INFERRING METHOD EFFECT SUMMARIES FOR NESTED HEAP REGIONS

BY

MOHSEN VAKILIAN

B.S., University of Tehran, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Research Associate Professor Ralph E. Johnson

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, that 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: inferring effects annotations automatically saves significant programming burden; and inferred effects are comparable to those in manually-annotated programs, while in many cases they are more accurate.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to Ralph Johnson and Vikram Adve who supervised this project. Also thanks to my colleagues Robert Bocchino, Danny Dig and Jeffrey Overbey for their invaluable contribution to this project. And finally, thanks to my wife and parents who have always offered me support and love.

This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programs written in mainstream, imperative languages have side effects on memory. Programmers have embraced this paradigm because it avoids copying program's state between different functions. 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 function 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 [24,20,18].

Although effect systems 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 thesis 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*. The key novel capability in the algorithm is that it is able to handle effect inference for programs even on "nested heap structures," including recursive as well as non-recursive data structures.

We have used the algorithm to develop an effect inference tool for a previously developed extension to Java, called *Deterministic Parallel Java (DPJ)* [15,6], which aims to enable programmers to write safe parallel programs. The effect system in DPJ is based on "regions," which are partitions of the heap, down to the granularity of individual fields. We also illustrate briefly how the algorithm could be used for a very different class of effect systems based on "object ownership" [12].

DPJ is an explicitly parallel language that uses a region and effect system to guarantee that any program that type-checks will have deterministic visible behavior, i.e., the program will produce identical externally visible results in all executions for a given input. Such deterministic semantics can greatly simplify parallel program design, debugging, testing and maintenance because it allows programmers to reason about programs with a simple, sequential semantic model and debug and test the program using versions of familiar sequential tools. Using DPJ, we have safely parallelized several programs [6]; 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 non trivial, error-prone, and time consuming. For example, a Monte Carlo financial application contains 2877 LOC, 29 region annotations, and 24 effect annotations. A Barnes-Hut N-body application contains 682 LOC, 91 region annotations, and 46 effect annotations.

DPJizer 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. 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 heart of DPJizer is an algorithm that statically infers side effects on field regions. The input of the algorithm is a program where shared fields are annotated with region information. The algorithm infers for each method the *method effect summary* that covers the effects (reads/writes) on declared regions. When summarizing the effect information, DPJizer eliminates redundant effects, which makes the effect annotations concise and easier to understand.

The inference algorithm 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 [6] in DPJ or object "levels" in the object ownership system, JOE [12]). It handles *recursive* structures by summarizing these nested structures 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, as noted earlier, 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 thesis makes the following contributions:

1. Algorithm. To the best of our knowledge, this thesis 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 accurate.

Chapter 2

Overview of DPJ

Deterministic Parallel Java (DPJ) [6] is a 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 be 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.

2.1 Region Names

Figure 2.1 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 6 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."

[1] The full DPJ language [5,6] also allows commutativity annotations that specify noninterference directly, without checking reads and writes. Here we focus on inferring read and write effects.

```
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 2.1: Using field region names to distinguish writes to different object fields. In Section 3, we will show how to infer the underlined method effect summaries.

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.

2.2 Region Parameters

As shown in section 2.1, 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 [17] and allow us to instantiate different object instances of the same class with different regions.

Figure 2.2 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 8. 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 13 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

```
 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) writes L, R {
11          cobegin {
12              /* writes L */
13              if (left != null) left.setMass(mass);
14              /* writes R */
15              if (right != null) right.setMass(mass);
16          }
17      }
18  }
```

Figure 2.2: Using region parameters to distinguish writes to different object instances.

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 2.1 to prove that the statements in lines 13 and 15 are noninterfering, since their write effects are on the disjoint regions `L` and `R`.

2.3 Region Path Lists (RPLs)

In conjunction with array index regions and index-parameterized arrays (discussed further in Section 3.4.1), 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 2.1. 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 recursive substitution we can generate arbitrarily long chains of names, such as `P:L:L:R`.

Figure 2.3 illustrates the use of this technique to write a recursive tree update. The example is similar to the one shown in Figure 2.2, except that lines 4 and 5 use regions `P:L` and `P:R` instead of `L` and `R`, and the method invocations in lines 11 and 13 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 2.1 for method effect summaries are still

