplifications do not reduce the expressive power of the language, any Java program can be written in this form.

The SCJ API has been reduced to its essence. We assume that the Object class has final methods enter () and execInArea (x), which correspond to the SCJ methods enterPrivateMemory(r,l) and executeInArea(x). We omit newInstance(c) as it can be encoded with execInArea. The return type of enter () is the $\mathsf{S_{imm}}$ Object and it always returns the null value. Its behavior is to call the run method of the reciever. The execInArea method will jump to the scope of the receiver and execute the run method of its argument.

The notation is standard from [8] and [15]. An overbar, $\overline{\mathsf{x}}$, means a possibly empty sequence $\mathsf{x}_1 \ldots \mathsf{x}_n$. The shorthand $\overline{\mathsf{x}\ op\ \mathsf{y}}$ denotes the pointwise extension of any predicate *op* to a sequence, i.e. $x_1\ op\, \mathsf{x}_1, \ldots, \mathsf{x}_n\ op\, \mathsf{x}_n$. Unsurprising auxiliary definitions come from [15] and can be found in appendix in the extended version of this paper. Metavariable S represents ranges of scope names. Distinguished variables $\mathsf{S_{call}}$, $\mathsf{S_{this}}$, $\mathsf{S_{imm}}$ and $\mathsf{S_{unk}}$ represent the caller, this, immortal and unknown scopes. The features of Java we have omitted include interfaces, exceptions, primitive data types, arrays, generics, access modifiers, static methods, and threads. While important in their own right they are not required and adding them would not impact our result.

A. Static Semantics

The static semantics is given by the rules of Fig. 3. For simplicity we will assume that all types τ occurring in a SCJ program are well formed, i.e. if $\tau = \mathsf{SC}$ then either $scope(\mathsf{C}) \in \{\mathsf{S_{call}}, \mathsf{S}\}$ or $\mathsf{S} = \mathsf{S_{unk}}$. The function $scope(\mathsf{C})$ returns the declared scope of class C and $scope(\mathsf{C.m})$ return the scope of the return value of method m in class

$$
\begin{array}{lcl}
p & ::= & \overline{cd} \\
cd & ::= & \mathsf{S\ class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} \\
fd & ::= & \tau\ \mathsf{f} \\
md & ::= & \tau\ \mathsf{S}\ \ \mathsf{m}(\overline{\tau\ \mathsf{x}})\ \{\mathsf{s; return\ \ y}\} \\
\mathsf{s} & ::= & \mathsf{s;s}\ \mid \tau\ \ \mathsf{x = y.f}\ \mid \mathsf{x.f = y}\ \mid \tau\ \ \mathsf{x = (C)y}\ \mid \\
 & & \mid \tau\ \ \mathsf{x = new\ C}\ \mid \tau\ \ \mathsf{x = y.m(\overline{z})}\ \mid \\
 & & (\mathsf{x} \overset{@}{=} \mathsf{y})\ \mathsf{x.f = y}\ \mid \mathsf{return\ \ x} \\
\tau & ::= & \mathsf{S\ \ C}
\end{array}
$$

Figure 2: SCJ's syntax. C, D are class names, f, m are field and method names, and x, y, z are names of variables or parameters. this is a distinguished variable. For simplicity, we assume that names of classes, fields, methods and local variables are distinct.

(T-CLASS)
$$\frac{\begin{array}{c}\overline{fd}\ \text{OK in}\ \mathsf{C} \quad \mathit{methods}(\mathsf{C})\ \text{OK in}\ \mathsf{C} \quad \mathsf{S} \neq \mathsf{S}_{\mathsf{unk}} \\ \mathsf{S} \neq \mathsf{S}_{\mathsf{this}} \quad \mathit{scope}(\mathsf{D}) \in \{\mathsf{S}_{\mathsf{call}}, \mathsf{S}\}\end{array}}{\mathsf{S\ class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\}\ \text{OK}}$$

(T-METHOD)
$$\frac{\begin{array}{c}E = \overline{\mathsf{x} : \tau_{\mathsf{x}}}, \mathsf{this} : \tau_{\mathsf{this}} \quad E\,\mathsf{C.m}\ \vdash\ \mathsf{s} \quad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\tau_{\mathsf{x}}} \to \tau, \mathsf{S}) \\ \mathsf{S} \neq \mathsf{S}_{\mathsf{unk}}\end{array}}{\tau\,\mathsf{S}\ \mathsf{m}(\overline{\tau_{\mathsf{x}}\,\mathsf{x}})\{\mathsf{s}\}\ \text{OK in}\ \mathsf{C}}$$

(T-FIELD)
$$\frac{\begin{array}{c}\mathit{scope}(\mathsf{C}) = \mathsf{S}' \quad \mathsf{S} \in \{\mathsf{S}', \mathsf{S}_{\mathsf{this}}\} \cup \mathit{parents}(\mathsf{S}') \\ \mathit{scope}(\mathsf{D}) \in \{\mathsf{S}, \mathsf{S}_{\mathsf{call}}\}\end{array}}{\mathsf{S\ D\ f}\ \text{OK in}\ \mathsf{C}}$$

(T-FIELD-THIS)
$$\frac{\mathit{scope}(\mathsf{C}) = \mathsf{S} \quad \mathit{scope}(\mathsf{D}) \in \{\mathsf{S}, \mathsf{S}_{\mathsf{call}}\}}{\mathsf{S}_{\mathsf{this}}\ \mathsf{D\ f}\ \text{OK in}\ \mathsf{C}}$$

(T-FIELD-UNK)
$$\frac{\mathit{scope}(\mathsf{C}) = \mathsf{S} \quad \mathit{scope}(\mathsf{D}) \in \{\mathsf{S}, \mathsf{S}_{\mathsf{call}}\} \cup \mathit{parents}(\mathsf{S})}{\mathsf{S}_{\mathsf{unk}}\ \mathsf{D\ f}\ \text{OK in}\ \mathsf{C}}$$

Auxiliary Definitions:

$$\frac{\mathsf{S} \neq \mathsf{S}_{\mathsf{this}}}{\mathsf{S}{\downarrow_{\mathsf{S}'}} = \mathsf{S}} \qquad \frac{\mathsf{S} = \mathsf{S}_{\mathsf{this}}}{\mathsf{S}{\downarrow_{\mathsf{S}'}} = \mathsf{S}'}$$

$$\frac{\mathsf{S} \neq \mathsf{S}_{\mathsf{call}} \quad S \neq \mathsf{S}_{\mathsf{this}}}{\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}} \qquad \frac{\mathsf{S} = \mathsf{S}_{\mathsf{call}}}{\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}}$$

$$\frac{\mathsf{S} = \mathsf{S}_{\mathsf{this}}}{\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}{\downarrow_{\mathsf{D}}}} \qquad \frac{\mathit{scope}(\mathsf{D}) \neq \mathsf{S}_{\mathsf{call}}}{\mathsf{S}{\downarrow_{\mathsf{D}}} = \mathsf{S}{\downarrow_{\mathit{scope}(\mathsf{D})}}} \qquad \frac{\mathit{scope}(\mathsf{D}) = \mathsf{S}_{\mathsf{call}}}{\mathsf{S}{\downarrow_{\mathsf{D}}} = \mathsf{S}}$$

(T-SELECT)
$$\frac{E(\mathsf{y}) = \mathsf{S\,C} \quad \mathit{type}(\mathsf{C.f}) = \mathsf{S}'\,\mathsf{C}' \quad \mathsf{S}' \neq \mathsf{S}_{\mathsf{this}}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S}'\,\mathsf{C}'\,\mathsf{x} = \mathsf{y.f}}$$

(T-SELECT-THIS)
$$\frac{E(\mathsf{y}) = \mathsf{S\,C} \quad \mathit{type}(\mathsf{C.f}) = \mathsf{S}_{\mathsf{this}}\,\mathsf{C}'}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S\,C}'\,\mathsf{x} = \mathsf{y.f}}$$

(T-WRITE)
$$\frac{E(\mathsf{x}) = \mathsf{S\,C} \quad E(\mathsf{y}) = \mathit{type}(\mathsf{C.f}) = \mathsf{S}'\,\mathsf{C}' \quad \mathsf{S} \neq \mathsf{S}_{\mathsf{unk}}\ \mathsf{S}' \neq \mathsf{S}_{\mathsf{this}}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{x.f} = \mathsf{y}}$$

(T-WRITE-THIS)
$$\frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{S\,C} \quad \mathit{type}(\mathsf{C.f}) = \mathsf{S}_{\mathsf{this}}\,\mathsf{C}' \quad E(\mathsf{y}) = \mathsf{S\,C}' \\ \mathsf{S} \neq \mathsf{S}_{\mathsf{unk}} \quad \mathit{scope}(\mathsf{D}) = \mathsf{S}\end{array}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{x.f} = \mathsf{y}}$$

(T-NEW)
$$\frac{\mathit{scope}(\mathsf{C}) \in \{\mathsf{S}{\downarrow_{\mathsf{D.m}}}, \mathsf{S}_{\mathsf{call}}\} \quad \mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S\,C\,x} = \text{new C}}$$

(T-CAST)
$$\frac{E(\mathsf{y}) = \mathsf{S}'\,\mathsf{C}' \quad \mathsf{C}' <: \mathsf{C} \quad \mathsf{S}'{\downarrow_{\mathsf{D.m}}} = \mathsf{S}{\downarrow_{\mathsf{D.m}}}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S\,C\,x} = (\mathsf{C})\mathsf{y}}$$

(T-CAST-UNK)
$$\frac{E(\mathsf{y}) = \mathsf{S}'\,\mathsf{C}' \quad \mathsf{C}' <: \mathsf{C}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S}_{\mathsf{unk}}\,\mathsf{C\,x} = (\mathsf{C})\mathsf{y}}$$

(T-CALL)
$$\frac{\begin{array}{c}E(y) = \mathsf{S}'\,\mathsf{C}' \quad \mathit{type}(\mathsf{C}'.\mathsf{m}') = \mathsf{S}'', \overline{\mathsf{S}_x\,\mathsf{C}_x} \to \mathsf{S}_m\,\mathsf{C}_m \\ E(\overline{\mathsf{z}}) = \overline{\mathsf{S}_z\,\mathsf{C}_m} \quad (\mathsf{S}_x{\downarrow_{\mathsf{S}'}}){\downarrow_{\mathsf{D.m}}} \in \{\mathsf{S}_{\mathsf{unk}}, \mathsf{S}_z{\downarrow_{\mathsf{D.m}}}\} \\ (\mathsf{S}''{\downarrow_{\mathsf{S}'}}){\downarrow_{\mathsf{D.m}}} \in \{\mathsf{S}_{\mathsf{call}}, \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}\} \\ \mathsf{S}{\downarrow_{\mathsf{D.m}}} \in \{\mathsf{S}_{\mathsf{unk}}, (\mathsf{S}_m{\downarrow_{\mathsf{S}'}}){\downarrow_{\mathsf{D.m}}}\} \quad \mathsf{S}' \neq \mathsf{S}_{\mathsf{unk}}\ \vee\ \mathsf{S}_x \neq \mathsf{S}_{\mathsf{this}}\end{array}}{E\,\mathsf{D.m}\ \vdash\ \mathsf{S\,C}_m\,\mathsf{x} = \mathsf{y.m}'(\overline{\mathsf{z}})}$$

(T-ENTER)
$$\frac{E(\mathsf{x}) = \mathsf{S\,C} \quad \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}} = \mathit{parent}(\mathit{runsin}(\mathsf{C.run}))}{E\,\mathsf{D.m}\ \vdash\ \tau\,\mathsf{y} = \mathsf{x.enter}()}$$

(T-EXEC-AREA)
$$\frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{S\,C} \quad E(\mathsf{x}') = \mathsf{S}'\,\mathsf{C}' \\ \mathit{runsin}(\mathsf{C}'.\mathsf{run}) = \mathsf{S} \in \mathit{parents}(\mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}})\end{array}}{E\,\mathsf{D.m}\ \vdash\ \tau\,\mathsf{y} = \mathsf{x.execInArea}(\mathsf{x}')}$$

(T-IF-SAME)
$$\frac{E(\mathsf{x}) = \mathsf{S\,C} \quad E(\mathsf{y}) = \mathsf{S}'\,\mathsf{C}' \quad \mathit{type}(\mathsf{C.f}) = \mathsf{S}_{\mathsf{this}}\,\mathsf{C}'}{E\,\mathsf{D.m}\ \vdash\ (\mathsf{x} \overset{@}{=} \mathsf{y})\ \mathsf{x.f} = \mathsf{y}}$$

(T-RETURN)
$$\frac{\mathit{type}(\mathsf{D.m}) = \mathsf{S}, \overline{\tau_{\mathsf{x}}} \to \mathsf{S}'\,\mathsf{C}' \quad E(\mathsf{x}) = \mathsf{S}'\,\mathsf{C}'}{E\,\mathsf{D.m}\ \vdash\ \text{return x}}$$

Figure 3: Static semantics.

