Inferring Method Effect Summaries for Nested Heap Regions

Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, Ralph Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {mvakili2,dig,bocchino,overbey2,vadve,rjohnson}@illinois.edu

Abstract—**Effect systems are important for reasoning about the side effects of a program. Although effect systems have been around for decades, they have not been widely adopted in practice because of the large number of annotations that they require. A tool that infers effects automatically can make effect systems practical. We present an effect inference algorithm and an Eclipse plug-in, DPJizer, which alleviate the burden of writing effect annotations for a language called Deterministic Parallel Java (DPJ). The key novel feature of the algorithm is the ability to infer effects on nested heap regions. Besides DPJ, we also illustrate how the algorithm can be used for a different effect system based on object ownership. Our experience shows that DPJizer is both useful and effective: (i) inferring effect annotations automatically saves significant programming burden; and (ii) inferred effects are more precise than those written manually, and are fine-grained enough to enable the compiler to prove determinism of the program.**

I. Introduction

Programs written in mainstream imperative languages have side effects on the memory. Programmers have embraced this paradigm because it avoids copying the program's state between different method invocations. However, this paradigm also makes it harder for programmers or tools to understand or analyze programs in a modular fashion.

Knowing what parts of the program's state are mutated by a method can help programmers modify large programs without introducing subtle mutation errors and can serve as explicit, machine-checkable documentation. It can enable safety tools to detect inconsistencies between intended usage of API methods and their actual usage, it is a building block for several other compiler analyses (e.g., MODREF analysis), and it can enable compilers to check the safety of parallel programs [1]–[3].

Effect systems express the effects of methods in terms of reads and writes of a subsets of the heap. Such groups of memory locations are referred to as "regions." Modern effect systems such as DPJ [4], [5] and JOE [6] express effects in terms of *nested heap regions*. Nested heap regions specify logical inclusion of regions, which is useful for recursion, subtyping, etc.

Effect annotations describing the side effects of each method can enable modular analysis of effects. But, although these annotations have been around for decades, they have not been used much in practice. The reason is that manually writing such effects is tedious and error-prone. In this paper, we present an algorithm that automatically infers the effects of each program statement, and summarizes them at the level of method declarations as *method effect summaries*. There is prior work on inferring effects on flat regions [7]–[9]; however, the key novelty of our algorithm is its ability to infer effects for programs even on nested heap regions, including recursive as well as non-recursive data structures.

Deterministic Parallel Java (DPJ) [4], [5] is an explicitly parallel language that aims to enable programmers to write safe parallel programs. DPJ is an extension to Java with an effect system based on regions. DPJ gives static guarantees that a program that type-checks with the DPJ compiler is safe, i.e., the program's behavior is deterministic. A deterministic program produces identical externally visible results in all executions for a given input.

The heart of DPJ is a type system that checks whether the parallel constructs access the shared data without conflicts. The programmer (i) specifies the shared data by virtually partitioning the heap into regions and (ii) specifies which regions are read or written by each method.

Using DPJ, we have safely parallelized several programs [5]; the parallel programs are deterministic and they exhibit good speedup. However, to get these benefits the programmer has to write region and effect annotations by hand. This job is nontrivial, error-prone, and time consuming. For example, a Monte Carlo financial application contains 1502 LOC and 314 annotations. A Barnes-Hut N-body application contains 698 LOC and 148 annotations.

This paper presents our tool, DPJizer, which alleviates the programmer's burden when writing effect annotations. Given a program with region annotations, DPJizer infers the method effect summaries and annotates the program. When summarizing the effect information, DPJizer eliminates redundant effects, which makes the effect annotations concise and easier to understand. We implemented DPJizer as an extension to Eclipse's refactoring engine, thus it offers all the convenient features of a practical refactoring engine: previewing changes, selection of edits to be applied, undo/redo, etc.

The inference algorithm at the heart of DPJizer is built on a classical constraint-based type-inference approach, but we use it to infer effects. The algorithm generates constraints from primitive operations (variable access, assignment, method calls, and method overriding declarations),

using the appropriate parameter and type substitutions at method invocations. It then solves these constraints by processing them iteratively and propagating the constraints through the call graph until a fixed point is reached and no more constraints are discovered. The novelty in the algorithm lies in the constraint solving phase. This phase handles nested regions by taking advantage of the structure of region specifications in the target language (e.g., Region Path Lists [5] in DPJ or object "levels" in the object ownership system, JOE [6]). It handles *recursive* structures by summarizing these nested heap regions in each case.

Although DPJIZER is designed to help in porting a Java program to DPJ, its applicability goes well beyond DPJ. Given a concurrent program that uses shared memory, by inferring the method effects, DPJIZER helps a programmer discover the patterns of shared data. This information is crucial in helping the programmer find out the accesses to shared data that need to be protected. Moreover, the underlying algorithm is useful beyond concurrent programs. For example, we show how the algorithm can be used to infer effects for a different effect specification system based on object ownership, which is a general mechanism to reason about and express the side effects of object-oriented programs.

This paper makes the following contributions:

1. Algorithm. To the best of our knowledge, this paper presents the first algorithm for inferring method effect summaries for a full Object-Oriented language (e.g., aliasing, recursion, polymorphism, generics, arrays, etc.) with a sophisticated effect system (e.g., parameterized regions, nested regions for recursive data-structures, etc.).

2. Tool. We implemented the effect inference algorithm in an *interactive* tool called DPJIZER. A programmer can use DPJIZER to infer method effects for a Java or a DPJ program. DPJIZER writes the inferred effects into the source code as DPJ annotations or as code comments. DPJIZER is built as an Eclipse plugin that extends Eclipse's refactoring engine.

3. Evaluation. We used DPJIZER to infer method effects in several real programs. We compare the effects inferred with DPJIZER against effects manually inferred by programmers. The comparison shows that DPJIZER can drastically reduce the burden of writing annotations manually, while the automatically inferred effects are more precise.

II. OVERVIEW OF DPJ

Deterministic Parallel Java (DPJ) [5] is a programming language that ensures parallel tasks are *noninterfering*. Two tasks are noninterfering if for each pair of memory accesses, one from each task, either both accesses are reads, or the two accesses operate on disjoint sets of memory locations.[1]

Noninterfering tasks can run in parallel while still exhibiting the same behavior as if they were run sequentially.

DPJ provides a type system that *guarantees* noninterference of parallel tasks for a well-typed program. In DPJ, the programmer assigns every object field and array cell to a *region* of memory and annotates each method with a summary (called a *method effect summary*) of the method read and write effects. The programmer also marks which code sections to run in parallel, using several standard constructs, such as `cobegin` for parallel statement execution and `foreach` for parallel loops. The compiler uses the region annotations and method effect summaries to check that all pairs of parallel tasks are noninterfering.

A. Region Names

```
1   class Node {
2     region Mass, Force;
3     double mass in Mass;
4     double force in Force;
5     void setMass(double mass) writes Mass {
6       /* writes Mass */
7       this.mass = mass;
8     }
9     void setForce(double force) writes Force {
10      /* writes Force */
11      this.force = force;
12    }
13    void initialize(double mass, double force)
14      writes Mass, Force {
15        cobegin {
16          /* writes Mass */
17          this.setMass(mass);
18          /* writes Force */
19          this.setForce(force);
20        }
21    }
22  }
```

Figure 1. Using field region names to distinguish writes to different object fields. In Section III, we will show how to infer the underlined method effect summaries.

