Object Ownership in Program Verification

Werner Dietl - University of Washington
Peter Müller - ETH Zürich

Presentation by Roman Schmocker

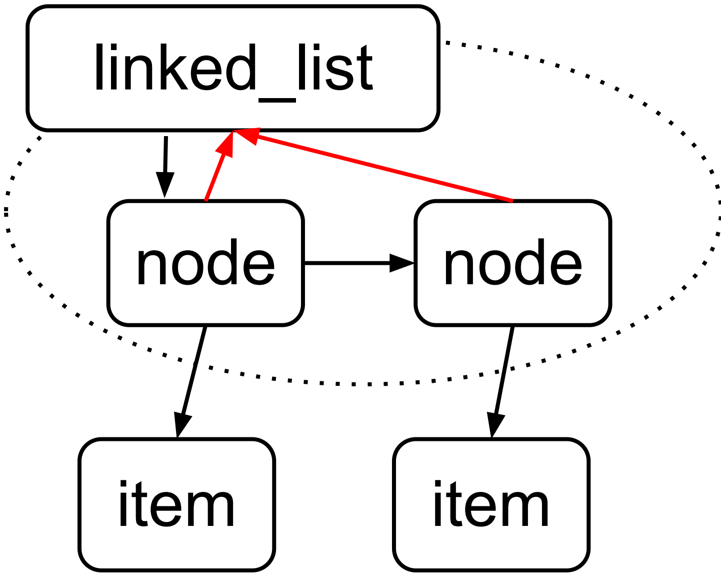
Motivation

```
void a_framing_problem (List a, List b)
   requires a != b;
{
   a.add (1);
   b.remove (1);
   assert (a.contains (1));
   // Does the assertion hold?
}
```

Object Ownership

The basic concepts

- Goal: Information on Heap structuring
  - Reasoning about aliasing
- Ownership topology
  - Objects can own other objects
  - At most one owner
  - Enforced by language
- Encapsulation
  - Protect owned objects from arbitrary modifications
  - Write access only for the owner
  - Readonly or no access for others



Dynamic Ownership

Ownership topology in Spec#

- Implicit ghost field: owner
  - Once set, cannot change
- Attributes on fields

```
[Rep] Node head;
```

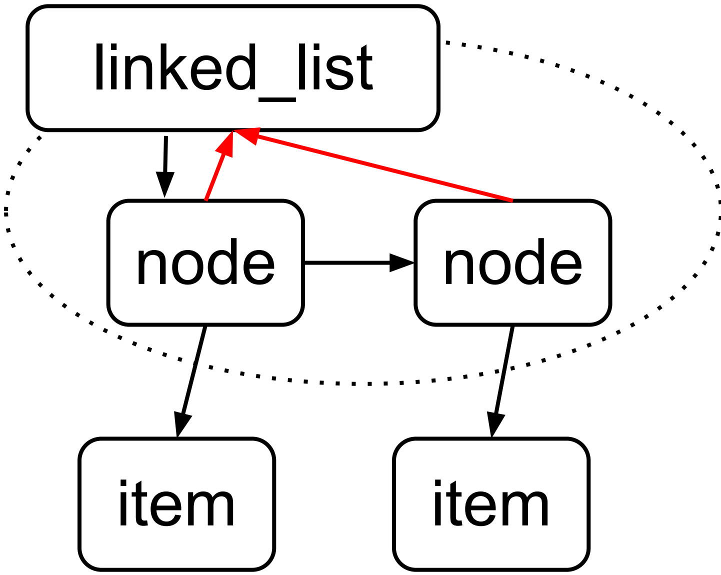
$\Longleftrightarrow$

```
invariant head.owner == this;
```

```
[Peer] Node next;
```

