Ownership Types for Safe Region-Based Memory Management in Real-Time Java

Chandrasekhar Boyapati
Alexandru Salcianu
William Beebee
Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology

Contribution

Ownership types
(Object encapsulation)

- Clarke et al. (OOPSLA '98) (OOPSLA '02)
- Boyapati et al. (OOPSLA '01) (OOPSLA '02)
- Boyapati et al. (POPL '03) (OOPSLA '03)
- Aldrich et al. (OOPSLA '02)

Region types
(Memory safety)

- Tofte, Talpin (POPL '94)
- Christiansen et al. (DIKU '98)
- Crary et al. (POPL '99)
- Grossman et al. (PLDI '02)

Unified type system for OO languages

- Object encapsulation **AND** Memory safety
- Foundation for enforcing other safety properties
  - Data race and deadlock freedom
  - Safe software upgrades
  - Safe real-time programming (Real-Time Java)

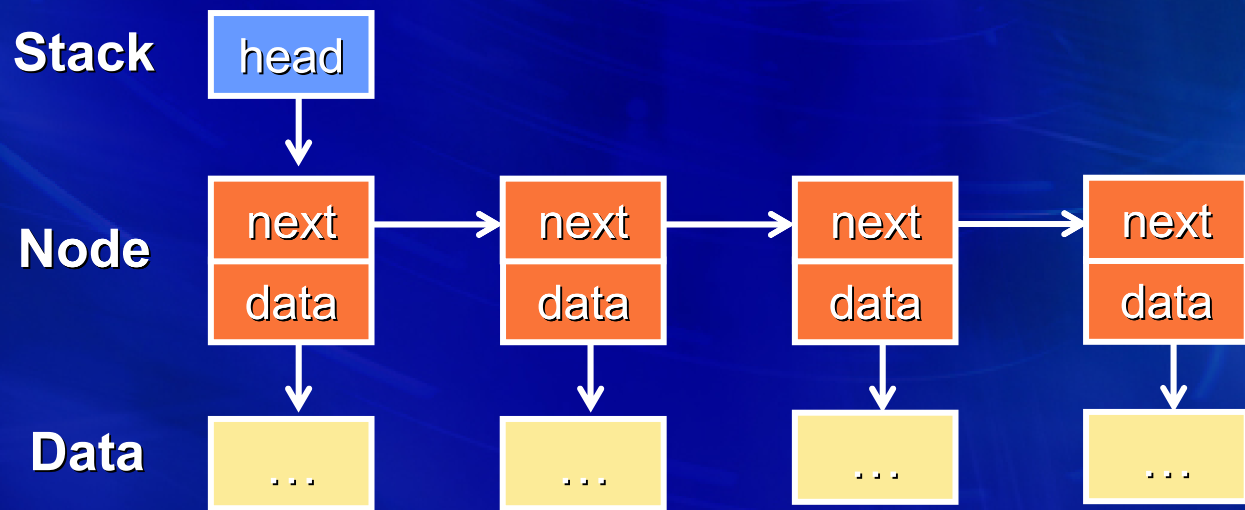
Talk Overview

- Type system for OO programs
  - Ownership types
  - Region types
  - Similarities
  - Unified type system
- Extensions for Real-Time Java
- Experience

Ownership Types

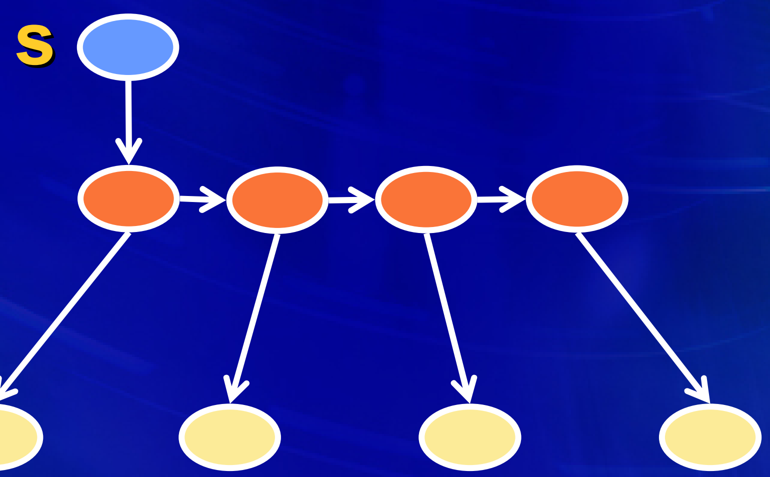
Ownership Types

- Say Stack s is implemented with linked list



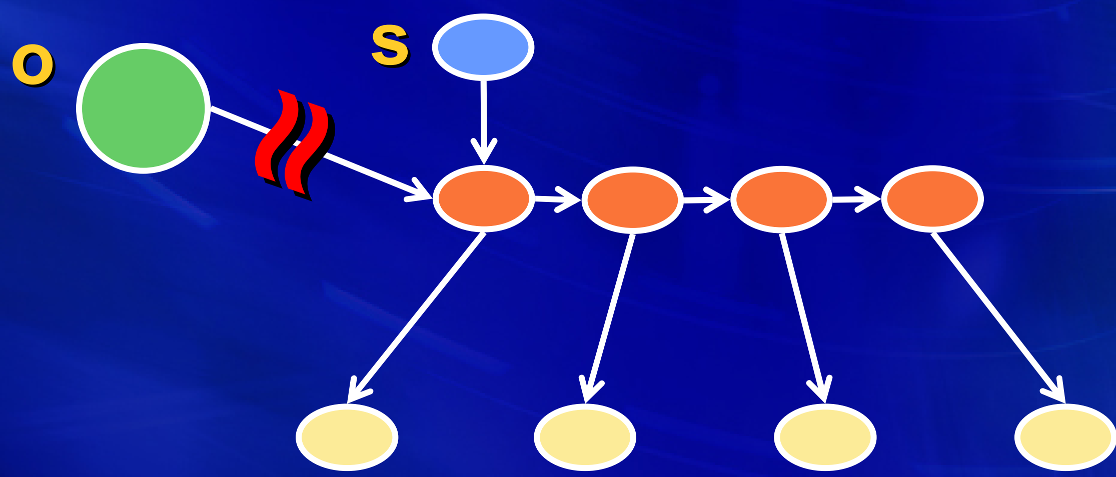
Ownership Types

- Say Stack s is implemented with linked list



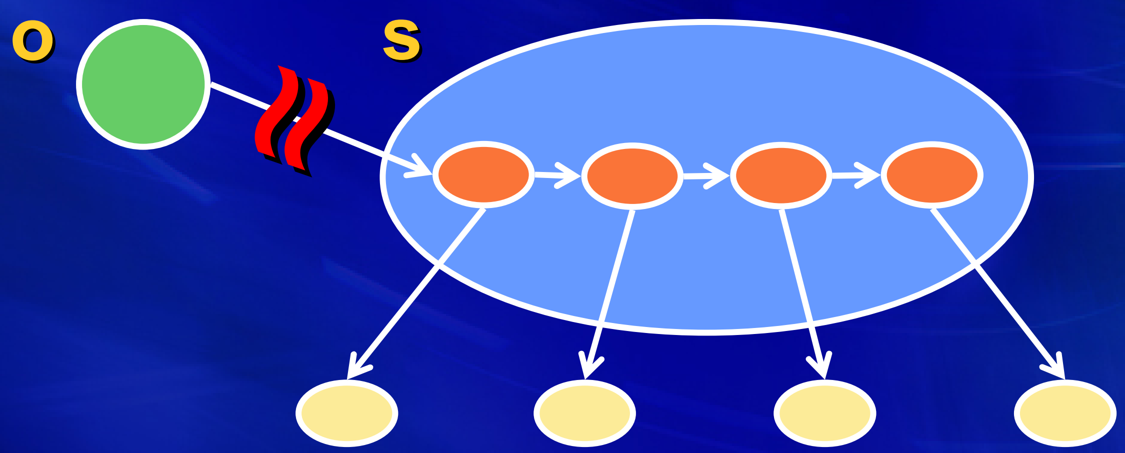
Ownership Types

- Say Stack s is implemented with linked list
- Outside objects must not access list nodes