Figure 1 illustrates the use of region names to distinguish writes to different fields of an object. Line 2 declares `Mass` and `Force` as region names that are available within the scope of class `Node`. These are called *field region declarations*. Lines 3 and 4 declare fields `mass` and `force` and place them in regions `Mass` and `Force`, respectively. Field region declarations are static, so there is one for each class. For example, all `mass` fields of all `Node` instances are in the same region, `Mass`.

Each method must have a *method effect summary* recording the effects that it performs on the heap, in terms of reads and writes to regions. For example, method `setMass` (line 5) has the summary `writes Mass`, because the effect of line 7 is to write the field `mass`, located in region `Mass`; and similarly for `setForce` (line 9). It is permissible for a method effect summary to be overly conservative; for example, `setMass` could have said `writes Mass, Force`. However, this may inhibit parallelism. It is an error for a method effect summary to be not conservative enough,

for example if `setMass` had said `pure`, meaning "no effect."

Together, the DPJ annotations allow the compiler to efficiently analyze noninterference of parallel code sections, as illustrated in the `initialize` method. From the method effect summaries, the compiler can infer that the effect of line 17 is `writes Mass` and the effect of line 19 is `writes Force`. The compiler can then use the distinctness of the names `Mass` and `Force` to prove noninterference: although both statements in the `cobegin` perform writes, the writes are to disjoint regions of the heap.

B. Region Parameters

As shown in section II-A, region names are useful for distinguishing parts of an object from each other. Often, however, we need to distinguish different *object instances* from each other. To do this, DPJ uses *region parameters*, which operate similarly to Java generic parameters [11] and allow us to instantiate different object instances of the same class with different regions.

```
1   class Node<region P> {
2     region L, R;
3     double mass in P;
4     Node<L> left in L;
5     Node<R> right in R;
6     void setMass(double mass) writes P {
7       /* writes P */
8       this.mass = mass;
9     }
10    void setMassOfChildren(double mass)
11      writes L, R {
12        cobegin {
13          /* writes L */
14          if (left != null) left.setMass(mass);
15          /* writes R */
16          if (right != null) right.setMass(mass);
17        }
18      }
19  }
```

Figure 2. Using region parameters to distinguish writes to different object instances.

Figure 2 illustrates the use of region parameters to distinguish writes to different object instances. In line 1, we declare class `Node` to have one region parameter `P` (we use the keyword `region` to distinguish DPJ region parameters from Java type parameters). As with Java generics, when we write a type using a class with region parameters, we provide an argument to the parameter, as shown in lines 4 and 5. The argument must be a valid region name in scope.

We can use the region name `P` within the scope of the class. For example, line 3 declares field `mass` in region `P`. When `this.mass` is accessed, the effect is on region `P`, as shown in line 7. However, when we access field `mass` through a selector other than `this`, we resolve the region `P` by substituting the actual argument given in the type of the selector. For example, the effect of `left.setMass` in line 14 is `writes L`. We get this by looking at the declaration `writes P` of `setMass` and substituting `L` for `P` from the type of `left`. (The read of field `left` also generates a read effect on region `L`; but in DPJ, write effects imply read effects, so the read is covered by `writes L`.) We can then use an analysis similar to the one discussed in Section II-A to prove that the statements in lines 14 and 16 are noninterfering, since their write effects are on the disjoint regions `L` and `R`.

C. Region Path Lists (RPLs)

In conjunction with array index regions and index-parameterized arrays (discussed further in Section III-E1), basic region names and region parameters can be used to express important parallel algorithms. However, it is often essential to be able to express a *nesting* relationship between regions. For example, to express tree-like recursive updates we need a nested hierarchy of regions. DPJ provides two ways to express nesting: *region path lists* and *owner regions*. Here we focus on region path lists; we defer the discussion of owner regions until after we have presented the effect inference algorithm.

A region path list (RPL) extends the idea of a simple region name introduced in Section II-A. An RPL is a colon-separated list of names that expresses the nesting relationship syntactically: if `P` and `R` are names, then `P:R` is nested under `P`. Nested RPLs are particularly useful in conjunction with region parameters: if we append names to parameters, such as `P:L` and `P:R`, then by left-recursive substitution we can generate arbitrarily long chains of names, such as `P:L:L:R`.

```
1   class Node<region P> {
2     region L, R;
3     double mass in P;
4     Node<P:L> left in P:L;
5     Node<P:R> right in P:R;
6     void setMassForTree(double mass) writes P:* {
7       /* writes P */
8       this.mass = mass;
9       cobegin {
10        /* writes P:L:* */
11        if (left != null)
12          left.setMassForTree(mass);
13        /* writes P:R:* */
14        if (right !=null)
15          right.setMassForTree(mass);
16      }
17    }
18  }
```

Figure 3. Using RPLs and region parameters to recursively update a tree in parallel.

Figure 3 illustrates the use of this technique to write a recursive tree update. The example is similar to the one shown in Figure 2, except that lines 4 and 5 use regions `P:L` and `P:R` instead of `L` and `R`, and the method invocations in lines 12 and 15 are recursive. To write the method effect summary in line 6, we need some new syntax: because the tree can be arbitrarily deep, and the RPLs arbitrarily long, we use a star (`*`) to stand in for any sequence of RPL elements.

Then we can write the method effect summary `writes P:*`, as shown in line 6. Note that the rules discussed in Section II-A for method effect summaries are still followed: by substituting the RPL arguments in the types of `left` and `right` in for `P`, we get the inferred effects shown in lines 10 and 13; and those effects are covered by the summary. Further, because the RPLs form a tree, we can conclude that all regions under `P:L` and all regions under `P:R` are disjoint, so effects of lines 12 and 15 are noninterfering.

III. Effect Inference Algorithm

We present our algorithm using *Core DPJ* [5], a small skeleton language that illustrates the ideas yet is tractable to formalize. To make the presentation easier to follow, we start with a simplified form of Core DPJ corresponding to the features introduced in Section II-A, i.e., basic region names with no region parameters or nesting. Then we build up the language to add region parameters, region path lists and owner regions. We also discuss how to handle array regions and inheritance. Finally, we discuss how to adapt the algorithm for use with other languages.

A. Basic region names

Figure 4 shows the syntax of the initial language. Note that we have moved field region declarations to the global scope to simplify the region names in the formal language. The algorithm consists of two phases, constraint generation and constraint solving.

Meaning	Symbol	Definition
Programs	*program*	*region-decl* * *class* *
Region decls	*region-decl*	`region` r
Classes	*class*	`class` C {*field** μ^*}
Fields	*field*	T f `in` r
Types	T	C
Methods	μ	T $m(T$ $x)$ { e }
Expressions	e	`this`.f | `this`.$f = e$ | $e.m(e)$ | `new` T | z
Variables	z	`this` | x

Figure 4. Syntax of Core DPJ with basic region names. C, f, m, x and r are identifiers.

