Technical Report: Region and Effect Inference for Safe Parallelism

Alexandros Tzannes, Stephen T. Heumann, Lamyaa Eloussi,
Mohsen Vakilian, Vikram S. Adve
Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science
{atzannes,heumann1,eloussi2,mvakili2,vadve}@illinois.edu

Michael Han
Autodesk Inc.
michael.han@autodesk.com

***Abstract*—In this paper, we present the first full regions-and-effects inference algorithm for explicitly parallel fork-join programs. We infer annotations inspired by Deterministic Parallel Java (DPJ) for a type-safe subset of C++. We chose the DPJ annotations because they give the *strongest* safety guarantees of any existing concurrency-checking approach we know of, static or dynamic, and it is also the *most expressive* static checking system we know of that gives strong safety guarantees. This expressiveness, however, makes manual annotation difficult and tedious, which motivates the need for automatic inference, but it also makes the inference problem very challenging: the code may use region polymorphism, imperative updates with complex aliasing, arbitrary recursion, hierarchical region specifications, and wildcard elements to describe potentially infinite sets of regions. We express the inference as a constraint satisfaction problem and develop, implement, and evaluate an algorithm for solving it. The region and effect annotations inferred by the algorithm constitute a checkable proof of safe parallelism, and it can be recorded both for documentation and for fast and modular safety checking.**

I. Introduction

In imperative parallel programs, the overwhelmingly many possible dynamic interleavings of instructions make it difficult for programmers to reason about the correctness of their program, and the non-deterministic or unstable nature of dynamic execution schedules makes it challenging to expose or reproduce parallelism bugs [1]. Many dynamic and static approaches have been proposed to address these challenges. Dynamic approaches [2], [3], [4] have been devised to address the issue, but all incur substantial performance overhead and some also require specialized hardware. Static approaches have zero runtime performance overhead but suffer from limited expressiveness and sometimes impose a heavy annotation burden on programmers. Ideally, we would like to have a highly expressive static checking approach with strong guarantees *and* minimal annotation burden on the programmer.

The strongest guarantees we know of in any parallelism checking approach – static *or* dynamic – are provided by Deterministic Parallel Java (DPJ) [5], [6]. For fork-join parallel programs, DPJ guarantees data race freedom, strong atomicity, determinism-by-default, and compositional reasoning for deterministic and non-deterministic parallel components, using only modular, static checking (cf. Section II). Using DPJ, however, can impose a significant burden: every variable, parameter, and field must be annotated with one or more region arguments specifying where it lives and/or points to; every method must be annotated with an effect summary specifying which regions the method may read or write; every type is optionally annotated with one or more region parameters, allowing the aforementioned annotations to take parametric forms, potentially allowing to distinguish between different dynamic instances of objects of the same type. In short, manually annotating the code involves conceptually sophisticated program annotations, which are likely to be too complex for general-purpose programmers to use in practice.

In this paper, we present the first regions and effects inference algorithm DPJ annotations. In particular, a programmer only has to parallelize a program using fork-join parallelism, and the algorithm infers the annotations needed for the DPJ checker to enforce all its guarantees. The current implementation of our algorithm has limitations that make some manual annotations necessary, but we will argue that all of them can be inferred using the same principles as we have used for the rest of the annotations, and it is merely a matter of implementation effort to support the necessary extensions.

The expressiveness (and complexity) of DPJ's annotations makes inference much more challenging than similar but less expressive static approaches for safe parallelism, such as type qualifier approaches (cf. Section V). Moreover, unlike other approaches that require programmer involvement when multiple solutions are possible, our annotation generation phase will automatically find a satisfying set of annotations (if one exists), and it can optionally be queried for alternate solutions. In order to allow programmers to guide the inference algorithm when necessary, we support partially annotated code: our inference must then find a solution that honors existing annotations.

Vakilian et al. [7] present an algorithm for inferring the DPJ effect annotations given the region annotations. This is insufficient to make DPJ practical because the region annotations, which the programmer must still provide, are significantly more challenging to select than the effect ones, and they are still too numerous. Also, in terms of the inference algorithm, effect inference has to solve only effect constraints with all regions therein completely specified (i.e., no region variables – only effect variables), whereas our inference has three types of constraints to solve, and those constraints include both effect and region variables to be determined. In short, the inference problem we are solving is substantially more challenging.

We tackle the complexity of inferring region and effect annotations by proceeding in *four phases* (cf. Section III). *First*, we introduce region and effect variables at all syntactic locations where manual annotations are missing. *Second*, we parse method declarations and their bodies to generate constraints according to the rules of the region and effect annotation language. *Third*, we process the generated constraints to

simplify them and to infer simpler constraints that effectively prune the search space. *Fourth*, we instantiate all region and effect variables produced in the first phase with values that satisfy the constraints output by the third phase.

The third phase, which simplifies the constraints generated during the second phase, is one of our core contributions. Its role is to prune the solution space by inferring properties of region variables and by simplifying the complex constraints produced by the previous phase, *without making the set of constraints unsatisfiable* (sound & complete). At its core are two lemmas and two theorems that transform sets of constraints matching a pattern into sets of constraints that are simpler to solve. The first theorem applies to cases where object distinction is necessary because two objects are updated in parallel, and the second one applies to recursive updates. Section III-C explains the theorems intuitively, and the Appendix provides the formal proofs.

Our contributions are: **(1)** A new algorithm that infers both region and effect annotations to statically guarantee that a parallel code is safe; This algorithm also works with partially annotated code, allowing the programmer to control multiple aspects of the parallelism checking, and allowing to modularize inference by providing manual annotations at module boundaries (Section III). **(2)** Formalized constraint generation rules for an object-oriented core language (Section III-B); **(3)** Constraint simplification rules that prune the solution space and greatly reduce time-to-solution, and formal proofs that these simplification rules are sound and complete (Section III-C); **(4)** A prototype implementation of our inference algorithm for a type-safe subset of C++, and an evaluation of it that demonstrates the feasibility of the inference.

II. Background

This section gives the needed background on the DPJ annotations to be inferred. A complete presentation of DPJ can be found in [5], [8]. Figure 1 gives a first running example.

```
1  class Point<region P> {
2     region X, Y;
3     double x in P:X;
4     double y in P:Y;
5
6     void setX(int v) writes P:X { x = v; }
7     void setY(int v) writes P:Y { y = v; }
8     void setXY(int vx, int vy) writes P:* {
9        cobegin { setX(vx); setY(vy); } }
10 }
11 region Foo;
12 void foo() writes Foo:* {
13    region R1, R2;
14    Point<Foo:R1> p1 = new Point();
15    Point<Foo:R2> p2 = new Point();
16    int x = 3, // effects on locals
17        y = 4; //       are ignored
18    cobegin{
19       p1.setXY(y,x); // writes Foo:R1:*
20       p2.setXY(x,y); // writes Foo:R2:*
21    }
22 }
```

Fig. 1. Example of annotated Point class (effects in comments are computed by the checker)

Region names, region parameters, and RPLs. To reason about memory accesses and aliasing at an appropriate granularity and level of abstraction, DPJ groups memory locations into *regions*. Regions are logical (not necessarily contiguous) sets of memory locations. Line 2 declares region names `X` and `Y`, which may be used in region arguments for types, as we will describe below. Line 1 indicates that the class takes a region parameter `P`, which allows to distinguish between different dynamic instances of objects, as we will see later. The field `x` is declared to be in region `P:X`, which is the region of field `x` of a particular instance of a `Point` object.

Definition 1 (RPL): A region path list (RPL) is a colon separated list of region parameters, region names, and wildcards that represents a set of regions (e.g., X, $P{:}X$, $P{:}\star{:}X$). The only wildcard we consider in this paper is $\star$, which stands for zero or more colon separated RPL elements.

Definition 2 (Fully Specified RPL): An RPL is fully specified if and only if it does not contain wildcards.

Similarly, field `y` is in region `P:Y`, which is *disjoint* from region `P:X`, and both are disjoint from regions P, X, and Y.

Definition 3 (Disjointness $\#$): Two RPLs are disjoint if they are distinct from the left ($\#_L$) or from the right ($\#_R$). RPLs R_1 and R_2 are *distinct from the left* if they are the same in the first n elements, they differ in element $n+1$, and neither contains a $\star$ in the first $n+1$ elements. Symmetrically, RPLs R_1 and R_2 are *distinct from the right* if they are the same in the last n elements, they differ in element $n+1$ from the end, and neither contains a $\star$ in the last $n+1$ elements.

RPLs allow us to describe nesting of regions, which is important for capturing many parallel idioms, such as parallel tree traversals, parallel divide-and-conquer algorithms, and parallel updates through an array of pointers. However, the semantics of RPL nesting may be slightly counterintuitive at first: we say that $P{:}X$ is *under* P and write $P{:}X \leq P$, but the two RPLs describe disjoint regions. In order to describe a parent region and all the child regions nested under it, we append a *star* to the RPL of the parent region ($P:\star$ in our example). Then $P{:}X$ is included in $P{:}\star$ ($P{:}X \subseteq P{:}\star$).

Definition 4 (Inclusion $\subseteq$): RPL R_2 includes RPL R_1 if the set of regions described by R_1 is a subset of the set of regions described by R_2: $R_1 \subseteq R_2$.

For example, $P{:}\star$ includes P, $P{:}X$, $P{:}Y$, $P{:}X{:}\star$, etc.

Effect Summaries. Each method is annotated with an *effect summary* which must *cover* the effects of its body. In this paper, we focus on read and write effects and leave other kinds of effects [6] for future work. We denote with $\emptyset$ the absence of (visible) side-effects, which we call *pure effect*. A read effect on an RPL R (reads R) indicates a read operation on one or more of the regions described by R. Similarly, a write effect on an RPL R (writes R) indicates a write operation.

Definition 5 (Coverage $\subseteq$): Effect e_1 on RPL R_1 covers effect e_2 on RPL R_2 ($e_2R_2 \subseteq e_1R_1$) if and only if $e_2 \leq e_1$ and $R_2 \subseteq R_1$, where $e_1, e_2 \in \{\text{reads}, \text{writes}, \emptyset\}$ and $\emptyset \leq \text{reads} \leq \text{writes}$.

Line 6 of Fig.1 defines method `setX`, which writes field `x` and thus has effect summary `writes P:X`. Similarly, method `setY` on line 7 `writes P:Y`. Method `setXY` calls the previously defined setter methods of the current object, and has the declared effect summary `writes P:*`, which is coarser than needed for illustration purposes. The effects of the body are invocations, so the static checker fetches the effect summaries of these methods, computes their union `writes P:X,P:Y`, and checks that it is covered by `writes P:*`.

The DPJ checker does not trust effect summary annotations: it checks them by inferring the effects of the method body using region information for all references in read and write operations and the declared effect summaries of the methods that are called. Having effect summaries allows the

DPJ checker to be modular, and inferring the effect summaries requires interprocedural analysis [7].

Non-Interference. Parallelism is introduced via `cobegin` in Lines 9 and 18; the DPJ techniques support a variety of other language constructs for nested fork-join parallelism. Each statement in a cobegin block *may* be executed in parallel in separate *tasks*, and all such tasks *must* complete before the statement after the cobegin may start. Parallelism is safe if the effects of concurrent tasks are *non-interfering*. We use the same symbol ($\#$) for non-interference as for RPL disjointness.

Definition 6 (Interference): Two effects interfere if one or both are *writes*, and their RPLs are not disjoint.

In the body of `setXY`, the two setter methods are called in parallel, so we check that the effects of the two tasks are non-interfering. Indeed `writes P:X` does not interfere with `writes P:Y` because the RPLs are disjoint.

Object Distinction. Lines 12 onward show a method `foo`, which creates and initializes two point objects in parallel. Line 14 declares object `p1` and provides region argument `Foo:R1` to the region parameter `P` of its type `Point`. Similarly, `p2` gets instantiated with region argument `Foo:R2` on line 15. The invocation of `setXY` through `p1` on line 19 induces the substitution $\sigma = [\texttt{P} \leftarrow \texttt{Foo:R1}]$ on its effect summary, which yields `writes Foo:R1:*`. Similarly, the effect of line 20 is `writes Foo:R2:*`. The checker confirms that these effects are non-interfering (distinction from the left) and confirms that they are covered by the declared effect summary of `foo`.

Recursion and the use of $\star$ in RPLs. The recursive example code in Figure 2 demonstrates a feature of the DPJ annotations that presents one of the primary challenges for the inference, namely summarizing an infinite set of regions using a star, and distinguishing between disjoint infinite sets of regions.

```
1  class TreeNode<region P> {
2   region M, L, R, Links;
3   double mass in P:M;
4   TreeNode<P:L> left  in Links;
5   TreeNode<P:R> right in Links;
6
7   double computeMass() reads Links  writes P:*:M {
8     if (left==null && right==null)// reads Links
9       return mass;                // reads P:M
10    double ml = 0, mr = 0;
11    cobegin {
12      if (left!=null)           // reads Links
13        ml=left.computeMass(); // rd Links wr P:L:*:M
14      if (right!=null)          // reads Links
15        mr=right.computeMass();// rd Links wr P:R:*:M
16    }
17    return mass = ml + mr;      // writes P:M
18  }
19 }
```

Fig. 2. Example of annotated TreeNode class (effects in comments are computed by the checker)

The `TreeNode` class has a `mass` field in `P:M` to enable parallel updates of the field in different tree nodes, and two references to child nodes. The references themselves reside in region `Links` (the references of all `TreeNode` objects will be in the same region because we do not care to distinguish them by object in this example). The `TreeNode` objects they refer to take region arguments `P:L` and `P:R`.

