Permission-Based Programming Languages

Jonathan Aldrich Ronald Garcia Mark Hahnenberg
Manuel Mohr Karl Naden Darpan Saini Sven Stork
Joshua Sunshine Éric Tanter Roger Wolff

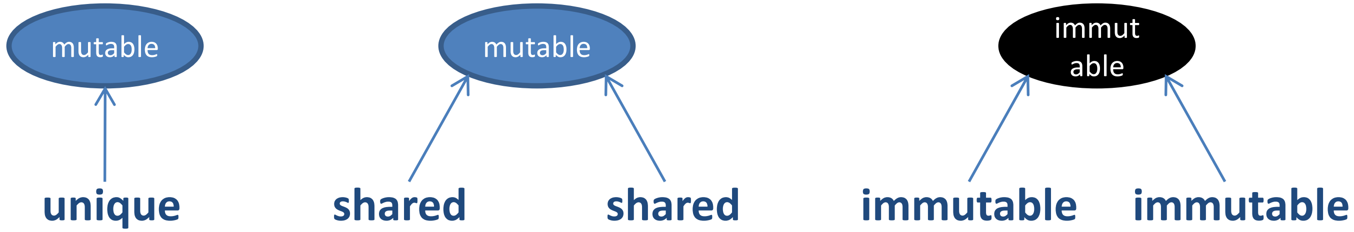
ICSE New Ideas and Emerging Results
May 25, 2011

Background: Permissions

- Permission systems associate every reference with both a type and a permission that restricts aliasing and mutability

```
var unique InputStream stream = new FileInputStream(…);
```

- Some permissions and their intuitive semantics [Boyland][Noble][…]



- Type system checks permission consistency
  - **unique**: no other references to the object
  - **immutable**: no-one can modify the object

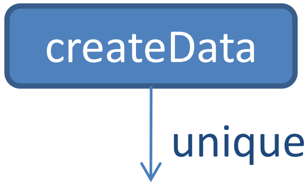
Permission-Based Language

- A language whose type system, object model, and run-time are co-designed with permissions in mind
  - Contrast: prior permission systems layered static permission checking onto existing languages

- Potential benefits
  - Design and encapsulation enforcement
  - Parallel execution
  - Explicit state change in the object model
  - Compile-time and run-time checking

Automatic Parallelization

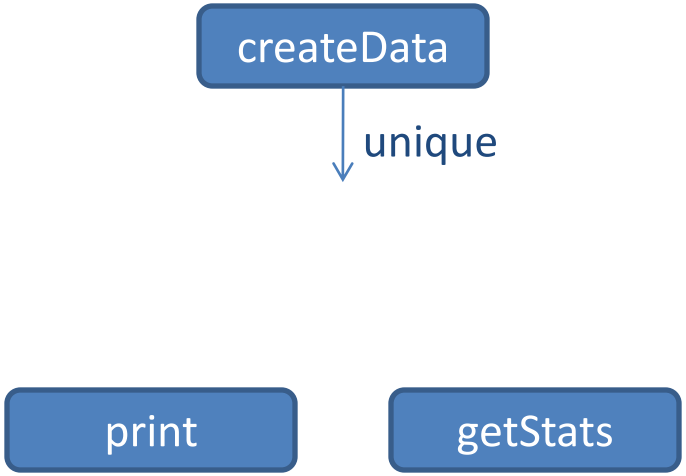
```
method unique Data createData();
```



```
val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
```

Automatic Parallelization

method unique Data createData();


createData

unique

print

getStats

val d = createData();

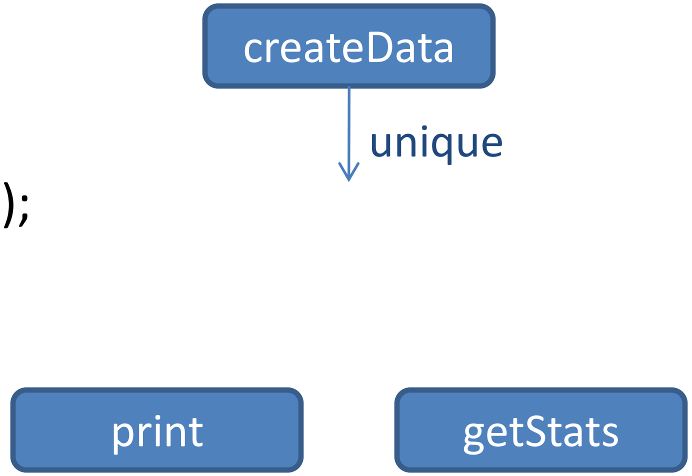
print(d);

val s = getStats(d);

manipulate(d, s);

Automatic Parallelization

```
method unique Data createData();
method void print(immutable Data d);
method unique Stats getStats(immutable Data d);
```



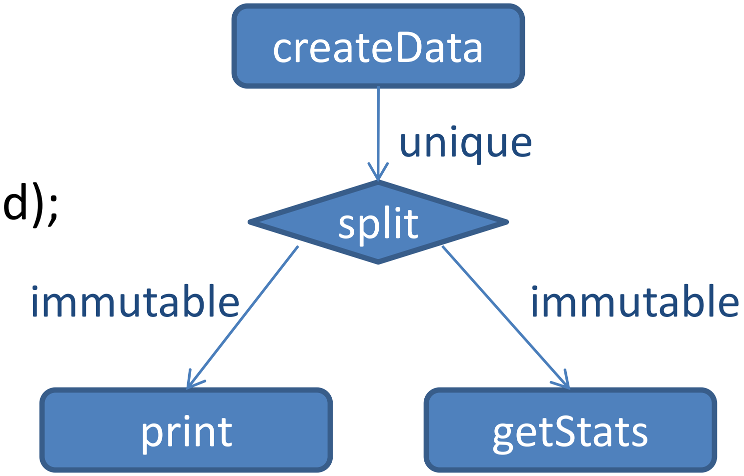
```
val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
```

Automatic Parallelization

method unique Data createData();

method void print(**immutable** Data d);

method unique Stats getStats(**immutable** Data d);



val d = createData();

print(d);

val s = getStats(d);