1) Constraint Generation: The constraint generation phase computes for each method μ a *constraint set* K_μ, where each element of K_μ is one of the constraints *reads r*, *writes r*, and *invokes* μ'. The first two constraints indicate the presence of a read or write effect in the method itself. The *invokes* constraint asserts that one method is invoking another, either directly or indirectly; these constraints will cause the solving phase (Section III-A2) to account for the read and write effects of callees.

The constraint generation phase visits each method body and walks the AST to generate a set of constraints. Figure 5 shows the constraint generation rules for the simplified language. The rules are similar to the ones for typing DPJ expressions [5], except that we do not check assignments or method formal parameter bindings for soundness (we

$$\text{(FIELD-ACCESS)} \quad \frac{(\text{this}, C) \in \Gamma \quad \textit{field}(C, f) = T \; f \; \text{in} \; r}{\Gamma \vdash \text{this}.f : T, \{\textit{reads } r\}}$$

$$\text{(ASSIGN)} \quad \frac{(\text{this}, C) \in \Gamma \quad \Gamma \vdash e : T, K \quad \textit{field}(C, f) = T' \; f \; \text{in} \; r}{\Gamma \vdash \text{this}.f = e : T, K \cup \{\textit{writes } r\}}$$

$$\text{(INVOKE)} \quad \frac{\Gamma \vdash e_1 : C, K_1 \quad \Gamma \vdash e_2 : T, K_2 \quad \textit{method}(C, m) = \mu}{\Gamma \vdash e_1.m(e_2) : T_r, K_1 \cup K_2 \cup \{\textit{invokes } \mu\}}$$

$$\text{(NEW)} \quad \frac{\cdot}{\text{new } C : C, \emptyset} \qquad \text{(VARIABLE)} \quad \frac{(z, T) \in \Gamma}{z : T, \emptyset}$$

Figure 5. Rules for computing the constraints generated by an expression.

assume that full DPJ type checking has been done as a separate pass).

The judgment $\Gamma \vdash e : T, K$ means that expression e has type T and generates constraint set K in environment Γ. The environment Γ is a set of pairs (z, T) binding variable z to type T. The term *method*(C, m) means the method named m defined in class C, while *field*(C, f) means the field named f defined in class C. For each method $\mu = T_r \; m(T_x \; x) \; \{ \; e \; \}$, let C_μ be the class where μ is defined. Then K_μ is just the set of constraints such that

$$\{(\text{this}, C_\mu), (x, T_x)\} \vdash e : T, K_\mu.$$

As an example, we show how to generate the constraints for the bodies of methods `setMass`, `setForce` and `initialize` in Figure 1. In line 7, rule ASSIGN generates the constraint *writes* `Mass` for assignment to the field in region `Mass`. Similarly, rule ASSIGN generates the constraint *writes* `Force` for method `setForce`. In method `initialize`, there are two method invocations (lines 17 and 19). Therefore, rule INVOKE generates two constraints *invokes* `setMass` and *invokes* `setForce`.

2) Constraint Solving: The constraint solving phase computes for each method μ an *effect set* E_μ, where each element of E_μ is one of the effects *reads r* or *writes r*. This phase comprises the following steps:

1) For each method μ, for each constraint *invokes* μ' in K_μ, add the elements of $K_{\mu'}$ to K_μ.
2) Repeat step 1 until no more constraints are added to any K_μ.

Step 1 prunes the constraint sets K_μ by never adding redundant constraints. For example, since writes cover reads, there is no need for any K_μ to contain both *reads r* and *writes r*; the second constraint suffices.

The algorithm terminates, because the total number of regions is bounded, so the total number of constraints that can be added to the K_μ is bounded. At the end of this process, for each μ we let $E_\mu = \textit{effects}(K_\mu)$, where the function *effects* extracts the read and write constraints (i.e., the effects) from K_μ. As an example, from Figure 1, the constraints *invokes* `setMass` and *invokes* `setForce` generate the effects *writes* `Mass` and *writes* `Force`.

B. Region Parameters

This section extends Core DPJ by adding region parameters. CoreDPJ allows only one region parameter to each class in order to simplify the formal rules. Also, CoreDPJ disallows type parameters, as they are irrelevant to our effect inference algorithm. Figure 6 shows the new syntax.

Meaning	Symbol	Definition
Classes	*class*	class $C\langle P\rangle$ {*field** μ^*}
Regions	R	$r \mid P$
Types	T	$C\langle R\rangle$

Figure 6.   New syntax of Core DPJ with region parameters. P is an identifier. Other syntactic elements are the same as in Figure 4.

1) Constraint Generation: Figure 7 shows the rules for generating *reads*, *writes*, and *invokes* constraints in Core DPJ with region parameters. The new rule INVOKE records the *region substitution* $\theta = \{P \mapsto R\}$ that the constraint solver will need when translating the effects of one method to another. The term *param*(C) represents the region parameter P of class C.

(FIELD-ACCESS) $\dfrac{(\texttt{this}, C\langle P\rangle) \in \Gamma \quad field(C, f) = T \; f \text{ in } R}{\Gamma \vdash \texttt{this}.f : T, \{reads\ R\}}$

(ASSIGN) $\dfrac{(\texttt{this}, C\langle P\rangle) \in \Gamma \quad \Gamma \vdash e : T, K \quad field(C, f) = T' \; f \text{ in } R}{\Gamma \vdash \texttt{this}.f = e : T, K \cup \{writes\ R\}}$

(INVOKE) $\dfrac{\Gamma \vdash e_1 : C\langle R\rangle, K_1 \quad \Gamma \vdash e_2 : T, K_2 \quad method(C, m) = \mu \quad \mu = T_r \; m(T_x \; x) \; \{ \; e \; \} \quad \theta = \{param(C) \mapsto R\}}{\Gamma \vdash e_1.m(e_2) : \theta(T_r), K_1 \cup K_2 \cup \{invokes\ \mu\ where\ \theta\ \}}$

(NEW) $\dfrac{\cdot}{\texttt{new } C\langle R\rangle : C\langle R\rangle, \emptyset}$ (VARIABLE) $\dfrac{(z, T) \in \Gamma}{z : T, \emptyset}$

Figure 7.   Rules for generating constraints in Core DPJ with region parameters.

As an example, we show how to generate the constraints for the code in Figure 2. According to rule AS-SIGN, line 8 generates the constraint *writes* P. In line 14, Rule FIELD-ACCESS generates the constraint *reads* L for accessing the field left. Then, because the type of this.left is Node<L>, rule INVOKE generates the constraint set {*reads* L, *invokes* setMass *where* {P ↦ L}}. Line 16 generates similar constraints, using region R instead of L.

2) Constraint Solving: The constraint solving phase is identical to the one described in Section III-A2, except that the algorithm applies substitutions θ in resolving *invokes* constraints:

1) For each method μ, for each constraint (*invokes* μ' *where* θ) in K_μ, add the elements of $\theta(K_{\mu'})$ to K_μ.
2) Repeat step 1 until no more constraints are added to any K_μ.

