Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks

Robert Bocchino, Vikram Adve
University of Illinois at Urbana-Champaign

http://dpj.cs.uiuc.edu/

UPCRC Illinois is sponsored by Intel Corporation and Microsoft Corporation.

Parallel Frameworks Are Very Useful...

Provide generic algorithm such as map, reduce, scan

- User fills in missing pieces
- Overrides methods specified in framework API

Advantages

- Easy for user (write sequential code)
- Reusable code for common patterns
- Separation of concerns: parallelism vs. domain expertise

Real world examples are emerging

- Algorithm templates in TBB
- `ParallelArray` framework in Java

UPCRC Illinois
Universal Parallel Computing Research Center

...But They Are Also Dangerous!

User can provide *arbitrary method* for parallel execution

What if the user's method...

- Updates a global variable?
- Updates a collection of objects with duplicate elements?
- Follows a link from one object in a collection to another?
- Etc., etc.

***Current approach*: Informal, unchecked specification**

- "Don't do that!"
- This is not satisfactory

Goal: Make Frameworks Safer

Idea: Use *design by contract* for framework APIs

- Formally specify and check conditions on use
- This idea has been well studied for other specifications
  - E.g., make sure that you follow the server protocol
- But not for shared-memory parallel frameworks

Key challenges

- Ensure that containers don't have duplicate objects
- Constrain the effects of user-supplied methods
- Make the types and effects sound yet generic enough to be useful

Contributions

We build on Deterministic Parallel Java (DPJ)

- Rich region-based type and effect system
- Leverage type information already built into framework APIs

We show how to...

- Use DPJ "off the shelf" to write safe APIs for container frameworks
  - Use local parameters, set properties, etc. to constrain aliasing and effects
- Extend the DPJ type system to add generic types and effects
  - Effect variables, type region parameters
- Make different forms of verification interoperate
  - Use program logic or testing to verify framework *internals*
  - Use the type system to verify framework *uses*

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

A Safe Container API Specialized to List Nodes

A Safe Container API with Generic Types and Effects

Verifying the Implementation

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

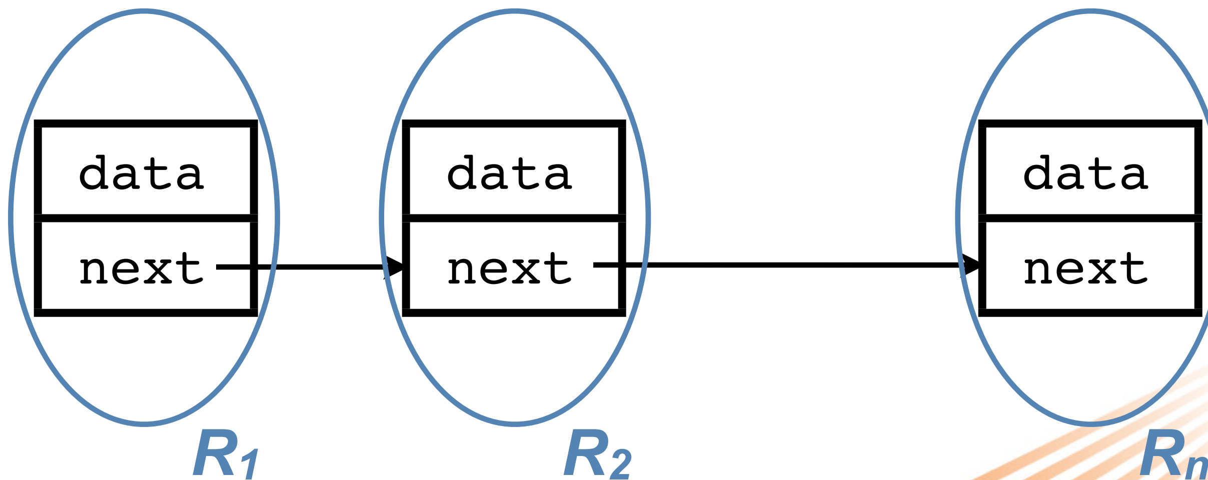
A DPJ List Node Class

Class is parameterized by region **R**

```
class ListNode<region R> {
  int data in R;
  ListNode<*> next in R;
}
```