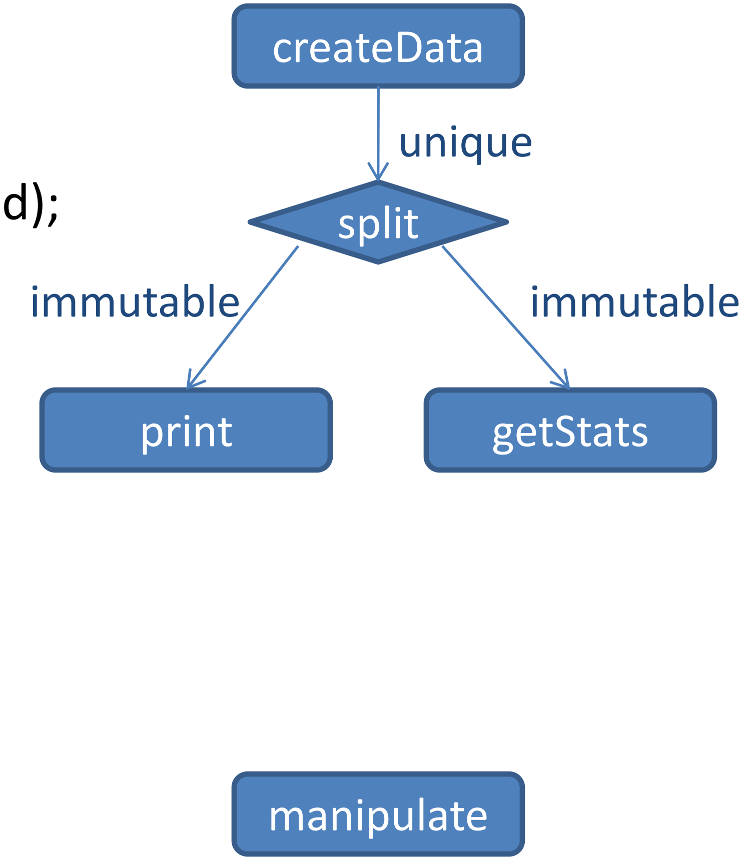
manipulate(d, s);

Automatic Parallelization

method unique Data createData();

method void print(**immutable** Data d);

method unique Stats getStats(**immutable** Data d);



val d = createData();

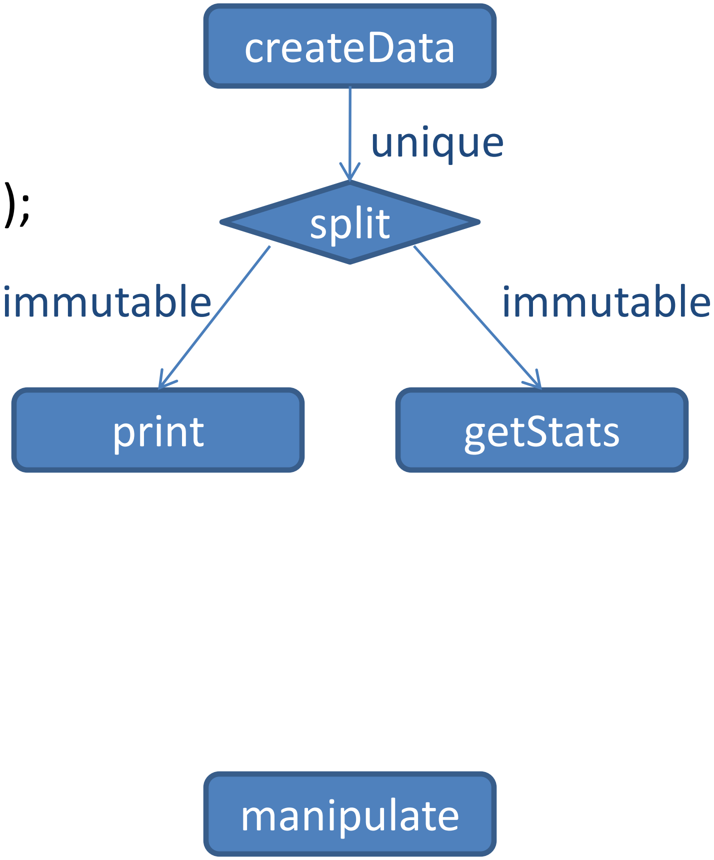
print(d);

val s = getStats(d);

manipulate(d, s);

Automatic Parallelization

```
method unique Data createData();
method void print(immutable Data d);
method unique Stats getStats(immutable Data d);
method void manipulate(unique Data d,
                                        immutable Stats s);


val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
```



Automatic Parallelization

method unique Data createData();

method void print(**immutable** Data d);

method unique Stats getStats(**immutable** Data d);

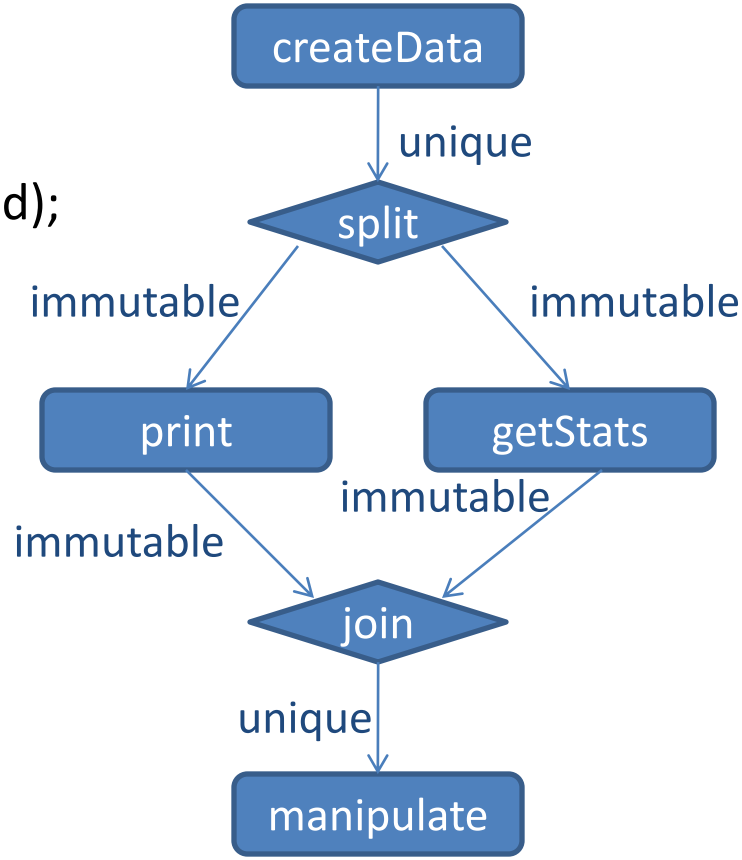
method void manipulate(**unique** Data d,
immutable Stats s);

val d = createData();

print(d);