```
 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) left.setMassForTree(mass);
12              /* writes P:R:* */
13              if (right !=null) right.setMassForTree(mass);
14          }
15      }
16  }
```

Figure 2.3: Using RPLs and region parameters to recursively update a tree in parallel.

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 12; 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 lines 11 and 13 are noninterfering.

Chapter 3

Effect Inference Algorithm

We present our algorithm using *Core DPJ* [6], 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 2.1, i.e., basic region names with no region parameters or nesting. Then we build up the language to add region parameters and region path lists. We also discuss how to handle owner regions, array regions, and inheritance. Finally, we discuss how to adapt the algorithm for use with other languages.

3.1 Basic Region Names

We start by showing how to infer effects for Core DPJ with basic region names, i.e., with no region parameters or nested regions. Figure 3.1 shows the syntax of the initial language. The algorithm consists of two phases, constraint generation and constraint solving.

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

Figure 3.1: Syntax of Core DPJ with basic region names. C, f, m, x and r are identifiers.

3.1.1 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 3.1.2) 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 3.2 shows the constraint generation rules for the simplified language. The rules are similar to the ones for typing DPJ expressions [6], except that we do not check assignments or method formal parameter bindings for soundness (we assume that full DPJ type checking has been done as a separate pass).

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

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

$$\text{(INVOKE)}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : C, K_1 \quad \Gamma \vdash e_2 : T, K_2 \quad \mathit{method}(C, m) = \mu \\ \mu = T_r\ m(T_x\ x)\ \{\ e\ \}\end{array}}{\Gamma \vdash e_1.m(e_2) : T_r, K_1 \cup K_2 \cup \{\mathit{invokes}\ \mu\}}$$

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

Figure 3.2: Rules for computing the constraints generated by an expression.

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 $\mathit{method}(C, m)$ means the method named m defined in class C, while $\mathit{field}(C, f)$ 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

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

As an example, we show how to generate the constraints for the bodies of methods `setMass`, `setForce` and `initialize` in Figure 2.1. 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`.

3.1.2 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 $E_\mu =$

effects(K_μ), where the function *effects* extracts the read and write constraints (i.e., the effects) from K_μ. As an example, from Figure 2.1, the constraints *invokes* `setMass` and *invokes* `setForce` generate the effects *writes* `Mass` and *writes* `Force`.

3.2 Region Parameters

This section extends Core DPJ by adding region parameters. Figure 3.3 shows the new syntax.

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

Figure 3.3: New syntax of Core DPJ with region parameters. P is an identifier.

3.2.1 Constraint Generation

Figure 3.4 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.

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

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

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

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

Figure 3.4: 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 2.2. According to rule ASSIGN, line 8 generates the constraint *writes* `P`. In line 13, 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` $\mapsto$ `L`$\}\}$. Line 15 generates similar constraints, using region `R` instead of `L`.

3.2.2 Constraint Solving

The constraint solving phase is identical to the one described in Section 3.1.2, except that the algorithm applies substitutions θ in resolving *invokes* constraints:

1. For each method μ, for each constraint ($invokes\ \mu'\ where\ \theta$) 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 algorithm terminates for the same reason given in Section 3.1.2.

Figure 3.5 illustrates the constraints and effects inferred by each iteration of the algorithm on the method `setMassOfChildren` in Figure 2.2. For brevity, we show only the effects coming from `left.setMass()`. The effect of method `setMass` is summarized as *writes* `P` in iteration 0 (just before execution of step 3). In iteration 1, the *invokes* effect leads the algorithm to infer the effect *writes* `L` by applying the substitutions `P` $\mapsto$ `L` on the effect of `setMass`. The algorithm does not infer any new effects in iteration 2, so it terminates after the second iteration.

	Iteration 0	Iteration 1
Effects	*reads* `L`	*writes* `L`
Constraints	*invokes* `setMass` *where* {`P` $\mapsto$ `L`}	

Figure 3.5: Effects and constraints inferred in each iteration of the algorithm for method `setMass-OfChildren` in Figure 2.2

3.3 Region Path Lists (RPLs)

This section adds RPLs to Core DPJ. Only the syntax for regions, shown in Figure 3.6, 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 3.6: Syntax of RPLs.

Constraint generation is the same as explained in Section 3.2.1. However, we need to extend the solving phase to handle recursion that would not terminate if we naively applied the algorithm from Section 3.2.2.

For example, that algorithm would not terminate on the code in Figure 2.3, 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 2.3.

3.3.1 Algorithm Description

Figure 3.7 shows the modified constraint solving algorithm. 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.

However, instead of just adding an *invokes* constraint, we first check whether the constraint is *recursive* and has an *expanding substitution*. A constraint $(invokes\ \mu\ where\ \theta) \in K_{\mu'}$ is recursive if $\mu = \mu'$, i.e., the method includes its own effects, through a chain of one or more invocations. A substitution $P \mapsto R$ is expanding if 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. If the constraint is recursive and has an expanding substitution, then we summarize the effects of K_μ by replacing each appearance of P in *effects*(K_μ) with $P : R : *$, and we add all new effects generated this way back into K_μ. In line 5, the function *summarize* takes $P \mapsto R$ to $P \mapsto R : *$.

If the constraint is either not recursive or not expanding (line 6), then we add the effects of $E_{\mu'}$ to E_μ after applying the substitution θ. We also add the non-recursive or non-expanding *invokes* constraints of $K_{\mu'}$ to K_μ, again after applying θ (line 10), so we can continue down the call graph until we hit an expanding recursion.

As before, all unions are up to redundant constraints and effects. For instance, once *writes* $R : *$ appears in K_μ, we never again add *reads* R or *writes* R to K_μ in line 5. Similarly, once *invokes* μ *with* $P \mapsto P : R$ appears in K_μ, we never add *invokes* μ *with* $P \mapsto P : R : R$ in line 10. This pruning ensures that the algorithm terminates (Section 3.3.3).

3.3.2 Example

Figure 3.8 illustrates the constraints and effects inferred by each iteration of the `repeat` loop in lines 1–11 of Figure 3.7 for the method `setMassForTree` in Figure 2.3. Iteration 0 lists the constraints and effects as of line 2 in Figure 3.7. In iteration 1, the algorithm detects the recursive *invokes* constraints with expanding substitutions, appends stars to these two substitutions, and applies the new substitutions on the effects discovered in iteration 0 to get the two new effects of iteration 1. The algorithm terminates after iteration 2 because it does not find any new effects in the second iteration.

Note that even though the effect summary *writes* `P:*` shown in Figure 2.3 is correct (i.e., it type-checks), DPJizer infers a more accurate (i.e., a more refined) summary. For this program, DPJizer infers *writes*