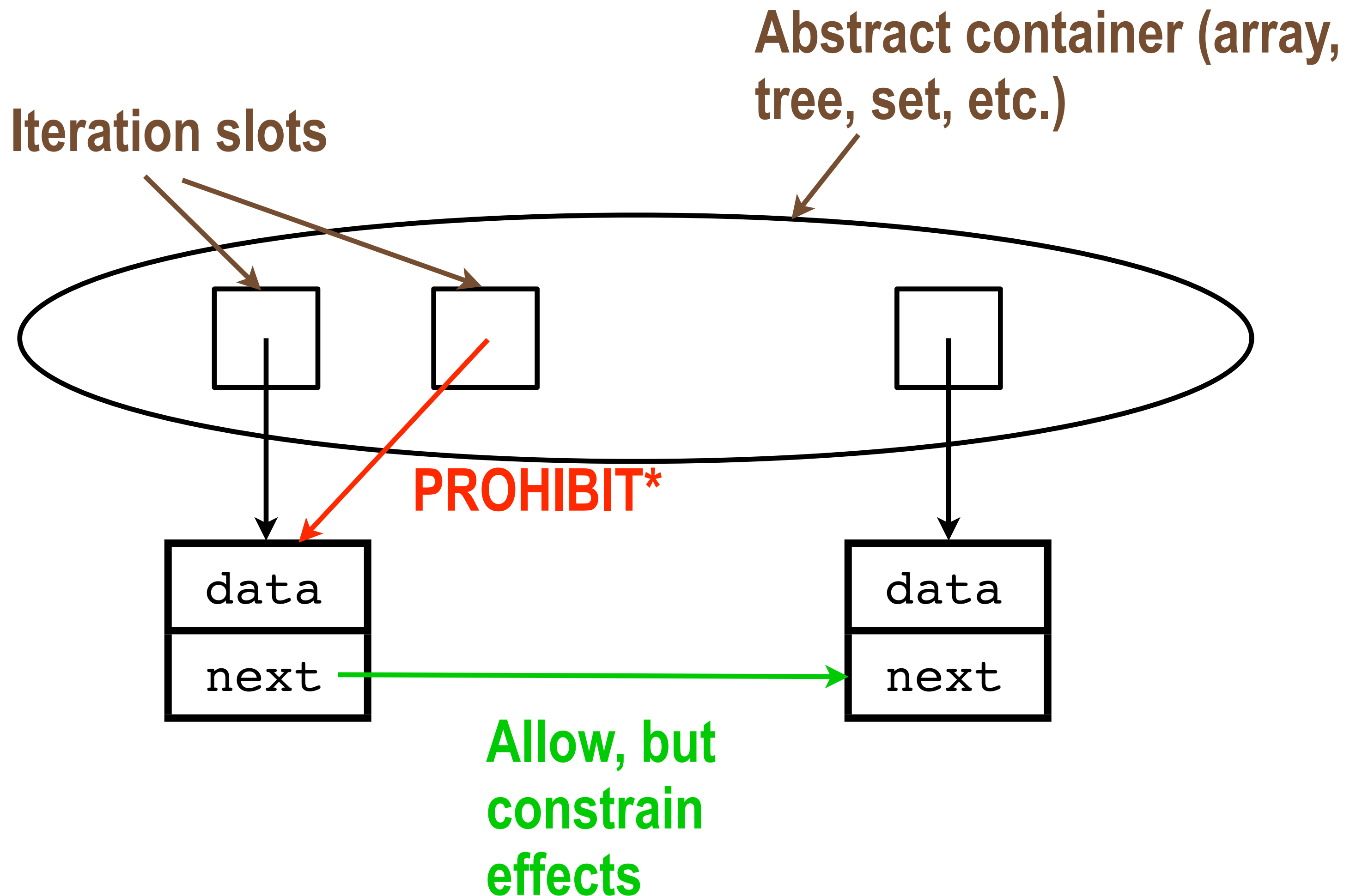
Data resides in **R**

next can point to any **ListNode**



UPCRC Illinois
Universal Parallel Computing Research Center

A List Node Container



*We say the container is *internally linear*

UPCRC Illinois
Universal Parallel Computing Research Center

Maintaining Linearity

Inductive argument based on API

- A container starts empty (trivially linear)
- Only API operations can change state (encapsulation)
- Every API operation is linearity-preserving