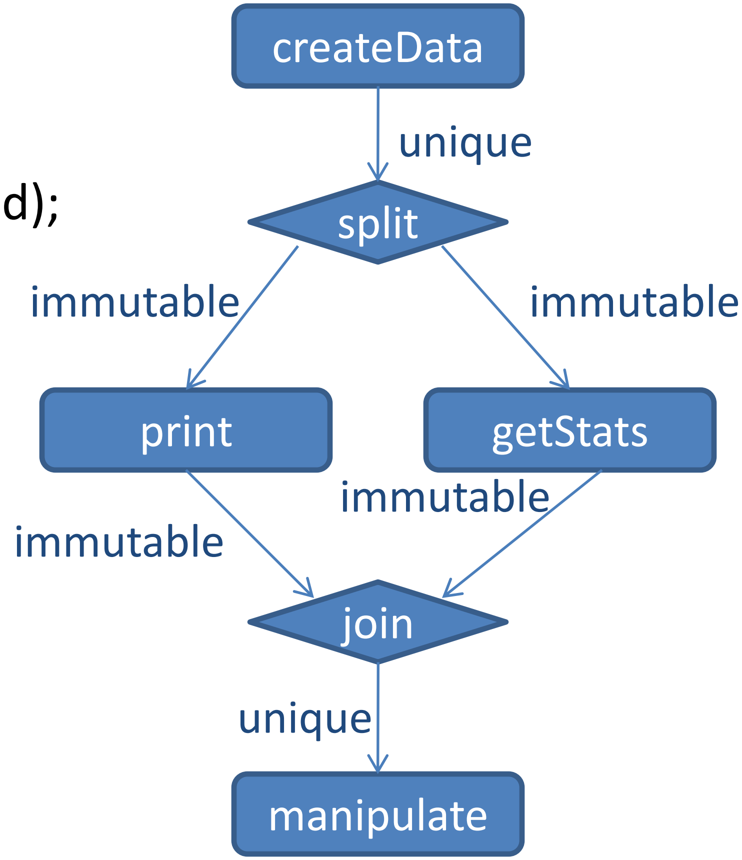
val s = getStats(d);

manipulate(d, s);



Automatic Parallelization

```
method unique Data createData();
method void print(immutable Data d);
method unique Stats getStats(immutable Data d);
method void manipulate(unique Data d,
                                   immutable Stats s);


val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
```


createData

unique

split

immutable

immutable

print

getStats

immutable

immutable

join

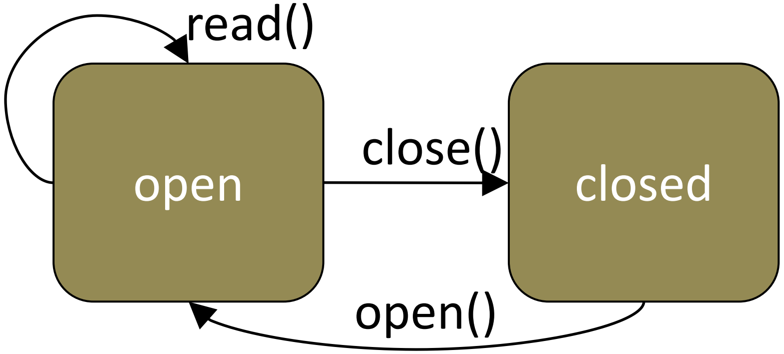
unique

manipulate

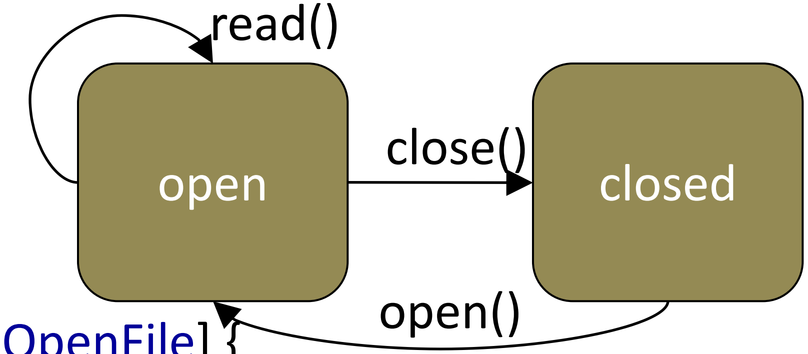
Casts can also be used to recover unique
The runtime checks the cast using reference counts

Explicit State Change

```
state File {
    val String filename;
}
```



Explicit State Change



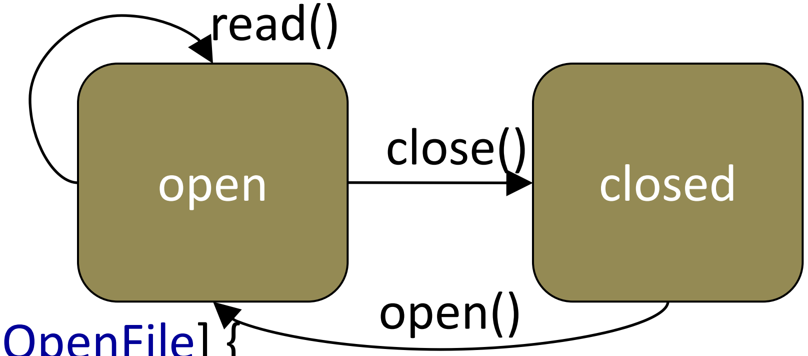
```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [unique ClosedFile>>OpenFile] {



    }
}
```

Explicit State Change

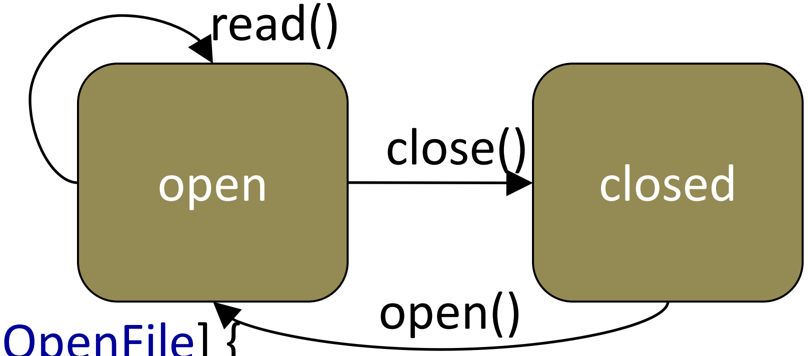
```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [unique ClosedFile>>OpenFile] {



    }
}
```

State transition

Permission / aliasing info.