The method `computeMass` returns the mass of leaf nodes (Line 8-9), and it recursively computes the total mass of the two subtrees for each non-leaf node (Lines 11-16), which it also stores in the mass field (Line 17). The effects of the method are `reads Links` from reading the `left` and `right` references, and `writes P:*:M` (meaning it writes the M region of all RPLs/nodes under P) from the write on Line 17 and the recursive invocations. The effects of the recursive invocation on line 13 are computed by applying the substitution $\sigma = [P \leftarrow P:L]$ to the declared effect summary of `computeMass`, yielding `reads Links writes P:L:*:M`. Similarly, the recursive invocation on line 15 has effects `reads Links writes P:R:*:M`. These effects are covered by the effect summary of `computeMass`. The parallelism is safe because the two tasks have non-interfering effects: {`reads Links, writes P:L:*:M`} # {`reads Links, writes P:R:*:M`}. Non-interference is decided by making all the pairwise comparisons, four in this case:

`rd Links # rd Links`	`rd Links # wr P:R:*:M`
`wr P:L:*:M # rd Links`	`wr P:L:*:M # wr P:R:*:M`

III. Inference

In this section, we start by giving an overview of the phases of our annotation inference algorithm, and we informally follow these phases for the `Point` class example described in the previous section. Then, we formalize the key phases of the algorithm in the following subsections.

A. Overview

Our annotation inference proceeds in four phases. In the first phase, we generate an RPL variable ρ_v for each unannotated variable v (field, parameter, or local variable) and for each unannotated type that requires a region argument, and we generate an effect summary variable E_m for each unannotated method m. Each RPL variable is also given a *domain* of valid region names and parameters from which it can be instantiated, which we describe later. In the second phase, we scan the code of the program to produce three kinds of constraints on these variables: assignments (including passing parameters and returning values at method calls) produce subtyping constraints; method definitions produce effect summary inclusion constraints; and parallelism produces task non-interference constraints. In the third phase, we process the constraints to produce simpler constraints, which greatly prune the solution space defined by the ρ_v and E_m variables. This pruning often makes full inference tractable, as we will demonstrate in the evaluation section. In Section III-C (and the appendix), we prove that these simplification rules are sound and complete. Finally, in the fourth phase, we search for an instantiation of the variables that satisfies the constraints. We use a *structure- and value-aware* algorithm, which guides the variable instantiation order and greatly reduces the amount of backtracking required, thus improving the solver's performance.

We initially attempted to use an SMT solver for the constraint processing and solving phases (3rd and 4th) instead of implementing our own from scratch, and we experimented with Z3 because its support for recursive rules was the most mature. Unfortunately, the complex recursive decision procedures required to check RPL disjointness and inclusion, and the fact that we had to provide these rules declaratively to Z3, resulted in solving times exceeding one minute for a simple disjointness constraint. So, we concluded that we would not be able to use an existing SMT solver to solve our constraints, and we implemented our own custom solver prototype in Prolog.

```
1  class Point { // Implicit region parameter P
2     double x; // in ρx
3     double y; // in ρy
4
5     void setX(int v) { x = v; }  // Ex (summary)
6     void setY(int v) { y = v; }  // Ey (summary)
7     void setXY(int vx, int vy) { // Exy (summary)
8        cobegin { setX(vx); setY(vy); } }
9  }
10
11 void foo() { // Ef (summary)
12    Point p1 = new Point(); // Point<ρp1>
13    Point p2 = new Point(); // Point<ρp2>
14    int v1 = 3, // no annotation needed for locals
15        v2 = 4; // unless their address is taken
16    cobegin{
17       p1.setXY(v1,v2); // Exy[P ← ρp1]
18       p2.setXY(v2,v1); // Exy[P ← ρp2]
19    }
20 }
```

Fig. 3. Example Point class *without* annotations.

The rest of this subsection explains the four phases in more detail, first informally through examples, then formally.

Phase 1: Introduction of Variables. Figure 3 shows the same code as Figure 1 but without annotations. We introduce a fresh region parameter P for the type `Point`, fresh RPL variables ρ_x and ρ_y for fields `x` and `y`, and fresh effect summary variables E_x, E_y, and E_{xy} for the setter methods, as shown in the comments (their names are chosen for clarity of presentation). For `foo`, we introduce effect summary variable E_f and RPL variables ρ_{p1}, ρ_{p2} for the region arguments of `p1` and `p2`. If a solution is found where the type `Point` does not require the region parameter P (i.e., ρ_x and ρ_y do not contain P, which would also mean that E_x, E_y, and E_{xy} do not contain P), then ρ_{p1} and ρ_{p2} can be dropped as their values are not used. We can reason locally about the effects on the local variables `v1` and `v2` and the method parameters `v`, `vx`, and `vy`, as long as their address is not taken and their type does not expect a region argument. In those cases, a region argument annotation is not necessary, and we need not report effects on those variables in effect summaries. (This is a weaker form of the property of *uniqueness* [9].)

Phase 2: Constraint Generation. For method `setX`, we generate the *effect inclusion constraint* that its effect summary should cover the effects of its body: $\{\text{writes } \rho_x\} \subseteq E_x$. Correspondingly, for `setY` we generate: $\{\text{writes } \rho_y\} \subseteq E_y$. The effect summary of `setXY`, which invokes the other two setter methods, must cover the union of their effects: $E_x \cup E_y \subseteq E_{xy}$. We will somewhat abuse the notation and write instead: $\{E_x, E_y\} \subseteq E_{xy}$. Moreover, the two tasks on line 8 must be non-interfering, so we generate the *non-interference constraint*: $E_x \# E_y$.

Line 17 invokes `setXY` on point `p1`, which has region argument ρ_{p1}. We compute its effects by applying the substitution $[P \leftarrow \rho_{p1}]$ to its effect summary E_{xy}, which we denote $E_{xy}[P \leftarrow \rho_{p1}]$. Similarly, the effects of the invocation on line 18 are $E_{xy}[P \leftarrow \rho_{p2}]$. The effect inclusion constraint for `foo` is then $\{E_{xy}[P \leftarrow \rho_{p1}], E_{xy}[P \leftarrow \rho_{p2}]\} \subseteq E_f$, and the non-interference constraint is $E_{xy}[P \leftarrow \rho_{p1}] \# E_{xy}[P \leftarrow \rho_{p2}]$. Table I summarizes the generated constraints.

Phase 3: Constraint Processing. The third phase helps prune the vast instantiation space defined by RPL and effect variables by processing the generated constraints *symbolically*, without instantiating any of the RPL or effect summary variables. All

TABLE I. CONSTRAINTS GENERATED FOR CODE IN FIGURE 3

No.	Constraint	Type
1	$\{\text{writes } \rho_x\} \subseteq E_x$	Effect Inclusion
2	$\{\text{writes } \rho_y\} \subseteq E_y$	Effect Inclusion
3	$\{E_x, E_y\} \subseteq E_{xy}$	Effect Inclusion
4	$E_x \# E_y$	Effect Disjointness
5	$\{E_{xy}[P \leftarrow \rho_{p1}], E_{xy}[P \leftarrow \rho_{p2}]\} \subseteq E_f$	Effect Inclusion
6	$E_{xy}[P \leftarrow \rho_{p1}] \# E_{xy}[P \leftarrow \rho_{p2}]$	Effect Disjointness

the transformations are **sound**, in that they do not introduce solutions that would violate the original set of constraints, and **complete**, in that if the original set of constraints had a solution, so will the resulting set, though it may be smaller.

Effect Inclusion to Equality. We can replace simple effect inclusion constraints (e.g., 1 and 2 in Table I) with effect equality constraints when the left hand side does not include invocations (cf. 1 and 2 in Table II), because the effects are fully defined (except for any region variables). Conversely, we cannot *yet* perform this simplification to constraint 5 in Table I because the left hand side contains invocations. Now that we have effect equality constraints for E_x and E_y, we can substitute them into constraints 3 and 4 to get $\{\text{writes } \rho_x, \text{writes } \rho_y\} \subseteq E_{xy}$ (which we can now replace with an equality constraint) and writes $\rho_x \#$ writes ρ_y; the latter is further simplified as $\rho_x \# \rho_y$ (disjointness of regions). With E_{xy} simplified, we can substitute it in the effect constraint for E_f, resulting in a long but simple constraint (#5 in Table II).

Notice that converting an effect inclusion constraint to an equality disallows potential solutions with less precise effect summaries, but it is a *complete* transformation according to the definition we gave above.

We can also substitute E_{xy} in constraint 6 and get: $\{\text{writes } (\rho_x[P \leftarrow \rho_{p1}],\ \rho_y[P \leftarrow \rho_{p1}])\}\ \#\ \{\text{writes } (\rho_x[P \leftarrow \rho_{p2}],\ \rho_y[P \leftarrow \rho_{p2}])\}$. This decomposes into four simpler pairwise disjointness constraints shown in the table (6.1-6.4), where we have also reduced the effect non-interference constraints to region disjointness ones by dropping the *writes*.

TABLE II. SIMPLIFIED CONSTRAINTS FOR CODE IN FIGURE 3

No.	Simplified Constraint	Type
1	$\{\text{writes } \rho_x\} = E_x$	Effect Equality
2	$\{\text{writes } \rho_y\} = E_y$	Effect Equality
3	$\{\text{writes } (\rho_x, \rho_y)\} = E_{xy}$	Effect Equality
4	$\rho_x \# \rho_y$	RPL Disjointness
5	$\{\text{writes } (\rho_x[P \leftarrow \rho_{p1}], \rho_y[P \leftarrow \rho_{p1}], \rho_x[P \leftarrow \rho_{p2}], \rho_y[P \leftarrow \rho_{p2}])\} = E_f$	Effect Inclusion
6.1	$\rho_x[P \leftarrow \rho_{p1}] \# \rho_x[P \leftarrow \rho_{p2}]$	RPL Disjointness
	$\Longrightarrow \rho_x = P{:}\widetilde{\rho_x} \ \wedge\ \rho_{p1} \# \rho_{p2}$	RPLvar param. & Disj.
6.2	$\rho_x[P \leftarrow \rho_{p1}] \# \rho_y[P \leftarrow \rho_{p2}]$	RPL Disjointness
6.3	$\rho_y[P \leftarrow \rho_{p1}] \# \rho_x[P \leftarrow \rho_{p2}]$	RPL Disjointness
6.4	$\rho_y[P \leftarrow \rho_{p1}] \# \rho_y[P \leftarrow \rho_{p2}]$	RPL Disjointness
	$\Longrightarrow \rho_y = P{:}\widetilde{\rho_y} \ \wedge\ \rho_{p1} \# \rho_{p2}$	RPLvar param. & Disj.

Disjointness Under Substitution. So far, the simplifications were straightforward, but the next steps, which are more tricky, are important for performance. From constraint 6.1, we can deduce two constraints. *First*, the substitution on ρ_x must produce a change, in other words ρ_x must contain the parameter P, otherwise the constraint would evaluate to $\rho_x \# \rho_x$, which is unsatisfiable. Because in DPJ a parameter can only appear at the head of an RPL, we infer the *structural constraint* $\rho_x = P{:}\widetilde{\rho_x}$, where $\widetilde{\rho_x}$ is a fresh tail-RPL variable, which is just a regular RPL variable, except it may not contain RPL elements that can only appear at the head position. Now, we can apply the substitutions on both sides of 6.1, and we get $\rho_{p1} : \widetilde{\rho_x} \# \rho_{p2} : \widetilde{\rho_x}$. Now, we can deduce the *second*

constraint: $\rho_{p1} \# \rho_{p2}$ (the *substituents* – i.e., the right-hand sides of the substitutions – must be disjoint in order for the disjointness constraint 6.1 to be satisfiable). Note that this is a necessary but not a sufficient condition if the fresh tail-RPL variable contains a wildcard, as we will show in Section III-C where we also determine the sufficient condition. The same reasoning is applied to constraint 6.4. Finally, we can use the structural constraints for ρ_x and ρ_y, substitute them into the constraints and further simplify them (e.g., constraint 5 becomes $\{\text{writes } (\rho_{p1}:\widetilde{\rho_x},\ \rho_{p1}:\widetilde{\rho_y},\ \rho_{p2}:\widetilde{\rho_x},\ \rho_{p2}:\widetilde{\rho_y})\} = E_f$).

Occasionally, in this phase we will reduce or infer a constraint that is clearly unsatisfiable, such as $\rho \# \rho$, in which case we cannot prove the parallelism to be safe, either because we cannot check it statically or because it is indeed unsafe, and we terminate the inference. Otherwise, we move on to the next phase, which attempts to find an annotation instantiation that allows the code to be checked successfully. Section III-C (and the appendix) show that the constraint processing transformations are both sound and complete.

Phase 4: Instantiation and Checking. In this phase, we instantiate RPL variables and check that they honor the constraints generated during the previous two phases. The naive approach is to first instantiate all RPL variables (in some arbitrary order) and then to check if they satisfy all the constraints, backtracking if they don't. This approach is not tractable because the space is too big, and a bad *instantiation order* will incur exponential amounts of unnecessary backtracking. Say you have a constraint A with variables ρ_1, ρ_2 and a constraint B with variables ρ_2, ρ_3, ρ_4, and the instantiation order is lexicographic. If constraint A is not satisfied, we will have to backtrack across all values of ρ_3, ρ_4 (which are not going to make A satisfiable), before trying a new value for ρ_2 which may make A satisfiable. The obvious solution is to check a constraint as soon as all its RPL variables become instantiated to trigger backtracking, but this approach is still too sensitive to the variable instantiation order and incurs too much overhead.