Here we apply the substitution θ elementwise to sets K_μ, and we apply θ to constraints as follows:

$$\begin{aligned}
\theta(reads\ R) &= reads\ \theta(R) \\
\theta(writes\ R) &= writes\ \theta(R) \\
\theta(invokes\ \mu\ where\ \theta') &= invokes\ \mu\ where\ \theta(\theta') \\
\theta(\{P \mapsto R\}) &= \{P \mapsto \theta(R)\}
\end{aligned}$$

The number of region parameters is finite, so the number of region substitutions is finite. Therefore, there are a finite number of *invokes* constraints. Thus, the algorithm terminates for the same reason given in Section III-A2.

Figure 8 illustrates the constraints and effects inferred by each iteration of the algorithm on the method setMassOfChildren in Figure 2. For brevity, we show only the effects coming from left.setMass(). Just before iteration 1, the effect of method setMass is summarized as *writes* P. In iteration 1, the *invokes* effect leads the algorithm to infer the effect *writes* L by applying the substitutions P ↦ L on the effect of setMass. The algorithm does not infer any new effects in iteration 2, so it terminates after iteration 2.

	Before Iteration 1	Iteration 1
Effects	*reads* L	*writes* L
Constraints	*invokes* setMass *where* {P ↦ L}	

Figure 8.   Effects and constraints inferred in each iteration of the algorithm for method setMassOfChildren in Figure 2

C. Region Path Lists (RPLs)

This section adds RPLs to Core DPJ. Only the syntax for regions, shown in Figure 9, is new. Root is a special name representing the root of the region tree.

Meaning	Symbol	Definition
Regions	R	Root $\mid r \mid P \mid R : r \mid R : *$

Figure 9.   Syntax of RPLs. Other syntactic elements are the same as in Figure 6.

Constraint generation is the same as explained in Section III-B1. However, we need to extend the solving phase to handle recursion that would not terminate if we naively applied the algorithm from Section III-B2. For example, that algorithm would not terminate on the code in Figure 3, because it would try to infer effects on infinite chains of RPL elements, such as $P : L : R : \cdots$. In such a case, we want to cut off the recursion and summarize the infinite set of RPLs with a *partially specified* RPL ending in $*$, as described in Section II-C.

1) Algorithm Description: We say that an *invokes* constraint (*invokes* μ *where* θ) $\in K_{\mu'}$ is *recursive* if and only if $\mu = \mu'$, i.e., the method includes its own effects, through a chain of one or more invocations. We define an *expanding substitution* to be a substitution such as $P \mapsto R$, where P

is the first RPL element of R, and R has more than one element. For example, $P \mapsto P : R$ is expanding but $P \mapsto P$ and $P \mapsto P' : R$ are not.

RPLs can become arbitrarily long when going under multiple region substitutions. We bound the length of RPLs to get readable effects and guarantee the termination of the algorithm. Usually, developers write RPLs of length at most three.

Figure 10 shows the modified constraint solving algorithm. In lines 1–6, the algorithm normalizes the *invokes* constraints of all methods. This normalization step truncates all long RPLs and appends a star to them. It also detects expanding substitutions in recursive *invokes* constraints and appends a star to the RPLs in such substitutions.

The *truncate* function makes sure that no RPL longer than a predefined length gets created. If it gets a long RPL, it cuts it off to fit within the limit and appends a star, e.g. $P : R : S : T$ becomes $P : R : *$ to have a length of three. It is necessary to make sure that the truncated RPL ends with a star so that it covers the original long RPL. The *summarize* function makes sure that each expanding substitution in the given recursive *invokes* constraint ends with a star. For example, *summarize* returns the substitution $P \mapsto P : R : *$ given the expanding substitution $P \mapsto P : R$.

The algorithm iterates until all the effect and constraint sets stabilize. As before, each iteration of the main loop iterates over the method set $\mathcal{M}$ and adds effects implied by the *invokes* constraints of K_μ. After adding the effects of the callee in line 10, the algorithm iterates over the *invokes* constraints of the callee, applies the substitution of the callee on the region substitution of the *invokes* constraint, and truncates the resulting region substitution to make sure that no long RPL occurs. If the truncated substitution is recursive and expanding, then the algorithm summarizes it before adding it back to K_μ.

As before, all unions are up to redundant constraints and effects. For instance, once *writes* $R : *$ appears in K_μ, the algorithm never again adds *reads* R or *writes* R to K_μ in line 10. Similarly, once *invokes* μ with $P \mapsto P : R : *$ appears in K_μ, the algorithm never adds *invokes* μ with $P \mapsto P : R : R$ in lines 13 and 15. This pruning ensures that the algorithm terminates (Section III-C3).

2) Example: Figure 11 illustrates the constraints and effects inferred by each iteration of the `repeat` loop in lines 7–16 of Figure 10 for the method `setMassForTree` in Figure 3. Before starting iteration 1, the constraints and effects are those computed up to line 7 of Figure 10. In this example, we let the cut-off limit for the *truncate* function be three. As a result, before iteration 1, the normalization step of the algorithm (Section III-C1), appends stars to the *invokes* constraints. In iteration 1, the algorithm detects two recursive *invokes* constraints with expanding substitutions. Then, it applies the substitutions of these two *invokes* constraints on the effects discovered before iteration 1. This

```
input  : Program P with region annotations
         Set M of methods
         Set K_μ of constraints for each method μ
output: A set of effects, E_μ, for each method μ
1  foreach μ ∈ M do
2      foreach c = (invokes μ' where θ) ∈ K_μ do
3          if μ' = μ and isExpanding(θ) then
4              c ← (invokes μ' where summarize(truncate(θ)))
5          else
6              c ← (invokes μ' where (truncate(θ)))
7  repeat
8      foreach μ ∈ M do
9          foreach c = (invokes μ' where θ) ∈ K_μ do
10             K_μ ← K_μ ∪ truncate(effects(K_μ'))
11             foreach c' = (invokes μ'' where θ') ∈ K_μ' do
12                 if μ'' = μ and isExpanding(θ(θ')) then
13                     K_μ ← K_μ ∪ (invokes μ'' where
                           summarize(truncate(θ(θ'))))
14                 else
15                     K_μ ← K_μ ∪ (invokes μ'' where truncate(θ(θ')))
16  until no K_μ changes
17  foreach μ ∈ M do
18      E_μ = effects(K_μ)
```

Figure 10.   The inference algorithm for RPLs.

	Before Iteration 1	Iteration 1
Effects	*reads* P:L, P:R *writes* P	*writes* P:L:*, P:R:*
Constraints	*invokes* setMassForTree() *where* {P ↦ P:L:*}, setMassForTree() *where* {P ↦ P:R:*}	

Figure 11.   Effects and constraints inferred in each iteration of Figure 10 for the method `setMassForTree` in Figure 3.

application discovers the two new effects of iteration 1. The algorithm terminates after iteration 2 because it does not find any new effects in iteration 2.