Explicit State Change



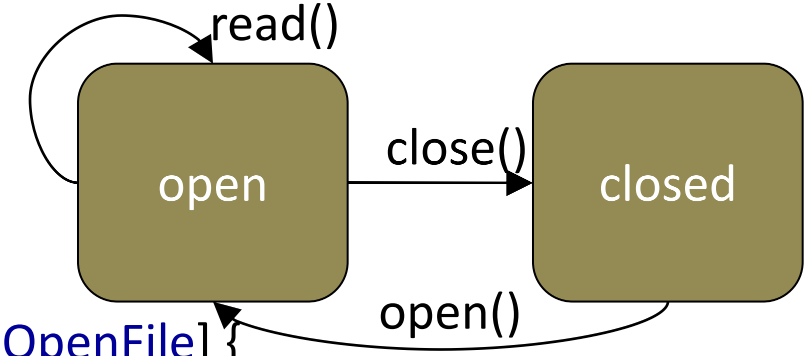
```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [unique ClosedFile>>OpenFile] {



    }
}
state OpenFile = File with {
    private val CFile fileResource;

    method int read();
    method void close() [OpenFile>>ClosedFile];
}
```

State transition

Permission / aliasing info.

New methods, Different representation

Explicit State Change



```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [unique ClosedFile>>OpenFile] {
        this <- OpenFile {
            fileResource = fopen(filename);
        }
    }
}
state OpenFile = File with {
    private val CFile fileResource;

    method int read();
    method void close() [OpenFile>>ClosedFile];
}
```

State transition

Permission / aliasing info.

State change primitive

New methods, Different representation

Plaid: A Permission-Based Language

- Currently exploring these ideas with Plaid
    - First-class abstractions for changing state
    - Naturally safe concurrent execution
    - Practical mix of static & dynamic checking

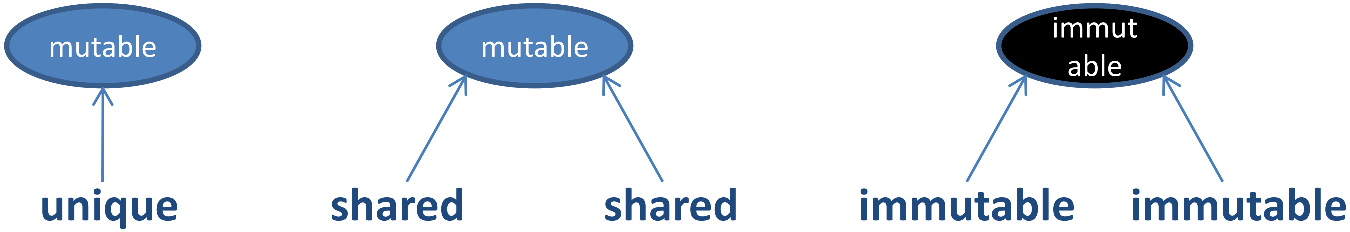
- Other research directions possible
    - Systems languages: permissions support memory management
    - Security: permissions help control access, information flow

- Status: compiler implemented, typechecker underway
    - Web-based interface available

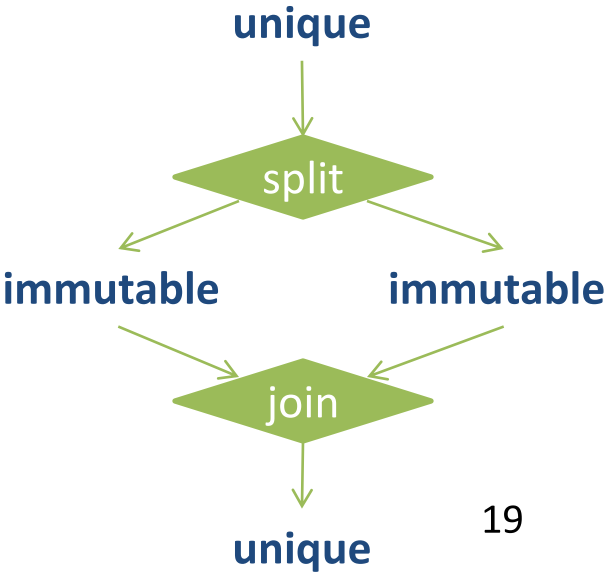
http://www.plaid-lang.org/

Permissions

- Annotations that describe aliasing and mutability [Noble][Boyland][…]


mutable
unique
mutable
shared
shared
immut able
immutable
immutable

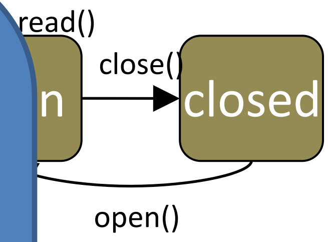
- Type system checks permission consistency
  - **unique**: no other references to the object
  - **immutable**: no-one can modify the object
- Enforced by splitting rules
  - E.g. split **unique** into two **shared** references
  - Can later join back into **unique**

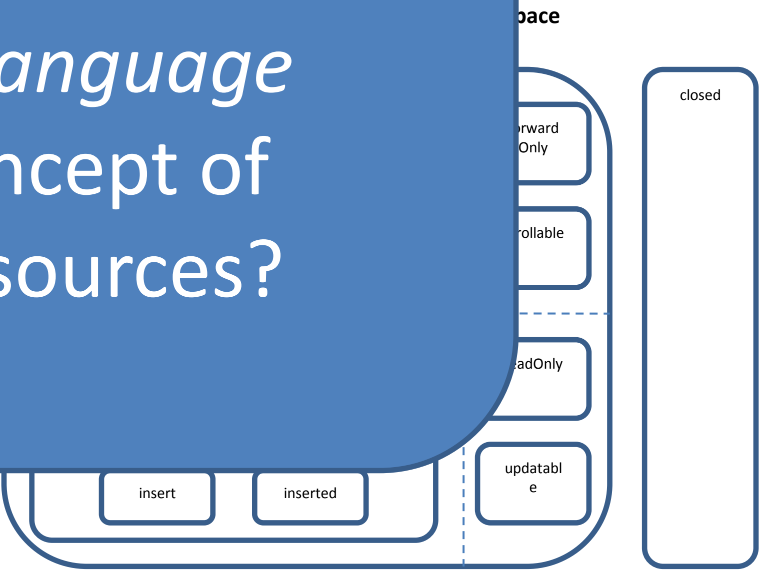

unique
split
immutable
immutable
join
unique

Resources and Composition

- Resource: stateful object whose use is constrained
  - I/O: fi
  - Sc
  - W
- Com
  - JD
- Easy
  - Of
  - Inadequate checking

<u>The Wild Idea:</u>
What if we organized a *programming language* around the concept of constrained resources?