C. $runsin(\mathsf{C.m})$ returns the runs-in scope of the method declaration. We assume the presence of an ordered set of symbolic scope names such that $parent(\mathsf{S})$ returns the direct parent of scope S, $parents(\mathsf{S})$ return the transitive closure of the parent relation, and $parents(\mathsf{S}_{\mathsf{imm}}) = \emptyset$. The function $type(\mathsf{C.m})$ returns the runs-in scope, sequence of arguments and the return type.

The static semantics is composed of three kinds of rules. The validity of a class is expressed as cd OK which says that a class is valid. Rules md OK in C and fd OK in C state that a method or field declaration is well-typed in the context of some class C. Lastly rules of the form E D.m $\vdash$ s state that statement s is well-typed in environment E and method m of class D. (T-CLASS) ensures that the scope of a class is either the same as its parent or its parent is $\mathsf{S}_{\mathsf{call}}$ and the class has a named scope. (T-METHOD) ensures that body of a method is well typed. The *override* auxiliary function ensures that the type and scope of an overriding definition match previous declarations of the method in parent classes. The three rules for a field declaration S D f ensure that if the field is declared of scope $\mathsf{S}_{\mathsf{this}}$ and class D occurring within class C, then class D has either same scope as the

Syntax:

$$
\begin{array}{llllll}
R & ::= & \epsilon \mid R\,\langle \rho\ c\,H\rangle & S & ::= & \epsilon \mid S\,\langle \mathsf{D.m}\,\mathsf{m}\,F\,\rho\rangle \mathsf{s} \\
H & ::= & [] \mid H[o \mapsto v] & F & ::= & [] \mid F[\mathsf{y} \mapsto r] \\
r & ::= & o^{\rho}_{c} & v & ::= & \mathsf{C}(\overline{r})
\end{array}
$$

Dynamic Semantics:

(D-RETURN)
$$\frac{S = S'\,\langle \mathsf{D.m}\,F'\,\rho\,\mathsf{x} = \mathsf{y'.m'}(\overline{\mathsf{z}});\mathsf{s}\rangle\langle \mathsf{D'.m'}\,F\,\rho'\,\mathsf{return\,y}\rangle}{R\,S \rightarrow R\,S'\,\langle \mathsf{D.m}\,F'[\mathsf{x} \mapsto F(\mathsf{y})]\,\rho\,\mathsf{s}\rangle}$$

(D-CAST)
$$\frac{}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x}{=}(\mathsf{C})\mathsf{y};\mathsf{s}\rangle \rightarrow R\,S\,\langle \mathsf{D.m}\,F[\mathsf{x} \mapsto F(\mathsf{y})]\,\rho\,\mathsf{s}\rangle}$$

(D-SELECT)
$$\frac{R = R'\langle \rho'\ c\,H\rangle R'' \qquad F(\mathsf{y}) = o^{\rho'}_{c} \qquad H(o) = \mathsf{C}(\overline{r}\,r_i\,\overline{r'})}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x}{=}\mathsf{y.f}_i;\mathsf{s}\rangle \rightarrow R\,S\,\langle \mathsf{D.m}\,F[\mathsf{x} \mapsto r_i]\,\rho\,\mathsf{s}\rangle}$$

(D-UPDATE)
$$\frac{\begin{array}{c} R = R''\langle \rho'\ c\,H\rangle R''' \quad F(\mathsf{x}) = o^{\rho'}_{c} \\ H(o) = \mathsf{C}(\overline{r}\,r_i\,\overline{r'}) \quad R' = R''\langle \rho'\ c\,H[o \mapsto \mathsf{C}(\overline{r}\,F(\mathsf{y})\,\overline{r'})]\rangle R''' \end{array}}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{x.f}_i{=}\mathsf{y};\mathsf{s}\rangle \rightarrow R'\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle}$$

(D-CALL)
$$\frac{\begin{array}{c} S = S'\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x}{=}\mathsf{y.m'}(\overline{\mathsf{z}});\mathsf{s}\rangle \quad R = R'\langle \rho\ c\,H\rangle R'' \\ F(\mathsf{y}) = o^{\rho}_{c} \quad H(o) = \mathsf{C}(\overline{r'}) \quad F(\overline{\mathsf{z}}) = \overline{r} \\ mbody(\mathsf{C.m'}) = (\overline{\tau_{\mathsf{x}}\,\mathsf{x'}};\,\mathsf{s'}) \quad F' = [\overline{\mathsf{x'} \mapsto r}][\mathsf{this} \mapsto r] \end{array}}{R\,S \rightarrow R\,S\,\langle \mathsf{C.m'}\,F'\,\rho\,\mathsf{s'}\rangle}$$

(D-NEW)
$$\frac{\begin{array}{c} R = R''\langle \rho\ c\,H\rangle R''' \quad o \notin dom(H) \\ R' = R''\langle \rho\ c\,H[o \mapsto \mathsf{C}(\overline{\mathsf{null}})]\rangle R''' \end{array}}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x} = \mathsf{new\,C};\mathsf{s}\rangle \rightarrow R'\,S\,\langle \mathsf{D.m}\,F[\mathsf{x} \mapsto o^{\rho}_{c}]\,\rho\,\mathsf{s}\rangle}$$

(D-IF-SAME-T)
$$\frac{\begin{array}{c} R = R''\langle \rho'\ c\,H\rangle R''' \quad F(\mathsf{x}) = o^{\rho'}_{c} \quad F(\mathsf{y}) = {o'}^{\rho'}_{c} \\ H(o) = \mathsf{C}(\overline{r}\,r_i\,\overline{r'}) \quad R' = R''\langle \rho'\ c\,H[o \mapsto \mathsf{C}(\overline{r}\,{o'}^{\rho'}_{c}\,\overline{r'})]\rangle R''' \end{array}}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,(\mathsf{x} \overset{@}{=} \mathsf{y})\,\mathsf{x.f} = \mathsf{y};\mathsf{s}\rangle \rightarrow R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle}$$

(D-IF-SAME-F)
$$\frac{F(\mathsf{x}) = o^{\rho}_{c} \qquad F(\mathsf{y}) = {o'}^{\rho'}_{c'} \qquad \rho \neq \rho'}{R\,S\,\langle \mathsf{D.m}\,F\,\rho\,(\mathsf{x} \overset{@}{=} \mathsf{y})\,\mathsf{x.f} = \mathsf{y};\mathsf{s}\rangle \rightarrow R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle}$$

(D-ENTER)
$$\frac{\begin{array}{c} S = S'\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.enter}();\mathsf{s}\rangle \\ R = R'\langle \rho\ c\,H\rangle\langle \rho'\ c'\ H'\rangle R'' \\ F(\mathsf{x}) = o^{\rho''}_{c''} \quad \langle \rho''\ c''\ H''\rangle \in R \quad H''(o) = \mathsf{C}(\overline{r}) \\ F' = [\mathsf{this} \mapsto r] \qquad mbody(\mathsf{C.run}) = \mathsf{s'} \\ if\ \ \rho' \in S\ \ then\ \ R''' = \langle \rho'\ c'\ H'\rangle\ \ else\ \ R''' = \langle \rho'\ c' + 1\,\epsilon\rangle \end{array}}{R\ S \rightarrow R'\,\langle \rho\ c\,H\rangle R'''\ R''\ S\ \langle \mathsf{C.run}\,F'\ \rho'\ \mathsf{s'}\rangle}$$

(D-EXEC-AREA)
$$\frac{\begin{array}{c} S = S'\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.execInArea}(\mathsf{x'});\mathsf{s}\rangle \\ R = R'\langle \rho\ c\,H\rangle R'' \qquad F(\mathsf{x}) = o^{\rho'}_{c'} \qquad \langle \rho'\ c'\ H'\rangle \in R \\ \rho' \in S \qquad F(\mathsf{x'}) = {o'}^{\rho''}_{c''} \qquad \langle \rho''\ c''\ H''\rangle \in R \\ H''(o) = \mathsf{C}(\overline{r}) \qquad F' = [\mathsf{this} \mapsto {o'}^{\rho''}_{c''}] \qquad mbody(\mathsf{C.run}) = \mathsf{s'} \\ S'' = S'\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.execInArea}(\mathsf{x'});\mathsf{s}\rangle\langle \mathsf{C.run}\,F'\ \rho'\ \mathsf{s'}\rangle \end{array}}{R\ S \rightarrow R\ S''}$$

Figure 4: Dynamic syntax and semantics.

enclosing class C, or is declared $\mathsf{S_{call}}$. If a field is declared $\mathsf{S_{unk}D}$ then the scope of D must either be $\mathsf{S_{call}}$ or one of the parents of C. Lastly, for all other cases the scope of the enclosing class C must be in the same as (or parent of) S.

The rules for expressions must take care of concretization of scope annotations and a number of special cases. (T-SELECT) states that reading a field y.f and storing is allowed if the type and scope of the field match the target local variable. If the field is annotated THIS then the target variable must have the same scope as that of the object that holds the field f. (T-WRITE-THIS) and (T-WRITE) cover assignments to fields and ensure that the scope of the field and of the variable to store in it match. (T-NEW) checks that a newly allocated object is created in a scope that, after concretization, is equal to the scope of the current method. The concretization predicate $\mathsf{S}{\downarrow_{\mathsf{D}}}$ yields S if the scope is not $\mathsf{S_{this}}$ or if $scope(\mathsf{D})$ is $\mathsf{S_{call}}$ and $scope(\mathsf{D})$ otherwise. $\mathsf{S}{\downarrow_{\mathsf{D.m}}}$ is similar except when the scope is $\mathsf{S_{call}}$, in which case the @RunsIn of the method is used. (T-CAST) allows upcasts as long as the scope is not modified, while (T-CAST-UNK) allows to cast any scope to unknown. (T-ENTER) makes sure that the target object has a run method with a scope that is a child of the caller. (T-EXEC-AREA) is required to run in a parent scope of the caller. (T-IF-SAME) allows the storing a variable of unknown scope in THIS field if a dynamic guard established that scopes are equivalent. (T-RETURN) ensures that the return type of a method has the declared scope.