Note that even though the effect summary *writes* P:* shown in Figure 3 is correct (i.e., it type-checks), DPJIZER infers a more precise (i.e., a more refined) summary. For this program, DPJIZER infers *writes* P, P:L:*, P:R:*. The effect *writes* P comes from the write access in line 8, while *writes* P:L:* comes from the recursive function in line 12. DPJIZER recognizes the recursive traversal of the data structure, and partially specifies the affected regions as P:L:*. The effect *writes* P:R:* comes from the recursive function in line 15.

3) Termination and Algorithmic Complexity: There are only a finite number of RPLs of a certain maximum length. So, the total number of effects and constraints that can be added to each set K_μ is finite. Because all K_μ sets are finite, the algorithm terminates.

We analyze the running time of our algorithm in terms of two parameters m and n and a constant c. The parameter m is the maximum number of method invocations in a method body, n is the number of possible RPL elements, and the constant c is the maximum length of RPLs.

Because the length of RPLs is bounded by c, the total number of possible RPLs is $O(n^c)$. Therefore, the number of *reads* and *writes* effects of a method is $O(n^c)$.

Each method makes at most m invocations and each

invokes constraint has at most n region substitutions of a length of at most n^c. Therefore, the number of *invokes* constraints of a method is $O(mn^{c+1})$. In other words, $\forall \mu \; |K_\mu| = O(mn^{c+1})$.

Assuming that the running time set union operation is linear in the size of its operands, the running of the algorithm is a polynomial in terms of m and n.

D. Owner Regions

DPJ provides a mechanism called *owner regions* for recursively partitioning a flat data structure (such as an array) in a divide-and-conquer manner. Figure 12 illustrates how to use owner regions to write parallel quicksort. The class `DPJArray` wraps an ordinary Java array and can be used to partition the array into subranges, and class `DPJPartition` is used to split the `DPJArray` into left and right segments `segs.left` and `segs.right`, as shown in lines 7 and 8.

The class `DPJPartition` dynamically partitions an array into two subarrays which are nested under the `this` region. Therefore, `QSort.sort` creates a binary tree of `QSort` objects, with each in its own region. The compiler verifies the noninterference of effects because the object references to `DPJPartition` are distinct and the subarrays are in disjoint regions nested under the object references.

In lines 11 and 12, the type of `segs.left` is `segs:DPJPartition.Left`, where `DPJPartition.Left` is a field region name (Section II-A) and the `final` local variable `segs` functions as an RPL. When a variable appears as an RPL, the region it represents is associated with the object reference stored in the variable at runtime, as in ownership type systems [6], [12]. The region of the variable is nested under the region bound to the first parameter of the variable's type. Here, `segs` has type `DPJPartition<P>`, so `segs` is nested under `P`. This fact allows us to write the method effect summary `writes P:*` covering both the write to `P` in line 6 and the recursive invocations of `sort` in lines 12 and 15. Given this effect summary, the compiler can use the inferred effects shown in lines 10 and 13 to prove that the statements in the `cobegin` block are noninterfering.

Figure 13 shows the syntax of Core DPJ extended to support owner regions. Note that we have changed the syntax of expressions in the following ways: (1) we add a `let` construct to simulate `final` local variables; and (2) we require the selector and actual argument of a method invocation expression to be variables to keep the typing rules simple.

Constraint generation works exactly as described in Section III-B1 except for rules LET and INVOKE, shown in Figure 14. In rule LET, we have to account for the fact that RPLs generated inside the `let` expression may contain a local variable z that is not in scope outside the body of the expression. Therefore, we coarsen any such RPL $z : R$

```
1   class QSort<region P> {
2     final DPJArray<P> A in P;
3     QSort(DPJArray<R> A) pure { this.A = A; }
4     void sort() writes P:* {
5       /* Quicksort partition: writes P */
6       int p = qsPartition(A);
7       final DPJPartition<P> segs =
8         new DPJPartition<P>(A, p);
9       cobegin {
10        /* writes segs:DPJPartition.Left:* */
11        new QSort<segs:DPJPartition.Left>
12          (segs.left).sort();
13        /* writes segs:DPJPartition.Right:* */
14        new QSort<segs:DPJPartition.Right>
15          (segs.right).sort();
16      }
17    }
18  }
19
20  class DPJPartition<region P> {
21    region Left, Right;
22    DPJArray<this:Left> left in this:Left;
23    DPJArray<this:Right> right in this:Right;
24
25    DPJPartition(DPJArray<P> A, int pivot) {
26      left = (DPJArray<this:Left>)
27        A.subarray(0, pivot);
28      right = (DPJArray<this:Right>)
29        A.subarray(pivot, A.length - pivot);
30    }
31  }
```

Figure 12. Using owner regions to write quicksort.

Meaning	Symbol	Definition
Regions	R	$\texttt{Root} \mid r \mid P \mid z \mid R:r \mid R:*$
Expressions	e	$\texttt{let } z = e \texttt{ in } e \mid \texttt{this}.f \mid \texttt{this}.f = e \mid$
		$z.m(z) \mid \texttt{new } C\langle R \rangle \mid z$

Figure 13. Core DPJ with owner regions.

to $R' : *$, where the type of z is $C\langle R' \rangle$. Rule INVOKE is nearly identical to the one shown in Figure 7, except that we record the substitutions $\texttt{this} \mapsto z_1$ and $x \mapsto z_2$ as well as the substitution $param(C) \mapsto R$. With these changes, the solving algorithm works exactly as described in Section III-C1.

$$
\text{(LET)} \quad \frac{\Gamma \vdash e_1 : C\langle R \rangle, K_1 \quad \Gamma \cup \{(x, T_1)\} \vdash e_2 : T_2, K_2 \quad \theta = \{x \mapsto R : *\}}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \; : \theta(T_2), \theta(K_1 \cup K_2)}
$$

$$
\text{(INVOKE)} \quad \frac{\{(z_1, C\langle R \rangle), (z_2, T)\} \subseteq \Gamma \quad method(C, m) = \mu}{\mu = T_r \; m(T_x \; x) \; \{ \; e' \; \}} \\
\frac{\theta = \{param(C) \mapsto R, \texttt{this} \mapsto z_1, x \mapsto z_2\}}{\Gamma \vdash z_1.m(z_2) : \theta(T_r), K \cup \{invokes \; \mu \; where \; \theta \}}
$$

Figure 14. Rules for generating constraints in Core DPJ with owner regions.

E. Other DPJ Features

We now show how we extended the algorithm described in Section III-C to handle the key remaining features of DPJ: arrays and inheritance.

1) Arrays: DPJ provides two capabilities for computing with arrays: *array RPL elements* and *index-parameterized*

arrays. An array RPL element is `[e]`, where `e` is an integer expression. Since array regions are just RPL elements (e.g., `Root:[e]:r`), the algorithm can handle them in exactly the same way as described for name RPL elements. We just need a constraint collection rule that says that if expression `e` uses a method-local variable that is out of scope at the point of the method prototype, then we replace the element `[e]` in any RPL with `[?]`, representing an *unknown* array index element in the DPJ type system.

An index-parameterized array allows the programmer to use an array index expression in the type of an indexed element. For example, the programmer can specify that the type of array index expression `A[e]` is `C<[e]>`. To handle index-parameterized arrays, we just add constraint generation rules for assignment and access through array index expressions that are nearly identical to the rules for field assignment and access shown in Figure 7. The rules are also similar to the rules for array access typing shown in [5].

