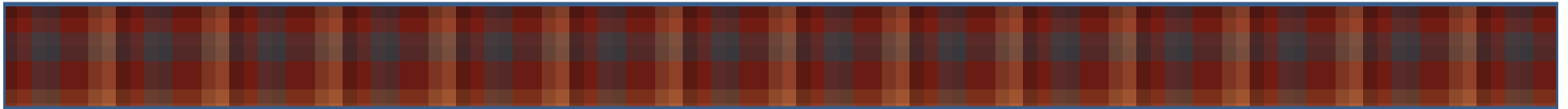


Permission-Based Programming in Plaid



IFIP 1.16 Working Group on Language Design

London

March 1, 2012

Types Should Do More

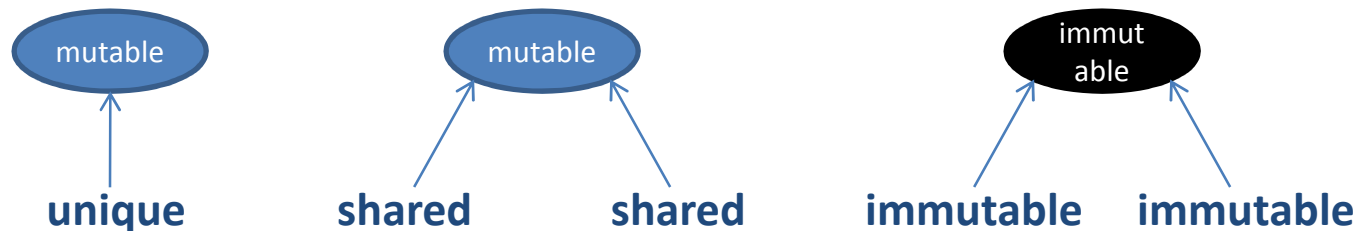
- One of our biggest problems is reasoning about aliased/concurrent mutable state
 - *the* hard problem in verification
 - *the* hard problem in concurrent programming
 - *the* hard problem in practical system building
- Nevertheless state is too useful to go away
 - Otherwise it would have been long gone by now
- Types should tell us what state is mutable and aliased
 - Safe, tool-supported reasoning about state updates
 - Safe, automatic concurrency
 - More productive programming in the large

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
- Prototype example: the **PLAiD** programming language at CMU
- Potential benefits
 - Explicit state change in the object model
 - Automatic parallel execution
 - Design and encapsulation enforcement
 - Compile-time and run-time checking



Outline

- **Plaid language audience and goals**
- Explicit state change in the object model
- Automatic parallel execution
- Design principles and lessons learned

Plaid Language Audience and Goals

- Target audience and domain
 - Professional programmers
 - Applications programming
 - Programs in which changing state is important
 - Programs for which concurrency is desired
- Goals
 - Explore the consequences of a permission-based programming language design
 - Explore an object model based on changing state
 - Explore an implicit concurrency model based on permissions
 - Enough realism to do large-scale case studies
 - Plaid is already bootstrapped
 - As much simplicity as is compatible with the above goals

Outline

- Plaid language audience and goals
- **Explicit state change in the object model**
- Automatic parallel execution
- Design principles and lessons learned

Empirical Studies of Typestate Protocols

- How commonly are protocols defined & used?
 - Corpus study on 2 million LOC: Java standard library, open source [Beckman, Kim, & A, ECOOP 2011]
 - 7% of all types define object protocols
 - c.f. 2.5% of types define type parameters using Java Generics
 - 13% of all classes act as object protocol clients
- Do protocols cause practical development challenges?
 - Empirical study of postings on an ASP.NET help forum
 - 75% of problems identified involved temporal constraints [Ciera Jaspan dissertation, 2011]

Typestate-Oriented Programming

A **new programming paradigm** in which:

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

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

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

– compare: prior typestate work considered only changing interfaces

[Strom and Yemeni, Deline and Fähndrich]