B. Dynamic Semantics

We formulate SCJ's dynamic semantics as a small-step operational semantics. Fig. 4 shows the syntax used for heaps, regions, references, call stacks, call frames, and objects. We model a region as a triple $\langle \rho\ c\,H\rangle$ where ρ is a region identifier, c is time stamp that is incremented every time the data in the region is deallocated. H is a mapping from unique addresses to objects. The key intuition is that whenever a reference is accessed we must check that the region that it points into has the same time stamp, if not an error has occurred. A configuration $R\,S$ is a pair of a region stack and the call stack of the single thread. Fig. 4 lists the rules for one step of computation $R\,S \rightarrow R'\,S'$. (D-UPDATE) presents the critical assignment rule. Its conclusion, $R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{x.f}_i{=}\mathsf{y};\mathsf{s}\rangle \rightarrow R'\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle$, states that in a configuration with topmost call frame containing an assignment to the i-th field of object referenced by x evolves in one step the configuration with a modified stack of regions R'. The antecedents for this rule to be applicable require that x and y be valid references, and in particular that y point to some scope ρ'' whose position in the scope stack is equal or above to the scope ρ' of x. Note that in (D-SELECT), execution will get stuck if the time stamp is not current. (D-ENTER) checks if the scope about to be entered is already on the call stack and if not clears the scope.

C. Properties

The goal of the type system is to prevent particular errors. In our case, the type system ensures, as usual, that a well-typed program will make disciplined use of object (i.e. all fields and methods are present when accessed, and the right types are passed/returned). Moreover it will guarantee that all references accessed by the program are valid (i.e. that they point to scopes that are on the scope stack, with valid time stamps). To prove that this is the case we will reason inductively on configuration. We will define the notion of a well-formed configuration, written $R\,S$ is WF, which states that the region and the call stack are well-formed and contains no invalid references. The proof then shows that any evaluation step preserves well-formedness and that a well-formed configuration can always take another step (unless the program has terminated).

Runtime Subtyping: To check that a configuration is well-formed we need to verify that references are well-typed. This is more subtle than it seems at first due to polymorphic annotations such as THIS which have no run-time equivalent. We thus define the run-time subtyping relation, $r <:^{R}_{\rho,\rho'} \tau$ indicates that a reference r is an instance of type τ at run-time, in the context of a scopes ρ and ρ' and the region stack R. Let $\tau = \mathsf{S}\,\mathsf{C}$, $r = o^{\rho''}_{c''}$, ρ be the scope of this, and ρ' be the scope of the method being executed. For subtyping to hold it must be the case that $R \equiv R'\langle \rho''\ c''\ H[o \mapsto v]\rangle R''$. The relation holds if $v = \mathsf{null}$. Otherwise if $v = \mathsf{D}(\overline{r})$, it must be the case that $\mathsf{D} <: \mathsf{C}$ and: If $\mathsf{S} = \mathsf{S_{this}}$, then $\rho'' = \rho$. If $\mathsf{S} = \mathsf{S_{unk}}$ the relation holds. If $\mathsf{S} = \mathsf{S_{call}}$ then $\rho'' = \rho'$. Otherwise, $\rho'' = |\mathit{parents}(\mathsf{S})|$.

Well-formed configurations: A *configuration* is well-formed, written $R\ S$ is WF, if the region stack and call stack are well-formed and the class table is well-typed. A region stack R is well-formed if it is empty or if all fields of all objects it contains are well-typed, meaning that the reference corresponding to each field is a runtime subtype of the static type of that field, the reference points to a region below in which the field's object belongs, and the region counter is equal to the counter of the region itself. A call stack S is well-formed if each frame $F \in S$ is well-formed. A frame F is well-formed if for each variable x in its domain, the reference is a runtime subtype of the static type of x and its region counter matches that of the region in which the referenced object lives. The rules appear in Fig. 5.

We prove type soundness of SCJ by showing preservation and progress. Here, preservation means that reduction of a well-formed configuration results in a well-formed configuration, and the proof of preservation states that after a step of reduction a well-formed configuration remains well-formed.

$$\frac{\forall \rho \in S, \langle \rho\ c\,H\rangle \in R \text{ is WF in } R \wedge \langle \rho\ c\,H\rangle \text{ nests in } R \qquad S \text{ is WF in } R}{R\,S \text{ is WF}}\ (\textsc{wf-configuration})$$

$$\frac{}{\langle 0\ c\,H\rangle \text{ nests in } R}\ (\textsc{wf-imm-region}) \qquad \frac{R = R'\langle \rho\ c\,H\rangle\langle \rho+1\ c'\ H'\rangle R''}{\langle \rho+1\ c'\ H'\rangle \text{ nests in } R}\ (\textsc{wf-region-nest})$$

$$\frac{}{\mathrm{wff}(\rho', \mathsf{null}, \tau, R)}\ (\textsc{wf-null-field}) \qquad \frac{}{\langle \rho\ c\,\epsilon\rangle \text{ is WF in } R}\ (\textsc{wf-empty-region})$$

$$\frac{\langle \rho\ c\,H[o \mapsto v]\rangle \in R \qquad \rho \le \rho' \qquad o^{\rho}_{c} <:^{R}_{\rho',\emptyset} \tau}{\mathrm{wff}(\rho', o^{\rho}_{c}, \tau, R)}\ (\textsc{wf-field})$$

$$\frac{R = R'\langle \rho\ c\,H\rangle R'' \qquad \langle \rho\ c\,H\rangle \text{ is WF in } R \qquad \forall i \in [1, |r|].\mathrm{wff}(\rho, r_i, \mathit{type}(\mathsf{C.f}_i), R) \qquad o^{\rho}_{c} <:^{R}_{\emptyset,\rho} \mathit{scope}(\mathsf{C})\ \mathsf{C}}{\langle \rho\ c\,H[o \mapsto \mathsf{C}(\overline{r})]\rangle \text{ is WF in } R}\ (\textsc{wf-region})$$

$$\frac{\begin{array}{c}\mathit{locals}(\mathsf{D.m}, F) = E \qquad \mathsf{D.m}\,E \vdash \mathsf{s} \qquad F(\mathsf{this}) = o'^{\rho'}_{c'} \\ E(\mathsf{x}) = \mathsf{S\,C} \qquad o^{\rho}_{c} <:^{R}_{\rho',\rho''} \mathsf{S}{\downarrow_{\mathsf{D.m}}}\ \mathsf{C} \qquad \langle \rho\ c\,H\rangle \in R \\ \langle \mathsf{D.m}\,F\,\rho''\,\mathsf{s}\rangle \text{ is WF in } R \qquad \mathsf{S}' = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}} \\ (\mathsf{S}' = \mathsf{S_{call}} \vee (\mathsf{S}' = \mathsf{S_{this}} \wedge \rho' = \rho) \vee \rho = |\mathit{parents}(\mathsf{S}')|\end{array}}{\langle \mathsf{D.m}\,F[\mathsf{x} \mapsto o^{\rho}_{c}]\,\rho''\,\mathsf{s}\rangle \text{ is WF in } R}\ (\textsc{wf-frame})$$

$$\frac{\mathit{locals}(\mathsf{D.m}, F) = E \qquad \mathsf{D.m}\,E \vdash \mathsf{s} \qquad \langle \mathsf{D.m}\,F\,\rho''\,\mathsf{s}\rangle \text{ is WF in } R}{\langle \mathsf{D.m}\,F[\mathsf{x} \mapsto \mathsf{null}]\,\rho''\,\mathsf{s}\rangle \text{ is WF in } R}\ (\textsc{wf-frame-null})$$

$$\frac{S = S''\langle \mathsf{D.m}\,F\,\rho\,\mathsf{y.m'}(\overline{\mathsf{z}})\rangle \text{ is WF in } R \qquad S' = \langle \mathsf{D'.m'}\,F'\,\rho\,\mathsf{s}\rangle \text{ is WF in } R}{SS' \text{ is WF in } R}\ (\textsc{wf-stack})$$

$$\frac{S = S''\langle \mathsf{D.m}\,F\,\rho\,\mathsf{y.enter()}\rangle \text{ is WF in } R \qquad E(\mathsf{y}) = \mathsf{S\,C} \qquad S' = \langle \mathsf{C.run}\,[\mathsf{this} \mapsto F(\mathsf{y})]\,\rho+1\,\mathsf{s}\rangle \text{ is WF in } R}{SS' \text{ is WF in } R}\ (\textsc{wf-stack-enter})$$

$$\frac{\begin{array}{c}S = S''\langle \mathsf{D.m}\,F\,\rho\,\mathsf{y.execInArea(z)}\rangle \text{ is WF in } R \qquad E(\mathsf{y}) = \mathsf{S\,C} \\ E(\mathsf{z}) = \mathsf{S'\,C'} \qquad S' = \langle \mathsf{C'.run}\,[\mathsf{this} \mapsto F(\mathsf{z})]\,\rho'\,\mathsf{s}\rangle \text{ is WF in } R \\ \rho' \le \rho \qquad \rho' = |\mathit{parents}(\mathit{runsin}(\mathsf{C'.run}))| = |\mathit{parents}(\mathsf{S}{\downarrow_{\mathsf{D.m}}})|\end{array}}{SS' \text{ is WF in } R}\ (\textsc{wf-stack-exec-area})$$

Figure 5: Well-formedness rules.

Theorem 4.1: Preservation. If $R\ S$ is WF and $R\ S \to R'\ S'$, then $R'\ S'$ is WF.

A thread is active, written *active(S)*, if it has not stuck on a null pointer exception and if there are more instructions to evaluate. Progress ensures that if the configuration is well-formed and the thread is active, then it is possible to take a step of reduction.

Theorem 4.2: Progress. If $R\ S$ is WF and $\mathit{active}(S)$ then $R\ S \to R'\ S'$.
Proofs are in the full paper on the project web page.

Case Study	LOC	Classes	Methods	SCJ Entities	@DefineScope	@Scope	@RunsIn	Annotation Density	Benchmarks LEON3	x86-rt
Quicksort	420	5	21	2	2	6	7	3.6%	39%	26.1%
Thruster	1 067	17	36	6	21	28	16	5.9%	—	—
Fast-MD5	1 148	6	31	2	2	2	2	0.5%	2.4%	18.4%
Webserver	1 679	28	40	6	14	18	14	2.7%	—	—
CDx	3 741	31	142	2	2	24	27	1.4%	1.4%	4.5%
Railsegment	5 413	50	197	23	23	86	170	5.1%	—	—
JPapaBench	12 708	150	757	17	17	45	56	0.9%	0.4%	1.7%
Total	26 176	287	1 224	58	81	209	292	2.2%	—	—

Table II: Case Studies. The SCJ Entities represent the number of MissionSequencer and Schedulable implemented. The Annotation Density column shows the percentage of the code in each benchmark that is annotated.