Our approach is a **structure and value sensitive** extension of the above idea. The constraints boil down to RPL inclusion or disjointness and have the form $\mathcal{R}_1\vec{\sigma_1} \bigoplus \mathcal{R}_2\vec{\sigma_2}$, where $\vec{\sigma}$ is a vector of substitutions and $\bigoplus$ stands for either the inclusion or disjointness operator. We use this **structure** to drive the instantiation process. First, we instantiate the left hand side. We start by instantiating any variables in $\mathcal{R}_1$, but before we proceed to instantiate $\vec{\sigma_1}$, we apply the substitution: if the instantiation **value** of $\mathcal{R}_1$ was not parametric, the entire substitution vector has no effect and we don't need to instantiate any of its variables. Then, we instantiate the right hand side using the same approach, and once both sides are instantiated we check if the constraint is satisfied.

Because RPL variables appear in multiple constraints, the above approach is still too sensitive to the instantiation order. To trigger backtracking earlier, we use an ***unsatisfiability test***: each time we instantiate an RPL variable ρ, we check for unsatifiability all constraints in which it appears: for each partially instantiated constraint independently, we try to find an instantiation of its uninstantiated variables that satisfies it, given the values of the instantiated variables; if such an instantiation does not exist, the value we used to instantiate ρ caused that constraint to become unsatisfiable, and backtracking is triggered. These checks are expensive but a missed opportunity for early backtracking can have an exponential cost.

To pick an instantiation for an RPL variable, we must define a *domain* for each variable. Such a domain lists all the region names and parameters that may be used to instantiate the corresponding RPL variable. Since we do not expect programmers to provide region name declarations, we must also implicitly provide those in order to instantiate the domains of RPL variables. A naive approach is to declare as many region names as there are RPL variables and allow all RPL variables to be instantiated from that global pool, but this approach allows too many isomorphic solutions (solutions that can be converted from one to another via alpha renaming). We use scoping and DPJ-specific knowledge to constrain the domains in ways that limit isomorphic instantiations, as described in the appendix Section VII-I.

One noteworthy choice was to restrict the instantiation length of RPL variables to two elements because we have not encountered the need for longer region arguments. This means that the size of the instantiation space of an RPL variable (which we call its *cardinality*) is roughly quadratic in the size of its domain. RPL arguments with length greater than two could be used to hide implementation details at API boundaries, but the constraint generation relies on having access to the code, so the desire of hiding the code will come with a need to manually annotate such module APIs. The annotations on the internals of such modules can still be automatically inferred. Finally, the restriction of the instantiation length does not carry to any other parts of the inference algorithm, which can work with RPLs of arbitrary length. Such longer RPLs can occur either from manual annotations, or due to substitutions applied to shorter RPLs (e.g., $P:L:\star:M$ in Fig.2).

The first and fourth phases (variable introduction and annotation instantiation) are relatively straightforward, so in the following sections, we focus on the constraint generation and constraint processing phases.

B. Constraint Generation

This section describes the constraint generation phase of our inference algorithm. For conciseness, we focus on a core calculus (Figure 4) in the style of Huang et al. [10], which models Java with a syntax in A-normal form [11], [12].

	Definition	**Meaning**
p ::=	$\overline{\text{cd}}\ \overline{\tau y}$ s	*program*
cd ::=	class C<region P> $\{\overline{fd}\ \overline{md}\}$	*class decl.*
fd ::=	τ f in $\mathring{\mathcal{R}}$	*field decl.*
md ::=	τ_r m(τ_x x) E $\{\overline{\tau y}$ s; return $y\}$	*method decl.*
τ ::=	C<$\mathcal{R}$>	*types*
$\mathcal{R}$ ::=	R | ρ	*RPLs*
R ::=	Root | P | R:r | R:*	*RPL annot.*
E ::=	$\emptyset$ | rd $\mathcal{R}$ | wr $\mathcal{R}$ | ϵ | E $\cup$ E	*Effects*
s ::=	s;s | s‖s | x = new τ() | x = y	*Statements*
	| x.f = y | this.f = y | x = y.f	
	| x = this.f | x = y.m(z) | x = this.m(z)	

Fig. 4. Syntax of a core OO language.

Programs in our language comprise class declarations, followed by global variable declarations and a statement to execute. Class declarations specify a class name C, a region parameter P, a list of field declarations and a list of method declarations. Field declarations have a type τ, a field name f, and a fully specified region $\mathring{\mathcal{R}}$ wherein the reference is stored. Types τ are composed of a name C and of a region argument $\mathcal{R}$. Method declarations are composed of a return type τ_r, a method name m, a formal parameter x with type τ_x, an effect summary E and a body. The body has a

list of local variable declarations $\overline{\tau y}$, a statement s, and a return statement. RPL arguments $\mathcal{R}$ are either RPL variables ρ introduced in phase 1 or RPL annotations given by the programmer (R). RPL annotations R are a colon-separated list of RPL elements that start either with the region `Root` or a parameter P, and may contain region names (r) and the star wildcard. An effect summary E is a possibly empty set of read (rd) or write (wr) effects on RPLs $\mathcal{R}$, or an effect summary variable ϵ, or a union of effect summaries. For brevity, we will write $\{\text{rd } \mathcal{R}_1, \mathcal{R}_2;\ \text{wr } \mathcal{R}_3, \mathcal{R}_4\}$ instead of $\{\text{rd } \mathcal{R}_1, \text{rd } \mathcal{R}_2, \text{wr } \mathcal{R}_3, \text{wr } \mathcal{R}_4\}$. A statement can be composed of two statements either sequentially (;) or in parallel ($\|$); or it can be an assignment statement, assigning a variety of right-hand-side (RHS) expressions to a variable or to a field of the current object (this.f) or of another object (y.f). The RHS expressions may be a heap allocation with a type argument, a variable, a field dereference (this.f or y.f), or a method call (this.m(z) or y.m(z)).

Subtyping for the explicitly or implicitly (i.e., with RPL variables introduced in phase 1) region-annotated types is equivalent to region inclusion of their RPL arguments:

$$\frac{\Gamma \vdash \mathcal{R}_1 \subseteq \mathcal{R}_2}{\Gamma \vdash C\langle \mathcal{R}_1 \rangle \leq C\langle \mathcal{R}_2 \rangle} \qquad \text{(SubTyping)}$$

The environment Γ is a set of variable-type bindings (τx) and parameter inclusion facts ($P \subseteq \mathcal{R}$) used by the capture rule, explained later in this section. The typing relation for statements $\Gamma \vdash s : \tau, E \mid K$ reads "statement s has type τ and effects E in environment Γ under the constraints K.

Next, we discuss the constraint generation as driven by the typing relation for the most interesting categories of statements and declarations. Due to lack of space, we omit rules that are similar to the ones we present.

For sequential composition, the type is that of the second statement, the effects are the union of the effects of the two statements, and the constraints are the union of the constraints. Parallel composition is similar, with only an additional non-interference constraint $\{E_1 \# E_2\}$.

$$\frac{\Gamma \vdash s_1 : \tau_1, E_1 \mid K_1 \qquad \Gamma \vdash s_2 : \tau_2, E_2 \mid K_2}{\Gamma \vdash s_1; s_2 : \tau_2, E_1 \cup E_2 \mid K_1 \cup K_2} \quad \text{(Seq)}$$

$$\frac{\Gamma \vdash s_1 : \tau_1, E_1 \mid K_1 \qquad \Gamma \vdash s_2 : \tau_2, E_2 \mid K_2}{\Gamma \vdash s_1 \| s_2 : \tau_2, E_1 \cup E_2 \mid K_1 \cup K_2 \cup \{\Gamma \vdash E_1 \# E_2\}} \quad \text{(Par)}$$

For a simple assignment (x=y), the class of x and y must be the same in the environment, and the region argument of y must be included in that of x. This is a straightforward application of the subtyping rule towards constraint generation. The rule for a statement allocating a new object is very similar, so we omit it. The effects on variables are not expressed in terms of RPLs because in the core language variables are *unique* (i.e., they cannot be aliased because their address cannot be taken). Thus, variables do not require an *in* clause, and effects on them simply use the variable name.

$$\frac{C\langle \mathcal{R}_x \rangle x \in \Gamma \qquad C\langle \mathcal{R}_y \rangle y \in \Gamma \qquad K = \{\Gamma \vdash \mathcal{R}_y \subseteq \mathcal{R}_x\}}{\Gamma \vdash x = y : C\langle \mathcal{R}_y \rangle, \{\text{rd } y; \text{wr } x\} \mid K} \quad \text{(Assign)}$$

Next, we discuss the constraints for aliasing a variable `x` to a field `f` accessed through variable `y` (`x=y.f`). The auxiliary function typeof() retrieves the type of field `f` from its declaration, and paramof() retrieves the region parameter P_y from the declaration of class C_y. `x` and `f` must have the same class `C`, substitution σ replaces the parameter P_y with the region argument $\mathcal{R}_y$ of `y`, and we apply it to the region $\mathring{\mathcal{R}}_f$ in which `f` lives. The type of the statement is $C\langle \mathcal{R}_f \rangle \sigma$, i.e., the type of the field after applying the substitution. Its effects are (i) reading the region in which `f` lives, after applying σ, (ii) reading y, and (iii) writing x. Finally, the generated region inclusion constraint requires that the region argument of the right hand side (`y.f`), which we get by applying σ to the region argument $\mathcal{R}_f$ of the type of `f`, be included in the region argument of `x`. We skip the simpler rule for `x=this.f`, as it follows from this one by omitting the substitution σ.

$$\frac{\begin{array}{c} C\langle \mathcal{R}_x \rangle x \in \Gamma \qquad \text{typeof}(f) = C\langle \mathcal{R}_f \rangle \text{ in } \mathring{\mathcal{R}}_f \\ C_y\langle \mathcal{R}_y \rangle y \in \Gamma \qquad \text{paramof}(C_y) = P_y \\ \sigma = [P_y \leftarrow \mathcal{R}_y] \qquad K = \{\Gamma \vdash \mathring{\mathcal{R}}_f \sigma \subseteq \mathcal{R}_x\} \end{array}}{\Gamma \vdash x = y.f : C\langle \mathcal{R}_f \rangle \sigma, \{\text{rd } y, \mathring{\mathcal{R}}_f \sigma; \text{wr } x\} \mid K} \quad \text{(ReadVField)}$$

The next rule assigns to a field through a variable (x.f=y). The substitution σ replacing the parameter of the type C_x of `x` with its region argument $\mathcal{R}_x$ is what one might expect, and it is used to determine the effect of the statement. We must use a different substitution σ', however, to check the legality of the assignment. In σ', we substitute the parameter P_x of C_x with a fresh parameter P, called the *capture parameter*, which can take values in $\mathcal{R}_x$, i.e., $P \subseteq \mathcal{R}_x$, and we require that $\mathcal{R}_y \subseteq \mathcal{R}_f \sigma'$ in the augmented environment that includes the fresh parameter P with its constraint. (This complication with the capture parameter is needed in this rule to avoid unsoundness in the presence of wildcards, as explained in more detail in [8]. It is similar to how Java handles the capture of a generic wildcard.) If $\mathcal{R}_x$ is fully specified, then $P = \mathcal{R}_x$ and σ' becomes equivalent to σ. Otherwise, the capture parameter tells us that the only information we have about the region argument of x is that it is included in $\mathcal{R}_x$.

$$\frac{\begin{array}{c} C_x\langle \mathcal{R}_x \rangle x \in \Gamma \qquad \text{typeof}(f) = C\langle \mathcal{R}_f \rangle f \text{ in } \mathring{\mathcal{R}}_f \\ C\langle \mathcal{R}_y \rangle y \in \Gamma \qquad \text{paramof}(C_x) = P_x \\ \sigma = [P_x \leftarrow \mathcal{R}_x] \qquad \sigma' = [P_x \leftarrow P], \text{ fresh } P,\ P \subseteq \mathcal{R}_x \\ K = \{\Gamma \cup \{P \subseteq \mathcal{R}_x\} \vdash \mathcal{R}_y \subseteq \mathcal{R}_f \sigma'\} \end{array}}{\Gamma \vdash x.f = y : C\langle \mathcal{R}_y \rangle, \{\text{rd } y; \text{wr } x, \mathring{\mathcal{R}}_f \sigma\} \mid K} \quad \text{(WriteVField)}$$

To type a method declaration, we type its body s in the augmented environment Γ' which includes the declarations of its formal parameter and local variables. Also, the effects E of the body must be covered by the method's effect summary E_m. Finally, the return statement generates the same constraint K' as an assignment would.

$$\frac{\begin{array}{cc} C\langle \mathcal{R}_y \rangle y \in \overline{\tau y} & \tau_r = C\langle \mathcal{R}_r \rangle \\ \Gamma' = \Gamma \cup \overline{\tau y} \cup \tau_x x & \Gamma' \vdash s : \tau, E \mid K \\ K' = \{\Gamma' \vdash \mathcal{R}_y \subseteq \mathcal{R}_r\} & K'' = \{\Gamma \vdash E \subseteq E_m\} \end{array}}{\Gamma \vdash \tau_r m(\tau_x x) E_m \{\overline{\tau y}\ s; \text{return } y\} : \emptyset, \emptyset \mid K \cup K' \cup K''} \quad \text{(Method)}$$

Lastly, we give the rule for a method call through a variable (`y.m(v)`) without explanation, as it is built up from the same principles we have discussed so far (including the use of a capture parameter P in the substitution applied to the region of argument x). The simpler rule for member method call (`this.m(v)`) is omitted.

$$\frac{\begin{array}{c} C\langle\mathcal{R}_z\rangle z \in \Gamma \qquad \text{typeof}(m) = \tau_r\ m(\tau_x x) E_m\{\ldots\} \\ \tau_r = C\langle\mathcal{R}_r\rangle \qquad C'\langle\mathcal{R}_v\rangle v \in \Gamma \qquad \tau_x = C'\langle\mathcal{R}_x\rangle \\ C_y\langle\mathcal{R}_y\rangle y \in \Gamma \qquad \text{paramof}(C_y) = P_y \qquad \sigma = [P_y \leftarrow \mathcal{R}_y] \\ K_1 = \{\Gamma \vdash \mathcal{R}_r \subseteq \mathcal{R}_z\} \qquad \sigma' = [P_y \leftarrow P],\ \text{fresh } P \\ K_2 = \{\Gamma \cup \{P \subseteq \mathcal{R}_y\} \vdash \mathcal{R}_v \subseteq \mathcal{R}_x\sigma'\} \end{array}}{\Gamma \vdash z = y.m(v) : \tau_r\sigma, \{\text{rd } v, y; \text{wr } z\} \cup E_m\sigma \mid K_1 \cup K_2} \text{(VCALL)}$$

C. Constraint Processing

In this section, we present two lemmas and two theorems that form the cornerstone of the effect processing phase because they demonstrate that the key constraint simplifications performed by this phase are sound (no invalid solutions are introduced) and complete (if the original constraint set was satisfiable, so is the simplified one).