Typestate-oriented Programming is embodied in the language



PLAID

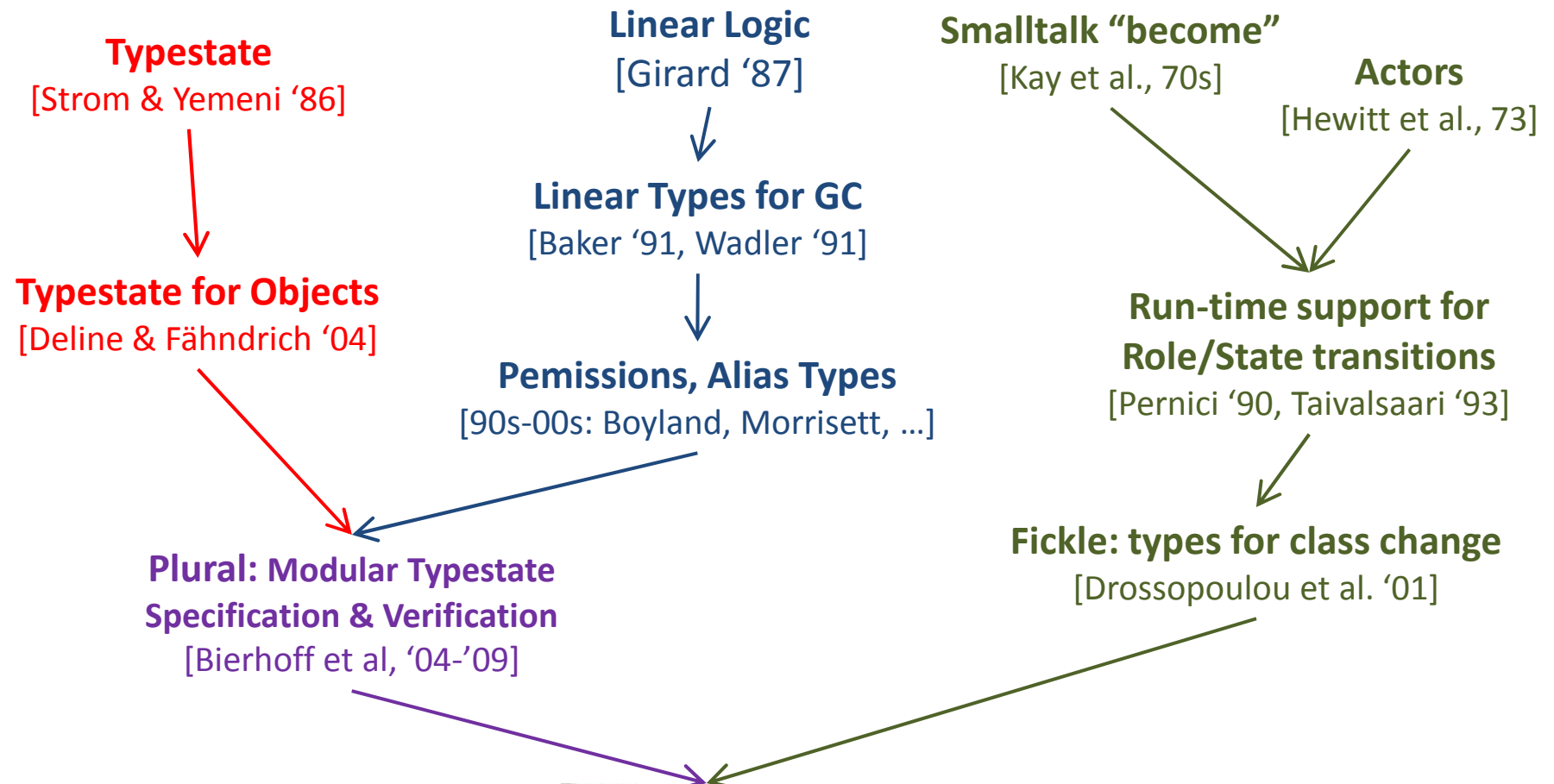
*Plaid (rhymes with “dad”) is a pattern of Scottish origin, composed of multicolored crosscutting threads