Ownership Types

- Say Stack s is implemented with linked list
- Outside objects must not access list nodes



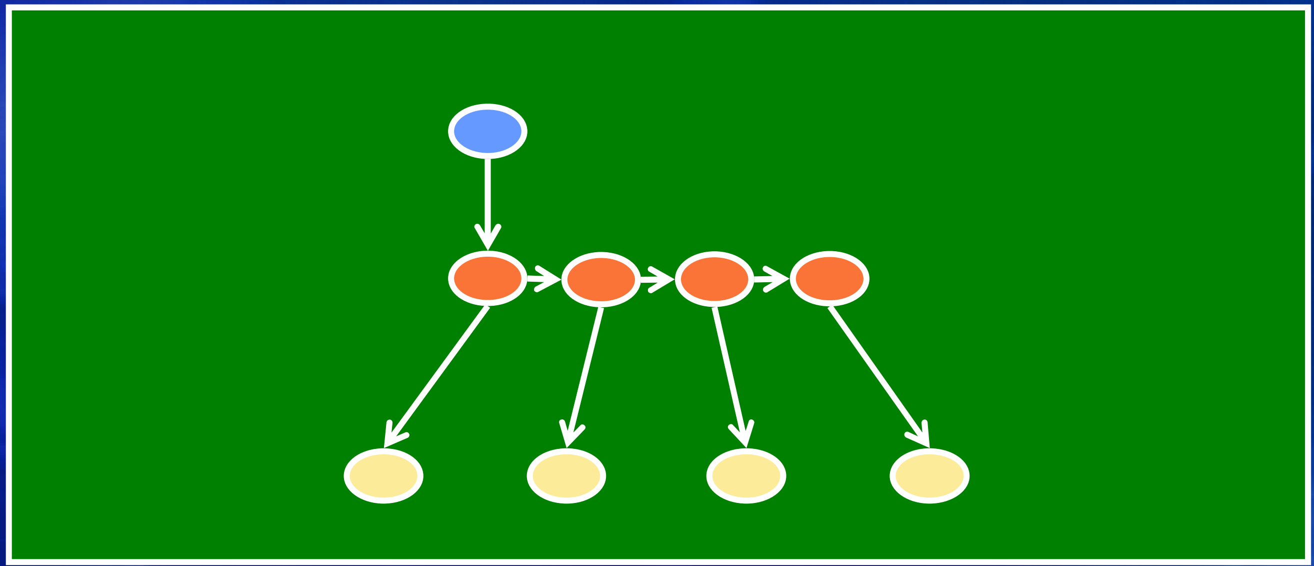
- Program can declare s owns list nodes
- System ensures list is encapsulated in s

Region-Based Memory Management

Region-Based Memory Management

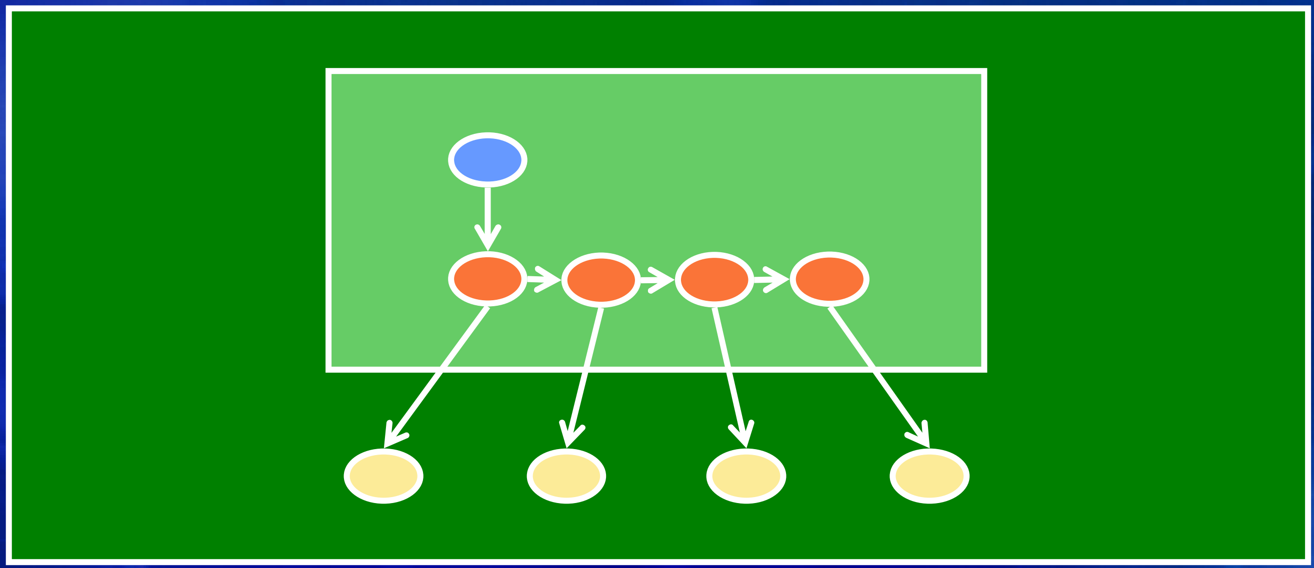
- Provides control over memory
  - For efficiency
  - For predictability
- While ensuring memory safety

Region-Based Memory Management



- Programs can create a region
- Allocate objects in a region
- Delete a region & free all objects in it