$\Longleftrightarrow$

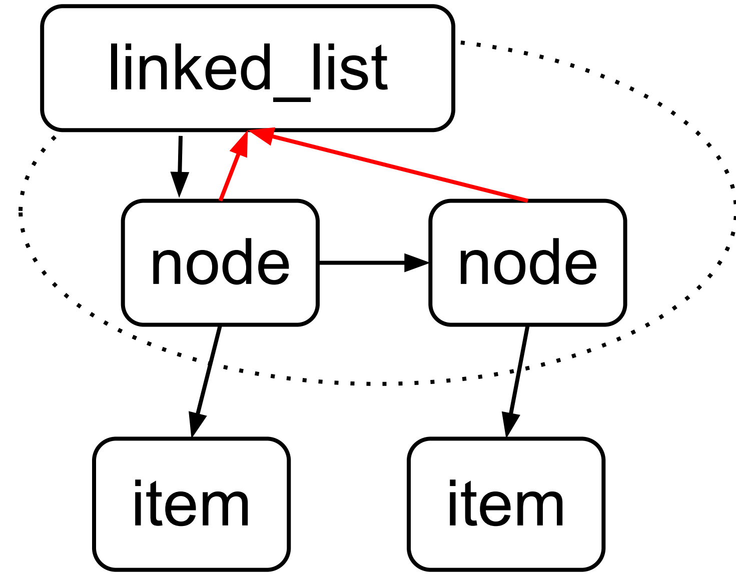
```
invariant next.owner == this.owner;
```



Dynamic Ownership

Ownership topology in Spec#

- Owner set automatically




```
class List {
   [Rep] Node head;

   List () {
      Node newHead = new Node();
         // newHead not owned yet
      this.head = newHead;
         // newHead.owner set to this
} }
```

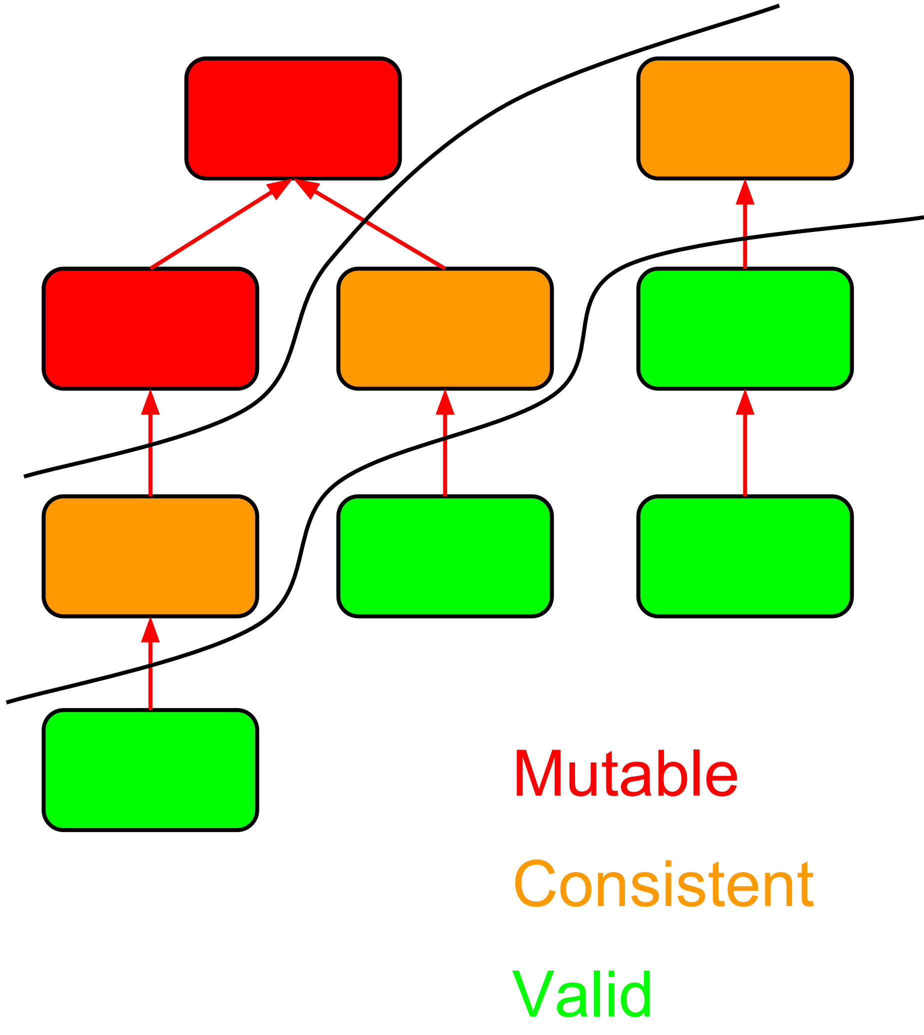
Encapsulation

- Goal: Do not circumvent owner!
  - Write access needs "permission" of owner
- Object states
  - Valid: Invariant holds, read access
  - Mutable: Invariant can be broken, read/write access
  - Consistent: Valid, with mutable owner
- Encapsulation invariant
  - Never allow a mutable object with a valid owner!