PLAID

Plaid: a Permission-Based
Programming Language

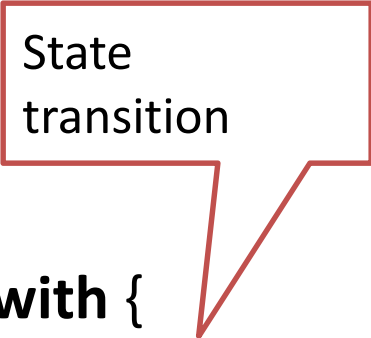
Historical Perspective



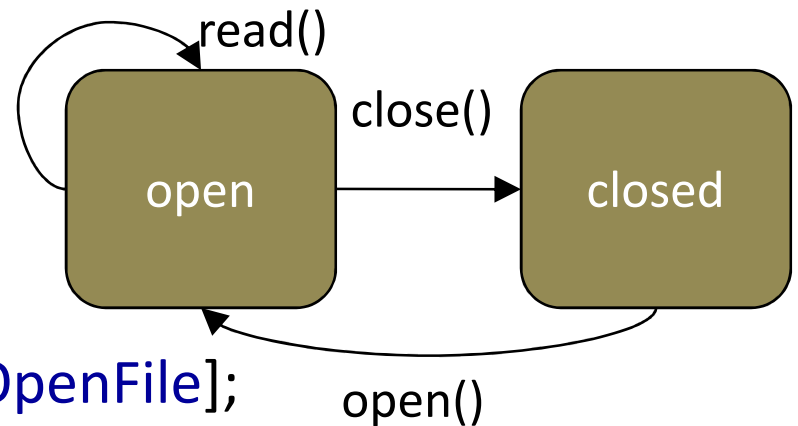
Plaid: a Permission-Based
Programming Language

Typestate-Oriented Programming

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



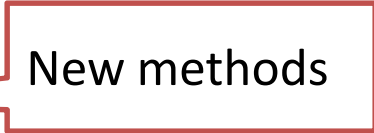
```
state ClosedFile = File with {  
  method void open() [ClosedFile>>OpenFile];  
}
```



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



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



Implementing Typestate Changes

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

Typestate change
primitive – like
Smalltalk *become*

Values must be
specified for
each new field

:

Why Typestate in the Language?

- The world has state – so should programming languages
 - egg -> caterpillar -> butterfly; sleep -> work -> eat -> play; hungry <-> full
- Language influences thought [Sapir '29, Whorf '56, Boroditsky '09]
 - Language support encourages engineers to **think** about states
 - Better designs, better documentation, more effective reuse
- Improved library specification and verification
 - Typestates define when you can call read()
 - Make constraints that are only implicit today, explicit
- Expressive modeling
 - If a field is not needed, it does not exist
 - Methods can be overridden for each state
- Simpler reasoning
 - Without state: fileResource non-**null** if File is open, **null** if closed
 - With state: fileResource always non-**null**
 - But only exists in the FileOpen state



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 (through an alias)?



Typestate Permissions

- **unique** OpenFile

- File is open; no aliases exist
- Default for mutable objects

pure resource-based programming

- **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

pure functional programming

- **shared** OpenFile@NotEOF

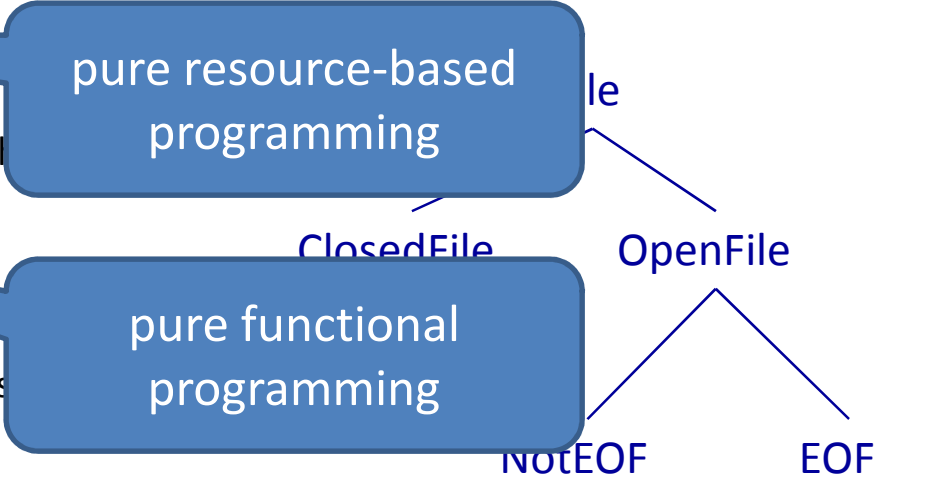
- File is aliased
- File is currently not at EOF
 - Any function call could change that, due to aliases
- It is forbidden to close the File
 - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases

shared OpenFile@OpenFile is (almost) traditional object-oriented programming

- **full** – like **shared** but is the exclusive writer

- **pure** – like **shared** but cannot write

Key innovations vs. prior work (c.f. Fugue, Boyland, Haskell monads, separation logic, etc.)