```
input  : Program $\mathcal{P}$ with region annotations
         Set $\mathcal{M}$ of methods
         Set $K_\mu$ of constraints for each method $\mu$
output: A set of effects, $E_\mu$, for each method $\mu$
1  repeat
2      foreach $\mu \in \mathcal{M}$ do
3          foreach $c = (invokes\ \mu'\ where\ \theta) \in K_\mu$ do
4              if $\mu' = \mu$ and $isExpanding(\theta)$ then
5                  $K_\mu \leftarrow K_\mu \cup summarize(\theta)(effects(K_\mu))$
6              else
7                  $K_\mu \leftarrow K_\mu \cup \theta(effects(K_{\mu'}))$
8                  foreach $c' = (invokes\ \mu''\ where\ \theta') \in K_{\mu'}$ do
9                      if $\mu'' \neq \mu'$ or not $isExpanding(\theta')$ then
10                         $K_\mu \leftarrow K_\mu \cup \theta(c')$
11 until no $K_\mu$ changes
12 foreach $\mu \in \mathcal{M}$ do
13     $E_\mu = effects(K_\mu)$
```

Figure 3.7: The inference algorithm for RPLs.

	Iteration 0	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 3.8: Effects and constraints inferred in each iteration of Figure 3.7 for the method `setMass-ForTree` in Figure 2.3.

`P, P:L:*, P:R:*`. The effect *writes* `P` comes from the write access in line 8. *writes* `P:L:*` comes from the recursive function in line 11. DPJizer recognizes the recursive traversal of the data structure, and partially specifies the affected regions as `P:L:*`. *writes* `P:R:*` comes from the recursive function in line 13.

3.3.3 Termination and Algorithmic Complexity

RPLs can become arbitrarily long when going under multiple region substitutions. We bound the length of RPLs to get readable effects. Usually, developers write RPLs of length at most three. We cut off RPLs longer than a certain limit and append a star.

There are only a limited number of RPLs of a certain maximum length. So, the total number of effects and constraints that can be added to each K_μ set is finite. Because, all K_μ sets are finite, the algorithm will finally terminate.

The algorithm is context-sensitive, so in some cases it will enumerate several call paths between two functions. While this could cause exponential complexity in the worst case, this is not likely to be a problem for realistic programs because it is very unlikely that the analysis will generate distinct constraints on exponentially many call paths. Besides, bounding the length of RPLs reduces the number of distinct call paths.

3.4 Other DPJ Features

We now show how we extended the algorithm described in Section 3.3 to handle the key remaining features of DPJ: arrays, owner regions, and inheritance.

3.4.1 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 3.4. The rules are also similar to the rules for array access typing shown in [6].

3.4.2 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 3.9 illustrates how to use owner regions to write parallel quicksort. Here 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 line 7.

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 line 11, the type of `segs.left` is `segs:DPJPartition.Left`, where `DPJPartition.Left` is a field region name (Section 2.1) 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 [13, 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.

