DPJizer: A Tool for Automated Java to DPJ Transformation

To appear in Automated Software Engineering 2009

Mohsen Vakilian (mvakili2@illinois.edu), Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, Ralph Johnson
An Effect System

```java
public FileWriter(String) throws IOException
```
Effect Systems

- Effect systems are important for reasoning about side effects.
- Effects are enforceable contracts.
- Effects are machine-checkable documentation.
- Effect system enable composability and modular reasoning.
Effect Systems for Parallel Programming

- Effects systems can be used to reason about correctness of parallel programs.
- **DPJ** = Deterministic Parallel Java
DPJ Effect System

• A **region** is a name for a collection of locations.

• An **effect** is a **read** or **write** operation on a region.

• Compiler checks that parallel tasks are **noninterfering**.
Region and Effect

• Programmer assigns regions to fields.
• Programmer summarizes effects of method bodies.
double mass in Mass;
double force in Force;
setMass(double) writes Mass;
setForce(double) writes Force;

setMass() writes mass

setForce() writes force
Why have not effect systems taken off yet?

• Writing annotations manually is **error-prone** and **tedious**.

• Whenever the code changes annotations have to be updated.

• On average, programmers annotate **8%** of lines of DPJ source code.
• DPJizer makes effect systems practical.
• DPJizer infers a summary of effects for each method.
DPJizer Preview Dialog

Changes to be performed

- Infer Effects

BinaryTree.java - TestProject/dpj-programs/01-flat-regions

Original Source

```java
public class BinaryTree {
    region L, R, M;
    BinaryTree left in L;
    BinaryTree right in R;
    double mass in M;

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

    void initTree() {
        left = new BinaryTree();
        right = new BinaryTree();
    }
}
```

Refactored Source

```java
public class BinaryTree {
    region L, R, M;
    BinaryTree left in L;
    BinaryTree right in R;
    double mass in M;

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

    void initTree() {
        left = new BinaryTree.L, Root : BinaryTree.M, Root : BinaryTree.R {
            left = new BinaryTree();
            right = new BinaryTree();
        }
    }
```
Highlight Source of Effects

```java
void setMass(double mass) {
    writes Root : BinaryTree.M,
    Root : BinaryTree.L,
    Root : BinaryTree.R {
        this.mass = mass;
        initTree();
    }
}
```
Region Names Ex. (1/2)

class Node {
    region Mass, Force;
    double mass in Mass;
    double force in Force;

    void setMass(double mass) writes Mass {
        this.mass = mass; // Effect: writes Mass
    }

    void setForce(double force) writes Force {
        this.force = force; // Effect: writes Force
    }
}
void initialize(double mass, double force)

writes Mass, Force {
    cobegin {
        this.setMass(mass); // Effect: writes Mass
        this.setForce(force); // Effect: writes Force
    }
}
}
Input: DPJ Program Annotated with Region Information

DPJizer

Constraint Generator

Constraints: reads, writes, and invokes

Constraint Solver

Output: DPJ Program with Region and Effect Annotations
class Node {
    region Mass, Force;
    double mass in Mass;
    double force in Force;

    void setMass(double mass) {
        this.mass = mass; // Constraint: writes Mass
    }

    void setForce(double force) {
        this.force = force; // Constraint: writes Force
    }
}
void initialize(double mass, double force) {
    cobegin {
        // Constraint: invokes setMass()
        this.setMass(mass);
        // Constraint: invokes setForce()
        this.setForce(force);
    }
}
Constraint Generation Output

initialize()

does not invoke setForce()

does not invoke setMass()

setMass() invokes setMass()

setForce() invokes setForce()

writes Mass

writes Force
Constraint Solving

- initialize() writes Mass, Force
- setForce() invokes initialize()
- setMass() invokes initialize()
- setMass() writes Mass
- setForce() writes Force

writes Force
setForce()
Region Parameters

• Region names distinguish different fields of an object.
• Region parameters distinguish different object instances.
Region Parameters Ex. (1/6)

class Node<region P> {
    region L, R;
    double mass in P;
    Node<L> left in L;
    Node<R> right in R;
    //...
}

class Node<region P> {
    region L, R;
    double mass in P;
    Node<L> left in L;
    Node<R> right in R;
    //...
}

Node<S>

mass in S

left

Node<L>

mass in L

right

Node<R>

mass in R
Region Parameters Ex. (3/6)

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

    void setMass(double mass) {
        this.mass = mass;
    }
}
class Node<region P> {
    region L, R;
    double mass in P;
    Node<L> left in L;
    Node<R> right in R;

    void setMass(double mass) \texttt{writes} P {
        this.mass = mass; \texttt{// Effect: writes P}
    }
}
void setMassOfChildren(double mass) {
    cobegin {
        if (left != null) left.setMass(mass);

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

    // Effect: writes R
    if (right != null) right.setMass(mass);
  }
}
Extending the Invokes Constraint

- The invokes constraint records region substitutions.
- $C<\text{region } P> \{ \quad /* \ldots */ \quad \}$
- `new C<R>().m()`
- `invokes m() where \{P \rightarrow R\}`
Constraints in Region Parameters Ex. (1/2)

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

    void setMass(double mass) {
        this.mass = mass; // Constraint: writes P
    }
}
void setMassOfChildren(double mass) {
    cobegin {
        // Constraint: reads L,
        // invokes setMass() where \{P \to L\}
        if (left != null) left.setMass(mass);