Lemma 1 (Effect Inclusion to Equality): An effect inclusion constraint $\{E_1, \ldots, E_n\} \subseteq ES$ can be replaced by an equality constraint $\{E_1, \ldots, E_n\} = ES$ if none of the effects $E_1, \ldots, E_n$ are invocation effects, and ES does not appear on the right-hand-side of any other effect inclusion constraint.

The goal of Lemma 1 is to replace an inclusion constraint with an equality constraint which is simpler and will allow substituting the effect variable wherever it appears in other constraints. This transformation is trivially sound. Intuitively, it is also complete because it simply restricts inferred effect summaries to be as precise as possible. See Section VII-F for the proof.

Lemma 2 (RPL Inclusion Chain): If $\rho_1 \subseteq \rho_2$, $\rho_2 \subseteq \rho_3$, $\ldots$, $\rho_{n-1} \subseteq \rho_n$, and $\rho_2, \ldots, \rho_{n-1}$ do not appear in any other RPL inclusion constraints, then inclusion is replaced by equality for all but the last RPL inclusion constraint: $\rho_i = \rho_{i+1}$, for $1 \leq i \leq n-2$.

Again, soundness is trivial and the same intuitive argument for completeness holds as for Lemma 1: we are only constraining potential solutions that would be less precise. The fact that RPL variable ρ_2 does not appear in other RPL inclusion constraints means it does not have to include any regions other than those of ρ_1. Conversely, if ρ_2 appeared in another RPL inclusion constraint, it could be necessary for $\rho_2 \neq \rho_1$, which is why the lemma is not applicable in that case.

The next two theorems target two more sophisticated sets of simplifications that arise for common parallelism patterns. The first theorem infers the need (and the associated constraints) for *object distinction*, which is necessary when we need to modify multiple objects of the same type in parallel. The second theorem infers the need for *recursive object distinction*, which is necessary when we need to modify multiple *nested* objects of the same type *recursively* in parallel.

The first theorem helps us deal with constraints of the form $\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]$. We saw this kind of constraint earlier and presented intuitively how to process it. Here we formalize our approach.

Theorem 1 (Disjointness Under Substitution):

$$\begin{array}{ll} \rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2] \iff \{\rho = P{:}\widetilde{\rho}, \text{fresh } \widetilde{\rho}\,\} \wedge & \\ (\,(\rho_1 \#_R \rho_2 \wedge \widetilde{\rho} \in \mathring{\mathcal{R}}) & (a) \\ \quad \vee (\rho_1 \#_L \rho_2 \wedge ((\rho_1 \not\leq \rho_2 \wedge \rho_2 \not\leq \rho_1) & (b) \\ \qquad \vee (\widetilde{\rho} \text{ does not start with wildcard})))) & (c) \end{array}$$

A disjointness constraint on a common RPL ρ with two different substitutions with the same base P is satisfied if and only if ρ starts with the substitution base P, and at least one of the three following cases holds: (a) the substituents ρ_1 and ρ_2 are distinct from the right and $\widetilde{\rho}$ is fully-specified, or (b,c) the substituents are distinct from the left and (b) neither substituent is under the other, or (c) $\widetilde{\rho}$ does not start with a wildcard.

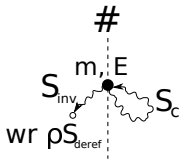
The formal proof is in the appendix Section VII-G. Here, we will give some intuition for why the result holds. First, unless ρ is parametric in P, the substitutions will have no effect and the non-interference constraint will not be satisfiable. Similarly, $\rho_1 \# \rho_2$ is necessary because these are the only RPL elements that will differ after the two substitutions. It is not, however, a sufficient condition, primarily because the rest of ρ may contain wildcards. For example, if $\rho = P{:}\star$, $\rho_1 = R_1$, and $\rho_2 = R_1{:}R_2$, then $\rho_1 \# \rho_2$ is true but $R_1{:}\star \# R_1{:}R_2{:}\star$ is not satisfiable because $R_1{:}R_2{:}\star \subseteq R_1{:}\star$.

The three cases (a-c) in the theorem structurally decompose all the ways in which $\rho_1 \# \rho_2$ can be used to prove that $\rho_1 : \widetilde{\rho} \# \rho_2 : \widetilde{\rho}$. (a) says if ρ_1 and ρ_2 are disjoint from the right, then the RPL elements following them ($\widetilde{\rho}$) must not contain any wildcards in order to prove the required property. If ρ_1 and ρ_2 are disjoint from the left, i.e., due to leading RPL terms, we can prove the resulting RPLs disjoint in one of two ways. (b) says we can prove it (regardless of $\widetilde{\rho}$) if ρ_1 and ρ_2 are in distinct subtrees of the region tree (i.e., neither is an ancestor of the other). (c) says we can also prove it if $\widetilde{\rho}$ is a subtree with a unique root because then appending that subtree to ρ_1 and ρ_2, which are distinct from the left, preserves their disjointness.

For simplicity, because constraints with disjunctions are expensive to check, our implementation of the constraint processing phase leverages this theorem only partially: it *adds* the simpler constraints $\rho = P{:}\widetilde{\rho}$ and $\rho_1 \# \rho_2$ to the original disjointness constraint. The first constrains the instantiation of ρ improving performance, and the second provides a necessary condition for disjointness which is quicker to check.

The second theorem deals with recursive write effects. Let m be a method with effect summary variable E that has some non-interference constraint (cf. Figure 5), and a (directly or indirectly) recursive method call, represented by a sequence of substitutions S_c (i.e., a *substitution cycle*). Effect inclusion for the body of m yields Eq. 1. Furthermore, let m include a `writes` effect, writes ρS_{deref}, possibly through a chain of invocations from m that contributes a substitution chain S_{inv} (if the statement contributing this writes effect is in m then S_{inv} will be empty), and let $S = S_{deref} S_{inv}$ (Eq. 2). Let the writes effect and the substitution cycle S_c be on opposing sides of a non-interference constraint (Eq. 3). Finally, we assume that the code compiles without errors, which allows us to claim that the substitutions S and S_c are *well-formed*, meaning they properly translate the RPL ρ into the context of m.

Theorem 2 (Non-interference of Recursive Writes):



$$ES_c \subseteq E \quad (1)$$
$$\text{writes } \rho S \subseteq E \quad (2)$$
$$\text{writes } \rho S \# ES_c \quad (3)$$

Fig. 5. Recursive Write Theorem

The premises Eq. 1–3 imply:

$$\rho = P' : \widetilde{\rho} \quad (4)$$
$$S = [P' \leftarrow P : \widetilde{s}] \quad (5)$$
$$S_c = [P \leftarrow P : \widetilde{s_c}] \quad (6)$$
$$\widetilde{s_c} \text{ isn't empty and doesn't start with } \star \quad (7)$$
$$\bigcup_{i=0}^{\infty} \text{writes } P : \widetilde{s_c}^{\,i} : \widetilde{s} : \widetilde{\rho} \subseteq E \quad (8)$$

Again, the theorem's transformation is sound and complete. For each write effect directly or indirectly within the body of m, described by Eq. 2, the inferred constraints of the theorem are *added* to the original set of constraints, trivially preserving soundness. The proof of the theorem guarantees completeness because the premises imply the conclusion (cf. appendix Section VII-H).

Adding these inferred constraints helps the solver by reducing the instantiation domain of several RPL variables and of substitutions that contain RPL variables: Eqs. 4, 5, and 6 require that the relevant RPL variables start with a specific parameter, and Eq. 7 puts some additional constraints on the instantiation of the tail of the RPL variable.

We briefly discuss why Theorem 2 holds. First, let the region parameter in the context where the write effect occurs be P', and the one in the context of m be P (possibly the same as P'). Then, Eq. 4 (i.e., ρ is parametric in P') is necessary to make the non-interference constraint in Eq. 3 satisfiable. This is because ρS contributes a term to E, and if the result did not include a region parameter, that term would be left unchanged by S_c, so Eq. 3 would be unsatisfiable.

Note that in Eqs. 5 and 6 the substitution chains S and S_c are shown as a single substitution. This is because well formed substitution chains can be simplified into a single substitution (cf. appendix). Eq. 5 follows from the well-formedness hypothesis. Eq. 6 says that the substitution cycle S_c can be simplified into a substitution which has the same parameter P both at its base and at the head of its substituent. In other words, S_c preserves the parameter P – it does not ground the effects it is applied to to fixed regions, which would then make the non-interference constraint unsatisfiable. Eq. 7 (in conjunction with Eq. 6) says that the substitution cycle S_c is *expanding* (i.e., s_c is not empty), meaning one or more region names are added to RPLs after S_c is applied, allowing the effects of an invocation of m and those of a recursive invocation of m to be over disjoint regions. The second requirement, that s_c not start with $\star$, stems from a technical detail of the DPJ annotations, whose justification can be found in the appendix Section VII-H.

Eq. 8 follows from Eqs. 1, 2, 4, 5, and 6 as follows. Eq. 1 and 2 yield $\bigcup_{i=0}^{\infty} \text{writes } \rho S S_c^i \subseteq E$, where S_c^i denotes i applications of S_c. Using Eq. 4, 5, and 6 to substitute the values of S, S_c, and ρ we get Eq.8. We can summarize this infinite set of regions with some loss of precision as $P : * : \widetilde{s} : \widetilde{\rho}$, but under certain conditions (adversarial partial annotations) this can be incomplete. It has not been an issue in practice, but in the future we plan to avoid this incomplete summarization by allowing the constraint solving engine to reason about RPLs of the form $P : \widetilde{s_c}^* : \widetilde{s} : \widetilde{\rho}$ which would precisely describe such infinite sets of regions.

IV. Implementation & Evaluation

We implemented the first two phases [13] (RPL and effect summary variable insertion and constraint generation) on top of our annotation checker tool written in Clang [14]. Our target language is a *type-safe* subset of C++11 with parallelism introduced via TBB [15] constructs and, in certain cases, a few manual annotations provided as attributes. We implemented the other two phases [16] (constraint processing and annotation instantiation) in Prolog because of its native support for unification and its declarative programming style, which made it extremely productive.

A. Benchmarks

We evaluated the feasibility and performance of our inference tool on a number of parallel benchmarks [16]. They include a variety of algorithms and parallelism styles, including computations on trees, recursive subdivision of arrays, and a parallel loop algorithm with a synchronized reduction phase. The parallel computations are written so that they can be proven deterministic using DPJ-style region and effect annotations.

List updates all the elements of a linked list and performs the update on the current node in parallel with the recursive call to update the rest of the list.

Tree builds a binary tree of a given depth: at each node, the two subtrees are built recursively in parallel.

MergeSort uses recursive parallelism, sorting the two halves of an array in parallel with each other before merging them. This algorithm allocates a new output array at each recursive step. This allows us to place the output arrays of each step in nested regions corresponding to the recursive tree structure of the computation, so the effects of each recursive step can be specified using an RPL containing the $\star$ wildcard (writes $\{Pout : A : \star, Pout : B : \star\}$, where A and B specify the left and right half of the array at each step).

KMeans is a clustering algorithm adapted from the STAMP benchmarks. It identifies k clusters from a set of points in N-dimensional space. It uses a parallel loop with a reduction step where information about each cluster may be updated. We use a manually-inserted block-scoped lock for this step, because our algorithm does not currently infer atomic blocks and atomic effects, which would be used by DPJ for this code. Our analysis assumes without checking that the locked block correctly synchronizes the operations in it, so its effects can be ignored. The remaining operations within the parallel loop are read-only, so the data they access can all be placed in a single region that is read by each iteration of the loop.

QuadTree builds a spatial partitioning tree where each node represents a region of a 2-D plane and internal nodes each have four child nodes corresponding to sub-quadrants of the parent node. This data structure has many applications such as collision detection, spatial indexing, image analysis, and simulation. 2-D points are inserted in the quad tree by starting from the root node and recursively moving down the tree by checking which quadrant the point belongs to. If we reach a

leaf node, we insert the point there if the node is not full; otherwise, we subdivide the node into four quadrants and recursively repeat the process. We parallelize the recursive computation by attempting to insert the point in each of a node's four subtrees in parallel; this will check the point's location against each subtree's bounding box in parallel and only proceed with the insertion in the appropriate subtree.