```
 1  class QSort<region P> {
 2      final DPJArray<P> A in P;
 3      QSort(DPJArray<P> 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>) A.subarray(0, pivot);
27      right = (DPJArray<this:Right>) A.subarray(pivot, A.length - pivot);
28    }
29  }
```

Figure 3.9: Using owner regions to write quicksort.

Figure 3.10 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.

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 3.10: Core DPJ with owner regions.

Constraint generation works exactly as described in Section 3.2.1 except for rules LET and INVOKE, shown in Figure 3.11. 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 $z : R$ to $R' : *$, where the type of z is $C\langle R'\rangle$. Rule INVOKE is nearly identical to the one shown in Figure 3.4, 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 3.3.1.

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

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

Figure 3.11: Rules for generating constraints in Core DPJ with owner regions.

3.4.3 Inheritance

DPJ supports inheritance, e.g., `class B<P> extends A<R>`. 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 [6]. 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 3.7, between lines 10 and 11, 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.

3.5 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 [12, 8, 7], 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 [12].

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 $o = \mathit{owner}(z)$ 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 3.11), 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 $o = \mathit{owner}(z)$. (We could also replace $z.n$ with `under` o, as our LET rule does for DPJ, but this would be less precise in the context of JOE.) 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) writes this, under this+1 {
5           /* writes this */
6           this.data = data;
7           /* invokes update where {this.n ↦ this.n+1}*/
8           let z = next in
9               /* invokes update where {this.n ↦ z.n+0}*/
10              if (z != null) z.update(data);
11      }
12  }
```

Figure 3.12: Example of inferring effects for JOE.

Figure 3.12 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 line 9, which is adjusted to the effect in line 7 by the LET rule discussed above. At the end of initial constraint collection, the constraints are as shown in lines 5 and 7. In the first iteration of the solving algorithm, the expanding substitution shown in line 7 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 4. The algorithm then terminates because there are no new effects to add.

Chapter 4

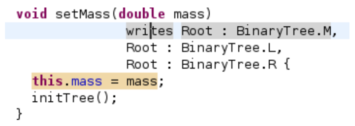
The DPJizer Tool

We have built an interactive tool, DPJizer, as an Eclipse plug-in, which incorporates the algorithm discussed in Section 3. 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 Fig. 4.1). 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 don't 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



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

Figure 4.1: The programmer selects an effect in the effect summary and DPJIZER highlights the statement that generated that effect.

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.

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 '*'.