Resources are Modeled with Typestates

Typestate-Oriented Programming

a **programming paradigm** in which:

programs are made up of dynamically created **resource objects**,

each object has a **typestate** that is **changeable**

and each state has an **interface**, **representation**, and **behavior**.

- Like Javascript, can add and remove fields and methods at run time
- Use **typestate** and **permissions** to statically typecheck these changes

Typestate-Oriented Programming is embodied in the language

PLAID

Implementing State Changes

```
method void open() [unique ClosedFile>>OpenFile] {
    this <- OpenFile {
        fileResource = fopen(filename);
    }
}
```

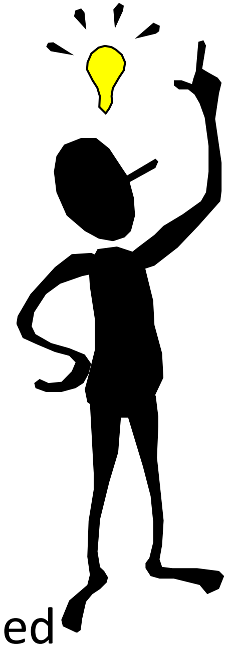
State change primitive

Values must be specified for each new field

:

Why Add State to the Language?

- The world has abstract states – so should programming languages
  - egg -> caterpillar -> butterfly; sleep -> work -> eat -> play; hungry <-> full
- Language influences thought [Boroditsky '09]
  - Language support encourages engineers to **think** about resources
    - Better designs, better documentation, more effective reuse
- Improved library specification and verification
  - Typestate defines when you can call read()
  - Make constraints that are only implicit today, explicit
- Expressive modeling and simpler reasoning
  - Without typestate: fileResource non-**null** if File is open, **null** if closed
  - With typestate: fileResource always non-**null**
    - But only exists in the FileOpen state



Research Challenges

- Practical typechecking
  - Use **linear permissions**, not types, to reason about limited resources
  - New **abstractions** that allow sharing
  - Efficient dynamic checks: how to cast to a **unique**?
  - Gradual types → **gradual permissions**
  - Checking states of **dynamic web pages**
  - Type **parameterization**
- Efficient compilation
  - New programming model requires new representation and optimization approaches
- Concurrency
  - Leveraging **permissions** for **safe deterministic concurrency**
  - Supporting controlled **non-determinism**

Try Plaid!

Try out Plaid in your browser!

Just click the run button at the bottom of the page to run your Plaid program. You can also insert some example code by using the menu on the right and experiment with it.

```
state Cell {
    method Cell getLeft() {
      left;
    }
    method getRight() {
      right;
    }
    val left;
    val right;

    method doPrint() {
      printVal();
      java.lang.System.out.print(" ");
    val rt = this.getRight().doPrint();
    }
    method print() {
  val lt = this.getLeft();
  lt.print();
    }
}

state Entry { }
```

Position: Ln 1, Ch 1 Total: Ln 253, Ch 4381

Select an example program:

Result:

```
running 1 state busy beaver:
0 1 0

running 2 state busy beaver:
0 1 1 1 1 0
```

BACKUP SLIDES

Related Work: Typestate

- Typestate [Strom and Yemeni ’86]
  - Captures a resource usage protocol as a set of states, with operations for each state
- Prior typestate work
  - Fugue: extension to objects [Deline & Fähndrich ’04]
  - Most systems forbid aliasing, nondeterminism, re-entrancy, concurrency, dynamic tests, flexible inheritance (all common in practice)
  - Very limited experience – only 1 significant case study (ADO.NET)
- Our Plural system had novel approaches to addressing limitations
  - State guarantees; state dimensions; new permission kinds; union and intersection types; re-entrant safe packing; additive conjunction; supertype invariants [OOPSLA’07]; atomicity [OOPSLA ’08]
- Plural is the first demonstrated to scale to real code [ECOOP’09]
  - Specification: JDBC (10 kLOC), Collections, Regular Expressions…
  - Verification: PMD (38 kLOC), Apache Beehive (aliasing challenges)

Checking Typestate

```
method void openHelper(ClosedFile>>OpenFile aFile) {
    aFile.open();
}
```

This method transitions the argument from ClosedFile to OpenFile

Must leave in the ClosedFile state

```
method int readFromFile(ClosedFile f) {
    openHelper(f);
    val x = computeBase() + f.read();
    f.close();
    return x;
}
```

Use the type of openHelper

f is open so read is OK

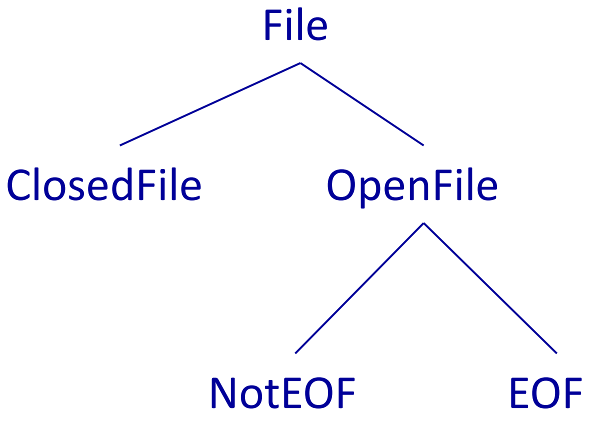
Correct postcondition; f is in ClosedFile

Question: How do we know computeBase doesn’t affect the file (thorugh an alias)?