CollisionTree implements a tree-based collision detection algorithm which was adapted from the open-source jMonkeyEngine game engine. We ported it from a version previously used to benchmark DPJ [5]. The Collision Tree algorithm processes a pair of spatial partitioning trees, each containing the triangles in a 3-D mesh, and finds any pairs of triangles in the two trees that intersect. It uses a recursive parallel approach, processing the two subtrees of one of the trees in parallel at each recursive step. This algorithm is read-only on the trees, but we want it to work with trees that could be built in parallel, so we kept manual annotations for the `left` and `right` pointers of the `TreeNode` class, similar to those shown in Figure 2. The inferred effects of each recursive step in the algorithm will therefore contain the $\star$ wildcard to designate effects on each subtree it is processing. This is an example of the inference working with partial manual annotations.

B. Evaluation

We evaluate the feasibility of our proposed inference algorithm and the performance improvements achieved by our simplification phase (Phase 3). To do so we measure the time-to-solution with different levels of simplification enabled as well as the size of the instantiation space of RPL variables.

Table III(a) shows how long it takes the constraint solver (in seconds, except where noted) to find a solution given different levels of simplification. The first column states the name of the program along with the number of lines of code as counted by `sloccount` [17]. The second column shows the time to solution with phase 3 disabled (only Lemma 1 is applied because it is so fundamental), the third column shows the time when Lemma 2 is enabled, the fourth shows the time when Theorem 1 is also enabled, and the fifth when Theorem 2 is also enabled (cf. Section III-C). Runs that took more than 24 hours are marked as T/O (time-out). We ran the solver on an Intel Core i7-2600, with 8G of RAM, running a 64 bit Ubuntu 10.14 using kernel 3.16.0-37-generic. The times reported are the average of five runs. The standard deviation of these runs was below 10%, or below 20% for the cases marked with an asterisk. This somewhat higher than expected variability seems tied to Prolog's runtime system.

The size of the instantiation space is computed as the product of the *cardinalities* of the RPL variables *reachable* by the union of all the non-interference constraints. An RPL variable v is reachable by a constraint C if it is included in C or if it is included in a constraint C' that contains an RPL variable v' that is reachable by C. Table III(b) shows the instantiation space at various stages before, during, and after the constraint processing phase. Notice that the simplifications significantly reduce the space and that different benchmarks benefit more or less from a specific simplification depending on the parallelism pattern they use. For example, tree based code reaps increased benefits from the simplifications of Theorem 2.

We make the following observations from the timing and space-size numbers: *(i)* the inference of region and effect annotations is tractable for these programs; *(ii)* in half of the cases, at least some of the simplifications are necessary to make the inference tractable, although Theorem 1 does not seem helpful on this set of benchmarks by itself. *(iii)* there is not a direct correlation between the time-to-completion and the space size (e.g., KMeans faster than Quadtree); *(iv)* even when the simplifications do not improve solving time, their overhead is minimal so it is reasonable to always perform them (cf. KMeans). For KMeans, because we currently trust that the atomic block is safe and the remaining parallelism only has read effects, finding a solution for read-only parallelism is very easy, as shown by the very quick time-to-solution; *(v)* Theorem 1 does not seem to improve time-to-solution on our set of benchmarks, which may be because it would mainly benefit flat

Finally, Table III shows the counts of the region arguments and effect summaries inferred automatically (c) and of the region parameter and argument annotations provided manually (d). Because our current implementation has some limitations, some manual annotations are needed. *(1)* If a method requires region parameters, they need to be manually provided and any region arguments using those parameters also need to be provided. This case will be easy to automate. Take the following annotated example from MergeSort:

```
void sort<Pin,Pout>(int*<Pin> in, int*<Pout> out)
```

Its formal parameters `in` and `out` have type pointer-to-int, which requires a region argument to describe which region the pointer points to. To enable `sort` to take arguments that point to various regions, we need to declare region parameters `Pin` and `Pout`, and use them as region arguments in the declarations of its formals. While this is currently not automated, the way to achieve it would be to count the number of RPL variables generated for the formals of a function, then create an equal number of fresh region parameters for the function and include them in the domains of the RPL variables we counted. Moreover, at each call-site, we need to generate a set of substitutions giving concrete values to those function region parameters. Currently, we require region arguments to enable the annotation checker to infer these substitutions, but we could lift that restriction by generating a substitution $[P \leftarrow \rho]$ with a fresh RPL variable ρ for each parameter per call-site. This is straightforward but tedious, so we opted for incorporating more benchmarks into our evaluation at the cost of adding some straightforward manual annotations. *(2)* We currently assume that each non-annotated user-defined type takes (at most) one parameter. This assumption is true for all types in our benchmarks except for functors implemented as classes with an overloaded `operator()` method, which are used by TBB parallel constructs. These are treated like methods with multiple parameters as in case (1) above, so it is equally straightforward to manually annotate them.

Overall, we have shown that our inference approach is successfully able to infer the complex regions and effects annotations for non-trivial parallel algorithms. In the future, we envisage three improvements. First, when the parallelism is unsafe (and thus the constraints are unsatisfiable), the solver takes too long to fail: procedures and heuristics must be devised to quickly detect an *unsatisfiable core*. Second, the algorithm must be extended to remove the need to provide manual annotations. In particular, lifting the assumption that unannotated types have a single region parameter without paying a steep performance penalty could be challenging. Third, there are several optimizations for constraint solving

TABLE III. EVALUATION RESULTS FOR BENCHMARKS.

	(a) Time-to-Solution (s)				(b) Instantiation Space Size				(c) Inferred		(d) Provided	
Program (LOC)	**None**	**Lem 2**	**Lem 2 & Thm 1**	**Lem 2 & Thms 1&2**	**Initial**	**Lem 2**	**Lem 2 & Thm 1**	**Lem 2 & Thms 1&2**	**#arg**	**#ES**	**#par**	**#arg**
List (64)	0.046	0.046	0.045	0.046	$1.1 \cdot 10^{13}$	$6.5 \cdot 10^{10}$	$6.5 \cdot 10^{10}$	$2.5 \cdot 10^{9}$	19	5	0	0
Tree (92)	0.140	0.112	0.117	0.115	$1.0 \cdot 10^{18}$	$2.1 \cdot 10^{13}$	$2.1 \cdot 10^{13}$	$1.7 \cdot 10^{11}$	31	9	0	0
MergeSort (60)	579.0	0.482	0.494	0.227	$8.5 \cdot 10^{17}$	$1.13 \cdot 10^{15}$	$8.5 \cdot 10^{14}$	$2.9 \cdot 10^{14}$	24	4	2	2
K-means (1063)	0.444	0.469	0.476	0.479	$5.2 \cdot 10^{87}$	$1.2 \cdot 10^{78}$	$1.2 \cdot 10^{78}$	$1.2 \cdot 10^{78}$	82	28	0	1
QuadTree (98)	T/O	3h52m*	3h58m*	42.9	$7.7 \cdot 10^{40}$	$1.2 \cdot 10^{39}$	$3.1 \cdot 10^{37}$	$1.1 \cdot 10^{36}$	42	12	6	6
CollTree (836)	T/O	112.0*	115.8*	98.0	$5.2 \cdot 10^{141}$	$9.8 \cdot 10^{120}$	$8.4 \cdot 10^{120}$	$8.3 \cdot 10^{115}$	125	32	32	66

algorithms that make a big difference to the speed and scalability of compiler alias analysis, taking algorithms like Andersen's [18] from unscalable to practical and widely used [19], [20], [21]. Although they will require extensive adaptations, these optimizations could prove beneficial to our algorithm. These are all exciting future directions.

V. RELATED WORK

a) Region Inference for Memory Management: There is substantial work on inference of regions for automatic memory management [22], [23], [24]. This body of work infers regions that group heap objects with similar lifetimes into common regions. The techniques used in those papers are not applicable to our problem because they are based on analysis of object lifetimes and not on analysis of effects or of non-interference between parallel tasks.

b) Languages and Systems for Safe Parallelism: Numerous systems use some form of static and/or dynamic checks to ensure determinism or other parallel safety properties (e.g., [25], [2], [26], [3], [27], [5], [4], [28], [9], [29]). Many of these require the programmer to write some form of annotations or to use special libraries to obtain safety guarantees, but only a few support automatic inference of these annotations; we discuss them below.

Systems like CoreDet [4] and Kendo [27] use dynamic mechanisms to provide deterministic execution for threaded code without special annotations, but they impose substantial run-time overheads. Also, the deterministic execution order provided by these systems does not correspond to the structure of the program's code, making it difficult to reason about.

c) Automatic Effect Inference: Bierman and Parkinson [30] present an inference algorithm for Greenhouse and Boyland's object-oriented effect system [31]. This system is designed to enable more aggressive compiler optimizations, not parallel safety, and has only limited hierarchical regions, e.g., they cannot express arbitrarily nested structures.

The work on inferring the side-effects of procedure calls in imperative languages [32], [33], [34], [35], [36] uses interprocedural dataflow analysis to propagate side-effect information over the call graph. All these techniques rely on the results of a pointer alias analysis, whereas we *infer* (at least, indirectly) the pointer aliases in a program as part of the algorithm through the subset constraints at assignments and function calls. (Technically, our algorithm is comparable to a flow- and context-insensitive, Andersen-style subset-based alias analysis [18] in terms of analysis power.)

Jouvelot and Gifford [37], Talpin and Jouvelot [38], and Tofte and Birkedal [39] all infer types and effects for mostly-functional languages with much simpler effect systems than ours: they cannot express effects on nested region structures, which are fundamental to many realistic parallel algorithms and significantly complicate the inference problem we address.

The work of Vakilian et al. [7] was discussed in Section I.

Immutability and purity type systems statically reason about the mutability of references or objects and purity of methods [40], [41], which can help to achieve safe parallelism. Several systems can infer mutability [42], [40], [43], and Pearce [41] describes a tool to infer pure methods. Immutability and purity are simple enough to be expressed using a few kinds of annotations. However, our annotations for expressing safe parallelism are nested and more complicated, which makes the inference more challenging.

d) Inference of Annotations for Safe Parallelism: SharC [26] infers type qualifiers indicating the sharing mode of objects using a whole-program sharing analysis that propagates information about which data may be shared between threads. SharC can guarantee data-race freedom but not determinism, so their annotations are much less expressive than the region system in DPJ, e.g., they do not *explicitly* describe aliasing (region types) or effects within hierarchical data structures.

Gordon et al. [9] propose an extension to the C# type system that adds uniqueness and immutability qualifiers to types with the intention to support safe parallelism. They infer allowable implicit conversions between their qualifier constraints, which reduces the annotation burden, but the programmer still needs to provide the bulk of the annotations. Their qualifier inference requires only intraprocedural analysis selecting between a small number of type qualifiers, making it much simpler than the problem we address.

Kawaguchi et al. use liquid effects to guarantee deterministic parallelism [28], and they use liquid type inference [44] to significantly reduce the annotation burden. However, the underlying type inference is intraprocedural and the remaining annotations are too difficult for general-purpose programmers.

e) Inference of Ownership Types and Universe Types: Ownership Types (OT) [45] and Universe Types (UT) [46] aim to restrict aliasing in object oriented languages in order to express the programmer's intended containment abstractions. UT and OT do not target safe parallelism explicitly, and their annotations involve only a small fixed set of type modifiers. Several systems [10], [47], [48] support inference of ownership or universe types. These systems use relatively simple inference approaches which are insufficient for our highly expressive type system, in which each annotation has a vast set of allowed values.

VI. DISCUSSION AND FUTURE WORK

There are *no* practical programming languages in widespread use today that are as expressive as Java or C++ yet guarantee safe use of concurrency constructs. The automatic region-and-effect inference algorithm described in this work takes a step in that direction by greatly simplifying the use of static checking techniques for such languages. Our implementation demonstrates that inferring rich and complex annotations, such as those of DPJ, can be tractable given a smart solver, and opens opportunities for further research.

More specifically: (a) support of the remaining DPJ constructs, such as index-parameterized arrays and atomic blocks, as well as scoped locks (cf. KMeans); (b) techniques to improve the performance of the algorithm, e.g., using more aggressive constraint processing rules, and perhaps heuristic techniques for navigating the solution space more efficiently; and (c) better diagnosis information when the constraints are unsatisfiable, to assist programmers in correctly parallelizing their programs.

REFERENCES

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[2] M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of Jade," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, May 1998.

[3] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: A dynamic dependence-based parallel execution model," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009.

[6] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman, "Safe nondeterminism in a deterministic-by-default parallel language," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[7] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson, "Inferring method effect summaries for nested heap regions," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[8] R. L. Bocchino, Jr., "An effect system and language for deterministic-by-default parallel programming," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.

[9] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and reference immutability for safe parallelism," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[10] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst, "Inference and checking of object ownership," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

[11] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style," *Lisp and Symbolic Computation - Special issue on continuations - part I*, vol. 6, no. 3-4, pp. 289–360, November 1993.

[12] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1993, pp. 237–247.

[13] Region and Effects Annotations Inference Engine https://github.com/hanm/clang/tree/dev/prolog-full.

[14] "clang: a C language family frontend for LLVM," http://clang.llvm.org/.

[15] "Threading Building Blocks," https://www.threadingbuildingblocks.org/.

[16] Region and Effects Annotations Inference Solver & Benchmarks https://bitbucket.org/atzannes/annotationinference.

[17] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount.

[18] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

[19] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998.

[20] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[21] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[22] M. Tofte and J.-P. Talpin, "Implementation of the typed call-by-value λ-calculus using a stack of regions," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[23] A. Aiken, M. Fähndrich, and R. Levien, "Better static memory management: Improving region-based analysis of higher-order languages," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995.