Chapter 5

Implementation

We have implemented DPJizer as two separate Eclipse plugins: DPJizer *back-end* and DPJizer *front-end*. The front-end is the user interface and the back-end implements the effect inference algorithm. The front-end depends upon the back-end. Using the front-end, the DPJ developer selects a DPJ project to infer effects for. The DPJizer back-end computes the effects of all the methods in the selected project and shows the developer a preview of the changes that will be applied. Once the developer confirms the changes, the DPJizer front-end adds the effects computed by the DPJizer back-end to method signatures.

5.1 DPJizer Back-end

The DPJizer back-end is implemented as an extension of the DPJ compiler. And, the DPJ compiler itself is built upon the OpenJDK [25] implementation of the Java compiler. The DPJizer back-end reuses the parser and the semantic analyzer of the DPJ compiler and extends the DPJ compiler by adding two phases. The first phase generates constraints from the input DPJ files and stores the constraints in the *constraint repository*. The constraint repository is an object which contains constraints of each method in map data structures. The constraint repository is the output of the constraint generation phase and is the input of the second phase. The second phase solves the generated constraints and returns a mapping from each method to the set of effects of that method.

5.1.1 Constraint Generation

The compiler uses the visitor pattern [16] to traverse the abstract syntax tree (AST). A visitor class has a visit method for each node of the AST. We place the code for generating constraints of each AST node in its corresponding visit method. The visitor executes the visit method of each node as it is traversing the AST. The constraint generation phase uses the visitor pattern to generate constraints of each AST node.

The constraint generation phase performs three analyses:

1. Computes the point at which the effects of each method should be inserted.
2. Stores the environment valid at the scope of each method signature.
3. Computes the constraints generated by each method.

The DPJ compiler stores the offset of each AST node in the original DPJ file. And, the DPJ back-end uses this information to compute the insertion point of method effects. Method effect summaries should be inserted after the method signature and before the method body.

An environment contains all the symbols valid in each program scope. Constraints generated in each method should be translated to constraints valid in the scope of the signature of that method. For instance, if there is a *reads* constraint *reads* $R : [i]$ in the body of a method where i is not visible at the scope of the method signature, the constraint generator translates this constraint into *reads* $R : [?]$. Another example is an out of scope variable used as a region. For example, if variable z is not in the scope of the method signature, the *invokes* constraint *invokes* μ *where* $\{this \mapsto z\}$ will not be valid at the scope of the method signature either. The constraint generator coarsens the constraint by replacing z by $R : *$ where R is the owner region of z. So, the resulting *invokes* constraint will be *invokes* μ *where* $\{this \mapsto R : *\}$. The constraint generation visitor computes the environment valid at each AST node as it traverses the tree and makes copies of the environments at method signatures for future translations.

The visitor class in the constraint generation phase overrides two visit methods:

1. the visit method for method invocation nodes
2. the visit method for method definition nodes

The visit method for method invocations generates the corresponding *invokes* constraint with all the region parameter substitutions. For example, the method invocation `left.setMassForTree(mass)` at line 11 of Figure 2.3 generates the *invokes* constraint *invokes* `setMassForTree` *where* $\{\texttt{P} \mapsto \texttt{P:L}\}$ for method `setMassForTree`. So, the constraint generation visitor creates a constraint object recording this *invokes* constraint when it visits the corresponding method invocation node in the AST. This constraint is added to the set of constraints of method `setMassForTree` in the constraint repository.

An *invokes* constraint indicates that the effects of the caller should cover those of the callee under the collected region substitutions. Thus, it is important that the region substitutions translate the effects of the callee into a set of effects valid at the scope of the method signature not the call site. As the region substitutions recorded at the call site might involve local regions, the visit method for method definition nodes translates the generated *invokes* constraints into constraints valid at the scope of the method signature.

While visiting a method definition node, the effect insertion point for that method is computed and stored. Besides, a snapshot of the environment valid at the scope of the signature of that method is taken and stored in a map from methods to their environments.

When executing the constraint generation phase, the set of effects of bodies of all methods have already been computed by the compiler. So, the constraint generation phase does not need to visit AST nodes for assignment, field access, ... The visit method for a method definition node generates all the three kinds of constraints generated by that method in three steps. First, the visit method transforms the set of effects of the method body into a set of corresponding read and write constraints valid in the environment of the signature of that method. Second, the set of invocation effects generated in the method body are transformed into constraints valid in the scope of the method signature. Third, the visit method generates an *isOverriddenBy* constraint if the method being visited overrides any other method.