V. Evaluation

We implemented a static verifier, SCJ-Checker, using the Checker Framework (JSR308) which is part of Java 7 and allows an extended annotation syntax. In particular it supports annotations on local variables. Our implementation is 5 KLOC and is integrated into the javac compiler. In order to conduct a field test of the annotation system, we have gathered an extensive set of SCJ applications. We used two synthetic benchmarks, Quicksort and Fast-MD5, three small RTSJ applications, Thruster, Webserver, and Railsegment – developed by A. Wellings, Sun, and Atego Inc. respectively and two larger RTSJ benchmarks: CDx [12] and JPapabench [9]. In all cases, we converted the RTSJ programs to SCJ, applied the annotations and verified each application using the SCJ-Checker. We also report performance numbers. The oSCJ VM was used to execute the applications [12]. We have modified oSCJ to omit scope checks for annotated programs.

A. Use Cases

Table II summarizes the development effort involved in adding annotations to our benchmark suite. The table gives the size of each application (LOC, number of classes and number of methods). The total size is 26 KLOC and 287 classes. More germane to the SCJ development effort, the table gives the number of SCJ entities (MissionSequencer and Schedulable classes which must be annotated); this ranges between 2 and 23. The key metric is the annotation density, or the ratio of SCJ annotations per line of code. This ranges between 0.5% and 5.9%. The higher numbers, Thruster and Railsegment, are either small or are frameworks with little behavior. CDx and JPapabench are the most representative of real programs and they have 0.9% and 1.4%. We conclude memory safety annotations meet our goal of being lightweight.

Some other issues have come up. *Class Duplication.* The most displeasing aspect of assigning classes to named scopes, mentioned in [1], is the necessity to duplicate classes that are to be used in different scopes. This can be sometimes alleviated by using polymorphic annotations (CALLER). We encountered this issue when refactoring JPapabench, where a task handler class was used to instantiate several different handlers. And since each handler must be explicitly annotated with a unique scope name, the annotation system required us to duplicate the class. *Restrictions.* The annotation system is conservative and may prevent the checking of correct programs. We have not encountered any such case in our benchmark suite. *String Literals.* Strings constants are statically allocated objects and thus should be implicitly IMMORTAL. However, this prevents users from assigning a string literal to a local variable even though the string literal is immutable. Therefore, we chose to treat these strings as CALLER so they may be safely assigned to local variables. *Standard Libraries.* Even though the system requires annotation of standard Java libraries, we believe that this one-time cost paid by JVM vendors is negligible in comparison to the costs of sanitizing those libraries to qualify them for safety certification and then actually gathering all of the required safety certification evidence. While refactoring CDx and Railsegment, a customized HashMap version had to be implemented to allow cross-scope invocation of its methods.

B. Case Study: CDx

Based on our experience, the effort required to refactor a correct RTSJ application to a verifiable SCJ program is small. RTSJ developers already do much of the work required to add annotations. They must envision the scope structure and reason about allocation context while coding. Thus adding annotation is usually a straightforward documentation step which aids in peer review and software maintanance. Consider the CDx benchmark, excerpted in Fig. 6. The class Handler is a periodic event handler. Its role in the application is to track aircrafts represented by their Sign and updates their positions in the Table. The classes are written with a minimum number of annotations (though the figure hides much of the logic which has no annotations at all). We divide the process of annotating the code into three simple refactoring steps.

First, we annotate MissionSequencers, their Missions, and Schedulables as dictated by SCJ. This does not represent any challenge since the annotations only reflect the scope-memory rules defined by SCJ. Thus users express these rules explicitly in the code, increasing the code's clarity. By experience, this simple step alone accounts for a large number of all annotations needed. In this step, @Scope("M") and @DefineScope("P","M") are added to the handler's declaration to indicate that instances will be allocated in the scope named M and that the PrivateMemory instance associated with the handler is referred to as P. The handleAsyncEvent() method is annotated @RunsIn("P") to indicate that while the enclosing class is allocated in mission memory, the method will be called from a scope with symbolic name P.

In the second refactoring step, we determine the @Scope of each class and the @RunsIn of their methods. Classes that will be used in different scopes need no @Scope annotation due to the defaulting rules. This is the case of Sign and V3d. The Handler class is already annotated, we only add

```
@Scope("M") @DefineScope(name="P", parent="M")
class Handler extends PeriodicEventHandler {

  Table t;
  Run r = new Run();

  @RunsIn("P") void handleAsyncEvent() {
    Sign s = ...;
    @Scope("M") V3d old = t.get(s);
    if (old == null) {
      @Scope("M") Sign ns = mk(s);
      put(ns);
    } else ...
  }

  @RunsIn("P") @Scope("M") Sign mk(Sign s) {
    @DefineScope(name="M", parent="IMMORTAL")
    @Scope(IMMORTAL) ManagedMemory m =
      (ManagedMemory) MemoryArea.getMemoryArea(t);
    @Scope("M") Sign ns = m.newInstance(
      Sign.class);
    ns.b = (byte[]) m.newArrayInArea(t,
      byte.class, s.b.length);
    ...
    return ns;
  }

  @RunsIn("P") void put(@Scope("M") Sign s) {
    r.s = s;
    ManagedMemory.getMemoryArea(t).
      executeInArea(r);
  }

  class Run implements SCJRunnable {

    Sign s;

    @RunsIn("M") void run() { t.put(s); }
  }
}

@Scope("M") class Table {

  @RunsIn(CALLER) @Scope("M")
  V3d get(@Scope(UNKNOWN) Sign s) {...}

  void put(@Scope("M") Sign s) {...}
}
```

Figure 6: Annotated CDx classes.

@RunsIn("P") to the remaining methods. Table is annotated @Scope("M") so that it can be referenced from Handler and from objects in the nested scope P. Further, the Table.get() method is annotated @RunsIn(CALLER), to enable cross-scope communication, and @Scope("M"), to return objects allocated in M. The argument is UNKNOWN because the method can potentially be called from any subscope. The Table.put() method has no @RunsIn annotation and thus defaults to the scope of its class.

The final step focuses on reference assignments. This is eased by the annotations from the previous steps. Consider the handleAsyncEvent() method. The first assignment is old=t.get(s). The left hand side is declared to be in M which matches the return type of Table.get(). The same reasoning holds for the assignment to ns. Handler.mk() method creates a copy of s in M. First, SCJ's getMemoryArea() method returns an object representing M, variable m holding the object is annotated with @Scope and @DefineScope to statically capture this information. The newInstance() method then returns instance of Sign.class in the scope represented by m. The resulting object is assigned to ns, which must be explicitly annotated @Scope("M"). Similarly, Sign's byte array is instantiated in M by newArrayInArea(); however, no annotation is needed, since the scope of the field cs is the same as the scope of its owner ns. The Handler.put() method is responsible for storing new Sign object into the Table. Since Table.put() is @RunsIn("M"), a special runnable is used to switch execution context to M by calling executeInArea(), and then the method can be safely invoked. Finally, a HashMap implementation annotated with @RunsIn(CALLER) annotations is necessary to complement the Table implementation. The checker can validate that this program is memory safe since all reference assignments and method invocations are valid.

C. *Performance*

Statically proving memory safety allows the VM to optimize out the runtime checks that verify reference assignment operations. We use two platforms to measure the performance impact of removing scope checks. The LEON3 is a Xilinx Spartan3-1500 board at 40Mhz with 64MB of PC133 SDRAM and with the RTEMS v4.9.3 OS. x86-rt is an Intel Pentium 4 3.80GHz single core machine with 3GB of RAM, Ubuntu Linux 9.04 with the 2.6.28-3-rt 32-bit RT PREEMPT kernel. The following four programs were suitable for throughput measurements: Quicksort, Fast-MD5, CDx and JPapabench. Results are presented in Table II as a percentage overhead of dynamic scope checks measured on both platforms. The largest improvement in performance are observed for Quicksort, with 26.1%. This is clearly because that workload is dominated by reference assignments. Other workloads show that removing scope checks have a smaller impact. CDx and JPapabench being representative real-time application show modest improvements of 4.5% and 1.7%, respectively, on x86-rt. The results on the LEON3 board are somewhat obscured by the board's lack of a floating point unit. Thus Fast-MD5, CDx and JPapabench show small relative improvement because the execution time is dominated by floating point overheads.

VI. Related Work

The Aonix PERC Pico virtual machine introduces stack-allocated scopes, an annotation system, and an integrated static analysis system to verify scope safety and analyze memory requirements. The PERC type system [10] introduces annotations indicating the scope in which a given object is allocated. A byte-code verifier interpreting the annotations proves the absence of scoped memory protocol errors. The PERC Pico annotations do not introduce absolute scopes identifiers. Instead, they emphasize scope relationships (e.g. argument A resides in a scope that encloses the scope of argument B). This allows more generic reuse of classes and methods in many different scopes, rather than requiring duplication of classes for each distinct scope context at the cost of a higher annotation burden. The PERC annotations address sizing requirements which are not considered here. The authors of [3] proposed a type system for Real-Time Java. Although the work is applied to a more general scenario of RTSJ-based applications, it shows that a type system makes it possible to eliminate runtime checks. In comparison to the approach in this work, the proposed type system provides a richer but a more complex solution. Scoped Types [17], [1] introduce a type system for RTSJ which ensures that no run-time errors due to memory access checks will occur. Furthermore, Scoped Types capture the runtime hierarchy of scopes and subscopes in the program text by the static hierarchy of Java packages and by two dedicated Java annotations. The authors demonstrates that it is possible to statically maintain the invariants that the RTSJ checks dynamically, yet syntactic overhead upon programmers is small. The solution presented by the authors is a direct ancestor of the system described in this paper.

VII. Conclusion

This paper presents an annotation system that prevents memory access errors in safety critical Java applications. The system is designed to be optional and, in contrary to related approaches, does not require changing the Java syntax or semantics. Formalization and the proof of soundness of the system demonstrate that well-typed programs are guaranteed to be free of memory access errors. We further apply the system on a suite of SCJ case studies (26 KLOC) and show that the annotations are lightweight while delivering required expressiveness. Our solution is integrated in the Java compiler, and its use leads to an observed performance improvements ranging from 1.7% to 26% due to the elimination of runtime scope checks.

References

[1] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 37(1), 2007.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[3] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI*, 2003.

[4] V. A. Braberman, F. J. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM*, 2008.

[5] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java programming language. In *OOPSLA*, 1998.

[7] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, 2002.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23, 2001.

[9] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive testing of Safety Critical Java. In *JTRES*, 2010.

[10] K. Nilsen. A type system to assure scope safety within safety-critical Java modules. In *JTRES*, 2006.

[11] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *ECOOP*, 1998.

[12] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing Safety Critical Java applications with oSCJ/L0. In *JTRES 10*, 2010.

[13] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 1992.

[14] D. Tang, A. Plsek, and J. Vitek. Static checking of safety critical java annotations. In *JTRES*, 2010.

[15] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, 2009.

[16] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 2008.

[17] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS*, 2004.

APPENDIX

Type Soundness

We prove type soundness of SCJ by showing preservation and progress. Here, preservation means that reduction of a well-formed configuration results in a well-formed configuration, and the proof of preservation states that after a step of reduction a well-formed configuration remains well-formed.

Theorem A.1: Preservation. If $R\,S$ is WF and $R\,S \rightarrow R'\,S'$, then $R'\,S'$ is WF.

Proof: We proceed by structural induction on the derivation of the reduction relation. For all cases, let E be the typing environment of the topmost method of the call stack S. Let $\mathit{rtype}(r)$ represent the runtime type of a reference r. That is, if $r = o^{\rho}_{c}$ and $R = R'\langle \rho\; c\, H[o \mapsto \mathsf{C}(\overline{r})]\rangle R''$, then $\mathit{rtype}(r) = \mathsf{C}$.

