

Resource Usage Protocols for Iterators

Christian Haack Clément Hurlin

Radboud University of Nijmegen

INRIA Sophia Antipolis

IWACO 2008 in Paphos (Cyprus) on July 7



<http://mobius.inria.fr>



Iterators

```
interface Collection {  
    void add(Object e);  
    Iterator iterator();  
}
```

```
interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

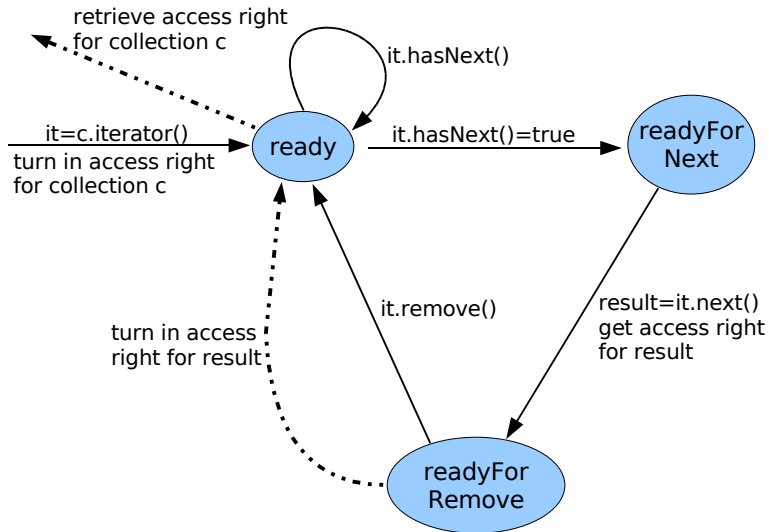
Prevent:

- concurrent modifications
- concurrent mutations of collection elements
- data races in concurrent programs

To this end:

- statically enforce disciplined iterator usage

Basic Usage Protocol



Outline

- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol
- 3 Allowing Concurrent Read-Only Iterators
- 4 Allowing Shallow Collections
- 5 Implementing the Iterator Protocol
- 6 Conclusion

Separation Logic: Formulas as Access Tickets

Atomic Formula:

$o.f \mapsto _$ access ticket for field $o.f$

Access tickets can be combined by resource conjunction:

$o.f \mapsto _ * o.g \mapsto _$ access tickets for fields $o.f$ and $o.g$

Separation Logic: Formulas as Access Tickets

This verifies:

```
//@ requires this.f  $\mapsto$  _ * this.g  $\mapsto$  _;  
//@ ensures this.f  $\mapsto$  _ * this.g  $\mapsto$  _;  
void m() { this.f = 1; this.g = 2; }
```

This is silly, but also verifies:

```
//@ requires this.f  $\mapsto$  _ * this.g  $\mapsto$  _;  
//@ ensures this.f  $\mapsto$  _;  
void m() { this.f = 1; this.g = 2; }
```

This does not verify:

```
//@ requires this.f  $\mapsto$  _;  
//@ ensures this.f  $\mapsto$  _;  
void m() { this.f = 1; this.g = 2; }
```

Abstract Predicates

Classes can define predicates:

```
class C {  
  
    int f;  
    int g;  
  
    pred space = this.f  $\mapsto$  _ * this.g  $\mapsto$  _;  
  
    //@ requires this.space;  
    //@ ensures this.space;  
    void m() { this.f = 1; this.g = 2; }  
  
}
```

- Object clients treat predicates abstractly.
- `this` knows the definitions of its own predicates.

The Space Predicate

- Classes implement the `space`-predicate to specify the heap space associated with `this`.

Default definition of `space` in `Object`:

```
class Object {  
    pred space = true;  
}
```

Implementation of `space` in `C`:

```
class C extends Object {  
    int f; int g;  
    pred space = this.f ↦ _ * this.g ↦ _;  
}
```

Extending Predicates

- Classes can **extend abstract predicates**.
- Predicate extensions in subclasses get ***-conjoined** with predicate extensions in superclasses.

Implementation of space in C:

```
class C extends Object {  
    int f;  
    pred space = this.f  $\mapsto$  _;  
}
```

Extension of space in D:

```
class D extends C {  
    int g;  
    pred space = this.g  $\mapsto$  _; // total space: this.f  $\mapsto$  _ * this.g  $\mapsto$  _  
}
```

Other Logical Operators: Resource Implication

Resource implication:

$F -* G$ ticket to trade ticket F for ticket G

Linear modus ponens:

$$F * (F -* G) -* G$$

Representing the right to make a state transition:

- If F and G represent abstract program states, then $F -* G$ represents the right to move from F to G .

Footnote for separation logic experts:

- We use the proof theory of (affine) linear logic with a separation logic semantics. Hence, no \Rightarrow .

Other Logical Operators: Choice

Resource conjunction:

- $F * G$
- resources F and G are available and are independent
 - ticket to use both resources F and G in any order

Choice:

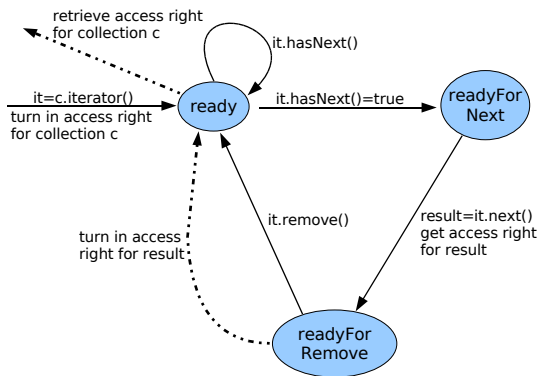
- $F \& G$
- resources F and G are available and are interdependent
 - ticket to use one of F and G , we can choose which one

When representing state machines, $\&$ is useful for representing non-determinism.

Outline

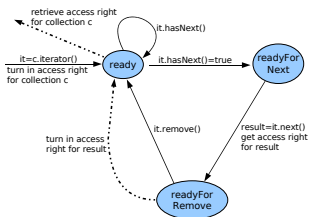
- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol**
- 3 Allowing Concurrent Read-Only Iterators
- 4 Allowing Shallow Collections
- 5 Implementing the Iterator Protocol
- 6 Conclusion

Formalization of Basic Iterator Protocol (Collection)



```
interface Collection {  
    //@ requires this.space * e.space; ensures this.space;  
    void add(Object e);  
    //@ requires this.space; ensures result.ready;  
    Iterator/*@<this>@*/ iterator();  
}
```

Formalization of Basic Iterator Protocol (Iterator)



```
interface Iterator/*@<Collection iteratee>@*/ {  
    //@ pred ready;  
    //@ pred readyForNext;  
    //@ pred readyForRemove<Object element>;  
    //@ axiom ready -* iteratee.space;  
    //@ axiom readyForRemove<e> * e.space -* ready;  
    //@ requires ready; ensures ready & (result -* readyForNext);  
    boolean hasNext();  
    //@ requires readyForNext;  
    //@ ensures readyForRemove<result> * result.space;  
    Object next();  
    //@ requires readyForRemove<_>; ensures ready;  
    void remove();  
}
```

Assessment of Basic Usage Protocol

Plus.

- prevents concurrent modifications
- prevents concurrent mutations of collection elements
- prevents data races in concurrent programs

Minus.

- disallows concurrent read-only iterators

Outline

- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol
- 3 Allowing Concurrent Read-Only Iterators**
- 4 Allowing Shallow Collections
- 5 Implementing the Iterator Protocol
- 6 Conclusion

Fractional Permissions

Superscript the points-to predicate with a fraction $\frac{1}{2^n}$:

$$o.f \stackrel{\pi}{\mapsto} e \quad \pi ::= \alpha \mid 1 \mid \frac{\pi}{2}$$

- Any superscript gives read access.
- Superscript 1 gives write access.

Split/merge axiom:

$$o.f \stackrel{\pi}{\mapsto} e \text{ *-* } (o.f \stackrel{\pi/2}{\mapsto} e * o.f \stackrel{\pi/2}{\mapsto} e)$$

Parameterizing space by a fractional permission:

```
class Object {  
    pred space<perm p> = true;  
    axiom space<p> *-* (space<p/2> * space<p/2>);  
}
```

Allowing Concurrent Read-only Iterators

```
interface Collection {
    /*@ <perm p> requires this.space<p>; ensures result.ready;
    Iterator/*@<p,this>@*/ iterator();
}

interface Iterator/*@<perm p, Collection iteratee>@*/ {
    /*@ pred ready;
    /*@ pred readyForNext;
    /*@ pred readyForRemove<Object element>;

    /*@ axiom ready -* iteratee.space<p>;
    /*@ axiom readyForRemove<e> * e.space<p> -* ready;

    /*@ requires ready; ensures ready & (result -* readyForNext);
    boolean hasNext();

    /*@ requires readyForNext;
    /*@ ensures readyForRemove<result> * result.space<p>;
    Object next();

    /*@ requires readyForRemove<_> * p==1; ensures ready;
    void remove();
}
```

Limitations of the Protocol

Observation.

- According to the protocol, the collection governs access to the collection elements (**deep collection**).

Are the following scenarios supported?

- Access right to an element stays with collection client who added the element (**shallow collection**).
(not supported by above protocol)
- In concurrent programs, access to elements is guarded by their object locks.
(possibly supported by above protocol if combined with separation logic rules for monitors)
- Prevention of concurrent modifications by runtime checks.
(not supported by above protocol)

Outline

- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol
- 3 Allowing Concurrent Read-Only Iterators
- 4 Allowing Shallow Collections**
- 5 Implementing the Iterator Protocol
- 6 Conclusion

Mutable and Immutable Object Space

For greater flexibility, we partition the object space into mutable and immutable part:

```
class Object {  
    //@ pred space<perm p> = true;           // mutable part  
    //@ axiom space<p> ** (space<p/2> * space<p/2>);  
  
    //@ pred ispace = true;                 // immutable part  
    //@ axiom ispace -* (ispace * ispace);  
}
```

Example:

```
class MapEntry extends Object {  
    int key;  
    int val;  
  
    //@ pred ispace = (ex perm p)(this.key  $\xrightarrow{P}$  int);  
    //@ pred space<perm p> = this.val  $\xrightarrow{P}$  int;  
}
```

Allowing Shallow Collections

Shallow collections:

- Immutable element space is shared between collection and client.
- Mutable element space stays with the client who added the element.

A generic collection interface:

```
interface Collection/*@<boolean isdeep>@*/ {  
    /*@ requires this.space<1> * e.ispace * (isdeep -* e.space<1>);  
    /*@ ensures this.space<1>;  
    void add(Object e);  
  
    /*@ <perm p> requires this.space<p>; ensures result.ready;  
    Iterator/*@<p,isdeep,this>@*/ iterator();  
}
```

Allowing Iterators over Shallow Collections

A (more) generic iterator interface:

```
interface
Iterator/*@<perm p, boolean isdeep, Collection<isdeep> iteratee>@*/ {
    /*@ pred ready;
    /*@ pred readyForNext;
    /*@ pred readyForRemove<Object element>;

    /*@ axiom ready -* iteratee.space<p>;
    /*@ axiom readyForRemove<e> * (isdeep -* e.space<p>) -* ready;

    /*@ requires ready; ensures ready & (result -* readyForNext);
    boolean hasNext();

    /*@ requires readyForNext;
    /*@ ensures readyForRemove<result> * result.ispace
    /*@      * (isdeep -* result.space<p>);
    Object next();

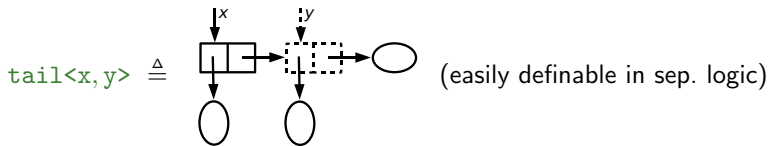
    /*@ requires readyForRemove<_> * p==1; ensures ready;
    void remove();
}
```

Outline

- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol
- 3 Allowing Concurrent Read-Only Iterators
- 4 Allowing Shallow Collections
- 5 Implementing the Iterator Protocol**
- 6 Conclusion

Implementing the Basic Iterator Protocol

```
class ListIterator/*@<Collection iteratee>@*/  
implements Iterator/*@<iteratee>@*/ {  
    Node cur, prev, pprev;  
    //@ pred ready =  
    //@   pprev  $\mapsto$  _ *  
    //@   (ex Node x,y) (prev  $\mapsto$  x * cur  $\mapsto$  y * front<iteratee,x> * tail<x,y>);  
    ...  
}
```



$\text{front}\langle \text{iteratee}, x \rangle \triangleq$ the front of the list already iterated over

How can this be defined formally?

$\text{front}\langle \text{iteratee}, x \rangle \triangleq x.\text{space} \text{ -* } \text{iteratee}.\text{space}$

Outline

- 1 Specifying Heap Access Protocols with Separation Logic
- 2 Formalization of Basic Usage Protocol
- 3 Allowing Concurrent Read-Only Iterators
- 4 Allowing Shallow Collections
- 5 Implementing the Iterator Protocol
- 6 Conclusion**

Summary

- We used separation logic to specify different iterator usage protocols that statically prevent concurrent modifications and concurrent mutations of collection elements.
- Ideally, we'd like a single generic iterator interface that supports all “reasonable” iterator uses.
- Our interface is pretty generic, supporting read-write and read-only iterators over shallow and deep collections through a single interface.
- However, there are reasonable iterator usages that are not supported by this interface (e.g., prevention of concurrent modifications through runtime checks).
- We implemented the generic iterator interface over a linked list collection. The implementation is generic in all interface parameters.

Related Work

Authors that used iterators as case studies for related systems:

- Parkinson (PhD thesis, 2005)
- Krishnaswami (SAVCBS 2006)
- Bierhoff (SAVCBS 2006), Bierhoff and Aldrich (OOPLSA 2007)
- Boyland, Retert, Zhao (IWACO 2007)