Encapsulation

- Heap topology
  - Forest of ownership trees
  - Belt of consistent objects

- expose(o) { ...}
  - o becomes mutable within code block
  - only possible on consistent objects



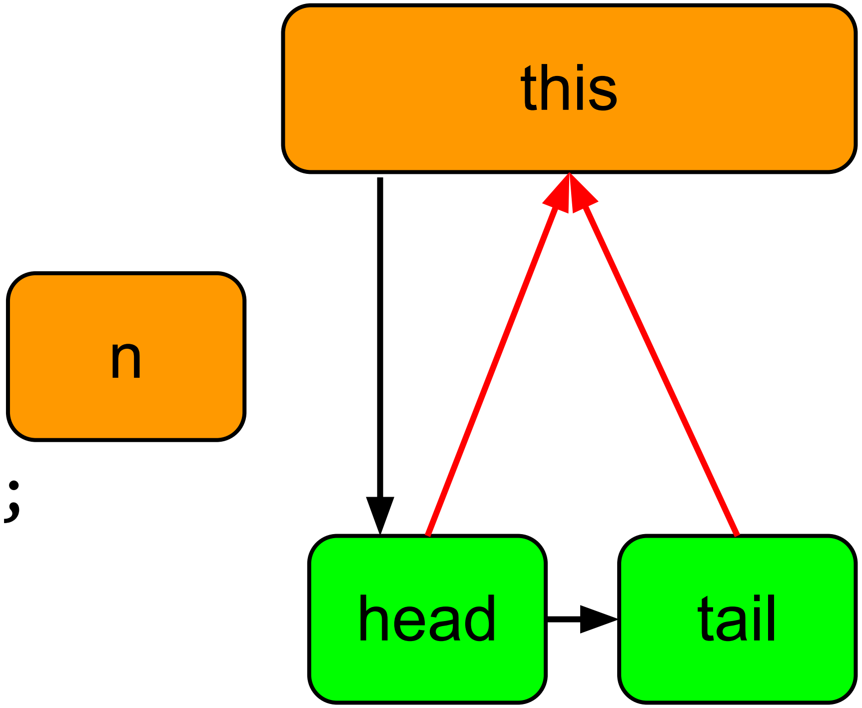
Encapsulation

- Mutating (impure) methods
  - Requires consistent receiver, argument, return value
  - Rationale:
    - May expose receiver
    - May call mutating methods on arguments
    - Caller should be able to modify return value

- Pure methods
  - Only requires valid receiver, argument, return value
  - Rationale: Not allowed to change values anyway

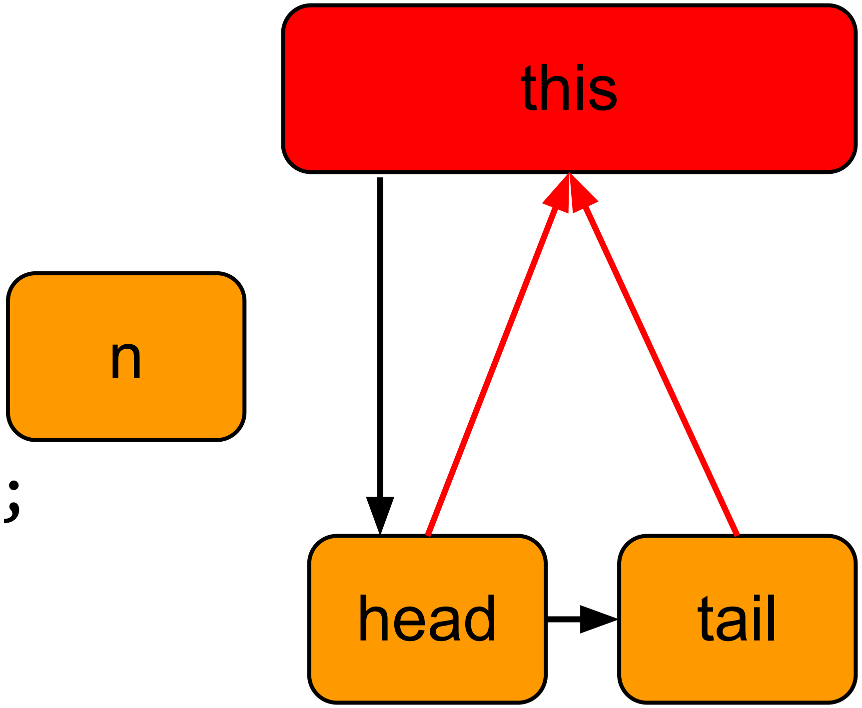
Example

```
class List {
   [Rep] Node head;

   void add (int i) {
      Node n = new Node(i);
      expose(this){
         expose(n) {
            n.next = head;
            head = n;
} } } }
```