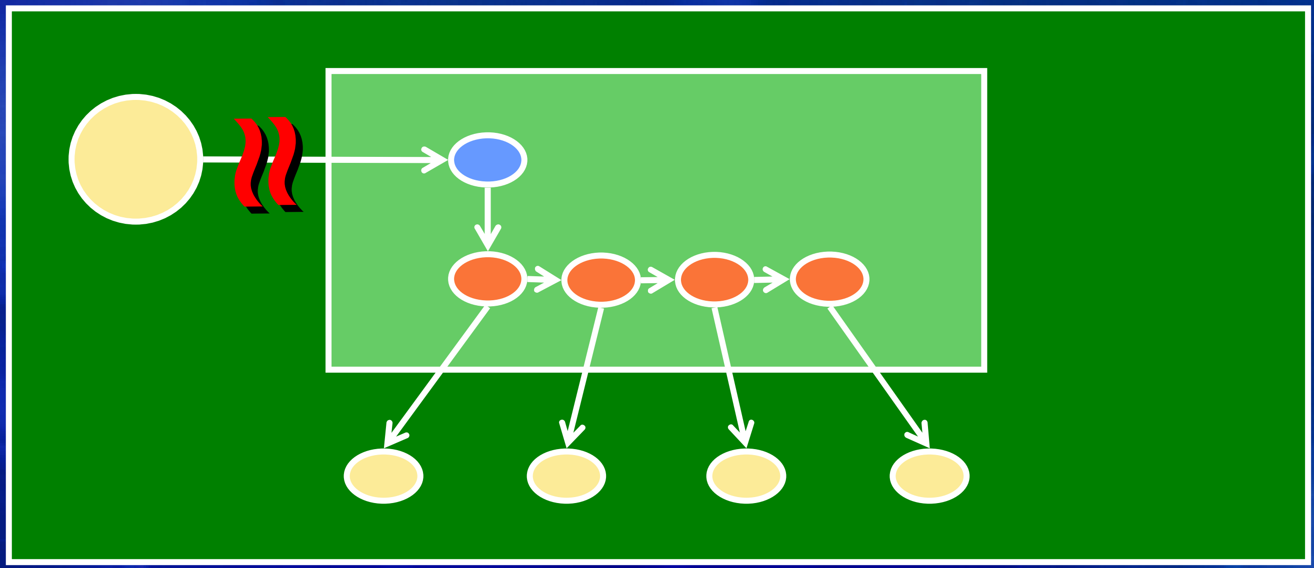
Region-Based Memory Management



- Programs can create a region
- Allocate objects in a region
- Delete a region & free all objects in it

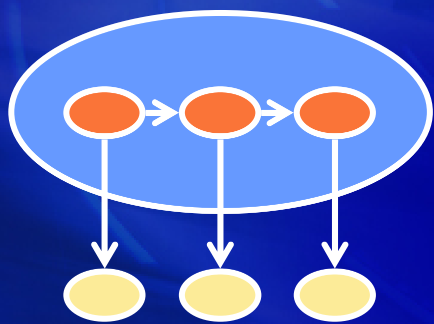
- Region lifetimes are nested

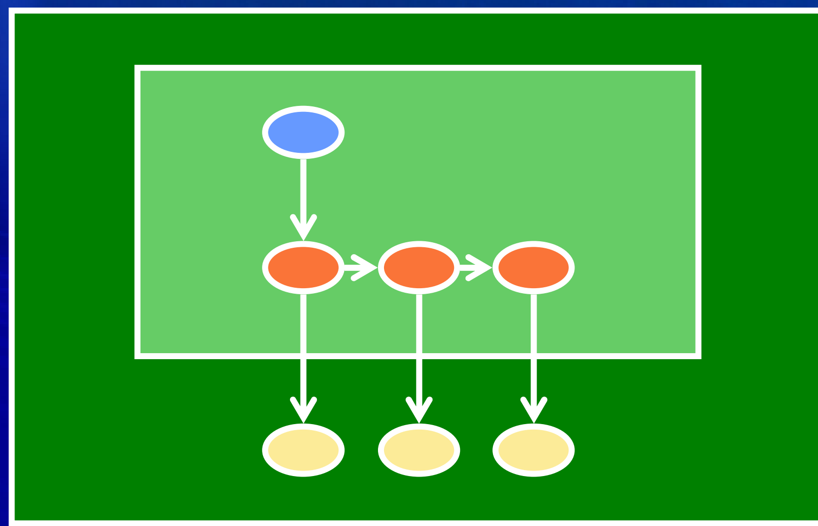
Region Types



- Ensure memory safety
- Disallow pointers from outside to inside

Similarities

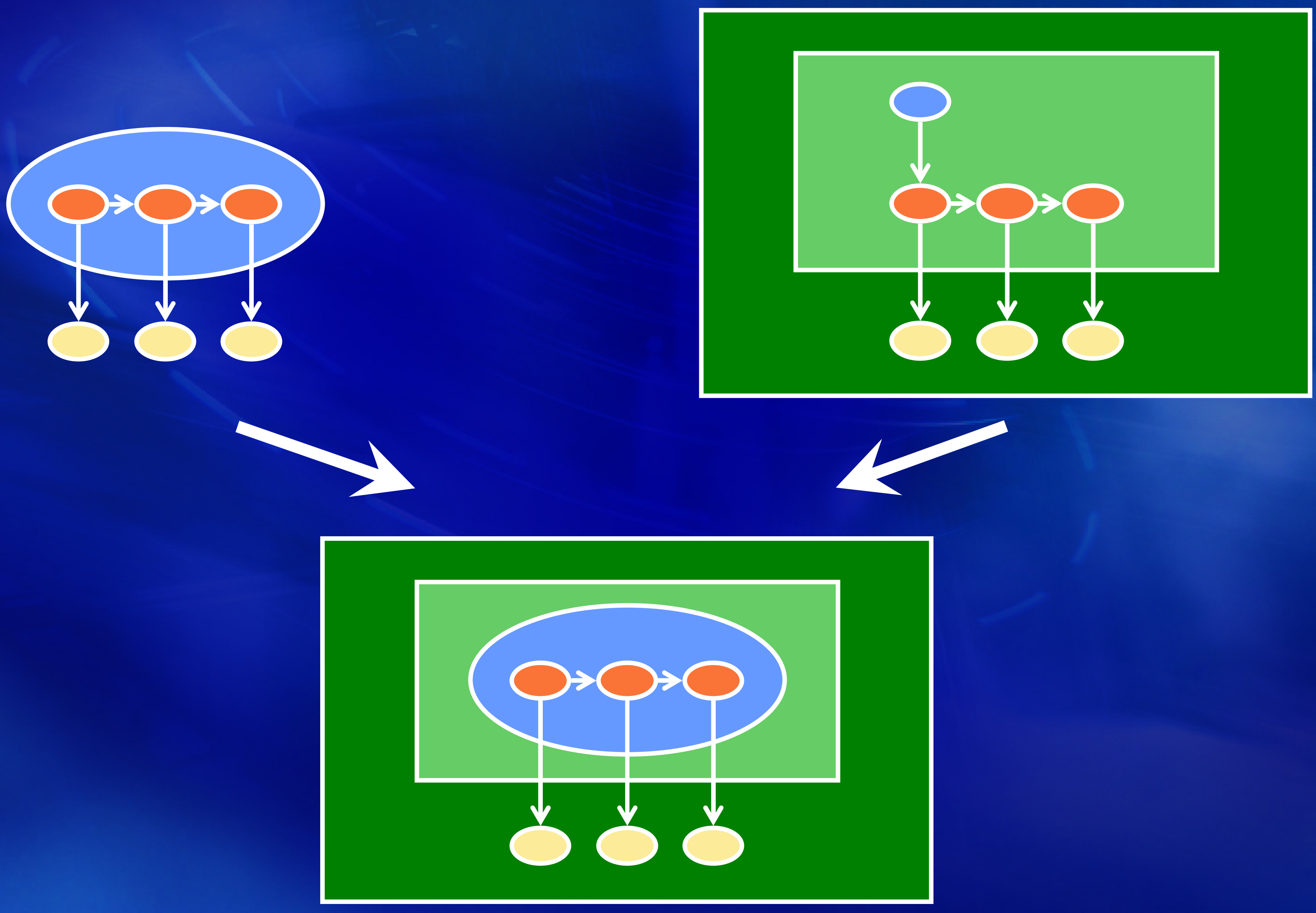




- Ownership types ensure object encapsulation
- Disallow pointers from outside to inside