        // Constraint: reads R,
        // invokes setMass() where \{P \to R\}
        if (right != null) right.setMass(mass);
    }
}
setMassOfChildren() invokes setMass() where \{P \rightarrow R\}

setMass() invokes setMass() where \{P \rightarrow L\}

setMassOfChildren() reads L, R

setMass() writes P
setMassOfChildren() invokes setMass() where \( \{P \rightarrow R\} \)

setMass() invokes setMass() where \( \{P \rightarrow L\} \)

writes L, R

writes P

writes P
Region Path Lists (RPLs)

• An RPL is a colon separated list of region names e.g. P : L : R : R
• \{ P : R1 : R2, P : R3 \} \subseteq P : *
• RPLs are useful to express tree-like recursive updates.
RPL Ex.

class Node<region P> {
    region L, R;
    double mass in P;
    Node<P:L> left in P:L;
    Node<P:R> right in P:R;
}

class Node<region P> {
    region L, R;
    double mass in P;
    Node<P:L> left in P:L;
    Node<P:R> right in P:R;
}

Node<S> mass in S

Node<S:L> mass in S:L

Node<S:L:L> mass in S:L:L

Node<S:L:R> mass in S:L:R

Node<S:R:L> mass in S:R:L

Node<S:R:R> mass in S:R:R

Node<S:R:L> mass in S:R:L

Node<S:R:R> mass in S:R:R
class Node<region P> {
    region L, R;
    double mass in P;
    Node<P:L> left in P:L;
    Node<P:R> right in P:R;
    void setMassForTree(double mass) {
        this.mass = mass;
        cobegin {
            if (left != null) left.setMassForTree(mass);
            if (right != null) right.setMassForTree(mass);
        }
    }
}
class Node<region P> {
    region L, R;
    double mass in P;
    Node<P:L> left in P:L;
    Node<P:R> right in P:R;
    void setMassForTree(double mass) writes P:* {
        this.mass = mass; // Effect: writes P
        cobegin {
            // Effect: writes P:L:*
            if (left != null) left.setMassForTree(mass);
            // Effect: writes P:R:*
            if (right != null) right.setMassForTree(mass);
        }
    }
}

Constraints in RPL Ex.

class Node<region P> {
    region L, R;
    double mass in P;
    Node<P:L> left in P:L;
    Node<P:R> right in P:R;
    void setMassForTree(double mass) {
        this.mass = mass; // Constraint: writes P
cobegin {
//Constraint: reads P:L,invokes setMassForTree where {P→ P:L}
        if (left != null) left.setMassForTree(mass);
//Constraint: reads P:R,invokes setMassForTree where {P→ P:R}
        if (right != null) right.setMassForTree(mass);
    }
}
}
Constraint Generation Output

invokes \texttt{setMassForTree} where \{P \rightarrow P: L\}

\begin{itemize}
  \item reads P:L, P:R
  \item writes P
\end{itemize}

\texttt{setMassForTree()}

invokes \texttt{setMassForTree} where \{P \rightarrow P: R\}
Constraint Solving

invokes setMassForTree where \( \{P \rightarrow P : L : *\} \)

reads P:L, P:R
writes P

setMassForTree()

invokes setMassForTree where \( \{P \rightarrow P : R : *\} \)
invokes setMassForTree where \( \{ P \rightarrow P : L : * \} \)

writes \( P \),
writes \( P : L : *, P : R : * \)

invokes setMassForTree where \( \{ P \rightarrow P : R : * \} \)
Evaluation

• Research Questions
  • Is DPJizer useful?
    • Does it alleviate the burden of writing effect annotations?
  • Are the inferred effects precise?
    • Do the inferred effects enable the compiler to prove determinism of the program?
Evaluation Methodology

• Quantitative
  • We compared the inferred effects to manual effects.

• Qualitative
  • We conducted a survey on DPJ programmers.
Problems of Manual Effects

- It is **time consuming** to write the effects manually.
- Programmers might write **redundant** effects
  - e.g. reads P, writes P
- Programmer might write **coarse-grained** effects
  - e.g. writes P:* instead of writes P
Comparing Manual to Inferred Effects

- We used DPJizer to infer method effect summaries of 11 programs.
  - Barnes-Hut, CollisionTree, IDEA, K-means, ListRanking, MergeSort, MonteCarlo, QuadTree, QuickSort, StringMatching, and SumReduce

<table>
<thead>
<tr>
<th>Total SLOC</th>
<th>Total # of Manually Written Effects</th>
<th>Total # of Effects Too Coarse By Keyword</th>
<th>Total # of Effects Too Coarse By Region</th>
<th>Total # of Redundant Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>5046</td>
<td>406</td>
<td>17</td>
<td>43</td>
<td>24</td>
</tr>
</tbody>
</table>
Summary

• DPJizer is an interactive tool for inferring effect annotations given the region annotations.
• DPJizer infers effects that are sufficiently precise and fine-grained.
• We are working on analyses to infer most of the region annotations as well.
Acknowledgement

This work is funded by Microsoft and Intel through the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois and by NSF grants 07-02724, 07-20772, 08-33128 and 08-33188.
References

• http://dpj.cs.illinois.edu
• Mohsen Vakilian et al., Inferring Method Effect Summaries for Nested Heap Regions, to appear in ASE 2009
• Robert Bocchino et al., A Type and Effect System for Deterministic Parallel Java, OOPSLA 2009.