PLAID:
A Resource-Based Language

Jonathan Aldrich
Carnegie Mellon University
University of Lisbon Talk
May 21, 2010
Resource Composition

• Modern programming – composition of programs from parts
  – Less emphasis on algorithms / data structures
  – Challenge: is that composition correct?

• Resource composition both important and difficult
  – Resource: stateful object whose use is constrained in some way
  – Example constraints: initialization, cleanup, lifecycle, coordination among threads
  – Even more challenging in a concurrent environment

• Scientific question
  – Could designing a language around resources help us to compose software more correctly and effectively, in a concurrent setting?
Resources are Complex

Java Database Connectivity (JDBC) Library State Space

A Resource-Based Language
Resources are Complex

Java Database Connectivity (JDBC) Library State Space

Statistics

- 33 unique states
- 69 simple state transitions
- 82 state transitions that depend on the initial state
- 11 methods whose result tests the state
- 18 methods that require a particular state
- 7 methods that return a result that depends on the ResultSet remaining in a state
- 0 methods where state does not matter
State Use in Practice

- Our empirical study found a substantial portion (~15-20%) of Java classes used or defined a protocol

- Empirically discovered “protocol design patterns”
  - **Initialization** before use – e.g. init(), open(), connect()
  - **Cleanup** – e.g. close()
  - **Non-redundancy** – can only call a method once, e.g. setCause()
  - **Boundary** check – e.g. hasNext()
  - **Marker** – marks a subset of objects with an interface, e.g. immutable collections
  - **Preparation** – e.g. call mark() before reset() on a stream
  - **Matching** – two operations called in a balanced way, e.g. lock/unlock
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)
Roadmap

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

Typestate-oriented Programming is embodied in the language

Plaid
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**

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

- **Typestate change primitive**
- **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 [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();
}

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

This method transitions the argument from ClosedFile to OpenFile
Must leave in the ClosedFile state
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

[Chan et al. ’98]

• **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
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) ...
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();
}

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
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

```java
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

TestMember tm = myList;
// compile-time error: TestMember
// does not support case analysis
```

```java
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 AEMINIUM 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

```scala
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);
```
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/