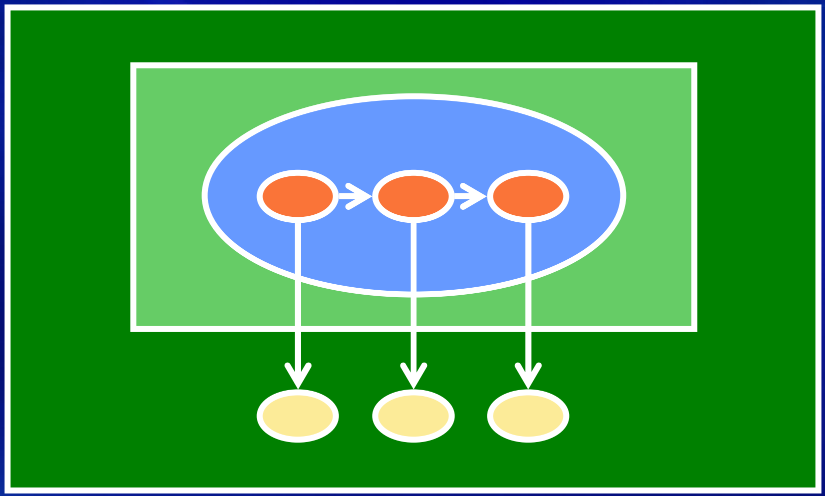
- Region types ensure memory safety
- Disallow pointers from outside to inside

Unified Type System



Unified Type System

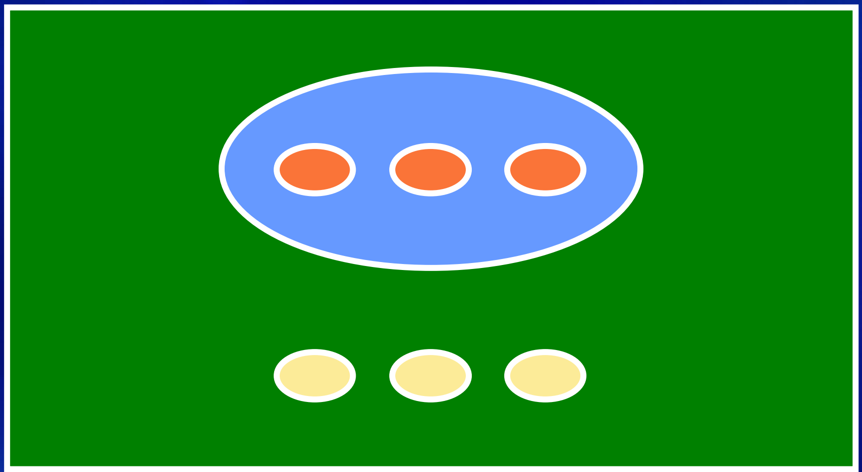
- Disallows pointers from outside to inside
- Ensures object encapsulation
- Ensures memory safety



Unified Type System

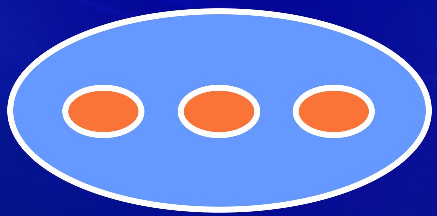
Unified Type System

- Every object has an owner
- Owner can be another object or a region
- Ownership relation forms a forest of trees



Unified Type System

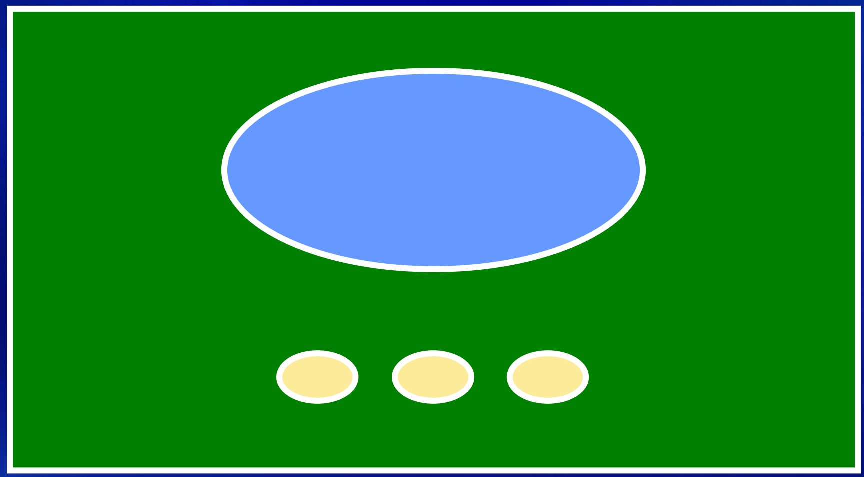
- An object owned by another object
  - Is an encapsulated subobject of its owner



Unified Type System

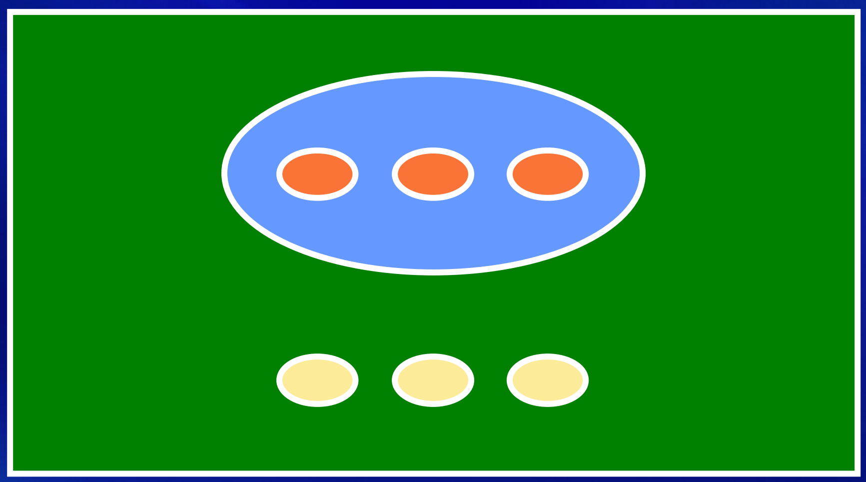
- An object owned by another object
  - Is an encapsulated subobject of its owner

- An object owned by a region
  - Is allocated in that region



Unified Type System

- An object owned by another object
  - Is an encapsulated subobject of its owner
  - Is allocated in the same region as its owner
- An object owned by a region
  - Is allocated in that region