Case (D-RETURN):

1) $R\,S\,\langle \mathsf{D'.m}\, F'\, \rho\, \tau\, \mathsf{x} = \mathsf{y'.m'(\overline{z}); s}\rangle\langle \mathsf{C''.m'}\, F\, \rho'\, \mathsf{return\, y}\rangle$ by (D-RETURN).
2) $F(\mathsf{this}) = o'^{\rho''}_{c'}$ by (WF-FRAME).
3) Let $\mathit{rtype}(F'(\mathsf{this})) = \mathsf{D'}$.
4) Let E' be the typing environment obtained by type checking $\mathsf{D'.m}$.
5) $E'(\mathsf{y'}) = \mathsf{S'''\, C''}$ by (T-CALL).
6) $\mathit{type}(\mathsf{C''.m'}) = \mathsf{S}, \tau_{\mathsf{x}} \rightarrow \mathsf{S'\, C'}$ by (T-CALL).
7) $E(\mathsf{y}) = \mathsf{S'\, C'}$ by (T-RETURN).
8) $F'(\mathsf{this}) = o''^{\rho'''}_{c''}$ by (WF-FRAME).
9) $F(\mathsf{y}) = r$ by (D-RETURN). We show that $r <:^{R}_{\rho''',\rho} \mathsf{S''}{\downarrow_{\mathsf{D.m}}}\ \mathsf{C'}$ by case analysis on r:
   a) If $r = \mathsf{null}$ then the relation holds. Otherwise, let $r = o^{\rho'}_{c}$.
   b) Otherwise, let $r = o^{\rho'}_{c}$. $o^{\rho'}_{c} <:^{R}_{\rho'',\rho'} \mathsf{S'}{\downarrow_{\mathsf{C''.m'}}}\ \mathsf{C'}$ by (WF-FRAME).
   c) $E'(\mathsf{x}) = \mathsf{S''\, C'}$ by (T-CALL).
   d) $o^{\rho'}_{c} <:^{R}_{\rho''',\rho} \mathsf{S''}{\downarrow_{\mathsf{D.m}}}\ \mathsf{C'}$ by case analysis on $\mathsf{S''}{\downarrow_{\mathsf{D.m}}}$:
      i) If $\mathsf{S''}{\downarrow_{\mathsf{D.m}}} = \mathsf{S_{unk}}$ then the relation holds.
      ii) If $\mathsf{S''}{\downarrow_{\mathsf{D.m}}} = \mathsf{S_{call}}$ then the relation holds if $\rho = \rho'$.
         A) If $\mathsf{m'}$ is either enter and $\mathsf{execInArea}$, this case does not apply because their return type is in $\mathsf{S_{imm}}$.
         B) Otherwise, this holds by (WF-STACK).
      iii) If $\mathsf{S''}{\downarrow_{\mathsf{D.m}}} = \mathsf{S_{this}}$ then the relation holds if $\rho' = \rho'''$. By (T-CALL), $(\mathsf{S'}{\downarrow_{\mathsf{S'''}}}){\downarrow_{\mathsf{D.m}}} = \mathsf{S_{this}}$. By definition of $\downarrow$, $\mathsf{S'''}$ and $\mathsf{S'}$ are also $\mathsf{S_{this}}$. By (WF-FRAME), $\rho' = \rho''$. Also by (WF-FRAME), $\rho'' = \rho'''$. Therefore, $\rho' = \rho'''$ and the relation holds.
      iv) Otherwise, $\mathsf{S''}{\downarrow_{\mathsf{D.m}}} = \mathsf{S'}{\downarrow_{\mathsf{C''.m'}}}$ by (T-CALL), so the relation holds.
10) $R\,S\,\langle \mathsf{D'}, \mathsf{m}\, F'[\mathsf{x} \mapsto r]\, \rho\, \mathsf{s}\rangle$ is WF by (9) and (WF-FRAME).

Case (D-CAST):

1) $R\,S\,\langle \mathsf{D.m}\, F\, \rho\, \tau\, \mathsf{x} = (\mathsf{C})\mathsf{y}; s\rangle$ by (D-CAST).
2) $E(\mathsf{x}) = \mathsf{S\, C}, E(\mathsf{y}) = \mathsf{S'\, C'}$ by (T-METHOD).
3) $F(\mathsf{this}) = o^{\rho'}_{c}$ by (WF-FRAME).
4) Let $\mathsf{S}_x = \mathsf{S}{\downarrow_{\mathsf{D.m}}}$ and $\mathsf{S}_y = \mathsf{S'}{\downarrow_{\mathsf{D.m}}}$. $\mathsf{S}_x = \mathsf{S}_y$ by (T-CAST).
5) $F(\mathsf{y}) = r$ and $r <:^{R}_{\rho',\rho} \mathsf{S}_y\, \mathsf{C'}$ by (WF-FRAME).
6) $\mathsf{C'} <: \mathsf{C}$ by (T-CAST-*).
7) We show that $r <:^{R}_{\rho',\rho} \mathsf{S}_x\, \mathsf{C}$ by (6) and case analysis on r:
   a) If $r = \mathsf{null}$ then the relation holds.
   b) Otherwise, let $r = o'^{\rho''}_{c'}$. We show by case analysis that $o'^{\rho''}_{c'} <:^{R}_{\rho',\rho} \mathsf{S}_x\, \mathsf{C}$ by case analysis on S_x:
      i) If $\mathsf{S}_x = \mathsf{S_{unk}}$ then the relation holds.
      ii) If $\mathsf{S}_x = \mathsf{S_{call}}$. By (5) we know that $\rho'' = \rho$ and the relation holds.
      iii) If $\mathsf{S}_x = \mathsf{S_{this}}$. By (5) we know that $\rho'' = \rho'$ and the relation holds.
      iv) Otherwise, S_x is a named scope. By (5) we know that $\rho = |\mathit{parents}(\mathsf{S}_y)|$. Since $\mathsf{S}_x = \mathsf{S}_y$, $\rho = |\mathit{parents}(\mathsf{S}_x)|$ and the relation holds.
8) $R\,S\,\langle \mathsf{D.m}\, F[\mathsf{x} \mapsto o'^{\rho''}_{c'}]\, \rho\, s\rangle$ by (7) and (WF-FRAME).

Case (D-SELECT):

1) $R\,S\,\langle \mathsf{D.m}\, F\, \rho\, \tau\, \mathsf{x} = \mathsf{y.f}_i; \mathsf{s}\rangle$ by (D-SELECT).
2) $E(\mathsf{x}) = \mathsf{S\, C}$

3) $F(\mathsf{y}) = o_c^{\rho'}, F(\mathsf{this}) = o'^{\rho''}_{c'}$ by (WF-FRAME).
4) $\langle \rho' \; c \, H[o \mapsto \mathsf{C}(\overline{r}, r_i, \overline{r}')]\rangle \in R$ by (WF-CONFIGURATION).
5) $\mathit{type}(\mathsf{C.f}_i) = \mathsf{S}' \, \mathsf{C}'$ by (T-SELECT).
6) From (T-SELECT), $\mathsf{C} = \mathsf{C}'$.
7) $r_i <:^R_{\rho',\emptyset} \mathsf{S}' \, \mathsf{C}'$ by (WF-FIELD).
8) If $r_i = \mathsf{null}$, $r_i <:^R_{\rho'',\rho} \mathsf{S}{\downarrow_{\mathsf{D.m}}} \; \mathsf{C}$ by definition of $<:$.
9) Otherwise, we show $r_i <:^R_{\rho'',\rho} \mathsf{S}{\downarrow_{\mathsf{D.m}}} \; \mathsf{C}$ by case analysis on $\mathsf{S}{\downarrow_{\mathsf{D.m}}}$:
   a) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{unk}}$, then by the definition of $<:$, the relation holds.
   b) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{this}}$, by (T-SELECT-THIS), $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$.
      i) If $\mathsf{S} = \mathsf{S}_{\mathsf{this}}$, then $\rho' = \rho''$ by (WF-FRAME).
      ii) If $\mathsf{S} = \mathsf{S}_{\mathsf{call}}$, $\mathit{runsin}(\mathsf{D.m}) = \mathsf{S}_{\mathsf{this}}$ by definition of $\downarrow$. $E(\mathsf{y}) = \mathsf{S}_{\mathsf{call}} \, \mathsf{C}''$ by (T-SELECT-THIS). $\mathsf{S}_{\mathsf{call}}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{this}}$. By (WF-FRAME), $\rho' = \rho''$.
      iii) Otherwise, this case does not apply.
   c) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{call}}$, then $\mathsf{S} = \mathsf{S}_{\mathsf{call}}$, $\mathit{runsin}(\mathsf{D.m}) = \mathsf{S}_{\mathsf{call}}$, and $E(\mathsf{y}) = \mathsf{S}_{\mathsf{call}} \, \mathsf{C}''$. By (WF-FRAME), $\rho' = \rho$. By (WF-FIELD), $r_i = o''^{\rho}_{c}$ and the relation holds.
   d) Otherwise, by case analysis on S:
      i) $\mathsf{S} = \mathsf{S}_{\mathsf{this}}$, then $\mathit{scope}(\mathsf{D})$ is a named scope. By (T-SELECT-THIS), $E(\mathsf{y}) = \mathsf{S}_{\mathsf{this}} \, \mathsf{C}''$ and $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$. By (WF-FRAME), $\rho' = \rho''$. By (WF-FIELD), $r_i = o''^{\rho'}_{c}$, so the relation holds.
      ii) $\mathsf{S} = \mathsf{S}_{\mathsf{call}}$, $E(\mathsf{y}) = \mathsf{S}_{\mathsf{call}} \, \mathsf{C}''$ and $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$ by (T-SELECT-THIS). By (WF-FRAME), $\rho = \rho'$ and $r_i = o''^{\rho}_{c}$ by (WF-FIELD). By definition of $\downarrow$, $\mathsf{S}_{\mathsf{call}}{\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$. By (WF-FRAME), $\rho = |\mathit{parents}(\mathsf{S}_{\mathsf{call}}{\downarrow_{\mathsf{D.m}}})|$ and the relation holds.
      iii) $\mathsf{S} = \mathsf{S}_{\mathsf{unk}}$, the case does not apply.
      iv) Otherwise, $\mathsf{S} = \mathsf{S}'$ by (T-SELECT) and by definition of $<:$, the relation holds.
10) $R \, S \, \langle \mathsf{D.m} \, F[\mathsf{x} \mapsto r_i] \, \rho \, \mathsf{s}\rangle$ is WF by (8), (9) and (WF-FRAME).