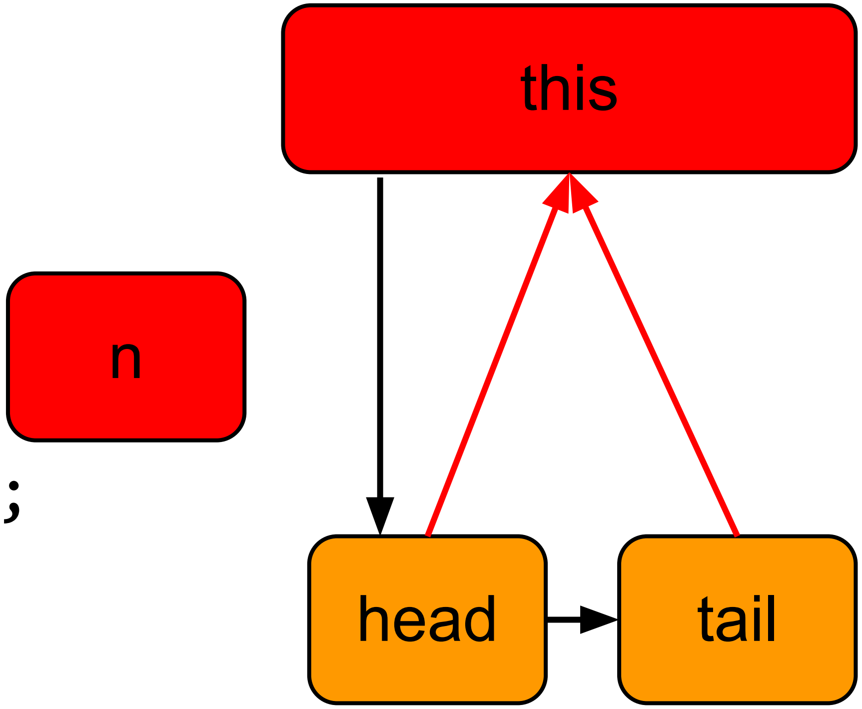
Example

```
class List {
   [Rep] Node head;

   void add (int i) {
      Node n = new Node(i);
      expose(this){
         expose(n) {
            n.next = head;
            head = n;
} } } }
```



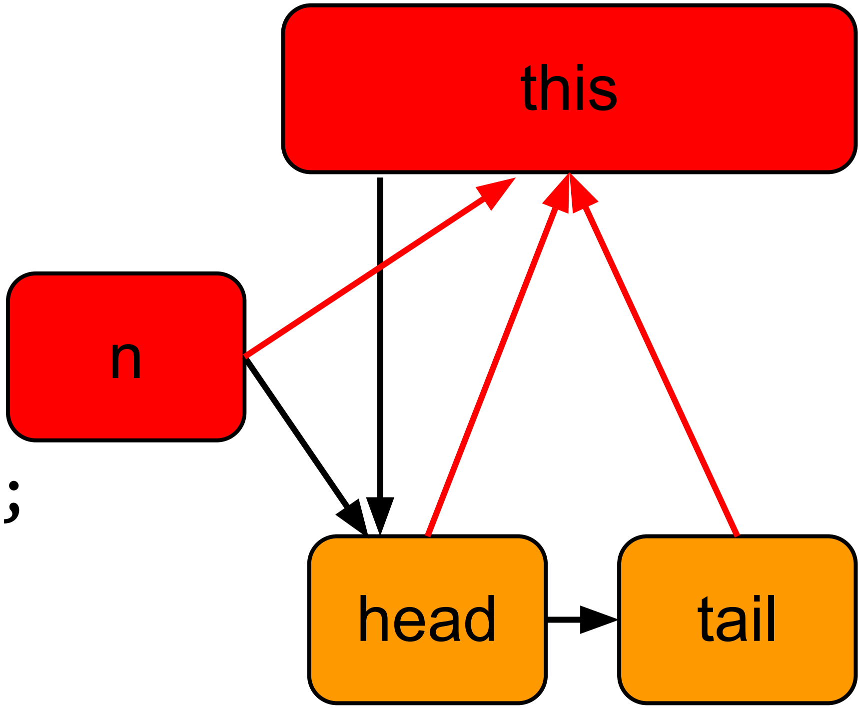
Example

```
class List {
   [Rep] Node head;

   void add (int i) {
      Node n = new Node(i);
      expose(this){
         expose(n) {
            n.next = head;
            head = n;
} } } }
```