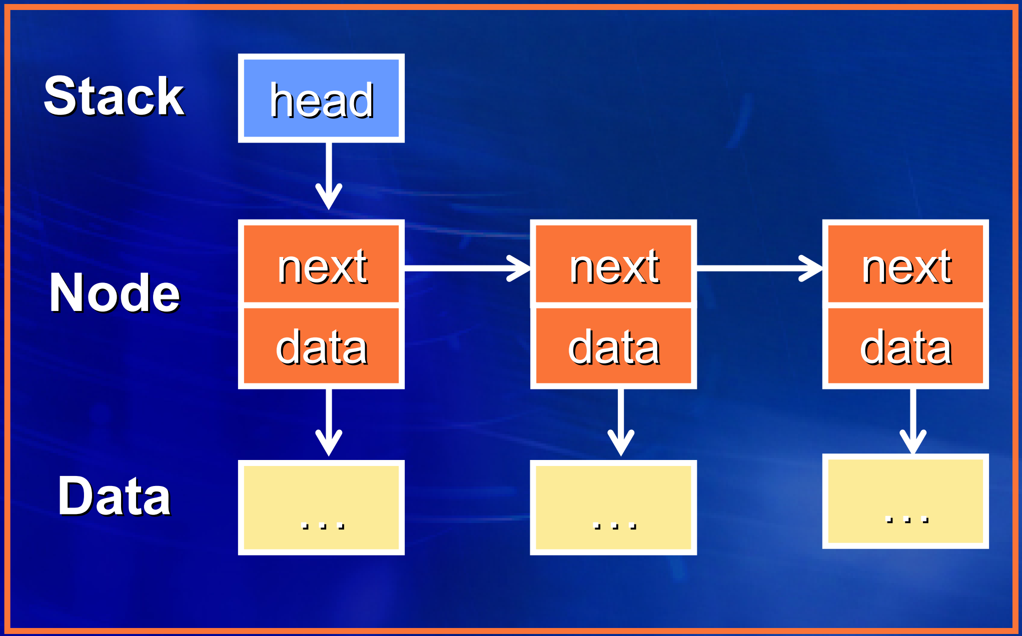


Unified Type System

- Programmers specify
  - Owner of every object
  - In types of variables pointing to objects
- Type checker statically verifies
  - No pointers from outside to inside

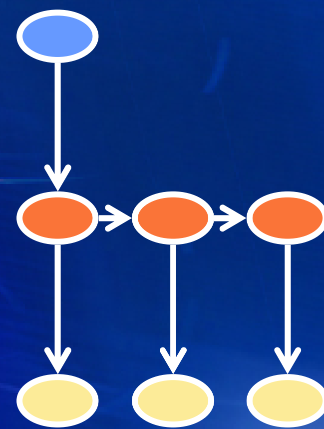
Unified Type System

```
class Stack {
    Node head;

    void push(Data data) {…}
     Data pop() {…}
}

class Node {
    Node next;
    Data data;
    …
}
```



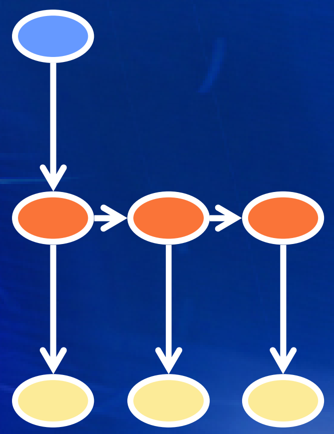
Unified Type System

```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
```



Unified Type System

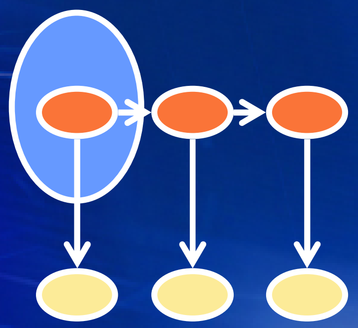
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
```



Classes are parameterized with owners
First owner owns the corresponding object

Unified Type System

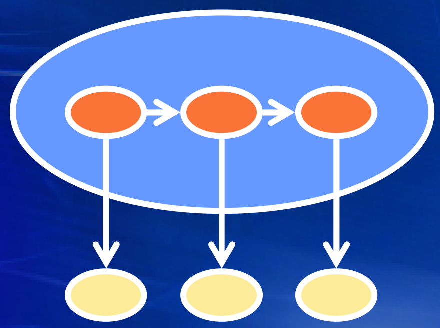
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
```



Stack owns the head Node

Unified Type System

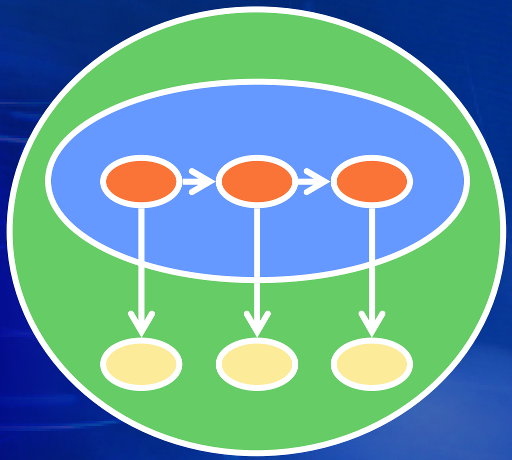
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
```



All Nodes have the same owner

Ownership Types for Safe Regions

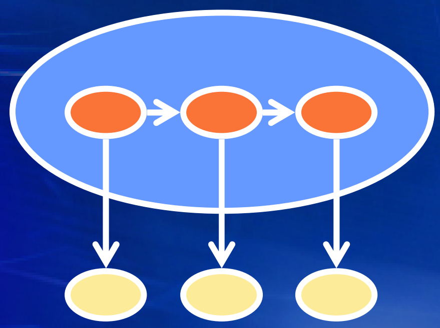
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
class Client {
    Stack⟨this, this⟩  s;
}
```



s is an encapsulated stack with encapsulated elements

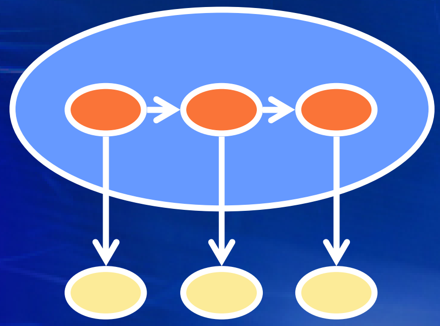
Unified Type System

```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
```