Case (D-UPDATE):
1) $R \, S \, \langle \mathsf{D.m} \, F \, \rho \, \mathsf{x.f}_i = \mathsf{y}; \mathsf{s}\rangle$ by (D-UPDATE)
2) $E(\mathsf{x}) = \mathsf{S} \, \mathsf{C}, \mathit{type}(\mathsf{C.f}) = \mathsf{S}' \, \mathsf{C}'$ by (T-WRITE)
3) $F(\mathsf{x}) = o_c^{\rho'}, F(\mathsf{y}) = r$ by (WF-FRAME).
4) We show that $\mathrm{wff}(\rho', r, \mathsf{S}' \, \mathsf{C}', R)$ holds by case analysis on r:
   a) If $r = \mathsf{null}$, it is true by (WF-NULL-FIELD).
   b) Otherwise, let $r = o'^{\rho''}_{c'}$. We show that $\mathrm{wff}(\rho', o'^{\rho''}_{c'}, \mathsf{S}' \, \mathsf{C}', R)$ holds by case analysis on S':
      i) If $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$, then wff holds if $o'^{\rho''}_{c'} <:^R_{\rho',\emptyset} \mathsf{S}_{\mathsf{this}} \, \mathsf{C}'$. In otherwords, it holds if $\rho'' = \rho'$. $E(\mathsf{y}) = \mathsf{S} \, \mathsf{C}'$ by (T-WRITE-THIS). By (WF-FRAME), $\rho'' = \rho'$ so wff holds.
      ii) Otherwise, S' is some named scope. $E(\mathsf{y}) = \mathsf{S}' \, \mathsf{C}'$ by (T-WRITE). For wff to hold, $\rho'' \leq \rho'$ and $o'^{\rho''}_{c'} <:^R_{\rho',\emptyset} \mathsf{S}' \, \mathsf{C}'$ must be true. By (T-FIELD), S is also a named scope. We show that $\rho'' \leq \rho'$ is true by case analysis on S':
         A) If $\mathsf{S} = \mathsf{S}'$, then by (WF-FRAME) $\rho'' = \rho'$.
         B) Otherwise, $\mathsf{S}' \in \mathit{parents}(\mathsf{S})$ by (T-FIELD). $|\mathit{parents}(\mathsf{S})|$ is necessarily larger than $|\mathit{parents}(\mathsf{S}')|$. By (WF-FRAME), $\rho' = |\mathit{parents}(\mathsf{S})|$ and $\rho'' = |\mathit{parents}(\mathsf{S}')|$, therefore $\rho'' < \rho'$.

         By (WF-FRAME), $o'^{\rho''}_{c'} <:^R_{\rho',\rho} \mathsf{S}' \, \mathsf{C}'$, therefore $\rho'' = |\mathit{parents}(\mathsf{S}')|$ by definition of $<:$. Thus, $o'^{\rho''}_{c'} <:^R_{\rho',\emptyset} \mathsf{S}' \, \mathsf{C}'$ also, so wff holds.
5) Let $R = R''\langle \rho' \; c \, H[o \mapsto \mathsf{C}(\overline{r} \, r_i \, \overline{r}')]\rangle R'''$ and $R' = R''\langle \rho' \; c \, H[o \mapsto \mathsf{C}(\overline{r} \, o'^{\rho''}_{c'} \; \overline{r}')]\rangle R'''$.
6) $\langle \rho' \; c \, H[o \mapsto \mathsf{C}(\overline{r} \, o'^{\rho''}_{c'} \; \overline{r}')]\rangle$ is WF in R' by (4) and (WF-REGION).
7) $R' \, S \, \langle \mathsf{D.m} \, F \, \rho \, \mathsf{s}\rangle$ is WF by (6) and (WF-CONFIGURATION).

Case (D-NEW):
1) $R \, S \, \langle \mathsf{D.m} \, F \, \rho \, \tau \, \mathsf{x} = \mathsf{new} \, \mathsf{C}; \mathsf{s}\rangle$ by (D-NEW).
2) $R = R''\langle \rho \; c \, H\rangle R'''$ by (D-NEW).
3) $o_c^{\rho} <:^R_{\emptyset,\rho} \mathit{scope}(\mathsf{C}) \, \mathsf{C}$ by case analysis on $\mathit{scope}(\mathsf{C})$:
   a) If $\mathit{scope}(\mathsf{C}) = \mathsf{S}_{\mathsf{call}}$, then the relation holds.
   b) If $\mathit{scope}(\mathsf{C}) = \mathsf{S}$, $\mathsf{S} = \mathsf{S}'$ by (T-NEW), where $\mathsf{S}' = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$. $|\mathit{parents}(\mathsf{S})| = |\mathit{parents}(\mathsf{S}')| = \rho$, so the relation holds.

4) Let $R' = R''\langle \rho\; c\, H[o \mapsto \mathsf{C}(\overline{\mathsf{null}})]\rangle R'''$.
5) $\langle \rho\; c\, H[o \mapsto \mathsf{C}(\overline{\mathsf{null}})]\rangle$ is WF in R' and nests in R' by (3).
6) $E(\mathsf{x}) = \mathsf{S}\,\mathsf{C}$ by (WF-FRAME).
7) $F(\mathsf{this}) = {o'}^{\rho'}_{c'}$ by (WF-FRAME).
8) $o^{\rho}_{c} <:^{R}_{\rho',\rho} \mathsf{S}{\downarrow_{\mathsf{D.m}}}\ \mathsf{C}$ by case analysis on $\mathsf{S}{\downarrow_{\mathsf{D.m}}}$:
   a) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{unk}}$ then the relation holds.
   b) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{call}}$ then the relation holds.
   c) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{this}}$ then $\mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}} = \mathsf{S}_{\mathsf{this}}$ by (T-NEW). By (WF-FRAME) $\rho = \rho'$ and the relation holds.
   d) Otherwise, $\rho = |\mathit{parents}(\mathsf{S}{\downarrow_{\mathsf{D.m}}})|$ by (WF-FRAME) and the relation holds.
9) $S' = S\langle \mathsf{D.m}\, F[\mathsf{x} \mapsto o^{\rho}_{c}]\, \rho\, s\rangle$ is WF by (7) and (WF-FRAME).
10) $R'\, S'$ is WF by (5), (9) and (WF-CONFIGURATION).

Case (D-CALL):
1) $R\, S''\langle \mathsf{D.m}\, F\, \rho\, \tau\, \mathsf{x} = \mathsf{y.m}'(\overline{\mathsf{z}}); \mathsf{s}\rangle$ by (D-CALL).
2) $E(\mathsf{y}) = \mathsf{S}\,\mathsf{C}$ by (T-CALL).
3) $\mathit{mbody}(\mathsf{C.m}') = \overline{\tau_{\mathsf{x}}\, \mathsf{x}'}; \mathsf{s}'$ by (D-CALL).
4) $F' = [\overline{\mathsf{x}' \mapsto F(\mathsf{z})}][\mathsf{this} \mapsto F(\mathsf{y})]$ by (D-CALL).
5) Let $S' = S''\langle \mathsf{D.m}\, F\, \rho\, \tau\, \mathsf{x} = \mathsf{y.m}'(\overline{\mathsf{z}}); \mathsf{s}\rangle\langle \mathsf{C.m}'\, F'\, \rho\, \mathsf{s}'\rangle$. S' is WF in R if (WF-FRAME) holds for the top frame and (WF-STACK) holds.
6) By (D-CALL), (WF-STACK) holds.
7) We show than (WF-FRAME) holds:
   a) Let E' be the typing environment obtained by type checking $\mathsf{C.m}'$.
   b) $F(\mathsf{y}) = o^{\rho'}_{c}$ by (D-CALL).
   c) We show that the frame is well-formed with respect to the designated allocation context by case analysis on $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}}$:
      i) If $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}} = \mathsf{S}_{\mathsf{unk}}$ the case does not apply due to (T-METHOD).
      ii) If $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}} = \mathsf{S}_{\mathsf{this}}$ then it must be the case that $\rho = \rho'$. In this case, $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C}) = \mathsf{S}_{\mathsf{call}}$. By (T-CALL), $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$. We show that $\rho = \rho'$ by case analysis on $\mathsf{S}{\downarrow_{\mathsf{D.m}}}$:
         A) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{unk}}$ this case does not apply due to (T-METHOD).
         B) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{call}}$ then $\rho = \rho'$ by definition of $<:$.
         C) If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{this}}$ then $\mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}} = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{D}) = \mathsf{S}_{\mathsf{call}}$ by definition of$\downarrow$. Let $F(\mathsf{y}) = {o''}^{\rho'''}_{c''}$. By (WF-FRAME), $\rho = \rho'''$ and $\rho' = \rho'''$ so $\rho = \rho'$.
         D) Otherwise $\mathsf{S}\ {\downarrow_{\mathsf{D.m}}}$ is a named scope and $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{D.m})\ {\downarrow_{\mathsf{D}}})|$. Since $\mathsf{S}\ {\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$ and $\rho' = |\mathit{parents}(\mathsf{S}{\downarrow_{\mathsf{D.m}}})|$, $\rho = \rho'$.
      iii) If $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}} = \mathsf{S}_{\mathsf{call}}$ then it is well-formed.
      iv) Otherwise, it must be the case that $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}})|$. Since $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}}$ is a named scope, either $\mathit{runsin}(\mathsf{C.m}')$ is a named scope or $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C})$ is a named scope.
         A) If $\mathit{runsin}(\mathsf{C.m}')$ is a named scope, then $\mathit{runsin}(\mathsf{C.m}') = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$ by (T-CALL). By (WF-FRAME), $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{D.m})\ {\downarrow_{\mathsf{D}}})|$. Therefore, $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{C.m}'))|$. Since $\mathit{runsin}(\mathsf{C.m}')$ is a named scope, $|\mathit{parents}(\mathit{runsin}(\mathsf{C.m}')) = |\mathit{parents}(\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}})|$, so the equality holds.
         B) If $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C})$ is a named scope, $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathit{runsin}(\mathsf{D.m}){\downarrow_{\mathsf{D}}}$. Since $\mathit{scope}(\mathsf{C})$ is a named scope, $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S} = \mathit{scope}(\mathsf{C})$. By (WF-FRAME), $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{D.m})\ {\downarrow_{\mathsf{D}}})|$. Also, $\rho = |\mathit{parents}(\mathsf{S})|$. Because of the values of $\mathit{runsin}(\mathsf{C.m}')$ and $\mathit{scope}(\mathsf{C})$, $\mathit{runsin}(\mathsf{C.m}'){\downarrow_{\mathsf{C}}} = \mathsf{S}$ and the equality holds.
   d) We show that each parameter x where $E'(\mathsf{x}) = \mathsf{S}'\,\mathsf{C}'$ (T-CALL) contributes to the well-formedness of the frame, where z is the actual and $E(\mathsf{z}) = \mathsf{S}''\,\mathsf{C}''$:
      i) If $F(\mathsf{z}) = \mathsf{null}$ then $F' = F''[\mathsf{x} \mapsto \mathsf{null}]$ is WF by (WF-FRAME-NULL).
      ii) Otherwise, let $F(\mathsf{z}) = {o'}^{\rho''}_{c'}$ and $F'(\mathsf{this}) = o^{\rho'}_{c}$. $F' = F''[\mathsf{x} \mapsto {o'}^{\rho''}_{c'}]$ is WF by (WF-FRAME) if ${o'}^{\rho''}_{c'} <:^{R}_{\rho',\rho} \mathsf{S}'{\downarrow_{\mathsf{C.m}'}}\ \mathsf{C}'$. We show that this relation holds by case analysis on $\mathsf{S}'{\downarrow_{\mathsf{C.m}'}}$:
         A) If $\mathsf{S}'{\downarrow_{\mathsf{C.m}'}} = \mathsf{S}_{\mathsf{unk}}$, then by definition of $<:$, the relation holds.
         B) If $\mathsf{S}'{\downarrow_{\mathsf{C.m}'}} = \mathsf{S}_{\mathsf{this}}$, the definition holds if $\rho' = \rho''$. By definition of$\downarrow$, $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C}') = \mathsf{S}_{\mathsf{call}}$. By (T-CALL), $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}''{\downarrow_{\mathsf{D.m}}}$.
            – If $\mathsf{S}{\downarrow_{\mathsf{D.m}}} = \mathsf{S}_{\mathsf{unk}}$ then $\mathsf{S} = \mathsf{S}_{\mathsf{unk}}$ and the case does not apply due to (T-CALL).

