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

```java
var unique InputStream stream = new FileInputStream(...);
```

• Some permissions and their intuitive semantics [Boyland][Noble][...]

```
mutable

unique

mutable

shared

mutable

shared

immutable

mutable

immutable
```

• 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
method unique Data createData();

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

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

Casts can also be used to recover unique
The runtime checks the cast using reference counts
```java
state File {
    val String filename;
}
```
Explicit State Change

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

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

Permission-Based Programming Languages
Explicit State Change

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

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

Permission-Based Programming Languages
Explicit State Change

```scala
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];
}
```

Permission-Based Programming Languages
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];
}
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][…]

• 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
Resources and Composition

• Resource: stateful object whose use is constrained
  – I/O: files, database connections, device access
  – Software abstractions: initialization
  – Web 2.0 pages
  – AJAX state changes

• Complex
  – JDBC
    – ResultSet: 33 states

• Easy to misuse
  – Often poor documentation
  – Inadequate checking

The Wild Idea:
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
method void open() [\texttt{unique ClosedFile} \gg \gg \texttt{OpenFile}] {
    this \gets \texttt{OpenFile} {
        fileResource = fopen(filename);
    }
}
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 $\rightarrow$ 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.

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

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

state Entry {
}
```

Select an example program:

- examples/turing.plaid

Result:

- running 1 state busy beaver: 0 1 0
- running 2 state busy beaver: 0 1 1 1 1 0
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)
method void openHelper(ClosedFile >> OpenFile aFile) {
    aFile.open();
}

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

- **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

[Chan et al. ’98]

Diagram:
- File
  - ClosedFile
  - OpenFile
    - NotEOF
    - EOF

Resource-Based Programming in Plaid
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
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) ...

Resource-Based Programming in Plaid
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

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

```java
state LinkedList = AbstractCollection[add->addLast] with {
    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.

**Trait-based composition**

**Trait element renaming**

Resource-Based Programming in Plaid
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) { ... }
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
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);

Resource-Based Programming in Plaid

PLaID
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

API designers specify API protocols

Automatically check code against 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