Permission Splitting

- Permissions may not be duplicated
 - No aliases to a unique object!
- Splitting that follows permission semantics is allowed, however
 - **unique** → **full**
 - **unique** → **shared**
 - **unique** → **immutable**
 - **shared** → **shared, shared**
 - **immutable** → **immutable, immutable**
 - **X** → **X, pure** *// for any non-unique permission X*
- How do we get unique back after we give it up? Stay tuned...

Parametric Polymorphism

```
state Collection {  
  type TElem;
```

Type parameter must now include state and permission

```
  void add(TElem>>none e);
```

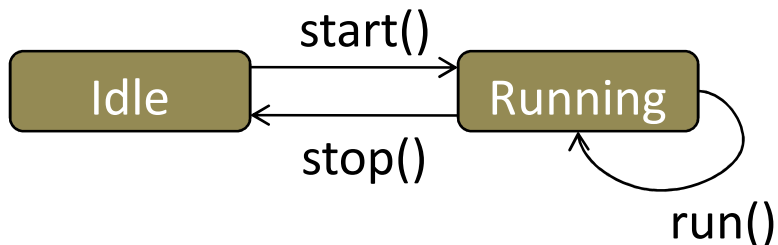
Adding an element to the collection removes the client's permission to it (e.g. to ensure unique objects are unaliased)

```
  TElem removeAny();
```

If we want to get an element, we must remove it from the collection (to avoid aliasing).

```
}
```

Example: Interactors




```
state Idle {  
    void start() [Idle >> Running];  
}  
state Running {  
    void stop() [Running >> Idle];  
    void run(InputEvent e);  
}
```

```
state MoveIdle extends Idle {  
    GraphicalObject go;  
    void start() [Idle >> Running] {  
        this <- Running {  
            void run(InputEvent e) {  
                go.move(e.x,e.y);  
            }  
            void stop() [Running >> Idle] {  
                this <- MoveIdle{}  
            }  
        }  
    }  
}
```

Typestate Checking Hypotheses

- Relatively simple permission mechanisms are sufficient to statically check typestate properties in most Plaid code
 - (for the exceptions, see Gradual Types, below)
- Both permissions and typestates express important design constraints, helping developers correctly evolve software
- Permissions can help make automated verification tools more effective

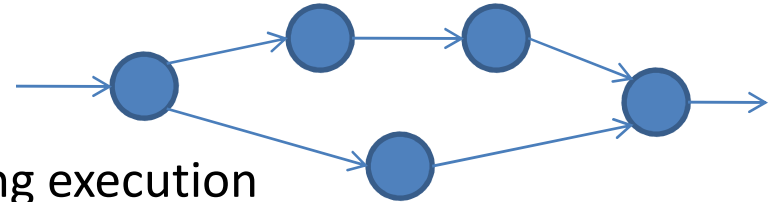
Outline

- Plaid language audience and goals
- Explicit state change in the object model
- **Automatic parallel execution in**  *
- Design principles and lessons learned

*The Æminium project is a collaboration between CMU and the University of Coimbra, located on the site of the ancient Roman town of Æminium