– If $\mathsf{S}\downarrow_{\mathsf{D.m}}= \mathsf{S}_{\mathsf{this}}$, let $F(\mathsf{this}) = o''^{\rho'''}_{c''}$. By (WF-FRAME), $\rho''' = \rho' = \rho''$ so the relation holds.
– If $\mathsf{S}\downarrow_{\mathsf{D.m}}= \mathsf{S}_{\mathsf{call}}$, by (WF-FRAME), $\rho = \rho' = \rho''$ so the relation holds.
– Otherwise, by definition of $<:$, $\rho' = |\mathit{parents}(\mathsf{S}\downarrow_{\mathsf{D.m}})| = \rho''$ and the relation holds.

C) If $\mathsf{S}'\downarrow_{\mathsf{C.m}'}= \mathsf{S}_{\mathsf{call}}$ the relation holds if $\rho = \rho''$. By definition of $\downarrow$, $\mathsf{S}' = \mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{call}}$. By (T-CALL), $\mathsf{S}_{\mathsf{call}}\downarrow_{\mathsf{D.m}}= \mathsf{S}''\downarrow_{\mathsf{D.m}}$ and $\mathit{runsin}(\mathsf{D.m})\downarrow_{\mathsf{D}}= \mathsf{S}''\downarrow_{\mathsf{D.m}}$.
– If $\mathsf{S}''\downarrow_{\mathsf{D.m}}= \mathsf{S}_{\mathsf{unk}}$ the case does not apply due to (T-METHOD).
– If $\mathsf{S}''\downarrow_{\mathsf{D.m}}= \mathsf{S}_{\mathsf{this}}$, let $F(\mathsf{this}) = o''^{\rho'''}_{c''}$. By (WF-FRAME), $\rho'' = \rho'''$. Also by (WF-FRAME), $\rho = \rho'''$ so the relation holds.
– If $\mathsf{S}''\downarrow_{\mathsf{D.m}}= \mathsf{S}_{\mathsf{call}}$ then by (WF-FRAME), $\rho'' = \rho$ and the relation holds.
– Otherwise by (WF-FRAME), $\rho = |\mathit{parents}(\mathsf{S}''\downarrow_{\mathsf{D.m}})| = \rho''$ and the relation holds.

D) Otherwise the relation holds if $\rho'' = |\mathit{parents}(\mathsf{S}'\downarrow_{\mathsf{C.m}'})|$. We show that this is true by case analysis on S':
– If $\mathsf{S}' = \mathsf{S}_{\mathsf{unk}}$ then the case does not apply.
– If $\mathsf{S}' = \mathsf{S}_{\mathsf{call}}$, either $\mathit{runsin}(\mathsf{C.m}')$ is a named scope or $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C})$ is a named scope. If $\mathit{runsin}(\mathsf{C.m}')$ is a named scope, then by (T-CALL), $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}''\downarrow_{\mathsf{D.m}}$. By definition of $\downarrow$, we can infer that $\mathsf{S}'\downarrow_{\mathsf{C.m}'}= \mathit{runsin}(\mathsf{C.m}')$ and also that $\mathsf{S}'\downarrow_{\mathsf{C.m}'}= \mathsf{S}''\downarrow_{\mathsf{D.m}}$. Since these are both named scopes, $\rho'' = |\mathit{parents}(\mathsf{S}'\downarrow_{\mathsf{C.m}'})|$ and the relation holds.
Otherwise, $\mathsf{S}'\downarrow_{\mathsf{D.m}}= \mathsf{S}''\downarrow_{\mathsf{D.m}}$ by (T-CALL). Since $\mathit{scope}(\mathsf{C})$ is a named scope and $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$, we can infer that $\mathsf{S} = \mathit{scope}(\mathsf{C})$. By (T-CALL), we know that $\mathsf{S}\downarrow_{\mathsf{D.m}}= \mathsf{S} = \mathit{runsin}(\mathsf{D.m})\downarrow_{\mathsf{D}}$. Given this and that $\mathsf{S}' = \mathsf{S}_{\mathsf{call}}$, $\mathsf{S}'\downarrow_{\mathsf{D.m}}$ must be concrete scope S. By (WF-FRAME), $\rho'' = |\mathit{parents}(\mathsf{S}''\downarrow_{\mathsf{D.m}})|$ and $\rho'' = |\mathit{parents}(\mathsf{S})|$. Since $\mathsf{S}' = \mathsf{S}_{\mathsf{call}}$, $\mathit{runsin}(\mathsf{C.m}') = \mathsf{S}_{\mathsf{this}}$ and $\mathit{scope}(\mathsf{C}) = \mathsf{S}$, $\mathsf{S} = \mathsf{S}'\downarrow_{\mathsf{C.m}'}$. Substituting this into the previous equation shows that the relation holds.
– If $\mathsf{S}' = \mathsf{S}_{\mathsf{this}}$ then $\mathit{scope}(\mathsf{C})$ is a named scope. By (T-CALL), $\mathsf{S}\downarrow_{\mathsf{D.m}}= \mathsf{S}''\downarrow_{\mathsf{D.m}}$; S must also be a named scope, therefore $\mathsf{S} = \mathsf{S}''\downarrow_{\mathsf{D.m}}$. By (WF-FRAME), $\rho'' = |\mathit{parents}(\mathsf{S}''\downarrow_{\mathsf{D.m}})|$. From this, $\rho'' = |\mathit{parents}(\mathsf{S})|$. Since S and $\mathit{scope}(\mathsf{C})$ are named scopes, it must be the case that $\mathsf{S} = \mathit{scope}(\mathsf{C})$, so $\rho'' = |\mathit{parents}(\mathit{scope}(\mathsf{C}))|$. By definition of $\downarrow$, $\mathsf{S}_{\mathsf{this}}\downarrow_{\mathsf{C.m}'}= \mathit{scope}(\mathsf{C})$ so the relation holds.
– Otherwise, S is a named scope and by (T-CALL), $\mathsf{S} = \mathsf{S}''\downarrow_{\mathsf{D.m}}$. By (WF-FRAME), $\rho'' = |\mathit{parents}(\mathsf{S}''\downarrow_{\mathsf{D.m}})| = |\mathit{parents}(\mathsf{S})|$. By definition of $\downarrow$, since S is a named scope, $\mathsf{S} = \mathsf{S}\downarrow_{\mathsf{C.m}'}$; therefore, $|\mathit{parents}(\mathsf{S})| = |\mathit{parents}(\mathsf{S}\downarrow_{\mathsf{C.m}'})|$ and the relation holds.

8) $R\,S'$ is WF by (6), (7), and (WF-CONFIGURATION).

Case (D-IF-SAME-T):
1) $R\,S\,\langle \mathsf{D.m}\,F\,\rho\,(\mathsf{x} \overset{@}{=} \mathsf{y})\,\mathsf{x.f}_i = \mathsf{y};\mathsf{s}\rangle$ by (D-IF-SAME-T).
2) $F(\mathsf{x}) = o^{\rho'}_{c}, F(\mathsf{y}) = o'^{\rho'}_{c}$ by (WF-FRAME) and (D-IF-SAME-T).
3) $E(\mathsf{y}) = \mathsf{S}'\,\mathsf{C}'$, $\mathit{type}(\mathsf{C.f}) = \mathsf{S}_{\mathsf{this}}\,\mathsf{C}'$ by (T-IF-SAME).
4) $o'^{\rho'}_{c} <:^{R}_{\rho',\emptyset} \mathsf{S}_{\mathsf{this}}\,\mathsf{C}'$ by definition of $<:$.
5) Let $R' = R''\langle\rho'\ c\,H[o \mapsto \mathsf{C}(\overline{r}\,o'^{\rho'}_{c}\,\overline{r}')]\rangle R'''$.
6) $\langle\rho'\ c\,H[o \mapsto \mathsf{C}(\overline{r}\,o'^{\rho'}_{c}\,\overline{r}')]\rangle$ is WF in R' by (WF-REGION).
7) $R'\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle$ is WF by (WF-CONFIGURATION).

Case (D-IF-SAME-F):
1) $R\,S\,\langle \mathsf{D.m}\,F\,\rho\,(\mathsf{x} \overset{@}{=} \mathsf{y})\,\mathsf{x.f} = \mathsf{y};\mathsf{s}\rangle$ by (D-IF-SAME-F).
2) $R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle$ is WF by (WF-CONFIGURATION).

Case (D-ENTER):
1) $R\,S\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.enter}();\mathsf{s}\rangle$ by (D-ENTER).
2) $R = R''\langle\rho\ c\,H\rangle\langle\rho'\ c'\,H'\rangle R'''$ by (D-ENTER).
3) $\rho' = \rho + 1$ by (WF-CONFIGURATION).
4) $E(\mathsf{x}) = \mathsf{S}\,\mathsf{C}$ by (T-ENTER).
5) $S' = S\,\langle \mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.enter}();\mathsf{s}\rangle\langle \mathsf{C.run}\,[\mathsf{this} \mapsto F(\mathsf{x})]\,\rho + 1\,\mathsf{s}'\rangle$
6) S' is WF in R if (WF-STACK-ENTER) and (WF-FRAME) hold.
   a) Since there are no parameters, (WF-FRAME) holds if $\rho + 1$ is well-formed with respect to its allocation context. By (T-ENTER), $\mathit{runsin}(\mathsf{C.run})$ must be a named scope, so (WF-FRAME) holds if $\rho + 1 = |\mathit{parents}(\mathit{runsin}(\mathsf{C.run}))|$. By (T-ENTER) we also know that $\mathit{runsin}(\mathsf{D.m})\downarrow_{\mathsf{D}}$ is a named scope. By (WF-FRAME), $\rho = |\mathit{parents}(\mathit{runsin}(\mathsf{D.m})\downarrow_{\mathsf{D}})|$. By (T-ENTER), $\mathit{parent}(\mathit{runsin}(\mathsf{C.run})) = \mathit{runsin}(\mathsf{D.m})\downarrow_{\mathsf{D}}$, therefore

$parents(runsin(\mathsf{C.run})) = \{runsin(\mathsf{D.m})\downarrow_{\mathsf{D}}\} \cup parents(runsin(\mathsf{D.m})\downarrow_{\mathsf{D}})$ and $|parents(runsin(\mathsf{C.run}))| = |parents(runsin(\mathsf{D.m})\downarrow_{\mathsf{D}})| + 1$. Therefore, $\rho + 1 = |parents(runsin(\mathsf{C.run}))|$ and (WF-FRAME) holds.

b) (WF-STACK-ENTER) holds by (5) and (6)(a).

7) We define R' and show that $R'\,S'$ is WF in R' by case analysis on $\rho + 1$:

a) If $\rho + 1 \in S$, then $R' = R$ and $R'\,S'$ is WF by (WF-CONFIGURATION).

b) If $\rho + 1 \notin S$, then $R' = R''\langle\rho + 1\ c + 1\,\epsilon\rangle R'''$. $R'\,S'$ is WF if no region present on the call stack has an object which points to another object in the region $\rho + 1$. By (WF-REGION), we know that no object in a lower region will point to an object in $\rho + 1$. By (WF-STACK-ENTER), we know that if $\rho + 1$ is not on the current call stack, then no upper region is on the call stack either. Region $\rho + 1$ itself has an empty heap by (D-ENTER). Therefore, by (WF-CONFIGURATION), $R'\,S'$ is WF.

