Alias Management with Ownership and Uniqueness

Johan Östlund, Elias Castegren, Stephan Brandauer and Tobias Wrigstad
Uppsala University, Sweden

UPMARC Uppsala Programming for Multicore Architectures Research Center

Aliasing is powerful

- Effective implementations, share a single copy of a datum and make in-place updates
- Model real-world scenarios with sharing

Aliasing is problematic

- Complicates programming and program reasoning
- Complicates verification and compiler optimisation
- Increasingly so in an in a parallel world

Aliasing must be controlled

- There is no formal theory but many patterns
- Programmer intention hidden between the lines
- No/little support in modern programming languages

Ownership Types

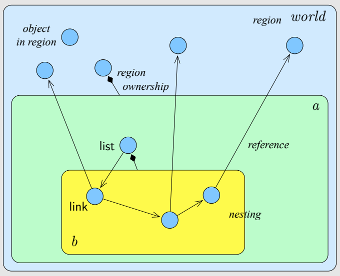
- Decomposes the heap in an hierarchic fashion; objects live in disjoint nested regions; the nesting relation forms a tree.
- Originally proposed by Clarke, Potter and Noble (1998) to formalise certain aspects of Noble, Vitek and Potter's work on Flexible Alias Protection (1998).

```
class List[Owner,Data] { // names of external regions
  Link[This,Data] first; // This = list's private region
}

class Link[Owner,Data] { // Owner => region where the
  Object[Data] element;  // current object resides
  Link[Owner,Data] next;
}
```

```
void prepend(Object[Data] elem) { // elements live in
  first = new Link(elem, first);  // the same region
}

// Returns internal data!
Object[This] leakyIfCalledExternally() {
  return first;
}
```

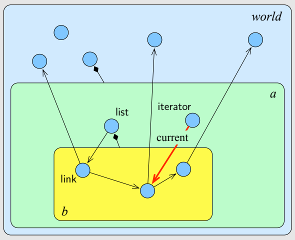
- Objects live in regions owned by other objects, and can be given permission to reference *external* regions.
- Ownership, nesting and permissions are reflected in types.
- Different ownership systems enforce different formal guarantees.
- Ownership types restrict access to *values*, not names.
- Containment invariant: `leaky...` can only be called from objects within the list aggregate.
- Hence: in ownership types a leak will not occur.

Ownership Types Example: List

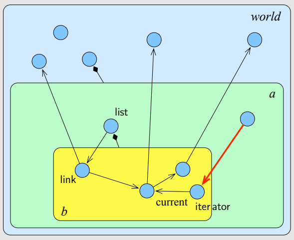


- No external access to links of a linked list.
- Nesting: $world > a > b$.
- b is the region of a linked list whose data elements live in region *world*.
- *Incoming* pointers (*e.g.*, $world \to a$ or $a \to b$) are statically prevented (see box below).
- *Outgoing* pointers (*e.g.*, $b \to a$ or $b \to world$) are allowed.

Breaching Encapsulation (Disallowed)



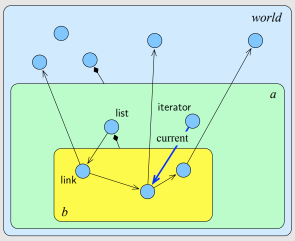
Iterator's direct access through `current` breaks encapsulation although it's implementation is *likely* benign.



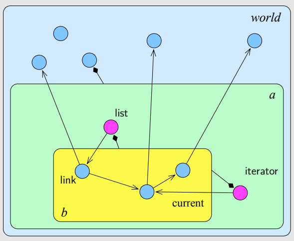
Iterators are allowed internal to the list, but then cannot be accessed externally.

Principled Relaxation of Restrictions

As the examples above show, the strong encapsulation of ownership types can destroy patterns with *intentional* breaches of encapsulation—like iterators. Subsequent work allow principled relaxations of ownership types (here Universes (Müller and Poetzsch-Heffter, 1999) and Ombudsmen (Östlund and Wrigstad, 2012)).