[24] S. Cherem and R. Rugina, "Region analysis and transformation for java programs," in *Proceedings of the 4th international symposium on Memory management*, Vancouver, Canada, 2004.

[25] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988.

[26] Z. Anderson, D. Gay, R. Ennals, and E. Brewer, "SharC: Checking data sharing strategies for multithreaded C," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[28] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala, "Deterministic parallelism via liquid effects," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[29] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[30] G. Bierman and M. Parkinson, "Effects and effect inference for a core Java calculus," *Workshop on Object Oriented Developments*, 2003.

[31] A. Greenhouse and J. Boyland, "An object-oriented effects system," in *Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer-Verlag, 1999.

[32] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979.

[33] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher, "A schema for interprocedural modification side-effect analysis with pointer aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 2, pp. 105–186, March 2001.

[34] P. Nguyen and J. Xue, "Interprocedural side-effect analysis for Java programs in the presence of dynamic class loading," in *Proceedings of the Twenty-eighth Australasian Conference on Computer Science - Volume 38*, 2005.

[35] A. Rountev, "Precise identification of side-effect-free methods in java," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004.

[36] A. Salcianu and M. C. Rinard, "Purity and side effect analysis for Java programs," in *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005.

[37] P. Jouvelot and D. Gifford, "Algebraic reconstruction of types and effects," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1991.

[38] J.-P. Talpin and P. Jouvelot, "Polymorphic type, region and effect inference," *Journal of Functional Programming*, vol. 2, pp. 245–271, July 1992.

[39] M. Tofte and L. Birkedal, "A region inference algorithm," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 724–767, Jul. 1998.

[40] S. Artzi, A. Kieun, J. Quinonez, and M. D. Ernst, "Parameter refer-

ence immutability: formal definition, inference tool, and comparison," *Automated Software Engineering*, vol. 16, pp. 145–192, 2009.

[41] D. J. Pearce, "JPure: A Modular Purity System for Java," *Compiler Construction*, pp. 104–123, 2011.

[42] J. Quinonez, M. S. Tschantz, and M. D. Ernst, "Inference of Reference Immutability," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, 2008, pp. 616–641.

[43] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[44] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Low-level liquid types," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[45] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[46] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers, "Formal methods for components and objects," F. S. Boer, M. M. Bonsangue, S. Graf, and W.-P. Roever, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Universe Types for Topology and Encapsulation, pp. 72–112.

[47] W. Dietl, M. D. Ernst, and P. Müller, "Tunable Static Inference for Generic Universe Types," in *Proceedings of the 25th European Conference on Object-oriented Programming*, 2011.

[48] C. Dymnikov, D. J. Pearce, and A. Potanin, "OwnKit: Inferring modularly checkable ownership annotations for java," in *Proceedings of the 22nd Australian Conference on Software Engineering*, 2013.

VII. Technical Appendix

In this appendix, we present the remaining constraint generation rules for Section III-B, as well as complete formal proofs for the lemmas and theorems discussed in Section III-C. In order to make following the proofs easier, we first give rigorous mathematical definitions of concepts discussed in Section II, such as disjointness and distinction from the left and from the right, and we define some additional concepts

A. Remaining Constraint Generation Rules

For completeness, we include the remaining constraint generation rules for our core language, which are simpler versions of the ones on which we have already elaborated.

$$\frac{C\langle\mathcal{R}_x\rangle x \in \Gamma \qquad \text{typeof}(f) = C\langle\mathcal{R}_f\rangle f \text{ in } \mathring{\mathcal{R}}_f \qquad K = \{\Gamma \vdash \mathcal{R}_f \subseteq \mathcal{R}_x\}}{\Gamma \vdash x = \text{this}.f : C\langle\mathcal{R}_f\rangle, \{\text{rd } \mathring{\mathcal{R}}_f;\quad \text{wr } x\} \mid K} \text{(ReadTField)}$$

$$\frac{\text{typeof}(f) = C\langle\mathcal{R}_f\rangle f \text{ in } \mathring{\mathcal{R}}_f \qquad C\langle\mathcal{R}_y\rangle y \in \Gamma \qquad K = \{\Gamma \vdash \mathcal{R}_y \subseteq \mathcal{R}_f\}}{\Gamma \vdash \text{this}.f = y : C\langle\mathcal{R}_y\rangle, \{\text{rd } y;\quad \text{wr } \mathring{\mathcal{R}}_f\} \mid K} \text{(WriteTField)}$$

$$\frac{C\langle\mathcal{R}_x\rangle x \in \Gamma \qquad \tau = C\langle\mathring{\mathcal{R}}_\tau\rangle \qquad K = \{\Gamma \vdash \mathring{\mathcal{R}}_\tau \subseteq \mathcal{R}_x\}}{\Gamma \vdash x = \text{new } \tau() : C\langle\mathring{\mathcal{R}}_\tau\rangle, \{\text{wr } x\} \mid K} \text{(New)}$$

$$\frac{\begin{array}{c} C\langle\mathcal{R}_z\rangle z \in \Gamma \qquad \text{typeof}(m) = \tau_r\ m(\tau_x x) E_m\{\ldots\} \\ \tau_r = C\langle\mathcal{R}_r\rangle \qquad C'\langle\mathcal{R}_v\rangle v \in \Gamma \qquad \tau_x = C'\langle\mathcal{R}_x\rangle \\ K_1 = \{\Gamma \vdash \mathcal{R}_r \subseteq \mathcal{R}_z\} \qquad K_2 = \{\Gamma \vdash \mathcal{R}_v \subseteq \mathcal{R}_x\} \end{array}}{\Gamma \vdash z = \text{this}.m(v) : \tau_r, \{\text{rd } v;\quad \text{wr } z\} \cup E_m \mid K_1 \cup K_2} \text{(TCall)}$$

B. Definitions: Distinction and Disjointness

Below is the formal declarative definition of distinction from the left. It is more involved than the one presented in Section II because we also include the case of distinguishing region parameters from region names or from other region parameters.

Definition 7 (Distinction from the left ($\#_L$)): Let $\rho = e_1 : \ldots : e_n$, $\rho' = e'_1 : \ldots : e'_m$, where ρ, ρ' are RPLs and e_i, e'_i are RPL elements. Let $h = \min(n, m)$.

1. $e_1 = e'_1 (\neq \text{wildcard})$:
   $\rho \#_L \rho' \iff \exists k, 1 \leq k \leq \min(n,m)$:
   $(\forall i \leq k : e_i = e'_i \wedge e_i \neq \text{wildcard}) \wedge$
   $e_{k+1} \neq e'_{k+1} \wedge e_{k+1} \neq \text{wildcard} \wedge e'_{k+1} \neq \text{wildcard}$
   Note: We allow at most one e_{k+1} or e'_{k+1} not to exist, in which case we take the inequality $e_{k+1} \neq e'_{k+1}$ to be trivially true. This covers the cases where ρ, ρ' are fully specified and $\rho \leq \rho'$ or $\rho' \leq \rho$.
2. $e_1 \neq e'_1$
   a. $e_1 = P,\ e'_1 = Q$ (P, Q are region parameters)
      $\rho \#_L \rho' \iff P \#_L Q$ [Parameter Constraint]
   b. $e_1 = P,\ e'_1 = R$ (P rgn parameter, R rgn name)
      $\rho \#_L \rho' \iff P \#_L R$ [Parameter Constraint]
   c. $e_1 = R,\ e'_1 = P$ [same as above]
   d. $e_1 = R,\ e'_1 = R'$ (R, R' region names)
      $\rho \#_L \rho' \iff R \neq R'$

The definition of distinction from the right is conceptually the mirror image of distinction from the left. Some slight differences occur because region parameters can only appear at the first position, creating some asymmetry.

Definition 8 (Distinction from the right ($\#_R$)): Let $\rho = e_n : \ldots : e_1$, $\rho' = e'_m : \ldots : e'_1$, where ρ, ρ' are RPLs and e_i, e'_i are RPL elements (note that the indexing starts from the right this time). Let $h = \min(n, m)$.

1. $\exists i < h : e_i \neq e'_i$
   $\rho \#_R \rho' \iff \exists k, 0 \leq k < h$:
   $(\forall i \leq k : e_i = e'_i \wedge e_i \neq \text{wildcard}) \wedge$
   $e_{k+1} \neq e'_{k+1} \wedge e_{k+1} \neq \text{wildcard} \wedge e'_{k+1} \neq \text{wildcard}$
   Note. If $k = 0$, we assume that $e_0 = e'_0 \wedge e_0 \neq \text{wildcard}$ holds trivially.
2. $\forall i < h : e_i = e'_i \wedge e_i \neq \text{wildcard}$
   2.0 $n = m,\ e_n = e'_m \rightarrow \rho \not\#_R \rho'$.
   2.1 $n = m,\ e_n \neq e'_m$
      2.1.1 $e_n = P,\ e'_m = Q$ (P, Q region parameters)
      $\rho \#_R \rho' \iff P \#_R Q$ [Parameter Constraint]
      2.1.2 $e_n = P,\ e'_m = R$ (P region parameter, R region name)
      $\rho \#_R \rho' \iff P \#_R R$ [Parameter Constraint]
      2.1.3 $e_n = R,\ e'_m = P$ [same as above]
      2.1.4 $e_n = R,\ e'_m = R'$ (R, R' region names)
      $\rho \#_R \rho' \iff R \neq R'$

   2.2 $n \neq m$ (assume $n < m$)
      It follows that $e_n \neq e'_n$ because e_n is a head RPL element and e'_n is not.
      2.2.1 $e_n = R.\ \rho \#_R \rho' \iff e'_n \neq \text{wildcard}$
      2.2.2 $e_n = P.\ \rho \#_R \rho' \iff P \#_R (e'_m : \ldots : e'_n)$ [Parameter Constraint]

Definition 9 (Distjointness ($\#$)): Let ρ, ρ' be RPLs. $\rho \# \rho' \iff \rho \#_L \rho' \vee \rho \#_R \rho'$

Corollary 1 (Star Suffix): $\rho{:}* \# \rho'{:}* \implies \rho \#_L \rho'$
Note: The inverse does not hold if ρ, ρ' are fully specified and $\rho \leq \rho'$ or $\rho' \leq \rho$. E.g., $R_1 \#_L R_1{:}R_2$ but $R_1{:}* \not\# R_1{:}R_2{:}*$.

Corollary 2 (Star Prefix): $* : \rho \# * : \rho' \implies \rho \#_R \rho'$

Corollary 3 (# Prefix): Let $\mathring{\rho}$ denote a fully specified RPL segment (i.e., $\mathring{\rho} \in \mathring{\mathcal{R}}$). The following hold:

1) $\rho_1{:}\rho \#_L \rho_2{:}\rho \implies \rho_1 \#_L \rho_2$
2) $\rho_1{:}\rho \#_R \rho_2{:}\rho \iff \rho_1 \#_R \rho_2 \wedge (\rho \in \mathring{\mathcal{R}})$
3) $\rho_1{:}\rho \# \rho_2{:}\rho \implies \rho_1 \# \rho_2$
4) $\rho_1{:}\mathring{\rho} \#_L \rho_2{:}\mathring{\rho} \iff \rho_1 \#_L \rho_2$
5) $\rho_1{:}\mathring{\rho} \# \rho_2{:}\mathring{\rho} \iff \rho_1 \# \rho_2$

Corollary 4 (# Suffix): The following hold:

1) $\rho{:}\rho_1 \#_L \rho{:}\rho_2 \iff \rho_1 \#_L \rho_2 \wedge (\rho \in \mathring{\mathcal{R}})$
2) $\rho{:}\rho_1 \#_R \rho{:}\rho_2 \implies \rho_1 \#_R \rho_2$
3) $\rho{:}\rho_1 \# \rho{:}\rho_2 \implies \rho_1 \# \rho_2$
4) $\mathring{\rho}{:}\rho_1 \#_R \mathring{\rho}{:}\rho_1 \iff \rho_1 \#_R \rho_2$
5) $\mathring{\rho}{:}\rho_1 \# \mathring{\rho}{:}\rho_1 \iff \rho_1 \# \rho_2$

Corollary 5 ($\#_L$ Prefix):
$\rho_1 : \rho \#_L \rho_2 : \rho \iff \rho_1 \#_L \rho_2 \wedge ((\rho_1 \not\leq \rho_2 \wedge \rho_2 \not\leq \rho_1) \vee (\rho$ does not start with a wildcard))

C. Definitions: Substitutions

Definition 10 (Substitution): A substitution $S = [P \leftarrow X]$, where P is a region parameter and X is an RPL, results in replacing all occurrences of P with X. We call P the ***base*** of the substitution S and X its ***substituent*** (which may or may not contain P or another parameter).

Figure 6 shows how substitutions are categorized and how they relate to each other. The categories are defined below.

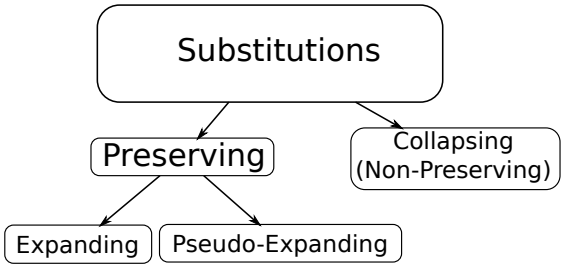


Fig. 6. Substitution Categories

Definition 11 (Collapsing Substitution): A substitution S is ***collapsing*** if and only if its substituent is not parametric (does not contain any region parameters).