Unified Type System

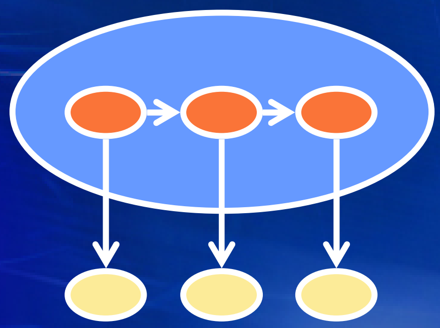
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
(RegionHandle⟨r⟩ h) {
    …
}
```



r is the region name. It is a compile time entity.
h is the region handle. It is a runtime value.

Unified Type System

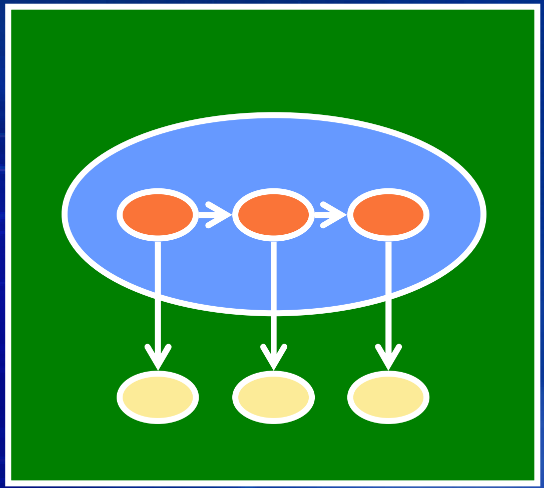
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
(RegionHandle⟨r1⟩ h1) {
    (RegionHandle⟨r2⟩ h2) {
        Stack⟨r1, r1⟩  s1;
        Stack⟨r2, r1⟩  s2;
}}
```



Region r2 is nested inside region r1

Unified Type System

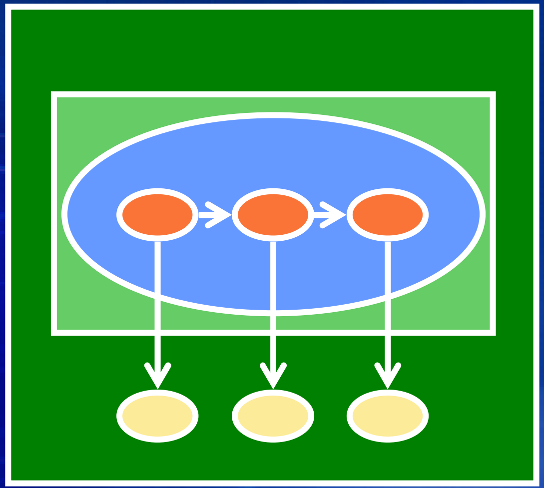
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
(RegionHandle⟨r1⟩ h1) {
    (RegionHandle⟨r2⟩ h2) {
        Stack⟨r1, r1⟩  s1;
        Stack⟨r2, r1⟩  s2;
}}
```



Stack and its elements are in the same region

Unified Type System

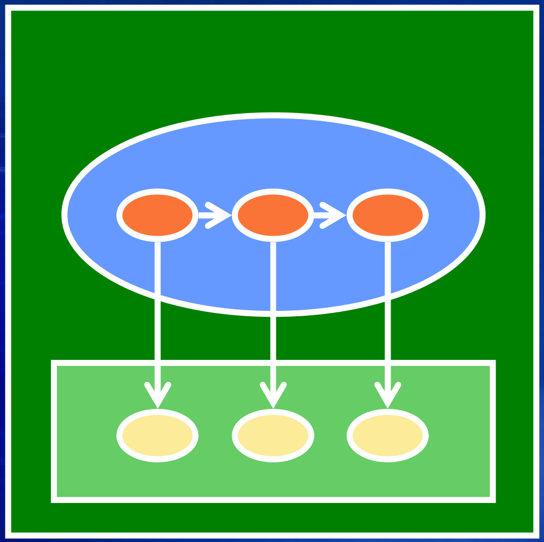
```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
(RegionHandle⟨r1⟩ h1) {
    (RegionHandle⟨r2⟩ h2) {
        Stack⟨r1, r1⟩  s1;
        Stack⟨r2, r1⟩  s2;
}}
```



Stack and its elements are in different regions

Unified Type System