2) Inheritance: DPJ supports inheritance, e.g., class $B\langle P \rangle$ extends $A\langle R \rangle$. Inheritance raises two issues for the inference algorithm. First, we must translate inherited methods and fields from the superclass to the subclass. We do this by applying the *translating substitution* θ implied by the chain of `extends` clauses from the superclass to the subclass. For example, if class $C_1\langle P_1 \rangle$ extends $C_2\langle R_1 \rangle$, and $C_2\langle P_2 \rangle$ extends `Object`, then the translating substitution from C_2 to C_1 is $\{P_2 \mapsto R_1\}$.

Second, DPJ requires that the declared effects of a method include the effects of all overriding methods [5]. This gives rise to a constraint similar to the one we represented by an *invokes* constraint, except that it is simpler, because there is no recursion in the inheritance graph. To handle this constraint, we make two simple extensions to the algorithm. First, in the constraint collection phase, after collecting constraints from each method body, we add to each K_μ the constraint *isOverriddenBy* μ' *where* θ, for each method μ' such that μ is overridden by μ'. Here, θ is the translating substitution defined above. Second, in the constraint solving phase, in each iteration of the `repeat` loop in Figure 10, between lines 7 and 16, we add another iteration over all methods $\mu \in \mathcal{M}$ to add $\theta(K_{\mu'})$ to K_μ for each constraint *isOverriddenBy* μ' *where* θ in K_μ.

F. Applicability Beyond DPJ

The relevance of our effect inference algorithm is not limited to DPJ: with suitable modifications, the algorithm can be adapted to infer effects for other object-oriented effect systems, such as ownership-based systems [6], [13], [14], that have features similar to DPJ's. Here, we illustrate how the inference algorithm might be adapted to work on the ownership-based effect system by Clarke and Drossopoulou called Java with Ownership and Effects, or JOE [6].

JOE also employs method effect summaries and supports effects on regions similar to DPJ's owner regions, except that JOE has no RPLs. Instead, JOE uses *effect shapes* of the form $p.n$ and `under` $p.n$, where p is a `final` local variable or context parameter (similar to a DPJ region parameter), and $n \geq 0$ is a natural number. The shape $p.n$ refers to all descendants of p in the region tree that are n levels below p in the tree, with $p.0$ being equivalent to p. The shape `under` $p.n$ is similar to an RPL with $*$ at the end and refers to all $p.n'$ such that $n' \geq n$. The key rule of JOE, which defines the region tree, is that if variable z has type $C\langle o \rangle$, then the shape $z.n$ is covered by the shape $o.n + 1$, where $o = owner(z)$ is the region bound to the owner parameter in the type of z.

To adapt our algorithm to JOE, we make the following modifications. First, instead of substitutions $P \mapsto R$, we use substitutions $p.n \mapsto p'.n + k$, for $k \geq 0$. Second, in rule LET (Figure 14), when a variable z goes out of scope, we generate an effect for the outer scope by applying the substitution $z.n \mapsto o.n + 1$, where $o = owner(z)$. (We could also replace $z.n$ with `under` o, as our LET rule does for DPJ, but replacing $z.n$ with $o.n + 1$ is more precise.) Third, we define an expanding substitution to be $p.n \mapsto p.n + k$, for $k > 0$, and when we see an expanding substitution, we replace its right-hand side with `under` $p.n + k$. Otherwise, the algorithm works as described in the previous sections.

```
1   class List<o> {
2     int data;
3     List<this> next;
4     void update(int data)
5       writes this, under this+1 {
6         /* writes this */
7         this.data = data;
8         /* invokes update where
9            {this.n ↦ this.n+1} */
10        let z = next in
11          /* invokes update where
12             {this.n ↦ z.n+0} */
13          if (z != null) z.update(data);
14      }
15  }
```

Figure 15.   Example of inferring effects for JOE.

Figure 15 shows how the algorithm infers effects for a simple recursive JOE program. This code traverses and updates a list such that each node of the list owns the next node. The initial constraints gathered in the constraint collecting phase are shown in the comments. Rule INVOKE generates the effect in lines 11–12, which is adjusted to the effect in lines 8–9 by the LET rule discussed above. At the end of initial constraint collection, the constraints are as shown in lines 8–9 and 11–12. In the first iteration of the solving algorithm, the expanding substitution shown in lines 8–9 gets summarized as `this.`$n \mapsto$ `under this.`$n + 1$. Applying this substitution to the effect *writes* `this` and putting the result back into the constraint set yields the inferred effects shown in line 5. The algorithm then terminates because there

are no new effects to add.

IV. THE DPJIZER TOOL

We have built an interactive tool, DPJIZER, incorporating the algorithm discussed in Section III. We implemented DPJIZER as an Eclipse plug-in. Given a partial DPJ program with legal region annotations, DPJIZER produces a legal DPJ program with region and effect annotations. In addition, the tool has some valuable interactive features. A programmer can select an effect in a method summary, and DPJIZER highlights the statements or expressions that generated that effect (as seen in the screen fragment in Figure 16). Alternately, the programmer can select a statement or expression, and DPJIZER highlights its corresponding effect in the effect summary. Thus, when the compiler reports interference warnings, the developer can use DPJIZER to localize the problem and refine the region annotations accordingly.

We now describe in more detail how DPJIZER helps programmers write DPJ programs. Typically, a DPJ programmer carries out the following steps to convert a given sequential Java program to DPJ.

1) Choose which sections of code are to be run in parallel, but do not yet insert the parallel constructs (`cobegin`, `foreach`, etc.).
2) Devise a strategy for using region declarations, region parameters, and RPLs to express the noninterference of the parallel sections. Add these annotations and make sure they pass the type checker. At this point, the methods all have an empty summary, which in DPJ means the most conservative effect, i.e., *writes* `Root:*`. Such effect summaries will pass the type checker but will not allow parallelism to be safely expressed.
3) For methods transitively invoked by a parallel section, refine the method summaries as necessary to make the parallel tasks mutually noninterfering.
4) Add the parallel constructs to the parallel sections.
5) If there are any interference warnings, then revisit steps two and three to revise the region annotations and/or method effect summaries to eliminate the interference.

DPJIZER helps this process in the following ways. First, step three is completely automated. This automation removes a lot of work from the development process, particularly if the user has to do two or more iterations of steps two and three. While the compiler provides error information of the form "effect E is missing from the summary of method m" that helps the user fix bad summaries, step three is still time consuming and difficult. Code with many methods and invocations requires a lot of summaries. Further, it is difficult for a user to manually propagate effects backwards along the call graph and around cycles. DPJIZER automates this process.

```
void setMass(double mass)
          writes Root : BinaryTree.M,
          Root : BinaryTree.L,
          Root : BinaryTree.R {
    this.mass = mass;
    initTree();
}
```

Figure 16.   The programmer selects an effect in the effect summary and DPJIZER highlights the statement that generated that effect.