5.1.2 Constraint Solving

The effect inference algorithm is described in Chapter 3. And, the constraint solver is the part of the DPJizer back-end that implements this algorithm. The constraint solver is the second phase we have added to the DPJ compiler. It takes as input the set of constraints collected for each method. These constraints are stored in the constraint repository. The constraint solver takes the constraint repository and comes up with a set of method effect summaries that satisfy all the constraints.

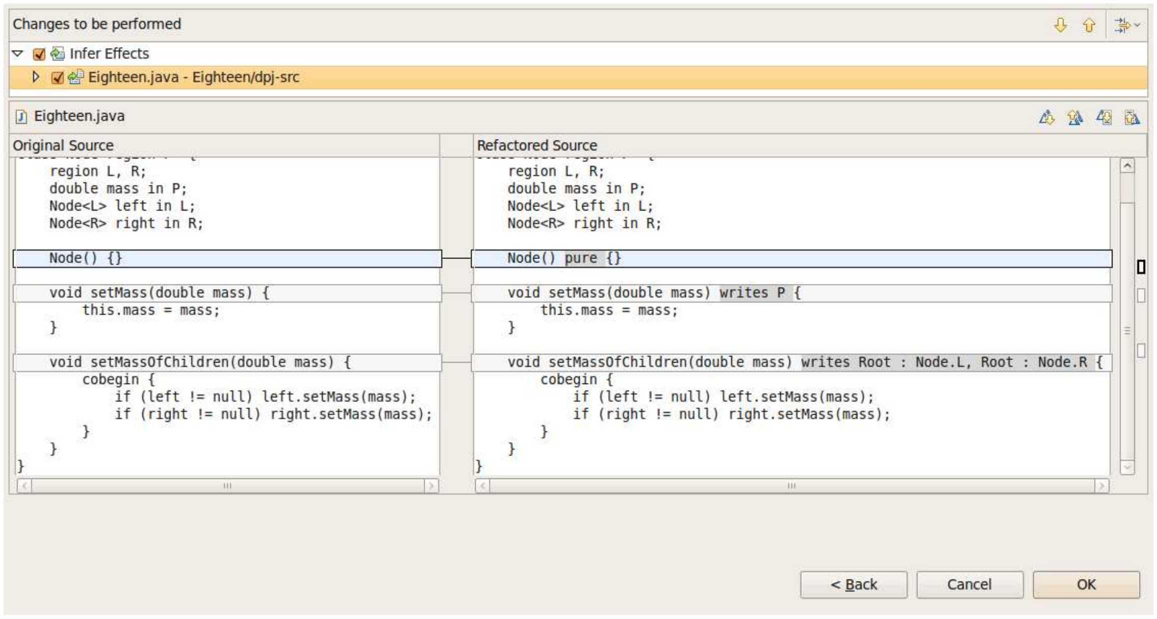
Having solved the constraints into effects, the constraint solver creates the method effect summary strings. For example, if the solver has inferred two effects *reads* R1 and *reads* R2 for a method μ, it converts these two effects into a single string "`reads R1, R2`" and associates this string to the object corresponding to the method μ in a map. The object of method μ contains the name and the offset at which the effects of that method should be inserted in the input DPJ files. This map from methods to effects is the input to the DPJizer front-end.

5.2 DPJizer Front-end

The DPJizer front-end implements the user interface of DPJizer. The developer selects a DPJ project on which to run the effect inferencer. The DPJizer front-end invokes the DPJizer back-end. The DPJizer back-end runs the effect inference algorithm and returns a map associating each method to its effect summary. Then, the DPJizer front-end looks up the insertion point of each effect summary in the map and creates the set of textual changes. These textual changes insert effects into the method effect summaries.

The DPJizer front-end is implemented using the Eclipse LTK framework. LTK is a framework for implementing transformations in Eclipse. Eclipse refactorings are implemented using LTK. Developers are already familiar with standard user interface for applying refactorings in Eclipse. So, implementing DPJizer using the same infrastructure leads to a user interface that developers know how to use.