ÆMINIUM: Explicit Dependencies for Concurrency

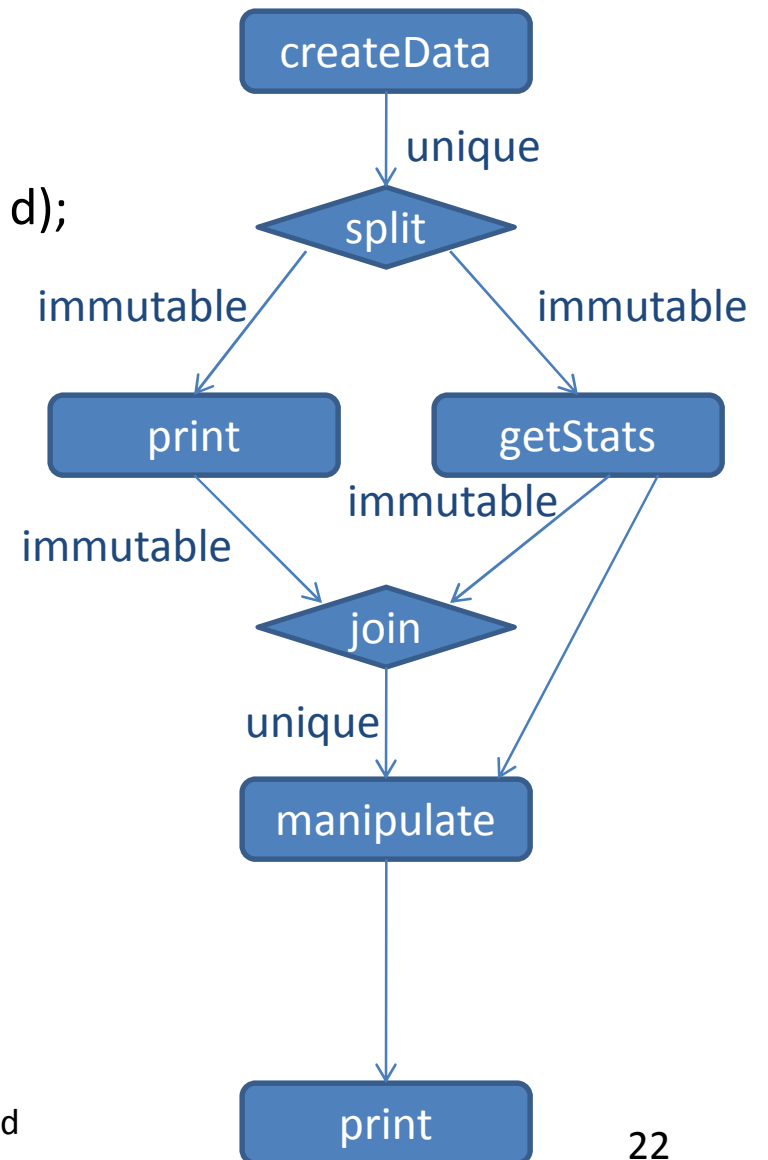
- Concurrency is a major challenge
 - Avoiding race conditions, understanding execution
- Inspiration: functional programming is “naturally concurrent”
 - Up to data dependencies in program
- Idea: use permissions to construct dataflow graph
 - Easier to track dependencies than all possible concurrent 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);

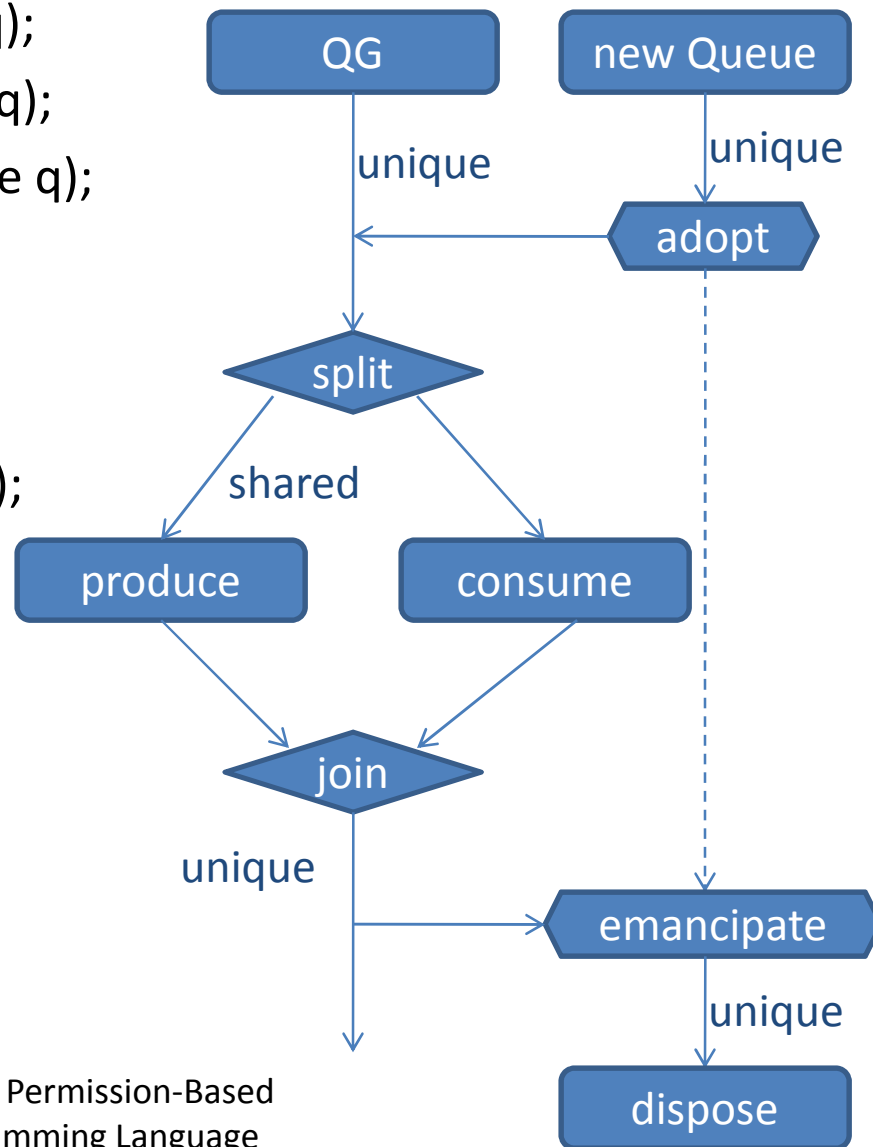
```
val d = createData();  
print(d);  
val s = getStats(d);  
manipulate(d, s);  
print(d);
```