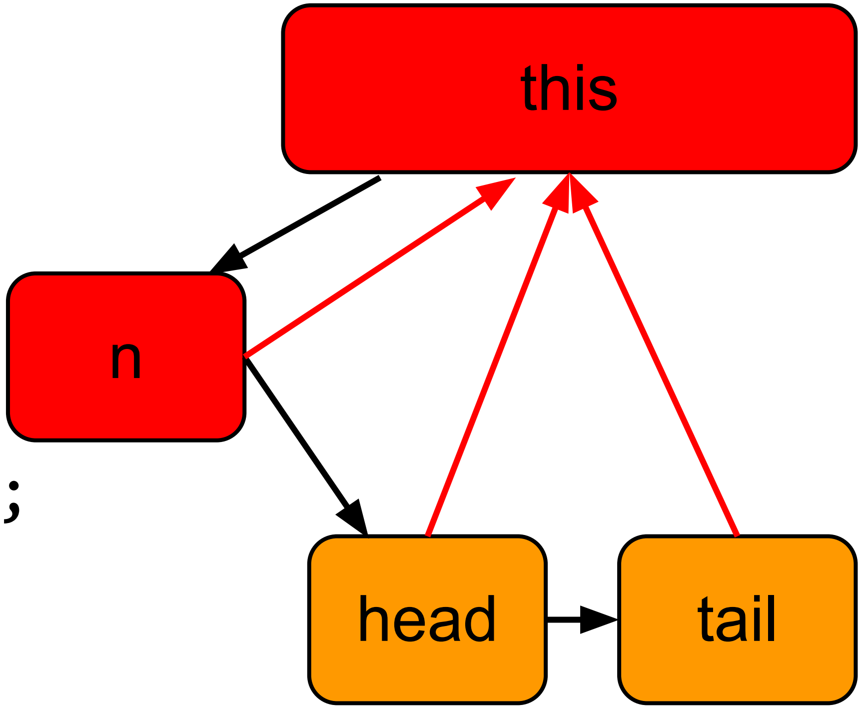
Example

```
class List {
   [Rep] Node head;

   void add (int i) {
      Node n = new Node(i);
      expose(this){
         expose(n) {
            n.next = head;
            head = n;
} } } }
```



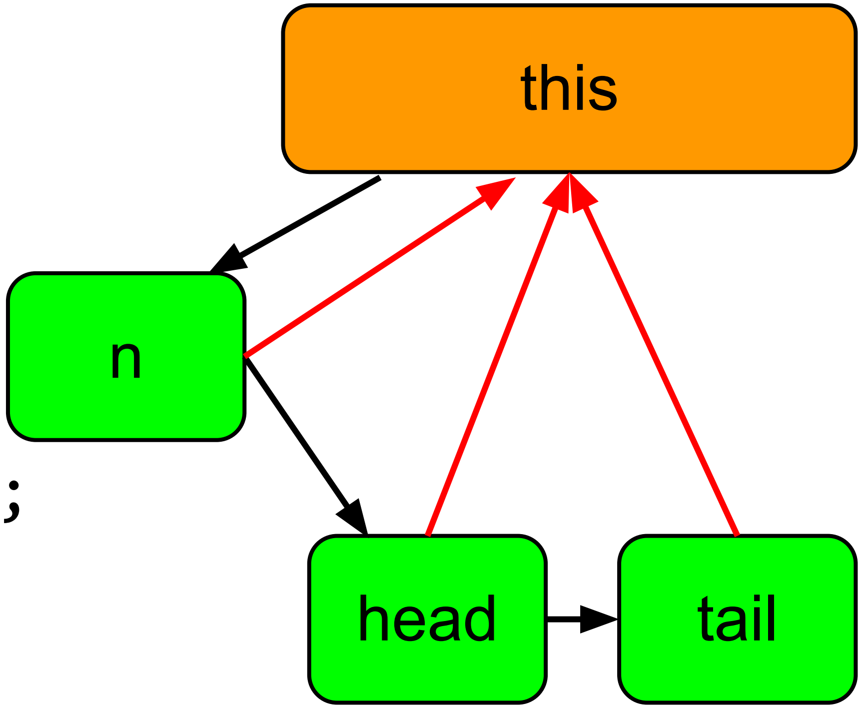
Example

```
class List {
  [Rep] Node head;

  void add (int i) {
    Node n = new Node(i);
    expose(this){
      expose(n) {
        n.next = head;
        head = n;
} } } }
```