Allow incoming **read-only aliases**



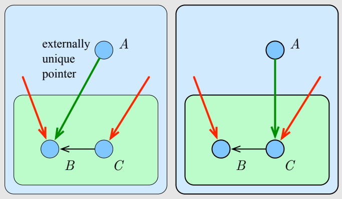
Allow defining multiple **bridge objects**

External Uniqueness

- Originally proposed by Clarke and Wrigstad (2003) as a natural way to combine ownership with unique pointers.
  → Introduce a relaxation of traditional uniqueness: only a single pointer to an object *from outside of the object.*
  → External pointer acts as a guard to access the object, and guarantees internal aliases are unreachable.
- Accessing a unique object needs an explicit borrowing operation:

```
unique List[Data] myList;
// myList is unique
borrow myList { // myList no longer unique
  myList.add(new Object[Data]);
  ... // omitted
}
// myList is unique again
```

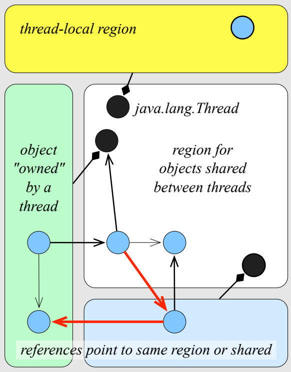
- In practise, almost all borrowing can be *inferred*
- May allow global read-access *outside* of borrowing
- *During* borrowing, exclusive access by the current thread with static guarantee that no aliases exist that can witness mutation



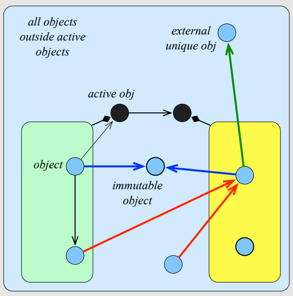
- External uniqueness introduces an additional enclosure to which there is only a **single incoming reference**.
- There may be multiple top-level objects in the enclosure (*e.g.*, B and C).
- Borrowing chooses which top-level object is pointed to by the single incoming reference (left B, right C).

Ownership for Thread-Locality

- Wrigstad *et al.* (2009) propose a simple ownership system where threads own regions in a flat hierarchy
- Experiments by Zaza (2012) show that few annotations (1/250 LOC) can capture large %-age of thread-locality (84% of all thread-local objects and 97% of all thread-local memory in DaCapo Xalan)
- Thread-local accesses statically safe, access to shared data area unsafe
- Eclipse plugin (now deprecated) and Java 8 checker front-end implementations



Joelle: Ownership for Active Objects



- Clarke *et al.* (2008) apply ownership types "minimally" to create isolated active objects
- Östlund and Wrigstad extend this with more complicated alias management to allow internal parallelism in active objects (ongoing)
- Brandauer (2012) shows implementation speed comparable to Scala and Erlang
- Castegren, Östlund and Wrigstad add structured parallelism to Joelle (ongoing)

References

1. Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.
2. Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin / Heidelberg, 2008.
3. P. Müller and A. Poetzsch-Heffter. Universes: a type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.
4. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.
5. Stephan Brandauer. Task Scheduling Using Joelle's Effects. Master Thesis, Dept. of IT, UU, 2013.
6. Johan Östlund and Tobias Wrigstad. Multiple aggregate entry point for ownership types. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 156–180. Springer Berlin / Heidelberg, 2012.
7. Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness, and immutability. In *TOOLS (46)*, pages 178–197, 2008.
8. Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 445–469. Springer Berlin / Heidelberg, 2009.
9. Nosheen Zaza. Evaluating the Accuracy of Annotations in the Loci 3.0 Pluggable Type Checker Master Thesis, Dept. of IT, UU, 2012.

LaTeX TikZposter class