Typestate Permissions

- **unique** OpenFile
  - File is open; no aliases exist
  - Default for mutable objects
- **immutable** OpenFile
  - Cannot change the File
    - Cannot close it
    - Cannot write to it, or change the position
  - Aliases may exist but do not matter
  - Default for immutable objects

[Chan et al. ’98]



- **shared** OpenFile@NotEOF [OOPSLA ’07]
  - File is aliased
  - File is currently not at EOF
    - Any function call could change that, due to aliasing
  - It is forbidden to close the File
    - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases
- **none** – no permission

Roadmap

- Introduction
- Typestate-Oriented Programming
- Plaid’s Compositional Object Model
- Parallel by Default Programming
- Conclusion

Object Model Goals

- Support for object-oriented and functional programming
  - Objects and subtyping; functions and type abstraction
- Abstract, flexible interfaces
  - Support after-the-fact interface extraction without modifying code
    - compare Java: must modify classes to implement the new interface
- Clean, effective code reuse
  - Same level of convenience as multiple inheritance
  - Avoid problems like name conflicts, unintentional open recursion
- Flexibility
  - Ways to escape from type system when it is too strict
- Information hiding
  - Avoid violations of abstraction
    - e.g. instanceof on a datatype that's not conceptually a tagged union

Functional Programming Support

```
val ADT = new {
    type set = List;
    method set<T> union(
            set<T> s1, set<T> s2) {
        s1.appendList(s2);
    }
} as {
    type set <: { type E; };
    val union: set<T> * set<T> -> set<T>
}
```

```
method List<U>
    map(’T -> ’U f)(List<T> lst) {
    match(lst) {
        case Cons(e,rest) =>
            makeCons(f(e), map(f)(rest))
        case Nil => Nil
    }
}

… map (fn (int x) => x + 1) (myIntList) …
```

Structural Types

```
type IntCollection = {
    method IntCollection add(int newInt);
}

type IntList = {
    method IntList add(int newInt);
    method int get(int index);
}

IntList list = makeMyList();
IntCollection coll = list;      // implicit structural subtyping
```

Safe Code Reuse via Composition

```
state AbstractCollection = {
    method void addAll(Collection other) {
        other.do (fn (int x) => add(x))
    }
    requires open method void add();
}
```

Reusable abstract state

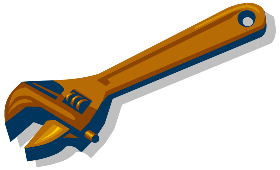
Selective open recursion [SAVCBS '04]: open recursion is only used in calls to methods marked **open**. The **open** keyword documents that subclasses can override self-calls to this method. Other methods can be overridden but self calls are unaffected.

```
state LinkedList = AbstractCollection[add->addLast] with {
    method void add() { … }
}
```

Trait element renaming

Trait-based composition

Static & Dynamic Checking in Plaid



- Typestate and permissions express *design intent*
  - Typechecking verifies intent statically
  - But sometimes static checking fails, even for OK programs
  - Need to have dynamic checks as a fallback
- Principle
  - All assertions about typestate and permissions can be checked either statically or dynamically
- Features
  - Gradual types [Siek and Taha '06]
    - can omit some types, statically check as much as possible
  - Casts to types, states, and permissions
- Research questions
  - How does gradual typing generalize to permissions?
  - How to check casts to **unique**?

Information Hiding Challenges: Dynamic Types and Pattern Matching

```
set = new Collection with {
    val List<E> members;
    method Set<E> union(Set<E> other);
} as Collection with {
     method Set<E> union(Set<E> other);
}

dynamic dset = set;       // dynamic typing
dset.members.add(e); // FAIL at run time
```

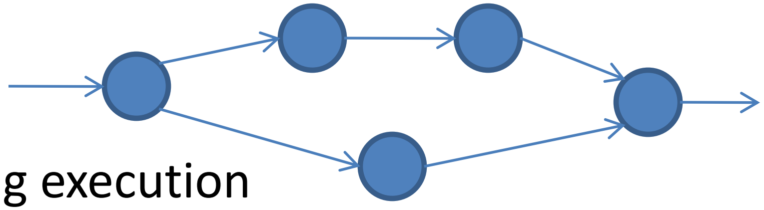
```
type TestMember = {
    boolean isMember(E e); }
state List = { … }
state ArrayList case of List = { … }

List myList = new ArrayList{};
// match OK – ArrayList a case of List
match (myList) {
    case ArrayList al { … }
}

TestMember tm = myList;
// compile-time error: TestMember
// does not support case analysis
match (tm) { … }
```

Demonstration

Roadmap

- Introduction
- Typestate-Oriented Programming
- Plaid's Compositional Object Model
- Parallel by Default Programming
  - Plaid's instantiation of the ÆMINIUM project
- Conclusion

Explicit Dependencies in Plaid

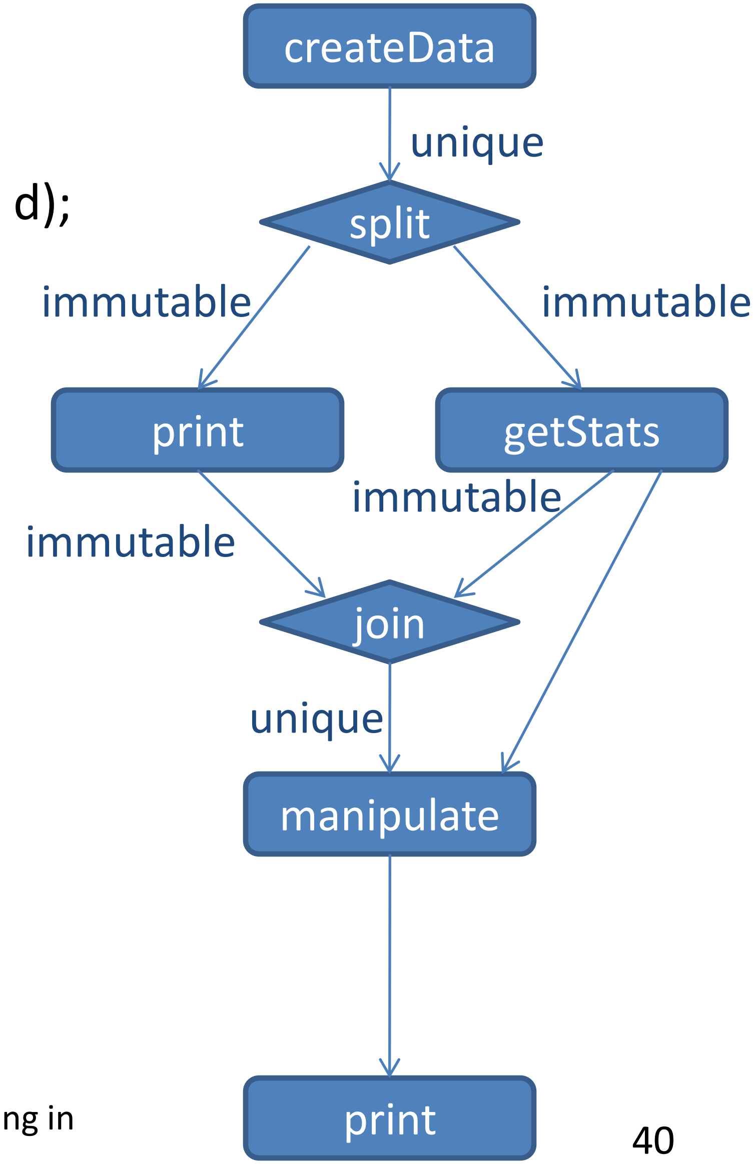


- Concurrency is a major challenge
  - Avoiding race conditions, understanding execution
- Inspiration: functional programming is “naturally concurrent”
  - Up to data dependencies in program
- Idea: use permissions to construct dataflow graph
  - Easier to track dependencies than all possible concurrent executions
  - Functional programming passes data explicitly to show dependencies
  - For stateful programs, we **pass permissions explicitly** instead
- Consequence: stateful programs can be naturally concurrent
  - Furthermore, we can provide strong reasoning about correctness