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();
```



Concurrency by Default Hypotheses

- It is easier for programmers to think correctly about dependencies rather than multiple threads of control
- Programmers using the Parallel by Default model will expose more concurrency than is typically exposed in explicit concurrency models
- Making non-deterministic concurrency *explicit* (with **split** blocks) and *scoped* (to the body of the split block, and to the data whose permission is split) helps avoid programmer errors
- **EMINIUM** can support additional kinds of reasoning about concurrent programs:
 - Consistent synchronization
 - Typestate protocol verification
 - Synchronization granularity (sufficient to ensure typestate)

Outline

- Plaid language audience and goals
- Explicit state change in the object model
- Automatic parallel execution
- **Design principles and lessons learned**
- Other topics
 - Gradual permissions and typestate
- Status, Demonstrations and Conclusions

Principles and Lessons

- Principles
 - Everything should make sense in the dynamic language
 - Semantics do not depend on typechecking
 - Even permissions asserted in the static code have an interpretation in the dynamic code (see gradual permissions)
 - Document, don't restrict
 - Don't restrict the language to allow reasoning. Always provide a general base case.
 - Instead, allow programmers to document cases where restrictions are *desired*
- Lessons
 - Self hosting may not have been a good idea
 - Spent considerable time
 - The compiler may not be the best example for Plaid

Outline

- Plaid language audience and goals
- Explicit state change in the object model
- Automatic parallel execution
- Design principles and lessons learned
- **Other topics**
 - **Gradual permissions and typestate [ECOOP '11]**
 - **Trait composition in Plaid [OOPSLA '11]**
 - **Making borrowing more practical [POPL '12]**
 - **Novel class/prototype hybrid model [hidden in OOPSLA '11]**
- Status, Demonstrations and Conclusions

Gradual Permissions and Assertions

- Static typechecking is good, but imperfect
 - Sometimes type information is lost and must be recovered
 - Type information makes programs more verbose, and must be changed when the program changes
 - Some programmers prefer to work in dynamically typed languages
- Principle: all assertions about typestate and permissions should be checkable either *statically* or *dynamically*
- Permission assertions
 - Check that a permission is consistent with other current permissions
- Gradual permissions
 - Programmer can selectively omit permissions
 - Statically checks as much as possible, dynamically check the rest
 - Based on gradual types [Siek and Taha '06]

Assertions

```
val unique File file = new File;  
fileCollection.add(file)  
// other code...  
fileCollection.removeAll();  
assert<unique File> file; // verify there are no aliases  
file.close();
```



- The assertion must verify
 - For **unique**, that there are no aliases
 - For **immutable**, that there are no write aliases (i.e. **unique** or **shared**)
 - For **shared**, that there are no **unique** or **immutable** aliases, and the asserted state invariant is consistent with other **shared** invariants

Assertion Checking Strategies

- Infeasible: garbage collect on each cast
- Eager reference counting
 - Keep track of how many references exist to each object, and what permissions they have
 - Cast succeeds if no conflict with other references
 - Benefits: early warnings about errors, can replace garbage collector
 - Drawback: hard to implement efficiently, especially in concurrent systems
- Lazy verification
 - All asserts succeed, but invalidate conflicting references
 - Raises a warning of reference with conflicting permission is used
 - Benefits: Only raises real errors, efficient to check
 - Drawback: raises error much later (*in contrast to assertions in C or Java*)



Gradual Permissions