```
class Stack⟨stackOwner, dataOwner⟩ {
    Node⟨this, dataOwner⟩ head;
}
class Node⟨nodeOwner, dataOwner⟩ {
    Node⟨nodeOwner, dataOwner⟩ next;
    Data⟨dataOwner⟩ data;
}
(RegionHandle⟨r1⟩ h1) {
    (RegionHandle⟨r2⟩ h2) {
        Stack⟨r1, r1⟩  s1;
        Stack⟨r2, r1⟩  s2;
        Stack⟨r1, r2⟩  s3;    // illegal
}}
```



Scoping alone does not ensure safety in presence of subtyping
First owner must be same as or nested in other owners

Unified Type System

- Other details
  - Special regions
    - Garbage collected heap
    - Immortal region
  - Runtime provides
    - Region handle of most nested region
    - Region handle of an object
  - Type checker statically infers
    - If a region handle is in scope

Unified Type System

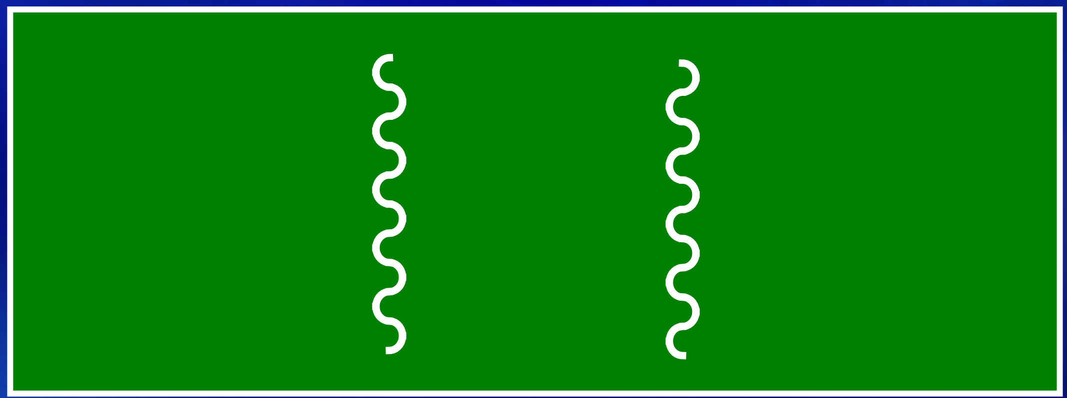
- Enforces object encapsulation
  - Boyapati, Liskov, Shrira (POPL '03)
- Enable safe region-based memory management
  - Boyapati, Salcianu, Beebee, Rinard (PLDI '03)
- Prevents data races and deadlocks
  - Boyapati, Rinard (OOPSLA '01)
  - Boyapati, Lee, Rinard (OOPSLA '02)
- Enables upgrades in persistent object stores
  - Boyapati, Liskov, Shrira, Moh, Richman (OOPSLA '03)

Talk Overview

- Unified type system for OO programs
- Extensions for Real-time Java
  - Multithreaded programs
  - Real-time programs
  - Real-time Java programs
- Experience

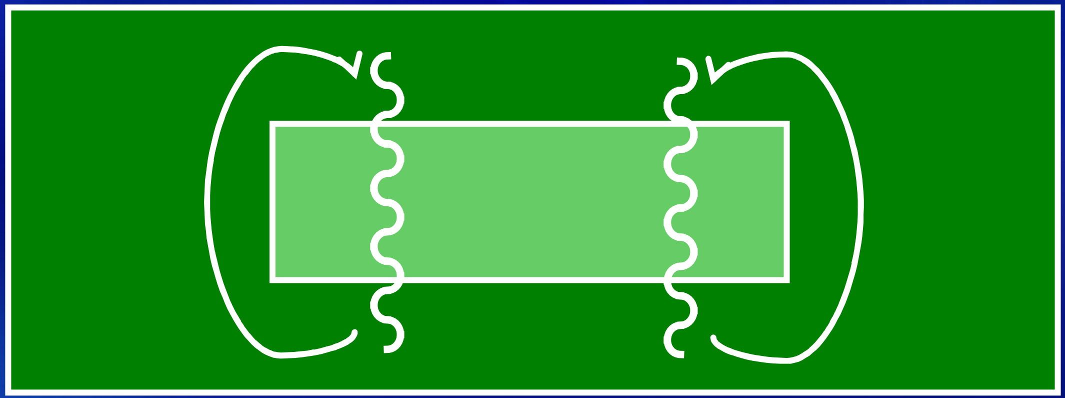
Regions for Multithreaded Programs

- **Shared regions** with reference counting
  - Grossman (TLDI '01)



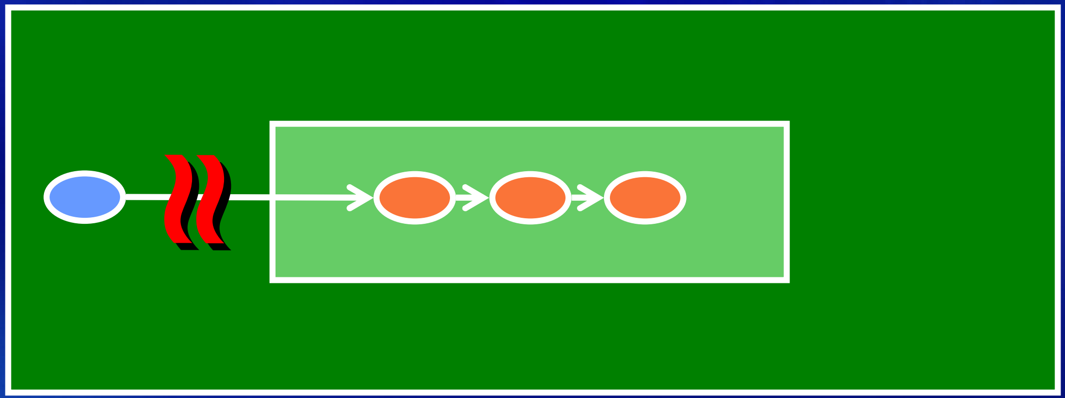
Regions for Multithreaded Programs

- Shared regions with reference counting
  - Grossman (TLDI '01)
- Sub regions within shared regions
- To avoid memory leaks in shared regions



Regions for Multithreaded Programs

- Shared regions with reference counting
  - Grossman (TLDI '01)
- Sub regions within shared regions
- To avoid memory leaks in shared regions

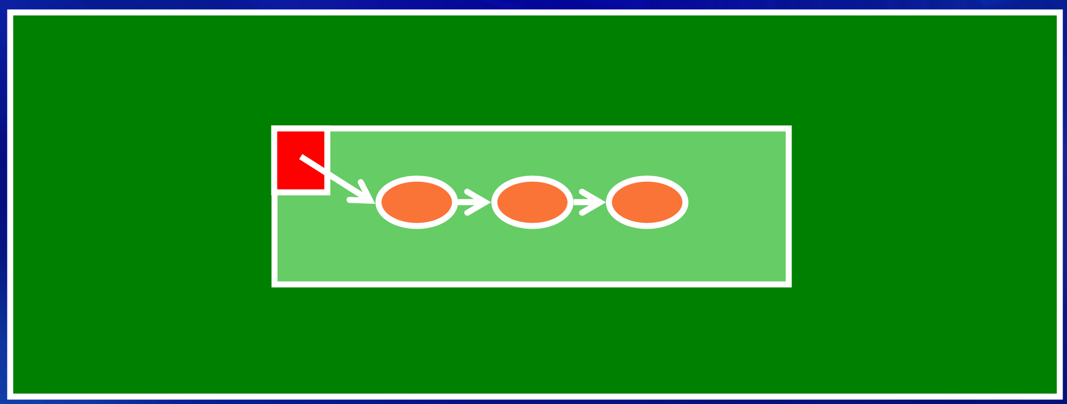


Regions for Multithreaded Programs

- Shared regions with reference counting
  - Grossman (TLDI ’01)

- Sub regions within shared regions
- To avoid memory leaks in shared regions

- Typed portal fields in sub regions
- To start inter-thread communication



Regions for Multithreaded Programs

- Shared regions with reference counting
  - Grossman (TLDI ’01)
- Sub regions within shared regions
- To avoid memory leaks in shared regions
- Typed portal fields in sub regions
- To start inter-thread communication
- Region kinds to make it all work

Talk Overview

- Unified type system for OO programs
- Extensions for Real-time Java
  - Multithreaded programs
  - Real-time programs
  - Real-time Java programs
- Experience

Regions for Real-Time Programs

- Real-time (RT) threads with real-time constraints

- RT threads cannot use garbage collected heap
- RT threads can use immortal memory
- RT threads can use regions

Regions for Real-Time Programs

- Real-time (RT) threads with real-time constraints

- RT threads cannot use garbage collected heap
- RT threads can use immortal memory
- RT threads can use regions

- RT threads cannot read heap references
- RT threads cannot overwrite heap references

Regions for Real-Time Programs

- Real-time (RT) threads with real-time constraints

- RT threads cannot use garbage collected heap
- RT threads can use immortal memory
- RT threads can use regions

- RT threads cannot read heap references