Figure 5.1 shows a screenshot of the DPJizer front-end user interface. In this screenshot, a preview of the set of changes that are going to be applied is shown to the programmer. This dialog lets the programmer browse the effects that DPJizer has inferred for each method before he confirms the changes.


Changes to be performed

Infer Effects

Eighteen.java - Eighteen/dpj-src

Eighteen.java

Original Source

```
    region L, R;
    double mass in P;
    Node<L> left in L;
    Node<R> right in R;

    Node() {}

    void setMass(double mass) {
        this.mass = mass;
    }

    void setMassOfChildren(double mass) {
        cobegin {
            if (left != null) left.setMass(mass);
            if (right != null) right.setMass(mass);
        }
    }
}
```

Refactored Source

```
    region L, R;
    double mass in P;
    Node<L> left in L;
    Node<R> right in R;

    Node() pure {}

    void setMass(double mass) writes P {
        this.mass = mass;
    }

    void setMassOfChildren(double mass) writes Root : Node.L, Root : Node.R {
        cobegin {
            if (left != null) left.setMass(mass);
            if (right != null) right.setMass(mass);
        }
    }
}
```

Figure 5.1: The programmer selects a project. DPJizer computes the effects of all the methods in that project and previews the changes to the programmer to get his confirmation.

Chapter 6

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:** Is the inference accurate? What is the granularity of the inferred effects?

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 6.1 provides *quantitative* answers, while the survey (Section 6.2) provides *qualitative* answers.

6.1 Case Study

6.1.1 Methodology

Table 6.1 lists the programs that we used as case studies. 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 and left only the region annotations, and we used DPJizer to infer the method effects.

To answer the first question (usefulness), we report the number of effects that programmers wrote manually, as well as the number of effects that DPJizer inferred automatically. To answer the second question (accuracy) we compared the effects written manually with the effects inferred by DPJizer.

6.1.2 Quantitative Results

Q1: Is DPJizer useful? From Table 6.1 one can see that if the programmers had used DPJizer to infer the method effects, they would have saved writing 201 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.

Also, note that in some cases (e.g., Barnes-Hut, KMeans), DPJizer generated many more effects than the programmers. This is because the programmers put in effects only for methods that were transitively

Program	**SLOC**	**Effects Inferred**	
		by Programmer	**by DPJizer**
Barnes-Hut	682	46	145
IDEA	228	6	6
K-means	501	3	42
ListRanking	105	4	30
MergeSort	295	17	28
MonteCarlo	2877	38	273
QuadTree	117	11	28
QuickSort	144	12	13
StringMatching	373	61	60
SumReduce	57	3	12
Total	**5379**	**201**	**637**

Table 6.1: Programs used as case studies. Program size is given in non-blank, non-comment lines of source code, counted by *sloccount*. The last two columns show the number of effects written by programmers or inferred by DPJizer.

invoked in a parallel construct. Because DPJizer infers effects for *all* methods in a program, these effects can serve for future parallelization of new code fragments, thus encouraging an incremental parallelization approach. In addition, a programmer can look over the inferred effects to detect patterns of data accesses.

Q2: Is the inference accurate? We carefully analyzed the programs in Table 6.1 and compare the effects written by programmers with the effects inferred automatically by DPJizer. Since programmers did not write effects for all methods, as explained above, we can only compare the effects for the methods that were annotated. Table 6.2 shows the number of inaccurate effects, i.e., effects that are too coarse-grained in comparison with the effects inferred by DPJizer. Note that in all cases, DPJizer infers effects that are the same as, or more precise (i.e., finer-grained) than, those written by the programmer: although DPJizer must be conservative, it is also quite accurate. Programmers can be at least as accurate if they choose, but sometimes choose to summarize effects, e.g., if they think the coarser effects will not inhibit parallelism.

Table 6.2 shows three sources of inaccuracy in the manually inferred effects. First, some of these 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.