```
method int read(shared File@OpenFile f) { ... }  
method getResource() { ... }
```

```
val file = getResource();  
// other code...  
read(file);
```



- C
 - T
- Research Hypothesis: Typestate-oriented programming has benefits even for dynamically-typed code:
- Code matches design better, facilitating evolution
 - Typestate defects are caught earlier and more explicitly, making them easier to isolate and fix
 - Permission assertions are useful in their own right, e.g. immutable or exclusive writer

Outline

- Plaid language audience and goals
- Explicit state change in the object model
- Automatic parallel execution
- Design principles and lessons learned
- Other topics
 - Gradual permissions and typestate
- **Status, Demonstrations and Conclusions**

Plaid Status

- Theoretical models and soundness results
 - PlaidCore – structural core of Plaid permissions system
 - Gradual Featherweight Typestate – runtime typestate & permissions
 - μ Aeminiun – absence of race conditions
- Implementation
 - Prototype compiler
 - bootstrapped in Java; reimplementaion in Plaid nearly complete
 - Typechecker for unique and immutable permissions
 - \mathcal{A} Eminium runtime supports a speedup on simple examples
- Prior research
 - Plural: permissions system, typestate checking for Java
 - Sync or Swim: typestate checking in a concurrent setting
 - Case studies: demonstrate practicality of permission/typestate system on ~50,000 lines of code

Demonstration

- Fun example: Turing machine
 - State-based computation model!
- Uses of states
 - State of the Turing machine controller
 - State of each cell on the tape

```
method writeOne() { this <- One }
```
 - Representing infinite tapes: LeftEnd, RightEnd, InnerCell
 - GUI states: *unchecked in Swing!*
 - Empty, Hidden, Visible windows
 - Initialization state
 - Size set
 - Parent set
- File example (open/closed)

Demonstration (2)

Game of Life in Plaid on the web:

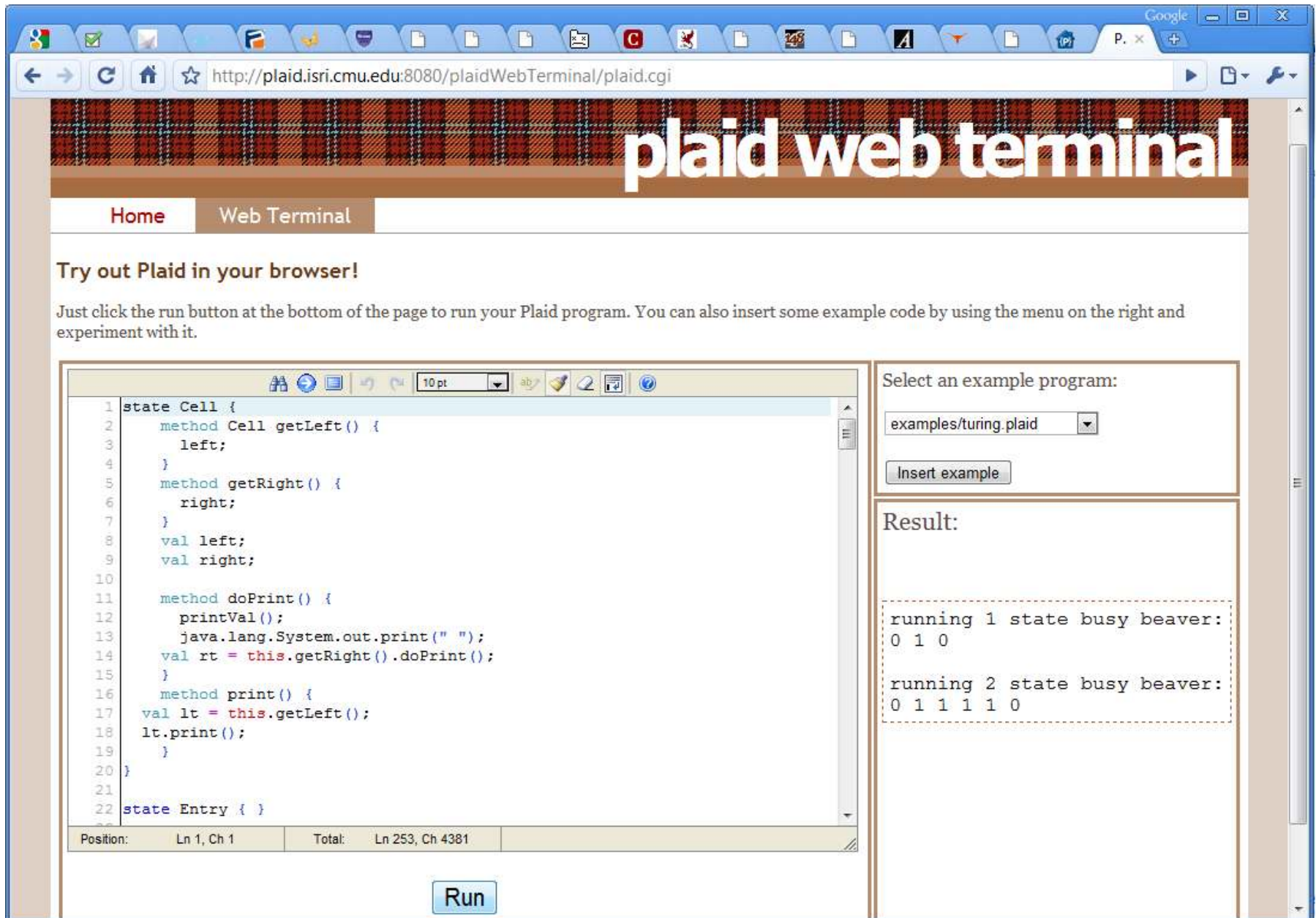
<http://www.cs.cmu.edu/~aldrich/plaid/life/>

States are used to represent the current state of a cell
(alive/dead) and the cell's situation (crowded, lonely, etc.)

Code:

<http://www.cs.cmu.edu/~aldrich/plaid/life/GameOfLife.zip>

Try Plaid!



The screenshot shows a web browser window with the URL `http://plaid.isri.cmu.edu:8080/plaidWebTerminal/plaid.cgi`. The page features a plaid-themed header with the text "plaid web terminal". Below the header, there are two navigation tabs: "Home" and "Web Terminal", with "Web Terminal" being the active tab. The main content area contains the text "Try out Plaid in your browser!" followed by instructions: "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."

The interface is divided into two main sections. On the left is a code editor with a toolbar at the top showing icons for undo, redo, and font size (set to 10 pt). The code in the editor is as follows:

```
1 state Cell {
2   method Cell getLeft() {
3     left;
4   }
5   method getRight() {
6     right;
7   }
8   val left;
9   val right;
10
11  method doPrint() {
12    printVal();
13    java.lang.System.out.print(" ");
14    val rt = this.getRight().doPrint();
15  }
16  method print() {
17    val lt = this.getLeft();
18    lt.print();
19  }
20 }
21
22 state Entry { }
```

At the bottom of the code editor, a status bar shows "Position: Ln 1, Ch 1" and "Total: Ln 253, Ch 4381". Below the code editor is a blue "Run" button.

On the right side of the interface, there is a section titled "Select an example program:" with a dropdown menu showing "examples/turing.plaid" and an "Insert example" button. Below this is a "Result:" section containing two lines of output, each enclosed in a dashed box:

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

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

Other Features: Plaid Object Model

- Pure object-oriented model
 - Everything is an object
 - Objects support procedural data abstraction
 - Structural types for after-the-fact interface abstraction
 - Nominal types to express variants, design intent
- Support for functional programming
 - First-class functions; type abstraction
- Traits for clean composition
 - Avoid problems like name conflicts, unintentional open recursion
 - State change in one trait “dimension” should not affect other traits
- Information hiding
 - Avoid violations of abstraction, e.g. from casts, instanceof, or dynamic typing

Current Plaid Language Research

- Core type system Darpan Saini (UCLA), 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, Manuel Mohr (U. Karlsruhe) 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

- A holistic, permission-based approach to managing state
 - First-class abstractions for characterizing state change
 - Use permission flow to infer concurrent execution
 - Practical mix of static & dynamic checking
- Opens a new area of research
 - Languages based on changeable states and permissions
- Potential benefits
 - Programs can more faithfully model the target domain
 - Permissions encode design constraints for static/dynamic checking
 - Naturally safe parallel execution model

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