Kinds of API operations

- Adding elements
- Transforming elements
- Moving elements around (e.g., tree rebalancing)
- Deleting elements

Needs careful API design

UPCRC Illinois
Universal Parallel Computing Research Center

Strategies for Adding Elements

Making one linear container from another

- E.g., tree from array or set
- Constructor API says incoming container must be linear
- Special case in DPJ = *index-parameterized array*

Populating container with user-supplied factory method

- Framework calls method once on each slot
- API must ensure the "factory method" really returns a fresh object!

Backing container with a set

- Sets are linear by definition
- Container semantics gives linearity "for free"

UPCRC Illinois
Universal Parallel Computing Research Center

User-Supplied Factory Method

Framework API

```
public interface NodeContainer<region N, C> {
  public interface NodeFactory {
    public <region R>ListNode<R> create(int i) pure;
  }
  ...
}
```

In interface, return type uses method region parameter...

User Code

```
public class MyFactory implements NodeContainer.NodeFactory {
  public <region R>ListNode<R> create(int i) pure {
    return new ListNode<R>(i, null);
  }
}
NodeContainer<N,C> cont =
  new NodeArray<N,C>(new MyFactory(), 10)
```

...so in user implementation, returned object must be freshly created

UPCRC Illinois
Universal Parallel Computing Research Center

Reasoning About Effects

Need to constrain effects of user-supplied methods

- Method should update only object(s) it is given

Conceptually this is straightforward (ordinary DPJ)

- Each element gets its own region
- Interface requires that method write at most under that region

But there's a catch

- If regions appear in the API, we can't swap elements between slots
  - Assigning `ListNode`<r_1> to `ListNode`<r_2> is unsound
  - General problem for region-based type systems (including DPJ)
- So instead we use the method region parameter trick
  - Write API's types and effects in terms of parameter `R`
  - Hide actual regions bound to `R` from user of API

UPCRC Illinois
Universal Parallel Computing Research Center

Effects, Continued

Framework API

```
public interface NodeContainer<region N, C> {
  public void performOnAll(Operation op) reads C writes N:*;
  public interface Operation {
    public <region R>void operateOn(ListNode<R> node) writes R:*;
  }
  ...
}
```

Subclass relation constrains effects

User Code

```
public class MyOperation implements NodeContainer.Operation {
  public <region R>void operateOn(ListNode<R> node) writes R {
     ++node.data;  // OK, writes R
     ++node.next.data;  // ERROR, writes *
  }
}
container.performOnAll(new MyOperation());
```

UPCRC Illinois
Universal Parallel Computing Research Center

Why Does This Work?

Framework ensures slot *i* has type `ListNode`$<R_i>$ s.t.

- R_i is under `N`, the node region of the container
- R_i`:*` # R_j`:*` for i =/= j

Therefore:

- (a) Effects of `performOnAll` are bounded by `writes N:*`
  - ⟹ the summarized effects are correct in the framework API
- (b) Effects of `operateOn` on slot *i* are bounded by `writes` R_i`:*`
  - ⟹ the effects are disjoint for different slots

Notice that

- These “hidden regions” are invisible to the user
- We could not show (a) or (b) with the type system alone

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

A Safe Container API Specialized to List Nodes

A Safe Container API with Generic Types and Effects

Verifying the Implementation

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Generic Effects

***Problem*: Superclass effects constrain subclass effects**

- Required for sound reasoning with polymorphism
- But then superclass writer must “know” all future subclass effects
- This is unsatisfactory for frameworks

Solution*: Use *effect variables

- Parameterize classes, methods by effect variables `E`
- Write method effects using `E`
- Provide actual effects for `E` when classes, methods are *instantiated*