Third, some manually inferred effects are redundant. For example, the programmer may specify *reads* `R` *writes* `R`, but the read effect is subsumed by the write. Alternatively, the programmer inferred *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

Program	**# of Effects Too Coarse By**		**# of Redundant**
	keyword	**region**	**Effects**
Barnes-Hut	1	0	3
IDEA	0	0	0
K-means	0	0	0
ListRanking	1	2	0
MergeSort	0	4	0
MonteCarlo	1	3	6
QuadTree	1	2	1
QuickSort	0	3	3
StringMatching	5	24	10
SumReduce	0	0	2
Total	**9**	**38**	**25**

Table 6.2: Number of inaccurate effects written by programmers. We report effects that are too coarse-grained by keyword (e.g., programmer wrote *writes* `R` instead of *reads* `R`), by region (e.g., programmer wrote *reads* `R:*` instead of *reads* `R`). Last column shows the number of redundant effects (e.g., programmer wrote both *reads* `R` *writes* `R`).

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 summaries do not contain redundant effects.

6.2 User Survey

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

1. We elided the effect annotations on the programs previously parallelized by these users, but retained the region annotations they had inserted.
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 the tool before the survey).

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 compile 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 recommend 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 8.)

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."

Chapter 7

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 [23, 18] use summaries, as do several systems combining object ownership with some form of effects [10, 12, 7]. Our work presents an algorithm and tool that can be used to infer such summaries.

Effect inference. The seminal work on inferring effects is from Jouvelot and Gifford [21]. They use a technique called *algebraic reconstruction* to infer types and effects in a mostly functional language. Talpin and Jouvelot [30] 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 [4] present an inference algorithm for Greenhouse and Boyland's effect system [18]. The features they consider are similar to the smallest subset of Core DPJ we covered in section 3.1, plus support for unique reference annotations and a limited form of nesting. Again there is no unbounded nesting.

Side-effect analysis [3, 28, 29, 27] 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 [14] 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 [32, 11, 2, 19]. Several researchers have studied the problem of inferring types or type qualifiers for imperative programs with references. Kiezun et al. [22] show how to infer Java generic parameters and arguments. Agarwal and Stoller show how to do type inference for parameterized race-free Java [1]. Quinonez et al. [26] 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 [31] present a type inference algorithm for deterministic parallelism using *linear types* supplemented with fractional permissions [9].

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.

Chapter 8

Conclusions

We have presented an effect inference algorithm and a tool, DPJizer, that ease the burden of writing DPJ programs. The DPJizer algorithm is also applicable to other effect systems that rely on method effect summaries. As future work, we plan to extend the capabilities of DPJizer 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.

References

[1] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, 2004.

[2] Anindya Banerjee, Michael Barnett, and David A. Naumann. Boogie meets Regions: A verification experience report. In *VSTTE*, 2008.

[3] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, 1979.

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

[5] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel Programming Must Be Deterministic By Default. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[6] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. to appear in OOPSLA 2009.

[7] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[8] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

[9] John Boyland. Checking interference with fractional permissions. *SAS*, 2003.

[10] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. *OOPSLA*, 2007.

[11] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *PLDI*, 2004.

[12] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.

[13] David G. Clarke et al. Ownership types for flexible alias protection. *OOPSLA*, 1998.

[14] Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 1997.

[15] DPJ homepage. http://dpj.cs.uiuc.edu.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, 2005.

[18] Aaron Greenhouse and John Boyland. An object-oriented effects system. *ECOOP*, 1999.

[19] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. *PLDI*, 2002.

[20] R. T. Hammel and D. K. Gifford. FX-87 performance measurements: Dataflow implementation. Technical Report MIT/LCS/TR-421, 1988.

[21] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *POPL*, 1991.

[22] Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, 2007.

[23] K. Rustan M. Leino et al. Using data groups to specify and check side effects. 2002.

[24] J. M. Lucassen et al. Polymorphic effect systems. In *POPL*, 1988.

[25] OpenJDK homepage. http://openjdk.java.net/.

[26] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP*, 2008.

[27] Atanas Rountev. Precise identification of side-effect-free methods in java. In *ICSM*, 2004.

[28] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *TOPLAS*, 2001.

[29] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, 2005.

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Prog.*, July 1992.

[31] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 2008.

[32] Mads Tofte and Lars Birkedal. A region inference algorithm. *TOPLAS*, 1998.