Second, DPJIZER helps step five by identifying the statements and expressions within a method that are contributing "bad" effects. A key part of this step is understanding the statements and methods that contribute these effects; the tool simplifies that greatly by allowing users to map effects back to the expressions that produce them, and vice versa. With additional programming (not yet implemented for lack of time), the tool will also help understand better how effects are propagated from methods to call sites, along with the relevant substitutions and how they propagate around cycles in the call graph, in some cases leading to summarization with '*'.

V. EVALUATION

Research Questions. To evaluate the effectiveness of DPJIZER, we answer the following two questions:

- **Q1:** Is DPJIZER useful? Does it alleviate the burden of writing effect annotations?
- **Q2:** Are the inferred effects precise? Do the inferred effects enable the compiler to prove determinism of the program?

We answer these questions in two ways: with a case study running the tool ourselves, and with a survey in which we asked other programmers who have written DPJ programs to run the tool and describe their experience using it. The case study (Section V-A) provides *quantitative* answers, while the survey (Section V-B) provides *qualitative* answers.

A. Case Study

1) Methodology: Table I lists the programs that we used as case studies. Program size is given in non-blank, non-comment lines of source code, counted by *sloccount*. These programs were manually annotated with regions and effects by other programmers before the existence of DPJIZER. We took these programs, erased the effect annotations leaving only the region annotations, and used DPJIZER to infer the method effects.

To answer the first question (usefulness), we report the number of effects that programmers wrote manually. To answer the second question (precision), we check that the effects inferred by DPJIZER are not interfering in the parallel constructs (e.g., `cobegin`). We also compare the effects written manually with those inferred by DPJIZER.

Note that DPJIZER's analysis does not take parallel statements into account. When using DPJIZER, it is assumed that the code has an appropriate set of region annotations such

that a fine-grained enough set of effects would make the effects of statements in parallel statements noninterfering. So, the answer to the second question (precision) clarifies whether DPJIZER infers such fine-grained effects.

2) Quantitative Results:

Q1: Is DPJIZER **useful?** From Table I one can see that if the programmers had used DPJIZER to infer the method effects, they would have saved writing 406 effects. Further, the programmers would have saved the time it took to generate these effects by manually propagating constraints backwards through the call graph, around cycles, and up the inheritance graph.

Q2: Are the inferred effects precise? Do the inferred effects enable the compiler to prove determinism of the program? We carefully analyzed the programs in Table I and compared the effects written by programmers with the effects inferred automatically by DPJIZER. Since programmers did not write effects for all methods, we can only compare the effects for the methods that were annotated.

Table I shows the number of differences between the manually and automatically inferred effects. Note that in all cases, DPJIZER infers effects that are the same as, or more precise than those written by the programmer. In terms of precision, there are two categories of differences between manually and automatically inferred effects: (i) *granularity* and (ii) *redundancy*.

In terms of granularity, some of the manual effects are too coarse-grained in the choice of effect keyword, e.g., *writes* R instead of *reads* R. This is legal (i.e., it type-checks) but unnecessarily coarse and forbids the parallel execution of two methods (e.g., two get() methods) that only read region R and otherwise could have been executed in parallel.

Second, some manually inferred effects are too coarse-grained in the region specification. For example, the programmer specified *writes* P:* when the appropriate effect inferred by DPJIZER was *writes* P, P:L:*, P:R:*. The coarser region inferred by programmer forbids any other method that writes in a subregion of P to run in parallel. This is an unnecessary restriction because the method only writes in subregions created using the L or R prefixes, so that another method that writes into P:M should be allowed to run in parallel.

In terms of redundancy, some of the manually written summaries contain redundant effects. For example, the programmer specified *reads* R *writes* R, but the read effect is subsumed by the write. Alternatively, the programmer wrote *writes* P, P:*, where the first region is redundant since it is subsumed by the second region. Such redundancies do not hinder parallelism but make the method effect summary unnecessarily verbose, which can hinder program understanding.

We carefully analyzed the source code of the methods, and indeed DPJIZER inferred the most fine-grained effects that are possible to express with the current DPJ language and the user-defined threshold on the RPL length, and the summaries do not contain redundant effects.

With respect to redundancy, DPJIZER does a better job than the programmer. With respect to granularity, there is a trade-off between expressible parallelism, reusability, and readability of code. Fine-grained effects enable more parallelism. However, the programmer might prefer coarser-grained effects to aid reusability. For example, the programmer might make effects of a method coarser-grained to allow future code extensions. For instance, she may prefer to summarize the effect of a method as *writes* R, even though *reads* R covers the effects of that method. But, she chooses the coarser-grained effect, *writes* R, because she anticipates subclasses that will override the method with the effect *writes* R. DPJIZER works in a *closed-world environment*: it only infers effects based on the available code and does not take reusability into account. Therefore, the programmer has to rerun DPJIZER each time she extends the code. Also, an effect like *writes* P:* is coarser-grained than *writes* P, P:L:*, P:R:*, but it is more readable.

For each program, the programmer wrote a set of regions and parallel constructs. Then, DPJIZER inferred a set of sufficiently fine-grained effects that enabled the compiler to prove determinism of the program. That is, DPJIZER inferred fine-grained effects that did not interfere in the parallel statement, and enabled all of the specified parallel constructs to run safely. To verify this, we checked that the compiler did not report any interference warnings in the code with inferred effects. The other way to verify this is to notice that DPJIZER inferred effects that were finer-grained than those written manually. Therefore, because none of the manually written effects interfered in the parallel statements, none of the inferred effects interfered either.

B. User Survey

We also conducted a preliminary survey of other programmers who have previously written DPJ programs. This study comprised the following steps:

1) We elided the effect annotations on the programs previously parallelized by these users, but retained the region annotations they had written.
2) The users then used DPJIZER to infer the effect annotations for those programs.
3) The users finally filled out a brief questionnaire asking about the results, usability and overall experience of using DPJIZER.

This study is limited because it only has a small number of users and they all know the study authors. Nevertheless, it provides some preliminary feedback on the usefulness of the tool from experienced DPJ programmers not involved in designing or building DPJIZER (none of them had seen or even participated in discussions about DPJIZER before the survey).

Program	SLOC	# of Manually Written Effects	# of Effects Too Coarse By		# of Redundant Effects
			keyword	region	
Barnes-Hut	698	47	1	0	3
CollisionTree	1021	83	4	14	0
IDEA	299	3	0	0	0
K-means	540	3	0	0	0
ListRanking	106	4	1	2	0
MergeSort	147	7	0	4	0
MonteCarlo	1502	179	5	0	16
QuadTree	143	13	1	2	0
QuickSort	150	12	0	0	3
StringMatch	380	54	5	21	2
SumReduce	60	1	0	0	0
Total	**5046**	**406**	**17**	**43**	**24**

Table I

LIST OF PROGRAMS, SIZES OF THOSE PROGRAMS, AND THE NUMBER OF
EFFECTS PROGRAMMERS HAVE WRITTEN. WE ALSO REPORT THE
MANUAL EFFECTS THAT ARE TOO COARSE-GRAINED BY KEYWORD
(E.G., PROGRAMMER WROTE *writes* R INSTEAD OF *reads* R) OR BY
REGION (E.G., PROGRAMMER WROTE *reads* R : * INSTEAD OF *reads* R).
THE LAST COLUMN SHOWS THE NUMBER OF REDUNDANT EFFECTS
(E.G., PROGRAMMER WROTE BOTH *reads* R *writes* R).