Example

```
class List {
   [Rep] Node head;

   void add (int i) {
      Node n = new Node(i);
      expose(this){
         expose(n) {
               n.next = head;
               head = n;
} } } }
```



Applications

Framing

```
void a_framing_problem (List a, List b)
   requires a != b;
{
   a.add (1);
   b.remove (1);
   assert (a.contains (1));
   // Does the assertion hold?
}
```

Applications

Framing

- Case 1: Shared node structures
- Case 2: a transitively owns b
- Case 3: a == b

```
void a_framing_problem
            (List a, List b)
    requires a != b;
{
    a.add (1);
    b.remove (1);
    assert (a.contains (1));
}
```

Applications

Framing

- Case 1: Shared node structures
  - No: contradicts topology invariant (only one owner)
- Case 2: a transitively owns b
- Case 3: a == b

```
void a_framing_problem
            (List a, List b)
    requires a != b;
{
    a.add (1);
    b.remove (1);
    assert (a.contains (1));
}
```

Applications

Framing

- Case 1: Shared node structures
  - No: contradicts topology invariant (only one owner)
- Case 2: a transitively owns b
  - No: Illegal call, since a and b cannot be both consistent
- Case 3: a == b

```
void a_framing_problem
                (List a, List b)
    requires a != b;
{
    a.add (1);
    b.remove (1);
    assert (a.contains (1));
}
```

Applications

Framing

- Case 1: Shared node structures
  - No: contradicts topology invariant (only one owner)
- Case 2: a transitively owns b
  - No: Illegal call, since a and b cannot be both consistent
- Case 3: a == b
  - No: see precondition

```
void a_framing_problem
                (List a, List b)
    requires a != b;
{
    a.add (1);
    b.remove (1);
    assert (a.contains (1));
}
```

Applications

Multi-Object Invariants

- Multi-Object Invariants
  - Invariants on state of referenced objects
- Problem
  - Objects may break the invariant of another object they didn't even know existed
  - Hard to check statically
  - A temporary break may actually be necessary

Applications

Multi-Object Invariants

- Admissible Invariants
  - Only allow multi-object invariants on [Rep] objects
  - Objects can only break invariant of their owner
  - OK, since owner is mutable anyway

- Modular invariant checking
  - At the end of expose() block
  - At the end of constructor

Applications
Immutable Objects

- Readonly interfaces
  - Can be casted away easily
- Wrapper classes
  - Make sure no mutable inner structure is leaked
  - Boilerplate code
  - (In Java:) Runtime checking, Exceptions
- Immutable objects
  - Only pure methods + constructor
  - Leaking still problematic
  - Inflexible object construction
  - Usually no inheritance allowed

Applications

Immutable Objects

- Freezer object
  - Cannot be exposed

- Ownership solution
  - Just set owner to the Freezer!

```
void freeze_example () {
    List l = new List();
    l.add (42);          // ok: l is consistent
    freeze l;            // set l.owner to Freezer
    l.add (43);          // error
}
```

Applications

Immutable Objects

- Transitive for all owned objects
  - especially useful for data structures
- No boilerplate code necessary
  - Any object can become immutable
- Static checking
  - Inner structures safe from write access
- Allows complex initialization

Conclusion

- Provides encapsulation for object structures
  - Statically checked!
- Some nice applications
  - Interesting ones shown in talk
  - Further applications: Termination proof, data race freedom, effect specialization
- Little annotation overhead
  - But also less flexibility
- Possible to integrate in other languages

About the paper

Historical Context

- 80s: Object-oriented programming emerges
  - Aliasing increasingly problematic
- 90s: Idea of Object ownership evolved
  - Most solutions inflexible and/or unsound
- 1998: Clarke et al: Ownership types
  - Flexible type system, soundness proven
- 2004: Microsoft releases Spec#
- 2012: This paper
  - Two implementations for Object ownership
  - Several applications

About the paper

- Assessment
  - Well written, self-contained
  - Many comparisons to other solutions
  - Main concepts actually come from another paper

- Current status
  - Dynamic Ownership implemented in Spec#
  - Framing and Multi-object invariants work
  - Freezing objects not implemented yet
  - Try it online: http://rise4fun.com/SpecSharp