The Plaid Language: Dynamic Core Specification

version 0.1.2

Jonathan Aldrich*

June 2010

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Plaid is the first typestate-oriented programming language. Typestate-oriented programming extends the object-oriented programming paradigm so that each object has a state that can change as a program executes. Each state has its own interface, behavior, and representation. Plaid combines a pure object model with good support for functional programming.

This document defines the syntax and semantics of the dynamically-typed core of the Plaid language.

This research is supported by DARPA grant HR00110710019 and NSF grant CCF-0811592.

Keywords: programming language, specification, typestate, Plaid

1 Conventions

This document uses the same grammar definition conventions as the Java Language Specification, Third Edition (JLS). Those conventions are described in chapter 2 of the JLS and are not repeated here.

2 Lexical Structure

The lexical structure of Plaid is based on that of Java, as defined in chapter 3 of the JLS. Specifically:

- Plaid uses the same definitions of line terminators as Java (JLS section 3.4), the same input elements and tokens (JLS section 3.5) except for a different keyword list, and the same definition of whitespace (JLS section 3.6).
- Plaid uses the same definition of comments as Java (JLS section 3.7).
- Plaid uses the same definition of identifiers as Java (JLS section 3.8).
- Plaid uses the same definition of literals as Java (JLS section 3.10) except that there is no Null Literal. Furthermore, Boolean objects named `true` and `false` exist in the standard library, but unlike in Java these are not keywords in Plaid. Version 1.0 of Plaid supports only String literals, plus a restricted form of Integer literal which are strings of digits.
- Plaid uses the same definition of Separators as Java (JLS section 3.11).

2.1 Keywords

The following character sequences, formed from ASCII letters, are reserved for use as *keywords* and cannot be used as identifiers:

Keyword: one of

```
case    default  import     fn      match
method     new      of    overrides  package
requires   state    this      val
var        with
```

2.2 Operators

We first define operator characters as follows:

OperatorChar: one of

```
= < > !  ~ ?  :  & | + - * / ^ %
```

Now an operator is a sequence of operator characters:

Operator:

OperatorChar

OperatorChar Operator

The exception to the grammar above is that the character sequences =, => and <− have other meanings in the language and may not be used as operators. Furthermore, operators containing the comment seequences /⋆ and // may not be used as operators.

3 Statements and Expressions

3.1 Exceptions

Several places below refer to an exception being thrown. The semantics of an exception being thrown is that the application halts with a run-time error. Future versions of this document will define facilities for propagating and catching exceptions.

3.2 Statements

Several locations in this document refer to an exception being thrown. The semantics of an exception being thrown is that the application halts with a run-time error. Future versions of this document will define facilities for propagating and catching exceptions.

Stmt:

Expr

VarDecl

VarDecl:

Specifier Identifier = *Expr*

Specifier:

`val`

`var`

Statements are either expressions, or variable declarations. A variable declaration must include an initial value. Variables can be declared with the **`val`** or **`var`** keyword; the former indicates an immutable let binding, whereas the latter indicates a mutable variable that can be reassigned later.

Statements evaluate to values, based on the expression in the statement or the value of the initializer for the variable. The last statement in a sequence is used for the return value of a method or the result of a block.

3.3 Expressions

Expr:

fn (*[Args]*) => *Expr*

Expr1

A first-class function.

Expr1:

[SimpleExpr .] Identifier = *Expr*

SimpleExpr <- *State*

match (*InfixExpr*) { *{*CaseClause*}* }

InfixExpr

The assignment form is for fields or for already-declared local variables, which must have been declared using **var**. If the identifier being assigned does not match a locally declared variable, and no qualifier is specified, then it is assumed to be an assignment to an already-declared field of the current object **this**.

The state change operator <- modifies the object to the left of the arrow as follows:

- All old members of the object are removed. In addition to members, objects have *tags* (as defined below) in order to support pattern matching; all old tags of the object are also removed.
- All tags from states referenced on the right of the operator are added to the object.
- All members on the right are added to the object.

The **match** expression matches an input expression to one of several cases using the *CaseClause* construct defined below. The overall match expression evaluates to whatever value the chosen case body evaluates to.

CaseClause:

case *Pattern BlockExpr*

default *BlockExpr*

Pattern:

QualifiedIdentifier [Identifier]

QualifiedIdentifier:

Identifier { . Identifier}

The value is matched against each of the cases in order. For the first case that matches, the corresponding expression list is evaluated. If no pattern matches, a `NoMatchException` is thrown.

The first kind of pattern syntax tests the value's tags against the *QualifiedIdentifier* given. The match succeeds if one of the tags of the value is equal to the tag *QualifiedIdentifier*, or if one of the tags of the value was declared in a state that is a transitive case of the *QualifiedIdentifier* specified.

The *QualifiedIdentifier* must resolve to a state declared with the **`case of`** construct; otherwise, a `IllegalCaseException` is thrown. If the match is successful, the matched value is bound to the *Identifier* given in the expression body.

For the default pattern, the match always succeeds. If there is a default pattern, it must be the last one in the match expression.

InfixExpr:

SimpleExpr

InfixExpr IdentifierOrOperator InfixExpr

IdentifierOrOperator:

Identifier | *Operator*

The operators defined in Java have the same precedence in Plaid as they do in Java. Identifiers as well as symbolic operators can be used as infix operators; both are treated as method calls on the object on the left of the operator. Non-Java operators and identifiers used as infix operators have a precedence above assignment and state change, and below all other operators.

SimpleExpr:

BlockExpr

`new` *State*

SimpleExpr2

The **`new`** statement creates an object initialized according to the *State* specification given (defined below). The object creation and field assignment is done atomically; there is never a "partially initialized" object.

BlockExpr:

{ *StmtListSemi* }

StmtListSemi:

Stmt { ; Stmt} [;]

Block expressions have a semicolon-separated list of statements, with an optional semicolon at the end. The statement list evaluates to the value given by the last statement in the list.

SimpleExpr2:

SimpleExpr1

SimpleExpr2 BlockExpr

To enable control structures with a natural, Java-like syntax, we allow a function to be invoked passing a block expression as an argument. The block expression is essentially a zero-argument lambda.

SimpleExpr1:

Literal

Identifier

`this`

(*ExprList*)

SimpleExpr1 . *Identifier*

SimpleExpr1 ArgumentExpr

ExprList:

Expr { , Expr}

ArgumentExpr:

(*ExprList*)

Parenthesized expressions can in fact be a list of expressions, supporting easy tuple construction.

As in Scala, multiple argument expressions can be given at a call site, supporting currying. An argument expression is a traditional parenthesis-surrounded expression list.

4 Declarations

Decl:

`state` *Identifier [*`case of` *QualifiedIdentifier] [StateBinding] [;]*

{Modifier} MSpec ;

{Modifier} MSpec BlockExpr

{Modifier} FieldDecl ;

`state` declarations specify the implementation of a state, as specified in the state definition.

The **`case of`** construct means that this state is given its own *tag* that can be used to test whether objects are in that state. Only states declared with **`case of`** can be given in a pattern for a case in a **`match`** statement. Different cases of the same superstate are orthogonal; no object may ever be tagged with two cases of the same superstate.

The final two cases are for method and field declarations. The method declaration has a method header and an optional method body. If the body is missing then the method is abstract and may be filled in by sub-states or when the state is instantiated.

Fields are discussed in more detail below.

StateBinding:

= *State*

{ *{Decl}* }

State:

*StatePrim {***with** *StatePrim}*

StatePrim:

SimpleExpr1 [{ {Decl} }]

{ *{Decl}* }

A state is a composition of primitive states separated by the **with** keyword. These primitive states can be literal blocks with a series of declarations, or they can be references to some previous object or state definition.

Composition is in general symmetric, as in traits. It is an error if more than one state defines a member, unless one of the primitive states is a block and the member is given the **overrides** modifier there.

FieldDecl:

ConcreteFieldDecl

AbstractFieldDecl

ConcreteFieldDecl:

[Specifier] Identifier = *Expr*

AbstractFieldDecl:

Specifier Identifier

MSpec:

method *IdentifierOrOperator* (*[Args]*)

Args:

Identifier { , Identifier}

The *FieldDecl* form should be familiar from Java-like languages. If no expression is given then the field is abstract. All fields can only be assigned from within the state. When fields are first defined a specifier (**var** or **val**) must be given; later, when the field is overridden and given a concrete value, the specifier must be omitted. **var** fields are mutable, **val** fields are not.

Field initialization expressions are run in the order given; initializers are run even if a field is overridden. Fields defined earlier are in scope during the execution of a field initializer. If the field initializer is run from inside a constructor, the object "this" being constructed is not in scope, as it will not even be created until all field initializers are run. If the field initializer is run from a state change operation, then the old "this" object is in scope, with whatever permissions are available in the context. In the case of state update an expression of the form "this.f" refers to the old value of a field, while "f" refers to the new value if "f" has been re-initialized already, otherwise the old value.

The method header *MSpec* also has a standard format. If the return type is missing, then the type is **`dynamic`**. The square brackets contain optional permissions and state changes to variables in scope . Permissions to the receiver can be specified by an *ArgSpec* with or without the **`this`** keyword. If no permission to the receiver is given, the *PermKind* modifier on the state declaration is used, or **`dynamic`** if there is no *PermKind* modifier. All other variables in scope needed by the method are specified with *Args*.

It is an error to have two methods, two fields, or a method and a field with the same name, unless one of the two has the "overrides" keyword.

Modifier:

`overrides`

`overrides` indicates that a method overrides a function of the same name during composition.

5 Compilation Units

CompilationUnit:

package *QualifiedIdentifier* **;** *Decls*

A compilation unit is made up of a (required) package clause followed by a sequence of declarations.

5.1 Imports

Decl:

import *QualifiedIdentifier [DotStar]* **;**

DotStar:

. *

An import statement imports a qualified name into the current scope so it can be referred to by the last identifier in the qualified name. If the import ends in .*, then all the members of the given *Name* are imported into the current scope. When a name from a wildcard import is used, the compiler must be able to find the actual declarations in that scope in order to properly resolve them; if the declarations are unavailable, each declaration used must be imported separately. Any top-level external identifier used in the file must be declared as an import, except for the identifier "plaid" for the Plaid standard libraries; this preserves the property that the compiler always knows what top-level identifiers are legal to access.

As in Java, importing the same simple name twice is an error unless the fully qualified name is the same. Importing a specific simple name always overrides importing all elements of a package where that name is defined, regardless of which definition goes first. In general, Plaid follows the Java Language Specification section 7.5.

5.2 Copmilation Unit Semantics

We define the semantics of a compilation unit in terms of ordinary Plaid objects. Each compilation unit is semantically an immutable object with an "instantiate()" method that can be used to instantiate a fresh module object.

Freshly generated module objects have methods defined for each of the top-level method declarations in the compilation unit; however, top level field declarations from the module are still missing. They also have abstract fields for the top levels of imported modules, which are given the types declared in the source code [1]. If no type for the import is given, the compiler looks up the imported module according to its current configuration[2] and uses the type of the module found for this import. Note that even though the type of a module found by the compiler may be used, the actual import is not bound until the module is linked.

[1] There is no syntax for this yet

[2] When targetting Java this is typically done with the PLAIDPATH, as defined below

Code within the module object may refer to imported modules. These references are references to the abstract fields provided by the imports. In the case of an import m.* declaration, m is imported and all names in the module that were found in m.* have an "m" prepended.

A module is linked to other modules by using the m <- p = p_def syntax, where p is the name of the imported module and p_def is the definition of the imported module, as specified at link time. Linking typically occurs whenever a module is first loaded into the run-time system.

A fully-linked module object contains a special method "initializeModule()" which creates the proper fields and invokes their initialization code. At this point, the methods defined in the module object can be invoked and the fields defined can be accessed.

5.3 Module Composition and Hierarchy

Two or more module objects may be composed using Plaid's **`with`** operation to form a single module that has a union of the declarations from each constituent module. Plaid's composition semantics will naturally merge imports with the same name from the two module objects, but renaming can be used to distinguish them if desired. The module objects' `initializeModule` methods will conflict, so these must be renamed and a new `initializeModule` method must be defined, which should call the previous `initializeModule` methods in order to properly initialize the constituent modules.

Modules may also be combined into a higher-level module, forming a module hierarchy. In this case, the higher-level module should be created by calling plaid.system.modules.newModule(), and then the <- operator should be used to add fields to the new module that refer to each of the constituent modules. Required fields should be defined for imports of the higher-level module, and imports of the constituent modules should be either linked to each other or should refer to the imports of the higher-level module. The `initializeModule` function should once again be defined to properly initialize each of the constituent modules.

Once a module has been linked or combined with another module, a new package representing the combination may be formed by calling the `asPackage` method. This method returns a new package object, such that calling "instantiate()" will generate fresh copies of the module and everything it was linked with. The method used to create the package is passed the fully qualified name the package is to have.

Package objects support code generation appropriate to the platform, allowing them to be written as binary code (or bytecode) in the file system and loaded later.

5.4 Package Loading

Programmers can load package objects from within Plaid using a package loader. "plaid.system.defaultPackageLoa
name>)." The default package loader looks up the module to be instantiated on the PLAIDPATH,
but other package loaders can[3] be used to search for modules in other ways.

[3] In the future, once we define the appropriate interface

5.5 Applications

An application is any globally-visible method that takes no arguments and is declared at the top level of a package.
The user can start an application using the following command line syntax:

```
plaid packagename.functionname
```

where `packagename` is a potentially qualified name. The Plaid runtime will look up the package referred to by the packagename and load it. The package will be instantiated and its imports will be linked with modules that are looked up and instantiated in the same way. Then `initializeModule` will be invoked, followed by the top-level function the user specified. The Plaid runtime can also be invoked with simply the name of a package, in which case the package's `main` function will be invoked.

5.6 File System Conventions

A compilation unit is stored in a file with extension .plaid. The file must be stored in a directory that can be found using the PLAIDPATH mechanism described below.

Each compilation unit may have one top-level declaration that has the same name as the file name (without the .plaid extension). This declaration is the only one that is visible from other files. All other declarations in the file are local.

Other public declarations may be placed in a special file named package.plaid. There may be one package.plaid file per package. All declarations in this file are public.

The Plaid compiler and runtime look up packages using the PLAIDPATH mechanism. The PLAIDPATH is a path specified in the same way as a CLASSPATH in Java. In particular, it is a sequence of files or directory names separated with a system-specific file separator character, which is ':' in unix and ';' in Windows. When looking up a qualified name, the PLAIDPATH is searched in order first of prefixes of the name, and next in the order of files in the PLAIDPATH, for matching .plaid files or binary or bytecode files. If a Plaid file is found with no corresponding binary or bytecode file, the Plaid file is dynamically compiled.

6 Java Language Binding

This section defines the Java language binding to Plaid, specifing how Plaid and Java code can interoperate.

Accessing Java from Plaid. Any java package, class, or class member can be referred to via a qualified name. Imported name(s) can include a package, class, or class member from Java. Instances of a Java class C may be created by invoking C.new(...) and passing appropriate arguments for one of the constructors of class C. A static method m of C may be invoked with the syntax C.m(...). An instance methods of a Java object o may be invoked with the syntax o.m(...). Arguments passed to calls of Java constructors and methods may be Java objects. Plaid integers,

strings, and booleans are converted to appropriate Java primitive, String, and numeric object types (e.g. java.lang.Integer) depending on the declared type of the method's formal parameters. If a Java method takes an Object or plaid.runtime.PlaidObject as an argument, then a Plaid object can be passed to it, allowing Java code to access Plaid objects.

Implementing Java Interfaces. When a Plaid object is first created, or a state is defined, the **`with`** operator can be used to compose a Java interface. In that case, any **`new`** expression that creates an object with that state will generate a Plaid object that extends the appropriate Java interface. The Plaid object may then be passed to a Java method that takes the interface type as an argument. Methods of the interface that are invoked by Java are converted into calls to Plaid methods of the same name and arguments, as described immediately below.

Accessing Plaid from Java. Java code may invoke methods of Plaid objects when those objects implement Java interfaces, as decribed above, or reflectively through the plaid.runtime.PlaidObject interface. When calling a Plaid method through this interface, Java objects of type Integer, String, Booleans, and other numeric objects are converted into the corresponding Plaid types. PlaidObjects and Java objects are passed through unchanged, and their methods may be invoked from Plaid in the usual way described above. The detailed interface of plaid.runtime.PlaidObject is specified in the javadoc for that interface.

The Plaid object representing a given Plaid package is accessible in Java via the static method packagename.getPackage(). Note that this only works for a top-level Plaid package; the package must have its imports resolved before sub-modules can be extracted from the package.

As a convenience, a Java function corresponding to each top-level function is created, which starts the Plaid runtime and executes the function as if the `plaid` command-line utility had been used. Thus Plaid programs can be invoked exactly like Java programs using the java command line tool, as long as the Plaid program files and the Plaid runtime jar file is in the CLASSPATH.