Definition 12 (Preserving Substitution): A substitution S is ***preserving*** if and only if its substituent is parametric (its parameter may differ from the base).

Note: A substitution is either collapsing or preserving.

Definition 13 (Fully Specified Substitution): A substitution S is ***fully specified*** if and only if its substituent is fully specified.

Definition 14 (Substitution Length): The length of a substitution is equal to the number of RPL elements of its substituent.

Definition 15 (Expanding Substitution): A substitution S is ***expanding*** if and only if it is preserving, fully specified, and has length of 2 or more.

Definition 16 (Pseudo-Expanding Substitution): A substitution S is ***pseudo-expanding*** if and only if it is preserving, has length of 2 or more, and is not fully specified.

Definition 17 (Substitution Chain): A substitution chain $\vec{S} = S_1 \ldots S_n$ is an ordered sequence of substitutions (to be applied in that order).

Definition 18 (Preserving Substitution Chain): A substitution chain $\vec{S} = S_1 \ldots S_n$ is ***preserving*** if and only if $\forall i.S_i$ is preserving and $S_i = P_i \leftarrow P_{i+1}{:}\rho_i$.

Definition 19 (Expanding Substitution Chain): A substitution chain $\vec{S} = S_1 \ldots S_n$ is ***expanding*** if and only if it is preserving and $\exists i.S_i$ is expanding.

Lemma 3 (Recursive&Disjoint Writes Effect Summarization): If $\bigcup_{i=0}^{\infty}(\text{wr } P : \widetilde{\rho})S_1 S_2^i$, S_2 is expanding, and $S_1 S_2^2$ is an expanding chain, then $\bigcup_{i=0}^{\infty}(\text{writes } P{:}\widetilde{\rho})S_1 S_2^i \subseteq \text{wr } P'{:}*{:}\widetilde{\rho}$

Proof: S_1 is preserving because $S_1 S_2^2$ is an expanding chain, so let $S_1 = [P \leftarrow P'{:}\rho_1]$, where ρ_1 is allowed to be empty. S_2 is expanding so let $S_2 = [P' \leftarrow P'{:}\rho_2]$, where ρ_2 is non-empty and fully specified. Note that the parameter of the substituent must be the same as the base (P') because S_2^2 is an expanding chain. By writing out the substitutions we get:

$$\begin{aligned}\bigcup_{i=0}^{\infty}(\text{wr } P{:}\widetilde{\rho})S_1 S_2^i &= \text{wr } \bigcup_{i=0}^{\infty} P{:}\widetilde{\rho}[P \leftarrow P'{:}\rho_1][P' \leftarrow P'{:}\rho_2]^i \\ &= \text{wr } \bigcup_{i=0}^{\infty} P'{:}\rho_2^i{:}\rho_1{:}\widetilde{\rho} \\ &= \text{wr } P'{:}\rho_2^*{:}\rho_1{:}\widetilde{\rho} \subseteq \text{wr } P'{:}*{:}\widetilde{\rho}\end{aligned} \tag{9}$$

■

D. Discussion: RPL Inclusion and Disjointness

The formal definition of inclusion is given in the DPJ literature (e.g., Bocchino et al. OOPSLA09). Intuitively, RPL inclusion is the same as set inclusion, with RPLs representing sets of regions.

Note that RPL disjointness is akin to deciding if the intersection of two RPLs (sets of regions) is empty, although disjointness – as we have defined it ($\#$) – is a bit more conservative (disjoint RPLs always have an empty intersection, but an empty intersection does not imply disjointness, because defining disjointness more precisely would make computing it much more complex). E.g., according to our definition of disjointness, $P : R : *$ is not disjoint from R' without an accompanying explicit constraint on P, (where P is a region parameter and R, R' are region names). The two RPLs are not disjoint from the right because one ends in a star, and they are not disjoint from the left because P and R' are not comparable (e.g., because P could be instantiated to star). Similarly, Root : $*$: R : $*$ is not disjoint (according to our definition) from Root : R', although it is easy to reason that the intersection of these two RPLs is empty simply by noticing that the region name R appears in one RPL but not in the other. Another example is Root : $*$: R : R : $*$ and Root : R: the intersection is empty by simple length comparison, but disjointness from the left or the right cannot reach that conclusion. Our definition of disjointness has the advantage it is easy to compute. Additional precision would come at an increased computational cost for deciding the relation, without any known benefits to the expressiveness of the annotation language and the static analysis. Also, just having the star to describe infinite sets of RPLs lacks the

expressiveness to allow proving the disjointness of $P:R_1^*:X$ and $P:R_2^*:X$, where R^* denotes zero or more occurrences of an RPL element R.

E. Properties: RPL Inclusion and Disjointness

The following corollaries follow from the definitions of RPL inclusion and disjointness.

Corollary 6: $(\rho_1 \subseteq \rho_2 \ \wedge \ \rho_2 \# \rho) \Rightarrow \rho_1 \# \rho$. The converse does not hold.

Corollary 7: $(\rho_1 \subseteq \rho_2 \ \wedge \ \rho_1 \not\# \rho) \Rightarrow \rho_2 \not\# \rho$. The converse does not hold.

Theorem 3 (RPL Inclusion Monotonicity wrt to Substitution): For all RPLs ρ_1, ρ_2, and substitutions S, $\rho_1 \subseteq \rho_2 \Rightarrow \rho_1 S \subseteq \rho_2 S$.

Proof: We proceed by cases (the cases are not mutually exclusive but their union covers all possibilities):

1) $\rho_1 = \rho_2$. Trivially true.
2) Both ρ_1 and ρ_2 are **non**-parametric. Trivially true, as S has no effect on ρ_1 and ρ_2.
3) $\rho_1 \subset \rho_2$. At least one of the two RPLs is a collection of regions (infinite if it includes $*$, finite if it includes $[?]$. Assume $S = [P_s \leftarrow \rho]$.
   a) Both ρ_1 and ρ_2 are parametric in P_s. W.l.o.g., assume they are in *normal form* (i.e., they contain at most one star). $\rho_1 = P_s:\widetilde{\rho}_{1f}:*_{OPT}:\widetilde{\rho}'_{1fOPT}$ and $\rho_2 = P_s:\widetilde{\rho}_{2f}:*_{OPT}:\widetilde{\rho}'_{2fOPT}$. OPT denotes an optional part and f denotes a fully specified RPL segment. The premise becomes: $P_s:\widetilde{\rho}_{1f}:*_{OPT}:\widetilde{\rho}'_{1fOPT} \subseteq P_s:\widetilde{\rho}_{2f}:*_{OPT}:\widetilde{\rho}'_{2fOPT}$. From the definitions of RPL relations include ($\subseteq$) and under $\leq$, we deduce that (i) $\widetilde{\rho'}_{2fOPT}$ is a suffix of $\widetilde{\rho'}_{1fOPT}$, and (ii) $\widetilde{\rho}_{2f}$ is a prefix of $\widetilde{\rho}_{1f}$. Using these two observations, it is easy to show that $\rho:\widetilde{\rho}_{1f}:*_{OPT}:\widetilde{\rho}'_{1fOPT} \subseteq \rho:\widetilde{\rho}_{2f}:*_{OPT}:\widetilde{\rho}'_{2fOPT}$, which is what we want.
   b) One RPL (say ρ_1) is parametric in P_s while the other isn't. There are two cases for ρ_2. If it is parametric (but not in P_s), then because inclusion constraints for parameters are currently unsupported, the premise ($\rho_1 \subseteq \rho_2$) cannot hold. Thus the proposition is vacuously true. The other case is for ρ_2 to **not** be parametric, then the premise becomes: $P_s:\widetilde{\rho}_{1f}:*_{OPT}:\widetilde{\rho}'_{1fOPT} \subseteq \rho_{2f}:*_{OPT}:\widetilde{\rho}'_{2fOPT}$. As before, from the definitions of $\subseteq$ and $\leq$, we can deduce that (i) $\widetilde{\rho'}_{2fOPT}$ is a suffix of $\widetilde{\rho'}_{1fOPT}$, and (ii) $\{P \subseteq R\} \in \Gamma$, where Γ is the environment, and ρ_{2f} is a prefix of $R:\widetilde{\rho}_{1f}$. Then, as long as the substitution S is valid (i.e., $\rho \subseteq R$), the conclusion holds trivially. In practice, either the annotation checker will enforce that the substitution is legal, or if it is unable to do so, it will generate a constraint to enforce that requirement.

■

The inclusion monotonicity theorem helps us chain together subtype constraints that boil-down to inclusion constraints.

E.g., $\left.\begin{array}{l}\rho S \subseteq \rho_1 \\ \rho_1 S_1 \subseteq \rho_2 \\ \rho_2 S_2 \subseteq \rho\end{array}\right\} \Rightarrow \rho S S_1 S_2 \subseteq \rho_1 S_1 S_2 \subseteq \rho_2 S_2 \subseteq \rho \Rightarrow \rho S S_1 S_2 \subseteq \rho$

F. Constraint Processing Lemmas

We restate the Effect Inclusion to Equality lemma.

Lemma 4 (Effect Inclusion to Equality): An effect inclusion constraint $\{E_1, \ldots, E_n\} \subseteq ES$ can be replaced by an equality constraint $\{E_1, \ldots, E_n\} = ES$ if none of the effects $E_1, \ldots, E_n$ are invocation effects.

Soundness is trivial: the resulting equality constraint only allows for solutions that are a subset of the solutions to the original inclusion constraint. Notice that effect equality constraints are always trivially satisfiable because each effect summary variable (henceforth effect variable) appears on the right hand side of an effect inclusion constraint exactly once and because we replace inclusion by equality only when the left hand side contains simple effects (reads and writes currently, and also atomic reads and writes in the future). Moreover, effect inclusion constraints are also always satisfiable by setting the effect variable to include all effects on all regions, but such a choice will often make other (non-interference) constraints unsatisfiable.

In order to prove completeness, we must show that, for any effect inclusion constraint C with a right-hand side effect variable e that was simplified to an equality constraint C', for any solution E of C that satisfies the entire set of generated constraints, the solution E' of C' also satisfies the entire set of generated constraints. The key thing to notice is that $E' \subseteq E$. We turn our attention to constraints in which the effect variable e may appear, namely non-interference constraints and the left-hand-sides of effect inclusion constraints (effect variables never appear in RPL inclusion constraints).

First, assume that e appears on the left-hand-side of an effect inclusion constraint C'' (via an invocation effect). We will show that, given any solution E of C, the set of solutions S of C'' is a subset of the set of solutions S' of C'' given the solution E' of C' (i.e., $S \subseteq S'$). C'' has the generic effect inclusion constraint form $E_1, \ldots, E_n, e\sigma_1, \ldots, e\sigma_m \subseteq e'$. From the RPL inclusion monotonicity theorem (Theorem 3 $\forall \sigma : \ E' \subseteq E \implies E'\sigma \subseteq E\sigma$, and therefore any value of e' that satisfies C'' given $e = E$ (i.e., covers the effects of the left hand side, including E), will also satisfy C'' when $e = E'$. Therefore, the effect inclusion to equality simplification does not reduce the solution space of other effect inclusion constraints.

Second, for non-interference constraints, the effect variable e may either appear directly or indirectly (through a chain of invocations) by means of an invocation effect. The generic form in both cases is $e\sigma \# ES'$, where σ is a chain of substitutions determined by the invocation chain, and ES' is a set of effects. Again, because $E' \subseteq E \implies E'\sigma \subseteq E\sigma$ (Theorem 3), if $E\sigma$ satisfies the non-interference constraint, so does $E'\sigma$ which is the solution of the simplified effect inclusion constraint. Thus we have shown that the effect inclusion to equality simplification is complete, in that if the original set of constraints was satisfiable, so will the resulting one.

Next, we restate and prove the RPL Inclusion Chain lemma.

Lemma 5 (RPL Inclusion Chain): If $\rho_1 \subseteq \rho_2$, $\rho_2 \subseteq \rho_3$, $\ldots$, $\rho_{n-1} \subseteq \rho_n$, and $\rho_2, \ldots, \rho_{n-1}$ do not appear in any other RPL inclusion constraints, then inclusion is replaced by equality for all but the last RPL inclusion constraint: $\rho_i = \rho_{i+1}$, for $1 \leq i < n-1$.

Proving soundness is trivial: any solution of the new equality constraints will also satisfy the original inclusion

constraints.

To prove completeness, i.e., that if the original set of constraints had a solution so will the new set of constraints, we follow a similar approach as for the earlier lemma. The chain of equality constraints can be though of as assigning the value of ρ_1 to all RPL variables $\rho_2 \dots \rho_{n-1}$, while ρ_1 and ρ_n remain unconstrained. Since $\rho_2 \dots \rho_{n-1}$ do not appear in any other RPL inclusion constraints, and because assigning the value of ρ_1 to them satisfies the RPL inclusion constraints that form the chain under consideration, they remain satisfiable. Next, it remains to show that given any solution S of the original set of constraints, we can find a solution for the new set of constraints. Let v_1 and v_n be the values of ρ_1 and ρ_n in a solution S, and let v_i be the value of $\rho_i, 1 < i < n$ in S. We know that $v_1 \subseteq v_i \subseteq v_n$, otherwise S would not be a solution. We proceed as for the previous lemma.

First, for any ρ_i that appears on the left-hand-side of an effect inclusion constraint C, let e be its effect variable whose value in S is E. Then, by the RPL inclusion monotonicity theorem (Theorem 3), and because $v_1 \subseteq v_i, 1 < i < n$, we conclude that $E' \subseteq E$, where E' is the effect summary of the simplified constraint set, where $v_i, 1 < i < n$ are replaced with v_1.