Features: Sharing and Dependencies

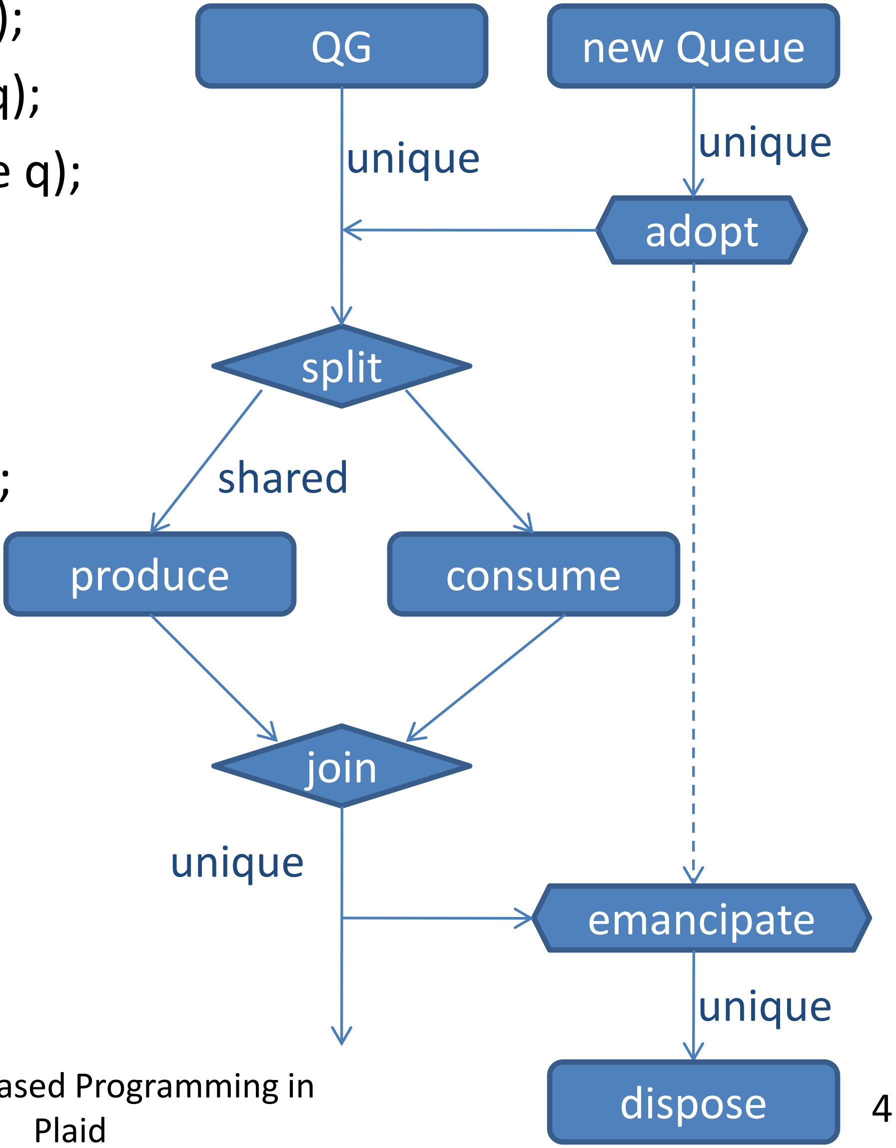
method unique Data createData();
method void print(**immutable** Data d);
method unique Stats getStats(**immutable** Data d);
method void manipulate(**unique** Data d,
immutable Stats s);

val d = createData();
print(d);
val s = getStats(d);
manipulate(d, s);
print(d);



Features: Sharing and Dependencies

```
method void produce(‘QG Queue q);
method void consume(‘QG Queue q);
method void dispose(unique Queue q);

group QG;
val QG Queue q = new Queue;
split QG: produce(q) || consume(q);
q.dispose();
```



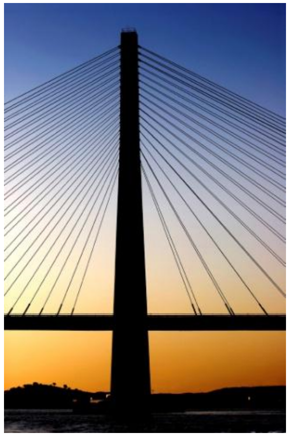
Consequences: Safe Concurrency

- Programmers think only about dependencies
  - Move away from a sequential model
- Programs execute in parallel by default
  - Execution is deterministic except for uses of **split**
- Compatible with shared state, nondeterminism when needed
  - Shared state is tracked with permissions
  - Non-determinism is explicit (in **split** blocks)
  - Non-determinism is scoped to a part of the program and to a specific group of shared data
- Reasoning support
  - Consistent synchronization
  - Typestate protocol verification
  - Synchronization granularity (sufficient to ensure typestate)

Roadmap

- Introduction
- Typestate-Oriented Programming
- Plaid’s Compositional Object Model
- Parallel by Default Programming
- Conclusion

A Bridge to Existing Languages



- Familiarity
  - use Java syntax wherever possible
  - when no clear language design choice, use Java's
    - fix some glaring problems like nulls (what Hoare calls his $1 billion mistake)
- Compatibility
  - compile to platforms, like the JVM, that have good existing libraries

Current Plaid Language Research

Core type system	Darpan Saini, Joshua Sunshine
Object model	Karl Naden
Typestate model	Filipe Militão, Luís Caires (FCT)
Gradual typing	Roger Wolff, Ron Garcia, Eric Tanter (U. Chile)
Concurrency	Sven Stork, Paulo Marques (U. Coimbra)
Web programming	Joshua Sunshine
Permission parameters	Nels Beckman
Compilation/typechecking	Karl Naden, Joshua Sunshine, Mark Hahnenberg, Sven Stork

