UPMARC Uppsala Programming for Multicore Architectures Research Center

UPPSALA UNIVERSITET

Race-free Parallelism using Refined Ownership Types with Effects

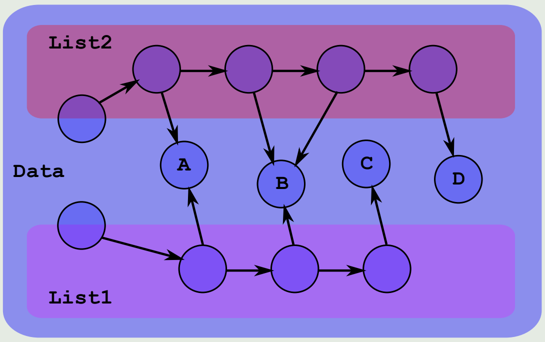
Elias Castegren
elias.castegren@it.uu.se

Johan Östlund
johan.ostlund@it.uu.se

Tobias Wrigstad
tobias.wrigstad@it.uu.se

Ownership Types and Effects

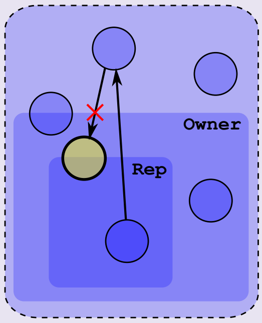
Ownership types allow partitioning the heap into disjoint regions which may be nested. Each object is "owned" by a region and is given explicit permission to reference objects in other regions. By annotating methods with effects - which regions they read and write - potential data-races can be detected statically.



- Both lists have elements that reside in the region Data.
- Mutating the elements of both lists in parallel is subject to data races.
- Each list has access to a private region of memory containing its own links and other private data.
- Parallel operations on separate regions can be performed without interference.

Ownership types allow expressing disjointness between different data structures, but cannot distinguish between elements in the same data structure. For example, a parallel map over List1 is free of races, but this can not be expressed with Ownership types without using inflexible region nesting.

Ownership Types Explained



References may pass outwards in the nesting hierarchy but not inwards.

Regions are named through parameters of the class or relative to the current **this**:

- **Rep** denotes the private region of the current **this**.
- **Owner** is the region containing the current **this**.

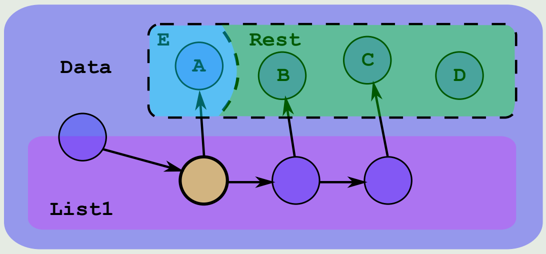
Refined Ownership Types

Refined ownership types extend ownership-based disjointness to allow static reasoning about objects in a single region. They work by introducing a local view of the region:

```
class List[Data]{
  Rep:Link[Data] first;
}
class Link[Data = E + Rest]{
  E:Object element;
  Owner:Link[Rest] next;
}
```

- In this code, the elements of a List-object reside in some region Data.
- The Link class refines Data into to E and Rest. The intuition is that Data is split into two disjoint regions; no object in E appears in Rest and vice-versa.
- The next-Link (and its successors) will have elements residing in Rest, and since Rest and E are disjoint, no alias of element can be reached by following next.

Refinement is a Local Property



- Here, the first Link has a local view of Data as disjoint regions E and Rest. A is in E but not in Rest.
- The second Link has a local view of Rest that splits it into a region containing only B (its element) and another owner containing C and D, and so forth.
- Another object can refine the same region differently (or not refine it at all), and still reference any object in the region.

Race-free Parallelism

```
class Node[Data = L + Root + R]{
  Root:Object element;
  Owner:Link[L] left;
  Owner:Link[R] right;

  void map(Function f) writes Data{
    par{
      left.map(f);          // writes L
      f.apply(element);// writes Root
      right.map(f);      // writes R
    }
  }
}
```

- Since L, Root and R are disjoint, writes to these regions cannot be conflicting.
- Race-freedom of statements in a **par**-block is checked statically at compile-time.
- The refinements are internal to the class. Local reasoning is enough to allow safe parallel access to objects in the refined region.

Key points

- We can give static guarantees that a data structure is tree-shaped and contains no duplicate elements.
- We can guarantee safety of parallel mutation of such data structures, including permutations (e.g. sorting or balancing).

Future Work

- Dynamically merging regions using disjointness information.
- Automatically inferring parallelism from disjointness information.