Technical challenges

- Sound subtyping (see paper)
- Effect constraints

UPCRC Illinois
Universal Parallel Computing Research Center

Generic Effects, Continued

Framework API

```
public interface NodeContainer<region N, C> {
  public interface Operation<effect E> {
    public <region R>void operateOn(ListNode<R> node)
      writes R:* effect E;
  }
  public <effect E | E # reads C writes N:* effect E>
    void performOnAll(Operation<E> op)
      reads C writes N:* effect E;
  ...
}
```

`Operation` is effect polymorphic

`performOnAll` constrains the effect of the supplied operation

User Code

```
public class MyOperation implements
  NodeContainer.Operation<pure> {
    public <region R>void operateOn(ListNode<R> node)
      writes R {
        ++node.data;
    }
}
cont.<pure>performOnAll(new MyOperation());
```

effects supplied by user code must obey constraints

UPCRC Illinois
Universal Parallel Computing Research Center

Generic Types

A container of list nodes is nice, but...

***Problem*: Must name region parameter of contained type**

- E.g., `void operateOn(ListNode<R> node) writes R:*;`
- Can't do this if we replace `ListNode<R>` with generic type `T`!

Solution*: Introduce *type region variables

- Kind of like C++ template template parameters
- Refer to the first *n* region arguments of a generic type argument

```
class C<type T<region R>>
```

UPCRC Illinois
Universal Parallel Computing Research Center

Generic Types, Continued

Framework API

```
public interface LinearContainer<type T<region R1>,
  region C> {
    public interface Operation<type T<region R2>,
    effect E> {
      public <region R3>void operateOn(T<R3> node)
        writes R:* effect E;
    }
    ...
}
```

User Code

```
public class MyOperation implements
  LinearContainer.Operation<ListNode<N>, pure> {
    public <region R4>void operateOn(ListNode<R4>
      node) writes R {
        ++node.data;
    }
}
cont.<pure>performOnAll(new MyOperation());
```

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

A Safe Container API Specialized to List Nodes

A Safe Container API with Generic Types and Effects

Verifying the Implementation

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Verifying the Implementation

Can use any desired method

- Testing
- Model checking
- Program logic

Full formal verification (proof) is future work

For now, sketch how to do it

- Use API to prove properties of framework internals
- Glue proved properties to the type system
- Properties + type system $\Rightarrow$ composite proof of noninterference

UPCRC Illinois
Universal Parallel Computing Research Center

Array Implementation

API, not assignment rules, constrains these types

```
public LinearArray<type T<region R>, region C> {
  T<R:*>[]<C> arr;
  ...
  public interface Operation<type T<region R2>,
  effect E> {
    public <region R3>void operateOn(T<R3> node)
      writes R:* effect E;
    }
  }
  public <effect E | E # reads C writes R:* effect E>
    void performOnAll(Operation<E> op)
      reads C writes R:* effect E {
        foreach (int i in 0, array.length)
          op.operateOn(arr[i]);
    }
  ...
}
```

Use type and effect constraints to push through "normal" DPJ proof for this loop

UPCRC Illinois
Universal Parallel Computing Research Center

Array: Verifying Type Constraints

Constraints: Want...

- `arr[`i`]` has type `T`<R_i>
- R_i`:*` # R_j`:*` if i =/= j

Proof: Inductive API argument (as for linearity)

- Trivial for fresh container
- Only API operations update state
- Each API operation preserves the property (e.g., permutation)

DPJ Proof

- Augment type system with predicate *disjoint-rgn(*z_i*,* z_j*)*
- Prove it using API constraints
- Use it to push through type-based proof

UPCRC Illinois
Universal Parallel Computing Research Center

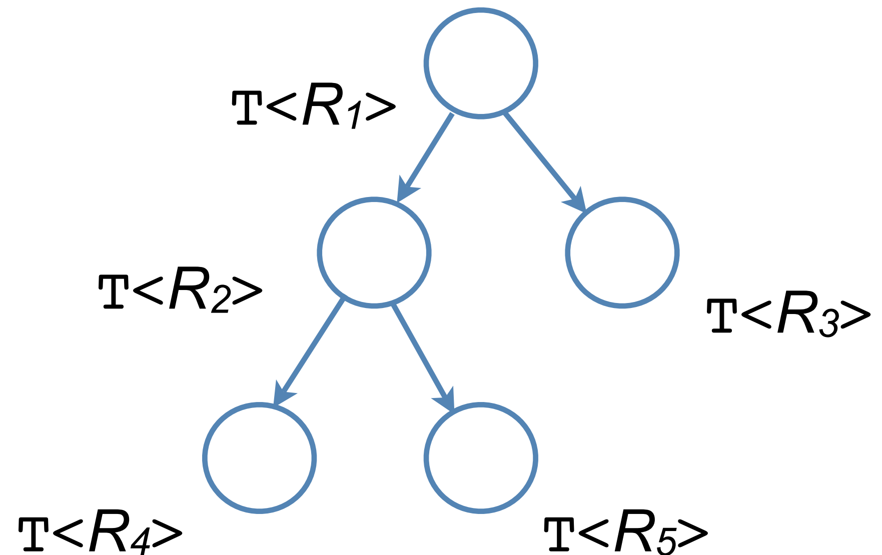
Tree Implementation

Same argument from API for data stored in nodes

Use logic to show

- Tree is a tree (shape)
- Parallel traversal touches disjoint nodes (traversal)

Replace recursive DPJ types with logic



R_i : * # R_j if i =/= j

UPCRC Illinois
Universal Parallel Computing Research Center

Outline

A Safe Container API Specialized to List Nodes

A Safe Container API with Generic Types and Effects

Verifying the Implementation

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Evaluation Overview

Wrote two parallel frameworks

- Array (based on Java ParallelArray)
- Tree (from scratch, inspired by tree algorithms)

Wrote two parallel algorithms

- *Monte Carlo* (Java Grande) using array framework
- *Barnes-Hut center of mass* (JOlden, SPLASH) using tree framework

Performed qualitative assessment

- *Expressivity:* Can we write the algorithms efficiently?
- *Ease of use:* For application writers (i.e., framework users)

Frameworks

Array framework

- `create`: User-supplied factory method
- `withMapping`: Array-to-array mapping $T_1[\,] \rightarrow T_2[\,]$
  - User supplies elementwise mapping function $T_1 \rightarrow T_2$
- `reduce`: Array-to-object mapping $T[\,] \rightarrow T$
  - User supplies binary mapping $(T, T) \rightarrow T$

Tree framework

- `buildTree`: Build a tree from a linear container
  - User supplies index function that determines which child to visit
- `visitPO`: Recursive parallel postorder tree traversal
  - User supplies visit method to apply to each node

UPCRC Illinois
Universal Parallel Computing Research Center

Algorithms

Monte Carlo financial simulation

- Map-reduce pattern
- Create many tasks, operate on them in parallel, reduce them

Barnes-Hut center of mass computation

- Recursive, divide-and-conquer tree update
- Octtree represents points in space
- Traverse tree and update each node with center of mass of subtree

Qualitative Assessment

Expressivity

- Safe frameworks express these algorithms well
- Combining arrays (e.g., concatenation) would be hard

Ease of use for application writers

- Region and effect annotations for API are sometimes tricky
- But user effect arguments are simple
  - `pure` or one or two read effects

Compared with "ordinary" DPJ

- User-written types are simpler
- Assignments are less restricted (e.g., reordering array)
- Code is more verbose

Outline

A Safe Container API Specialized to List Nodes

A Safe Container API with Generic Types and Effects

Verifying the Implementation

Evaluation

Conclusion

UPCRC Illinois
Universal Parallel Computing Research Center

Conclusion

Safe frameworks can help parallel programming

We show how to...

- Use DPJ "off the shelf" to write safe framework APIs
- Make the types and effects generic with novel techniques
- Combine type-based and other verification in a modular way

Future work

- Verify framework implementations
- Explore whether any more "glue" is needed to do composite proofs