Second, for effect non-interference constraints, the RPL varibles $\rho_i, 1 < i < n$ may appear either directly or indirectly through a chain of invocations. The generic form in both cases is $\rho_i \sigma \# ES$, where σ is a chain of substitutions determined by the invocation chain, and ES' is a set of effects. Again, because $v_1 \subseteq v_i \implies v_1\sigma \subseteq v_i\sigma$ (Theorem 3), if $v_i\sigma$ satisfies the non-interference constraint, so does $v_1\sigma$ which is the solution of the simplified RPL inclusion chain. Thus we have shown that the RPL inclusion chain simplification is complete, in that if the original set of constraints was satisfiable, so will the resulting one.

G. Disjointness Under Substitution Theorem

We restate the disjointness under substitution theorem.

Theorem 4 (RPL disjointness under Substitution):

$$\begin{aligned}\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2] \iff \rho = P{:}\widetilde{\rho}, \widetilde{\rho} \ \text{ fresh} \wedge \\ ((\rho_1 \#_R \rho_2 \wedge \widetilde{\rho} \in \mathring{\mathcal{R}}) \\ \vee (\rho_1 \#_L \rho_2 \wedge ((\rho_1 \not\leq \rho_2 \wedge \rho_2 \not\leq \rho_1) \\ \vee (\widetilde{\rho} \text{ does not start with wildcard}))))\end{aligned}$$

This can be rewritten as three inference rules that taken together cover all possible ways of satisfying the bottom

$$\frac{\rho = P{:}\widetilde{\rho} \qquad \widetilde{\rho} \in \mathring{\mathcal{R}} \qquad \rho_1 \#_R \rho_2}{\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]}$$
(DISTINCTION FROM THE RIGHT)

$$\frac{\rho = P{:}\widetilde{\rho} \qquad \rho_1 \#_L \rho_2 \qquad \rho_1 \not\leq \rho_2 \qquad \rho_2 \not\leq \rho_1}{\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]}$$
(DISTINCTION FROM THE LEFT NOT UNDER)

$$\frac{\rho = P{:}\widetilde{\rho} \quad \rho_1 \#_L \rho_2 \quad \widetilde{\rho} \text{ does not start with wildcard}}{\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]}$$
(DISTINCTION FROM THE LEFT UNDER)

Proof:

1.[$\implies$] If the substitutions evaluate to no-op, the result $\rho \# \rho$ is unsatisfiable, therefore at least one of the substitutions must modify ρ, so ρ must contain the parameter P. Region parameters can only appear at the head of an RPL, so $\rho = P{:}\widetilde{\rho}$, where $\widetilde{\rho}$ is a fresh, possibly empty, non-head RPL variable (if we are doing inference) whose domain is the same as that of ρ, but without its region parameters. The premise then becomes $\rho_1 : \widetilde{\rho} \# \rho_2 : \widetilde{\rho}$. From the definition of RPL disjointness we get that $\rho_1 : \widetilde{\rho} \#_R \rho_2 : \widetilde{\rho}$ or $\rho_1 : \widetilde{\rho} \#_L \rho_2 : \widetilde{\rho}$. In the first case, by Corollary 3 we get $\rho_1 : \widetilde{\rho} \#_R \rho_2 : \widetilde{\rho} \iff \rho_1 \#_R \rho_2 \wedge \widetilde{\rho}$ is fully specified. In the second case, by Corollary 5 we get $\rho_1 : \widetilde{\rho} \#_L \rho_2 : \widetilde{\rho} \iff \rho_1 \#_L \rho_2 \wedge ((\rho_1 \not\leq \rho_2 \wedge \rho_2 \not\leq \rho_1) \vee \widetilde{\rho}$ does not start with a wildcard).

2.[$\impliedby$] Given that $\rho = P : \widetilde{\rho}$, we just need to show that $\rho_1{:}\widetilde{\rho} \# \rho_2{:}\widetilde{\rho}$.

We proceed by cases:

1) $\rho_1 \#_R \rho_2 \wedge \widetilde{\rho} \in \mathring{\mathcal{R}}$ then by Corollary 3 $\rho_1{:}\widetilde{\rho} \# \rho_2{:}\widetilde{\rho}$
2) $\rho_1 \#_L \rho_2 \wedge ((\rho_1 \not\leq \rho_2 \wedge \rho_2 \not\leq \rho_1) \vee \rho$ does not start w. wildcard) then by Corollary 5 $\rho_1{:}\rho \#_L \rho_2{:}\rho$

■

Corollary 8 (Disjointness under Substitution – Necessary):

$$\frac{\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]}{\rho = P{:}\widetilde{\rho} \qquad \rho_1 \# \rho_2}$$
(DISJOINTNESS UNDER SUBSTITUTION – NECESSARY)

Note: this rule allows us to generate simpler constraints that will not miss any solutions, but may find solutions that are not solutions of the original program. This rule could be used to prune the search space and plug solutions found into the original problem to check if they satisfy it.

Corollary 9 (disjointness under Substitution – Sufficient):

$$\frac{\rho = P{:}\widetilde{\rho} \quad \widetilde{\rho} \in \mathring{\mathcal{R}} \quad \rho_1 \# \rho_2}{\rho[P \leftarrow \rho_1] \# \rho[P \leftarrow \rho_2]}$$
(RPL DISJOINTNESS UNDER SUBSTITUTION – SUFFICIENT)

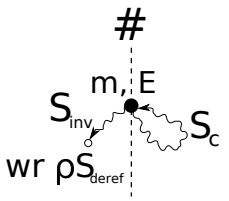
Note: this rule simplifies the original constraint and could lead to faster solving (by not having to explore several alternatives), but it is not complete so it could transform a satisfiable set of constraints into an unsatisfiable one.

In our implementation we use this disjointness under substitution theorem to infer the additional structural constraint $\rho = P{:}\widetilde{\rho}$ which constrains the instantiation space of ρ, but we do not try to exploit the more complex implications of the last two corollaries above. It will be a challenging and interesting future direction of this work to explore how to best exploit those in practice.

H. Disjoint Recursive Writes Theorem

Let m be a method with effect summary (variable) E that has some non-interference constraint (#), and let S_c be a substitution cycle along a call-graph cycle (direct or indirect recursion) (Eq. 10), and let m include a `writes` effect writes ρS_{deref}, possibly through a call chain from m that contributes a substitution chain S_{inv} (and let $S = S_{deref} S_{inv}$) (Eq. 11). Let the writes effect and the substitution cycle S_c be on opposing sides of the non-interference constraint (Eq. 12). See also Figure 7. Finally, we are assuming that the code

compiles without errors (there is no point in trying to prove parallel safety of a code that doesn't even compile), which in turn allows us to claim that the substitutions S and S_c are well-formed, properly translating the RPL ρ to the context of m (e.g., the variable in ρ being written to is declared within the context of some class or method and so is expressed using the region parameters of that context, but it must be translated to the context of m via the substitution S).



$$ES_c \subseteq E \quad (10)$$
$$\text{writes } \rho S \subseteq E \quad (11)$$
$$\text{writes } \rho S \# ES_c \quad (12)$$

Fig. 7. Recursive Write Theorem

Theorem 5 (Non-interference of Recursive Writes): Given the three premises described by equations Eq. 11, 10, and 12 we can infer that:

$$\rho = P' : \widetilde{\rho} \quad (13)$$
$$S = [P' \leftarrow P : \widetilde{s}] \quad (14)$$
$$S_c = [P \leftarrow P : \widetilde{s_c}], \ \ \widetilde{s_c} \text{ isn't empty} \quad (15)$$
$$\bigcup_{i=0}^{\infty} \text{writes } P : \widetilde{s_c}^{\,i} : \widetilde{s} : \widetilde{\rho} \subseteq E \quad (16)$$

Proof: From Eq. 11 and 10 we get:

$$\bigcup_{i=0}^{\infty} \text{writes } \rho S S_c^i \subseteq E \quad (17)$$

From Eq. 12 and 17 we get:

$$\text{writes } \rho S \# \bigcup_{i=1}^{\infty} \text{writes } \rho S S_c^i \iff \rho S \# \bigcup_{i=1}^{\infty} \rho S S_c^i \quad (18)$$

Note that i ranges from 1 to infinity and not from zero.

- Unless ρ is parametric, for any substitution S we have $\rho S = \rho$ and therefore $\rho S = \bigcup_{i=1}^{\infty} \rho S S_c^i$, which violates the premise. So, ρ must be parametric (let P' be the region parameter of the context where ρ appears): $\rho = P' : \widetilde{\rho}$ (QED Eq. 13).
- Unless S is preserving, for any substitution S' we have $\rho S S' = \rho S$ and therefore $\rho S = \bigcup_{i=1}^{\infty} \rho S S_c^i$ which violates the premise. So S must be preserving: $S = [P' \leftarrow P : \widetilde{s}]$ (QED Eq. 14). Also, we get $\rho S = P : \widetilde{s} : \rho$. We assumed S is well-formed, translating the writes effect from the context where it appears (i.e., parametric in P') to the context of m, which is parametric in some other region parameter, say P. Therefore, S must substitute P' (the region parameter of ρ), with something parametric in P.
- Assume S_c is collapsing: $S_c = [P \leftarrow s_c]$. Eq 18 becomes $P : \widetilde{s} : \widetilde{\rho} \# s_c : \widetilde{s} : \widetilde{\rho} \Rightarrow P \# s_c$. This is an inferred constraint on the region parameter P. However, this make the substitution S_c illegal, since it substitutes P with s_c. So, S_c cannot be collapsing, and thus it is preserving:

$$S_c = [P \leftarrow P : \widetilde{s_c}] \quad (19)$$

- Eq. 18 $\iff \forall i \geq 1. P : \widetilde{s} : \widetilde{\rho} \# P : \widetilde{s_c}^{\,i} : \widetilde{s} : \widetilde{\rho}$. From which we deduce that $\widetilde{s_c}$ cannot be empty:

$$|\widetilde{s_c}| \geq 1 \quad (20)$$

- Using Eq. 13, 14, and 19 we get $\bigcup_{i=0}^{\infty} \text{writes } \rho S S_c^i = \bigcup_{i=0}^{\infty} \text{writes } P : \widetilde{s_c}^{\,i} : \widetilde{s} : \widetilde{\rho}$ and since we know that $\widetilde{s_c}$ is non-empty (Eq. 20), the RPL segment $\widetilde{s_c}^{\,i}$ cannot be empty for all values of i (QED Eq. 16).

■

Discussion. We can summarize that infinite union of regions as $P : \widetilde{s_c}^{\,*} : \widetilde{s} : \widetilde{\rho}$, where $\widetilde{s_c}^{\,*}$ means zero or more occurrences of $\widetilde{s_c}$, but this notation is not part of the annotation language. The most precise single RPL that can describe this set of regions in the annotation language is $P : * : \widetilde{s} : \widetilde{\rho}$, which is a superset of the regions of Eq.16. We can show, by artificial counter-examples, that for any i, it may be necessary to *unroll* the effect set i times (e.g., for $i = 1$ we get $P : \widetilde{s} : \widetilde{\rho}, P : \widetilde{s_c} : * : \widetilde{s} : \widetilde{\rho}$) for the constraint transformation to be complete. In other words, summarizing $P : \widetilde{s_c}^{\,*} : \widetilde{s} : \widetilde{\rho}$ as $P : * : \widetilde{s} : \widetilde{\rho}$ is incomplete. However, in our implementation we use this incomplete heuristic because we have not encountered any non-artificial examples for which the heuristic caused us to miss solutions.

If we use this incomplete summarization, we can also infer that $\widetilde{s_c}$ doesn't start with a star wildcard ($*$). From Eq. 12 and using $P : * : \widetilde{s} : \widetilde{\rho} \subseteq E$ it follows that: writes $\rho S \#$ writes $P : * : \widetilde{s} : \widetilde{\rho} S_c \iff P : \widetilde{s} : \widetilde{\rho} \# P : \widetilde{s_c} : * : \widetilde{s} : \widetilde{\rho} \Rightarrow P \# P : \widetilde{s_c} : * \Rightarrow P \#_L P : \widetilde{s_c} \Rightarrow \emptyset \#_L \widetilde{s_c} \Rightarrow |\widetilde{s_c}| \geq 1$ and $\widetilde{s_c}$ doesn't start with a star. .

I. Isomorphic Solution Pruning Heuristic

In order to greatly reduce the number of isomorphic solutions (solutions that can be converted from one to another via alpha-renaming) during RPL variable instantiation, we use the following *symmetry breaking* technique: we implicitly declare a fresh region name each time we generate an RPL variable ρ, and we associate ρ with a domain $\mathbb{D}(\rho)$ that includes the region names declared so far (implicitly or explicitly) and any region parameters currently in scope. When instantiating an RPL variable, we are only allowed to use elements from its domain. In our example in Figure 3, for field `x` and RPL variable ρ_x, we implicitly declare region name R_x, so the domain for ρ_x includes the region parameter P and the region name R_x, i.e., $\mathbb{D}(\rho_x) = \{P, R_x\}$. Similarly, for RPL variable ρ_y associated with field `y`, we implicitly declare R_y, so $\mathbb{D}(\rho_y) = \{P, R_x, R_y\}$, which also includes the previously declared region name R_x. This symmetry breaking technique (the domains of ρ_x and ρ_y are different) reduces the isomorphic solutions: the domains of the two RPL variables have one region name in common (R_x) instead of having both R_x and R_y. Note that we did not preclude any *useful* solutions: there is an instantiation that places ρ_x and ρ_y in the same region (R_x, R_x), another that places them in different regions (R_x, R_y).