- RT threads cannot overwrite heap references

- Ownership types augmented with effects clauses
- To statically verify above properties

Real-Time Java (RTJ)

- Extension to Java for real-time programs
- Java Specification Request (JSR) 1
- http://www.rtj.org

Real-Time Java (RTJ)

- Extension to Java for real-time programs

- Java Specification Request (JSR) 1
- http://www.rtj.org

- Real-time (RT) threads
- Region-based memory management

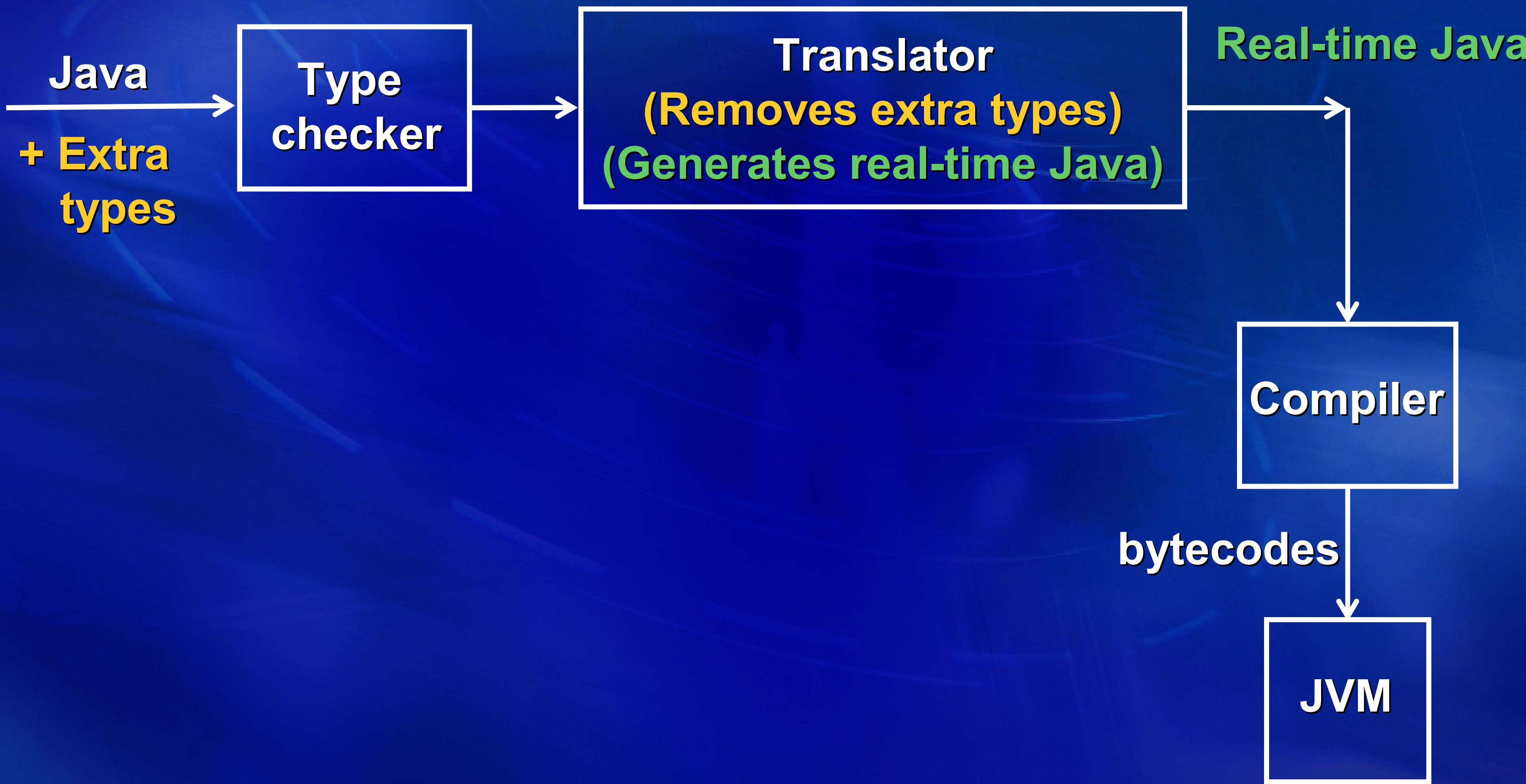
- Threads cant violate memory safety
- RT threads cant interact with garbage collector

Real-Time Java (RTJ)

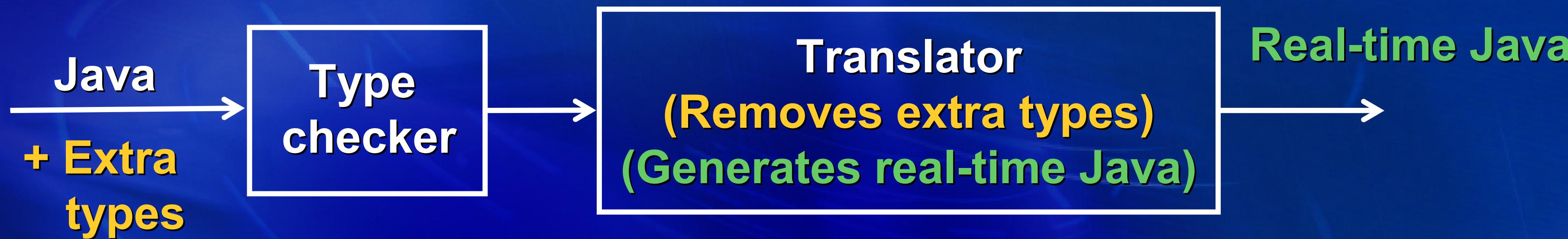
- Uses dynamic checks to ensure
  - No pointers from outer to inner regions
  - Nesting of regions forms a hierarchy
  - RT threads do not read heap refs
  - RT threads do not overwrite heap refs

- Introduces new failure modes
- Programming model is difficult to use

Region Types as Front-End for RTJ



Benefits of Using Region Types



- **Safety**
- **Checks errors at compile time**
- **Efficiency**
- **Avoids runtime checking overhead**

Experience

Reducing Programming Overhead

- Type inference for method local variables
- Default types for method signatures & fields
- User defined defaults as well

- Significantly reduces programming overhead

- Approach supports separate compilation

Programming Overhead

Program	# Lines of code	# Lines annotated
HTTP Server	603	20
Game Server	97	10
Database Server	244	24
java.util.Vector	992	35
java.util.Hashtable	1011	53
Image Recognition	567	8
Water	1850	31
Barnes	1850	16

RTJ Dynamic Checking Overhead

Program	Execution Time (sec)		Speed Up
	Dynamic Checks	Static Checks	
Water	2.55	2.06	24%
Barnes	21.6	19.1	13%
Image Recognition	8.10	6.70	21%
load	0.813	0.667	25%
cross	0.014	0.014	
thinning	0.026	0.023	10%
save	0.731	0.617	18%

Related Work

Related Work

- Ownership types
  - Clarke, Potter, Noble (OOPSLA '98), (ECOOP '01)
  - Clarke, Drossopoulou (OOPSLA '02)
  - Boyapati, Lee, Rinard (OOPSLA '01) (OOPSLA '02)
  - Boyapati, Liskov, Shrira, Moh, Richman (POPL '03) (OOPSLA '03)
  - Aldrich, Kostadinov, Chambers (OOPSLA '02)
- Region types
  - Tofte, Talpin (POPL '94)
  - Christiansen, Henglein, Niss, Velschow (DIKU '98)
  - Crary, Walker, Morrisett (POPL '99)
  - Grossman, Morrisett, Jim, Hicks, Wang, Cheney (PLDI '02)
  - Grossman (TLDI '03)

Our work unifies these areas

Related Work

- Systems that allow regions to be freed early
  - Aiken, Fahndrich, Levien (PLDI ’95)
  - Gay, Aiken (PLDI ’98) (PLDI ’01)
  - Crary, Walker, Morrisett (POPL ’99)
- Dynamic analysis to infer RTJ regions
  - Deters, Cytron (ISMM ’02)
- Static analysis to remove RTJ dynamic checks
  - Salcianu, Rinard (PPoPP ’01)
- Static analysis to help infer size of RTJ regions
  - Gheorghioiu, Salcianu, Rinard (POPL ’03)
- Real-time garbage collection
  - Baker (CACM ’78)
  - Bacon, Cheng, Rajan (POPL ’03)

Conclusions

Unified type system for OO languages

- Statically enforces several properties
  - Object encapsulation
  - Memory safety
  - Data race and deadlock freedom
  - Safe software upgrades
  - Safe real-time programming
- Type checking is fast and scalable
- Requires little programming overhead
- Promising way to make programs reliable

Ownership Types for Safe Region-Based Memory Management in Real-Time Java

Chandrasekhar Boyapati
Alexandru Salcianu
William Beebee
Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology