Team 15 – Peter Mikitsh, Ethan Peters, and Steven Brunwaser

History and Motivation

The Plaid Programming Language is a permission-based programming paradigm and is under the umbrella of object-oriented programming. Plaid supports 'state-based' objects that better model the real world as engineers see it. In state-based programming, objects have **typestates** that define a unique interface (set of functions) and unique sets of instance variables. The programmer is not responsible for the state transition, which can be an error-prone process to manage manually. One of Plaid's unique features is that it has built-in typestates, which provide the advantage of compile-time checking of protocol compliance. This removes the need for (in the past) throwing exceptions for not calling functions in a correct sequential order, e.g., calling the reset() function on a Buffer object before calling mark() would throw an IOException. In the state-based model, calling reset() in such a manner would not be possible because the state that represents an unmarked buffer would not include reset() in its interface. Consequently, the state-based model saves programmers from headaches via compile-time checking and it removes the need for the programmer to worry about calling functions in a sequential order.

Plaid was also developed as a response to changes in large-scale computing seen today. The ability to leverage concurrency is critical to systems utilizing technology with multi-core processing. The concurrency is achieved through **access permissions**. Various types of access permissions define the way multiple references to a single object co-exist with respect to the modifications a reference can make on an object. A special case of the access permission is "unique", where only one reference exists and it offers read/write permissions. Other permission types define all references as read-only, other references as read/write or read-only, and so on. Permissions can be split and joined through an object's lifetime.

Design Considerations

Typestate oriented - Plaid is typestate oriented. Typestate orientation is an extension to the object oriented programming model that adds an object's state as a consideration. This allows more natural modeling of the "real world" in software, as real world objects are often defined in terms of their current state in addition to other parameters. In Plaid, states are defined in a similar manner to classes in object oriented languages, each state having a unique set of characteristics (methods and variables). States can be made to extend other states or classes, so that a group of states can be associated with one object. For instance, if states A, B and C inherit from state D, then an instance of D can occupy any one of the three states A, B and C at a time, and can transition between them. While in state A, the object will have all of the methods and

data available to it that are defined in state A, and will not have access to definitions from states B and C. This allows for clean and intuitive representations and implementations of state machines, a powerful feature.

Concurrent by default - In Plaid, the programmer uses permissions to specify data dependencies within code, and the run time will use this information to determine the order of execution of the program's operations. In particular, the primary hurtle to concurrent code, the dreaded multiple references to the same data/object, is overcome through the use of permissions (associated with a given reference). A reference can have Unique, Immutable, Shared, Full or Pure permissions to an object (or of course, None, but that's kind of boring). Unique permission guarantees that a given reference is currently the only reference to its object; this is a simple way to eliminate race conditions and other fun hazards associated with concurrent programming. Immutable permission grants read-only access to the given object (duh) and also guarantees that all other references to the given object have immutable permission. Shared permission grants modifying access to an object for a reference, but makes no guarantees about the permissions of other references to the same object; this is useful for data that might not suffer from race conditions or read-after-write hazards and the like. Full permission grants modifying access to an object, and guarantees that no other references currently have write permission for the same object. Pure permission grants read-only access, but makes no guarantees about other references' permissions.

Heavily Java-based - Plaid runs on a JVM and allows for calls to standard Java libraries, as well as outright stealing most of its syntax from Java. This allows for easy integration of Java developers into Plaid projects/environments, which is fantastic.

Syntax, Semantics and Pragmatics

Plaid borrows heavily from Java in its syntax. Statements are semi-colon terminated, blocks are enclosed in curly braces, and states are declared in a manner similar to classes in Java:

```
state MyCons case of MyListCell {
  var value;
  var next;
  method append(elem) {
     this.next.append(elem);
  }
}
```

Note the lack of type specification for the variables and the method declared

above. Plaid is duck typed; that is, the type of an object is determined implicitly from its use and context. Additionally the form “<method_declaration> ::= method <method_name> (<argument_list>) [<state_name> >> <state_name>];” is used to indicate that a given method will cause a state transition for the object the method is called on, as follows:

```
state MarkedReader {

        var Integer markPosition ;

        method void reset () [ MarkedReader >> MarkedReader ] ;

        method void mark() [ MarkedReader >> MarkedReader ] ;

}

state UnmarkedReader {

        method void mark() [ UnmarkedReader >> MarkedReader ];

}
```

(Interesting) Keywords - atomic, dynamic, group, stateval, unpack, callonce, with

atomic - Provides safe access to a specified code block by a specified set of objects, in order to eliminate race conditions.

dynamic - Specifies a variable which is not statically guaranteed to be type safe.

group - Creates a new data group. A data group is a container for all shared references to a particular object.

stateval - used to declare variables associated with a particular state.

callonce - specifies that a method may only be called once

unpack - allows access to inner/nested groups declared in an object

with - when declaring an instance of a state, the keyword “with” is used to initialize data within the state:

```
state MyState {

        var myData;

}

…

var myStateInstance = new MyState with { var myData = 5; };

…
```

Strengths

* Plaid offers a reasonable approach to state-based programming. Typestates are easy to understand and would appear as an attractive option to experienced programmers. Typestates are used to extend existing code, and maintain and old codebase and provide an easy way to add new features via additions of new typestates.

* Plaid offers solutions and handling concurrency. Access permissions control how references have access to object's data through read/write permissions. This allows the programmer to employ concurrency to write more efficient code. For Software Engineers whose work focuses in concurrency in systems, this programming language may be a game-changer in the future.

Weaknesses

* The Access Permissions are not easy to summarize. They are complex to work with when it comes to understanding the proper ways one can split and merge permissions from references. The programmer must have a solid understanding of these rules before diving into coding permissions.

* Software Engineering Design Patterns exist for state machines in Object-Oriented Programming languages. These pre-existing patterns offer flexibility in an object-oriented environment. Creating a new programming language for an issue that has been more or less resolved in the Object-Oriented Programming community may restrict the language in its growth.

Sample Code

Good Examples

```
state my_base_class {

  var total;


}


state idle case of my_base_class {
```

```
    method deposit_coin(amount) [idle >> coin_deposit]
    {
       this <- coin_deposit{ var amount_deposited = amount;
}
    }
    method request_soda(price) [idle >> soda_eval]
    {
       this <- soda_eval{var price_of_request = price;}
    }
    method check_amount()
    {
       java.lang.System.out.println(total);
    }
  }

  state coin_deposit case of my_base_class {
    var amount_deposited;
    method add_amount() [coin_deposit >> idle]
    {
       total += amount_deposited;
       this <- idle{}
    }
  }

  state soda_eval case of my_base_class {
```

```
  var price_of_request;
  method evaluate()
  {
     if { price_of_request < total; }
     { this <- insufficient_fund {} }
     else {this <- vending{} };
  }
}

state insufficient_fund case of my_base_class {
  method result()
  {
     java.lang.System.out.println("Not enough money");
     this <- idle{};
  }
}

state vending case of my_base_class {
  method result()
  {
   java.lang.System.out.println("Enjoy your soda");
   this <- idle{};
  }
}
```

```
method main()
{
  var focus = new idle with {var total = 0;};
  focus.deposit_coin(50);
  focus.add_amount();
  focus.request_soda(75);
  focus.result();
  focus.deposit_coint(50);
  focus.request_soda(75);
  focus.result();
}
```

Bad Examples

The following code over uses conditionals instead of using the typestate system built into the language.

```
state my_base_class {
  var flag;
  method do_some_things()
  {
     if { flag == 0; }
     {
     java.lang.System.out.println("I'm in state 1");
     };
     if { flag == 1; }
     {
     java.lang.System.out.println("I'm in state 2");
```

```
        };
        if { flag == 2; }
        {
        java.lang.System.out.println("I'm in state 3");
        };
    }
}

method main()
{
    focus = new my_base_class with { flag = 0; };
    focus.do_some_things();
}
```

Status and Future

Plaid is currently under development at Carnegie Melon University. Its developer, Johnathon Aldrich, is advising PhD students researching the language. There is currently no standalone compiler available to the public, the only option in that department is one built into a web page on the CMU site.