Usefulness: The users said the tool saved "a significant fraction" of porting effort. One user said the tool saved "a lot of time in the process of writing/adding annotations ... and then compiling to find more methods to annotate."

Accuracy: One user thought the tool inferred too many annotations: he would prefer to see fewer effect annotations, and could re-run the tool if more were needed. Conversely, he said the tool did help eliminate some redundant annotations (compared with his manual effect summaries).

Requested features: The most requested features included incremental addition of annotations; presenting choices of annotations to the user and letting him choose; and recommending better region structure to produce more fine-grain effects. (The latter is outside the scope of the current work but is a subject of future work, as described in Section VII.)

Summary: Overall, all users said that they would use DPJIZER to help write DPJ programs. One user said "I think it will also help me redesign region structures to be more precise and effective."

VI. RELATED WORK

Method effect summaries. Many effect systems employ effect summaries to enable modular analysis and composability of program components. The original proposals for an object-oriented effect system [3], [15] use summaries, as do several systems combining object ownership with some form of effects [6], [14], [16]. Our work presents an algorithm and a tool that can be used to infer such summaries.

Effect inference. The seminal work on inferring effects is from Jouvelot and Gifford [7]. They use a technique called *algebraic reconstruction* to infer types and effects in a mostly functional language. Talpin and Jouvelot [8] build on this work to develop a constraint-based solving algorithm. These algorithms are tailored to a mostly-functional language with a much simpler effect system than DPJ's: nested effects cannot be expressed, so no summaries such as $R : *$ have to be inferred.

Bierman and Parkinson [9] present an inference algorithm for Greenhouse and Boyland's effect system [3]. The features they consider are similar to the smallest subset of Core DPJ we covered in section III-A, plus support for unique reference annotations and a limited form of nesting. Again there is no unbounded nesting.

Side-effect analysis [17]–[20] uses interprocedural alias analysis and dataflow propagation algorithms to compute the side effects of functions. There are two major differences between these algorithms and DPJIZER. First, DPJIZER operates on programmer-specified region types, which identify and express effects more precisely than alias analysis. Second, DPJIZER exploits the structure of RPLs to do a custom solution for recursive calls, which should significantly speed up convergence of the constraint solver.

Commutativity analysis [21] uses symbolic execution to collect the side effects of methods and reason about which pairs of methods commute with each other. The analysis is fully automatic, but less expressive than DPJ, because programs must be written in a certain restricted style in order for the analysis to work.

Region and type inference. There is extensive literature on *region inference* for region-based memory management [22]–[25]. Several researchers have studied the problem of inferring types or type qualifiers for imperative programs with references. Kiezun et al. [26] show how to infer Java generic parameters and arguments. Agarwal and Stoller show how to do type inference for parameterized race-free Java [27]. Quinonez et al. [28] present a tool called Javarifier for inferring *reference immutability* for variables (i.e., that the reference is never used to update the state of any object that it transitively points to). Terauchi and Aiken [29] present a type inference algorithm for deterministic parallelism using *linear types* supplemented with fractional permissions [30].

These algorithms are broadly similar to ours, in that they collect constraints across the whole program and solve them. However, the technical details are quite different because the problem domains differ from our problem of inferring effects for nested regions. The region and type inference techniques may be useful in extending DPJIZER to infer DPJ region annotations.

VII. Conclusions

We have presented an effect inference algorithm and a tool, DPJ<small>IZER</small>, that ease the burden of writing DPJ programs. Our experience shows that DPJ<small>IZER</small> infers effects that are are both readable and precise. The DPJ<small>IZER</small> algorithm is also applicable to other effect systems that rely on method effect summaries and nested heap regions. As future work, we plan to extend the capabilities of DPJ<small>IZER</small> so that it can help with *region inference*, i.e., inferring region declarations, region parameters, and region arguments. Region inference in DPJ is challenging, but preliminary work indicates that it should be possible to infer regions for many common parallel patterns.

VIII. Acknowledgments

References

[1] J. M. Lucassen *et al.*, "Polymorphic effect systems," in *POPL*, 1988.

[2] R. T. Hammel and D. K. Gifford, "FX-87 performance measurements: Dataflow implementation," Tech. Rep. MIT/LCS/TR-421, 1988.

[3] A. Greenhouse and J. Boyland, "An object-oriented effects system," *ECOOP*, 1999.

[4] "DPJ homepage," http://dpj.cs.illinois.edu.

[5] R. L. Bocchino, V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," to appear in OOPSLA 2009.

[6] D. Clarke and S. Drossopoulou, "Ownership, encapsulation and the disjointness of type and effect," in *OOPSLA*, 2002.

[7] P. Jouvelot and D. Gifford, "Algebraic reconstruction of types and effects," in *POPL*, 1991.

[8] J.-P. Talpin and P. Jouvelot, "Polymorphic type, region and effect inference," *J. Funct. Prog.*, July 1992.

[9] G. Bierman and M. Parkinson, "Effects and effect inference for a core java calculus," *Workshop on Object Oriented Developments (WOOD)*, 2003.

[10] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir, "Parallel Programming Must Be Deterministic By Default," in *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.

[12] D. G. Clarke *et al.*, "Ownership types for flexible alias protection," *OOPSLA*, 1998.

[13] C. Boyapati, B. Liskov, and L. Shrira, "Ownership types for object encapsulation," in *POPL*, 2003.

[14] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," in *OOPSLA*, 2002.

[15] K. R. M. Leino *et al.*, "Using data groups to specify and check side effects," 2002.

[16] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith, "Multiple ownership," *OOPSLA*, 2007.

[17] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *POPL*, 1979.

[18] 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," *TOPLAS*, 2001.

[19] A. Salcianu and M. C. Rinard, "Purity and side effect analysis for Java programs," in *VMCAI*, 2005.

[20] A. Rountev, "Precise identification of side-effect-free methods in java," in *ICSM*, 2004.

[21] P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *TOPLAS*, 1997.

[22] M. Tofte and L. Birkedal, "A region inference algorithm," *TOPLAS*, 1998.

[23] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard, "Region inference for an object-oriented language," in *PLDI*, 2004.

[24] A. Banerjee, M. Barnett, and D. A. Naumann, "Boogie meets Regions: A verification experience report," in *VSTTE*, 2008.

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

[26] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for parameterizing Java classes," in *ICSE*, 2007.

[27] R. Agarwal and S. D. Stoller, "Type inference for parameterized race-free Java," in *VMCAI*, 2004.

[28] J. Quinonez, M. S. Tschantz, and M. D. Ernst, "Inference of reference immutability," in *ECOOP*, 2008.

[29] T. Terauchi and A. Aiken, "A capability calculus for concurrency and determinism," *TOPLAS*, 2008.

[30] J. Boyland, "Checking interference with fractional permissions," *SAS*, 2003.