The Plaid Language

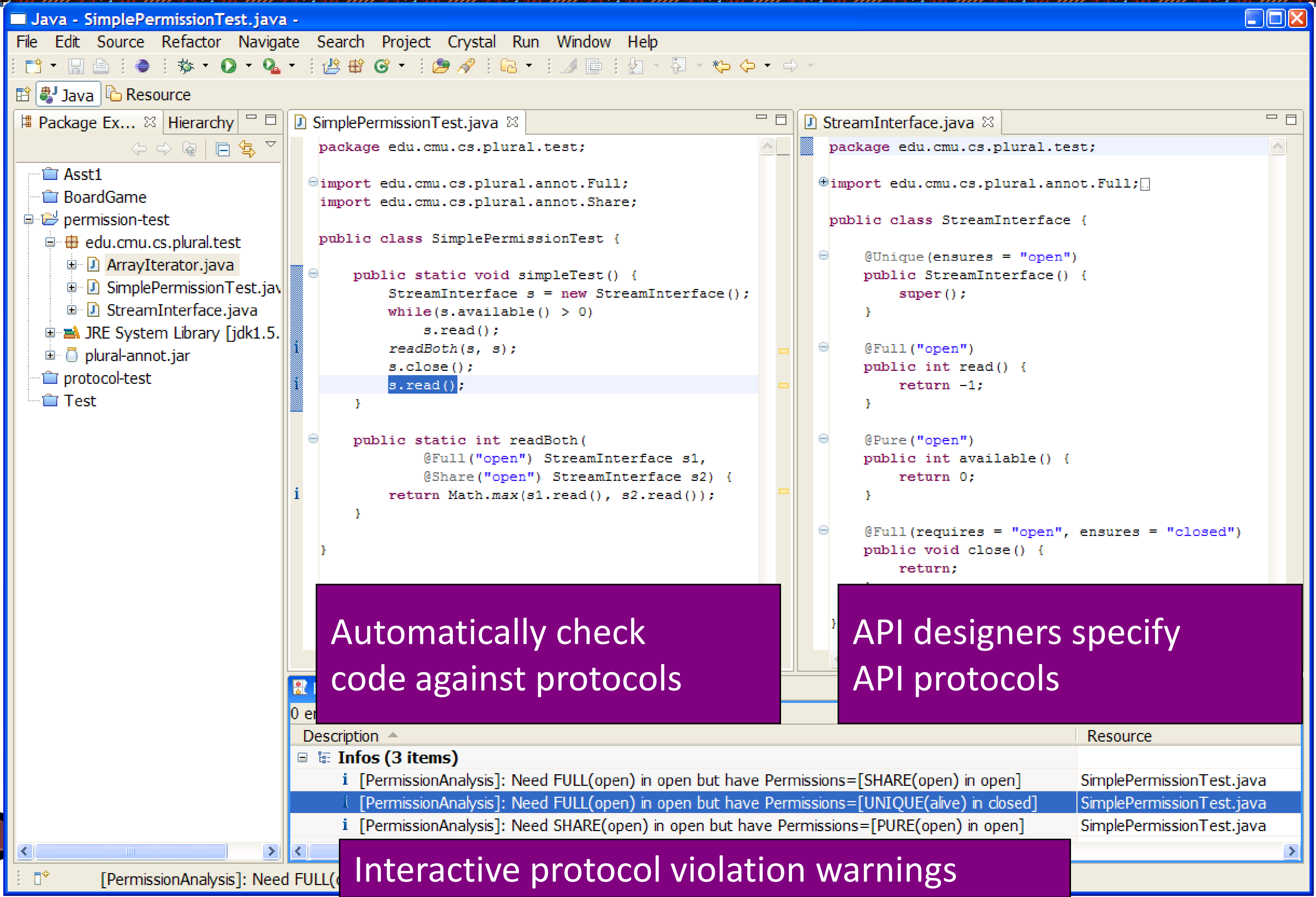
- Supports programming with resources
  - First-class abstractions for characterizing state
  - Naturally concurrent execution
  - Practical mix of static & dynamic checking
- Opens a new subfield of research
  - Languages based on changeable states and permissions
- Work in progress
  - Compiler implemented (in Java, for now)
  - Plaid typechecker (in Plaid) underway

http://www.plaid-lang.org/

Questions?

Additional Slides

Plural: Typestate in Java



```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;
import edu.cmu.cs.plural.annot.Share;

public class SimplePermissionTest {

    public static void simpleTest() {
        StreamInterface s = new StreamInterface();
        while(s.available() > 0)
            s.read();
        readBoth(s, s);
        s.close();
        s.read();
    }

    public static int readBoth(
            @Full("open") StreamInterface s1,
            @Share("open") StreamInterface s2) {
        return Math.max(s1.read(), s2.read());
    }

}
```

```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;

public class StreamInterface {

    @Unique(ensures = "open")
    public StreamInterface() {
        super();
    }

    @Full("open")
    public int read() {
        return -1;
    }

    @Pure("open")
    public int available() {
        return 0;
    }

    @Full(requires = "open", ensures = "closed")
    public void close() {
        return;
```

Description	Resource
Infos (3 items)	
[PermissionAnalysis]: Need FULL(open) in open but have Permissions=[SHARE(open) in open]	SimplePermissionTest.java
[PermissionAnalysis]: Need FULL(open) in open but have Permissions=[UNIQUE(alive) in closed]	SimplePermissionTest.java
[PermissionAnalysis]: Need SHARE(open) in open but have Permissions=[PURE(open) in open]	SimplePermissionTest.java

[PermissionAnalysis]: Need FULL(

Automatically check code against protocols

API designers specify API protocols

Interactive protocol violation warnings

State: Powerful but Dangerous

- Shared Mutable State
  - Convenient communication abstraction, increases efficiency
  - Models stateful aspects of the world
  - BUT creates potential for inconsistency

- Challenges: Composition
  - Stateful libraries impose ordering constraints
    - Cannot invoke read() after closing a file

- Challenges: Concurrency
  - Data race
    - Concurrent read and write to shared state violates an invariant
  - Abstract state race
    - Expedia reports an available flight, but it is gone when you try to reserve
  - Either may result in a crash, vulnerability, or incorrect results