Case (D-EXEC-AREA):

1) $R\,S\,\langle\mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.execInArea(x')};\mathsf{s}\rangle$ by (D-EXEC-AREA)
2) $F(\mathsf{x}) = o_{c'}^{\rho'}, F(\mathsf{x'}) = o'^{\rho''}_{c''}$ by (D-EXEC-AREA)
3) $S' = S\,\langle\mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{y} = \mathsf{x.execInArea(x')};\mathsf{s}\rangle\langle\mathsf{C.run}\,[\mathsf{this} \mapsto o'^{\rho''}_{c''}]\,\rho'\,s'\rangle$
4) S' is WF in R if (WF-STACK-EXEC-AREA) and (WF-FRAME) hold.

a) For (WF-FRAME) to hold, it must be the case that $\rho' = |parents(runsin(\mathsf{C.run}))|$. From (T-EXEC-AREA), $runsin(\mathsf{C.run})$ is the same named scope as the scope of x. By (WF-FRAME), $\rho' = |parents(runsin(\mathsf{C.run}))|$ so (WF-FRAME) holds for the new frame.

b) For (WF-STACK-EXEC-AREA) to hold, it must be the case that $\rho' \leq \rho$. By (T-EXEC-AREA), we know that $runsin(\mathsf{D.m})\downarrow_{\mathsf{D}}$ is a named scope and that $runsin(\mathsf{C.run})$ is a named ancestor of it. By (WF-FRAME), $\rho = |parents(runsin(\mathsf{D.m})\downarrow_{\mathsf{D}})|$ and $\rho' = |parents(runsin(\mathsf{C.run}))|$. Because of the parenting relation, we know that ρ' is strictly less than ρ so the relation holds.

5) $R\,S'$ is WF by (4) and (WF-CONFIGURATION).

■

Progress requires that if there exists an active thread in a well-formed configuration, this thread should be allowed to make a step. We expand the definition of $active(S)$. Let $S = S'\langle\mathsf{D.m}\,F\,\rho\,s\rangle$. As previously stated, $active(S)$ holds if there are more instructions to evaluate and the thread is not stuck on a null pointer exception. If $S' = []$ and $s = \mathsf{return\,y}$, then there are no more instructions to evaluate. We enumerate the cases where the thread can be stuck on a null pointer exception:

1) $s = \tau\,\mathsf{x} = \mathsf{y.f}_i$ and $F(\mathsf{y}) = \mathsf{null}$.
2) $s = \mathsf{x.f}_i = \mathsf{y}$ and $F(\mathsf{x}) = \mathsf{null}$.
3) $s = \tau\,\mathsf{x} = \mathsf{y.m(\bar{z})}$ and $F(\mathsf{y}) = \mathsf{null}$.
4) $s = \tau\,\mathsf{x} = \mathsf{y.enter()}$ and $F(\mathsf{y}) = \mathsf{null}$.
5) $s = \tau\,\mathsf{x} = \mathsf{y.execInArea}(z)$ and $F(\mathsf{y}) = \mathsf{null}$ or $F(\mathsf{z}) = \mathsf{null}$.
6) $s = (\mathsf{x} \overset{@}{=} \mathsf{y})\ \mathsf{x.f} = \mathsf{y}$ and $F(\mathsf{x}) = \mathsf{null}$ or $F(\mathsf{y}) = \mathsf{null}$.

Theorem A.2: Progress. If $R\ S$ is WF and $active(S)$ then $R\ S \to R'\ S'$.

Proof:

We obtain R'; S' by structural induction on s when $S = S''\langle\mathsf{D.m}\,F\,\rho\,\mathsf{s}\rangle$.

Case $[\mathsf{s'};\mathsf{s''}]$: Follows immediately by the induction hypothesis.

Case $[\tau\,\mathsf{x} = \ \mathsf{y.f}_i]$:

a) By (WF-FRAME), $F(\mathsf{y}) = r$.

b) By (a), either $r = \mathsf{null}$ or $r = o_c^{\rho}$.

i) If $r = \mathsf{null}$, then $active(S)$ does not hold.

ii) Otherwise, by (WF-FRAME) y refers to a valid object $\mathsf{C}(\overline{r})$ and by (T-SELECT) there is an r_i corresponding to f_i. Applying (D-SELECT) yields $R\,S'\langle\mathsf{D.m}\,F'\,\rho\,s'\rangle$.

Case $[\mathsf{x.f}_i\ =\ \mathsf{y}]$: Similar to the previous case.

Case $[\tau\,\mathsf{x} = \ (\mathsf{C})\mathsf{y}]$: Immediate by application of (D-CAST).

Case $[\tau\,\mathsf{x} = \ \mathsf{new\ C}]$: Immediate by (WF-CONFIGURATION) and application of (D-NEW).

Case $[\tau\,\mathsf{x} = \ \mathsf{y.m'(\bar{z})}]$:

a) If $F(\mathsf{y}) = \mathsf{null}$, $active(S)$ does not hold.

b) Otherwise, since the program is well-typed, $rtype(\mathsf{y}) = \mathsf{C}$ and C has a method $\mathsf{m'}$. By (WF-CONFIGURATION) and application of (D-CALL), we obtain $R\,S'\langle\mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x} = \ \mathsf{y.m'(\bar{z})}\rangle\langle\mathsf{D'.m'}\,F'\,\rho\,\mathsf{s'}\rangle$.

Case $[\tau\,\mathsf{x} = \mathsf{y.enter()}]$:

a) If $F(\mathsf{y}) = \mathsf{null}$, $active(S)$ does not hold.

b) Otherwise, by (WF-CONFIGURATION), (T-ENTER) and application of (D-ENTER), we obtain $R'\,S'\langle\mathsf{D.m}\,F\,\rho\,\tau\,\mathsf{x} = \mathsf{y.enter()}\rangle\langle\mathsf{D'.enter}\,F'\,\rho'\,\mathsf{s'}\rangle$.

Case $[\tau\,\mathsf{x} = \mathsf{y.execInArea(z)}]$: Similar to the previous case.

Case $[(\mathsf{x} \overset{@}{=} \mathsf{y})\ \mathsf{x.f}\ = \ \mathsf{y}]$:

a) By (WF-FRAME), $F(\mathsf{x}) = r$ and $F(\mathsf{y}) = r'$.

b) If $r = \mathsf{null}$ or $r' = \mathsf{null}$, $active(S)$ does not hold.

c) Otherwise, let $F(\mathsf{x}) = o^{\rho}_{c}, F(\mathsf{y}) = o'^{\rho'}_{c'}$.

   i) If $\rho = \rho'$, application of (D-IF-SAME-T) yields $R'\,S'\langle\mathsf{D.m}\,F\,\rho\,\mathsf{s'}\rangle$.

   ii) Otherwise, application of (D-IF-SAME-F) yields $R\,S'\langle\mathsf{D.m}\,F\,\rho\,\mathsf{s'}\rangle$.

Case $[s = \mathsf{return\ y}]$:

a) If $S' = []$, then $active(S)$ does not hold.

b) Otherwise, let $S' = S''\langle\mathsf{D'.m'}\,F'\,\rho\,\tau\,\mathsf{x} = \mathsf{y.m(\bar{z}); s'}\rangle$. By applying (D-RETURN), we obtain $R\,S''\langle\mathsf{D'.m'}\,F''\,\rho\,\mathsf{s'}\rangle$.

■

Subtyping:

$$\frac{}{\mathsf{C} <: \mathsf{C}} \qquad \frac{\mathsf{C\ extends\ D}}{\mathsf{C} <: \mathsf{D}} \qquad \frac{\mathsf{C} <: \mathsf{C'} \quad \mathsf{C'} <: \mathsf{D}}{\mathsf{C} <: \mathsf{D}}$$

Extends:

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \}}{\mathsf{C\ extends\ D}}$$

Type lookup:

$$\frac{\tau\ \mathsf{S}\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\ \{\ \overline{\tau_\mathsf{z}\,\mathsf{z}};\ \mathsf{s};\ \mathsf{return\ y}\ \} \in methods(\mathsf{C})}{type(\mathsf{C.m}) = \mathsf{S}, \overline{\tau_\mathsf{x}} \to \tau}$$

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \} \quad \mathsf{m}\ \textit{is not defined in}\ \overline{md}}{type(\mathsf{C.m}) = type(\mathsf{D.m})}$$

$$\frac{\tau\ \mathsf{f} \in fields(\mathsf{C})}{type(\mathsf{C.f}) = \tau}$$

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \} \quad \mathsf{f}\ \textit{is not defined in}\ \overline{fd}}{type(\mathsf{C.f}) = type(\mathsf{D.f})}$$

Method lookup:

$$\frac{\tau\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\ \{\ \overline{\tau_\mathsf{z}\,\mathsf{z}};\ \mathsf{s};\ \mathsf{return\ y}\ \} \in methods(\mathsf{C})}{mbody(\mathsf{C.m}) = (\overline{\tau_\mathsf{x}\,\mathsf{x}};\ \mathsf{s};\ \mathsf{return\ y})}$$

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \} \quad \mathsf{m}\ \textit{not in}\ \overline{md}}{mbody(\mathsf{C.m}) = mbody(\mathsf{D.m})}$$

Fields lookup:

$$\frac{}{fields(\mathsf{Object}) = \epsilon}$$

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \} \quad fields(\mathsf{D}) = \overline{fd'}}{fields(\mathsf{C}) = \overline{fd'}\ \overline{fd}}$$

Methods lookup:

$$\frac{}{methods(\mathsf{Object}) = \epsilon}$$

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \} \quad methods(\mathsf{D}) = \overline{md'} \quad \overline{md''} = \overline{md'} - \overline{md}}{methods(\mathsf{C}) = \overline{md}\ \overline{md''}}$$

Valid Method overriding:

$$\frac{type(\mathsf{C.m}) = \overline{\tau'} \to \tau'\ \textit{implies}\ \ \overline{\tau} = \overline{\tau'}\ \textit{and}\ \tau = \tau' \quad \mathsf{S} = runsin(\mathsf{C.m})}{override(\mathsf{m}, \mathsf{C}, \overline{\tau} \to \tau, \mathsf{S})}$$

Scope:

$$\frac{CT(\mathsf{C}) = \mathsf{S\ class\ C\ extends\ D}\ \{\ \overline{fd}\ \overline{md}\ \}}{scope(\mathsf{C}) = \mathsf{S}}$$

$$\frac{\tau\ \mathsf{S}\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\ \{\ldots\} \in methods(\mathsf{C}) \quad \tau = \mathsf{S'}\ \mathsf{D}}{scope(\mathsf{C.m}) = \mathsf{S'}}$$

$$\frac{\tau\ \mathsf{S}\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\ \{\ldots\} \in methods(\mathsf{C})}{runsin(\mathsf{C.m}) = \mathsf{S}}$$

$$\frac{parent(\mathsf{S}) = \mathsf{S'}}{parents(\mathsf{S}) = \{\mathsf{S'}\} \cup parents(\mathsf{S'})}$$

$$\frac{}{parents(\mathsf{S_{imm}}) = \emptyset}$$

Figure 7: Auxiliary definitions.