The Plaid Language: Typed Core Specification

version 0.4.0

Jonathan Aldrich **Karl Naden** **Sven Stork**
Joshua Sunshine

October 2011
CMU-CS-11-XXX

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This document defines the core of the Plaid language, including its initial type system

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

Keywords: programming language, typestate, Plaid, gradual typing, permissions

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 matches 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 literals are based on Java literals (JLS section 3.10), but there are several substantial differences. Plaid strings literals are the same as Java string literals. Plaid integer literals are of arbitrary length. Plaid rational literals are like Java double literals but are of arbitrary length. Furthermore, Boolean objects named `true` and `false` exist in the standard library, but unlike in Java these are not keywords in Plaid. All other Java literals are not currently supported but we plan to support in future versions of Plaid. One notable exception is the null literal which we never plan to support.
- 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

atomic	**case**	**default**	**dyn**	**dynamic**
exclusive	**fn**	**freeze**	**full**	**group**
immutable	**import**	**local**	**match**	**method**
mutable	**new**	**none**	**of**	**override**
package	**pure**	**readonly**	**remove**	**rename**
requires	**shared**	**split**	**state**	**stateval**
this	**top**	**type**	**unique**	**unpack**
val	**var**	**void**	**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 sequences `/*` or `//` may not be used as operators.

3 Statements and Expressions

3.1 Exceptions

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.

3.2 Statements

Stmt:

Expr

VarDecl

StateValDecl

VarDecl:

Specifier [Type] Identifier = *Expr*

StateValDecl:

`stateval` *Identifier StateBinding*

Specifier:

`val`

`var`

Statements are either expressions, or variable declarations. A variable declaration must include an initial value. Object variables are declared with the **`val`** or **`var`** keyword; the former indicates a final let binding, whereas the latter indicates a assignable variable that can be updated. State variables are declared with the **`stateval`** keyword.

An optional type may be given for variable declarations. If the type is omitted for a **`val`** declaration, then it is inferred to have the structure of the initializing expression and the permission that is the default for that structure. If no type is given for a **`var`** declaration the variable is considered to have type **`dynamic`**.

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`** *[MetaArgsSpec]([Args]) [[Args]] => Expr*
>
> *Expr1*

A first-class function. The optional arguments surrounded with [] support specifying types for variables that are currently in scope, including any changes to the types of these variables made by a call to the function.

Expr1:

> *[SimpleExpr .] Identifier = Expr*
>
> *SimpleExpr <− State*
>
> *SimpleExpr <<− State*
>
> **`match`** (*InfixExpr*) { *{*CaseClause*}* }
>
> **`atomic`** *MetaArgs* BlockExpr
>
> **`split`** *MetaArgs* BlockExpr
>
> **`unpack`** BlockExpr
>
> *InfixExpr*

The assignment form is for fields or for already-declared local variables, which must have been declared using **`var`**.

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

- All tags on the right are added to the object. Old tags are kept unless they are inconsistent with the new tags, i.e. the old tag and new tag are (transitively) different cases of the same state.
- All members that were declared from tags being removed, are removed from the object.

- All members on the right are added to the object. All old members on the left that are not explicitly removed according to the bullet above, are retained.
- Futher details on the semantics of state change can be found in **?**.

The replacement operator <<− removes all tags and members from the object on the left and adds all tags and members of the state on the right.

The type of either state change operations is **`void`**.

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

The **`atomic`** expression provides a save access environment to all shared objects which belong to the data groups mentioned by the **`atomic`** block. For a full definition of the semantics refer to **??**.

The **`split`** executes all statements of its body concurrently. To allow parallel access to shared data the **`split`** block will split the declared data group permissions into shared permissions, one for each statement. For a full definition of the semantics refer to [**??**].

The **`unpack`** is used to trade the group/access permission to the specified object to gain access to the inner/nested groups declared inside the object. For a full definition of the semantics refer to [**??**].

CaseClause:

`case` *Pattern BlockExpr*

`default` *BlockExpr*

Pattern:

QualifiedIdentifier

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, an exception 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 **`state`** keyword; otherwise, an exception is thrown.

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

CastExpr

InfixExpr IdentifierOrOperator InfixExpr

IdentifierOrOperator:

Identifier | *Operator*

CastExpr:

*SimpleExpr [***as** *Type]*

The operators defined in Java have the same precedence in Plaid as they do in Java, except the ternary operator and right shift operators which are unsupported. 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.

Cast expressions assert that a variable has a given type, and also assert the relevant permission for that variable. These casts are trusted by the typechecker, but unchecked. A program that executes an invalid cast may fail at may fail at any point later in the program's execution.

SimpleExpr:

BlockExpr

new *State*

SimpleExpr2

The **new** statement creates an object initialized according to the *State* specification given (defined below).

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 . **new**

SimpleExpr1 [MetaArgs] ArgumentExpr

ExprList:

Expr { , Expr}

ArgumentExpr:

(*[ExprList]*)

this represents the receiver of a method call as in Java. It is bound in method bodies declared as members of states. Unlike Java, **this** is not bound in field initializers.

Expressions can appear within parenthesis as a comma separated list representing a tuple.

Java constructors can be invoked by calling **new** on the Java class name.

Function and method invocation are handled uniformly by supplying the arguments as a tuple. Applications can be chained, supporting currying. Polymorphic arguments are specified at each call site as well.

4 Polymorphism

MetaArgsSpec:

< *MetaArgSpec [, MetaArgSpec]* >

MetaArgSpec:

group *[GroupPermission] Identifier*

GroupPermission:

exclusive

shared

protected

MetaArgs:

< *SimpleExpr1 [, SimpleExpr1]* >

Plaid supports polymorphism for data groups[1]. Plaid uses angle bracket to enclose polymorphic parameters and arguments (similar to Java's generics). A *MetaArgSpec* describes a single formal,polymorphic parameter. At the moment Paid only supports only group parameters. A group parameter consists of the **group** keyword to identify this parameter as group parameter, and optional *GroupPermission* (only optional for state declarations) and the name of the parameter. For more information about data groups and group parameters refer to [**??**].

5 Declarations

DeclOrStateOp:

Decl

StateOp

Decl:

{ModifierOrDefaultPermission} **state** *Identifier [*MetaArgs*] [***case of** *QualifiedIdentifier [*MetaArgs*]] [StateBinding] [;]*

{ModifierOrDefaultPermission} **stateval** *Identifier [*MetaArgs*] [StateBinding] [;]*

{Modifier} MSpec ;

{Modifier} MSpec BlockExpr

{Modifier} FieldDecl ;

{Modifier} GroupDecl ;

StateOp:

remove *Identifier* ;

rename *Identifier* **as** *Identifier* ;

state and **stateval** declarations specify the implementation of a state, as specified in the state definition. The **state** keyword 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 **state** can be given in a pattern for a case in a **match** statement.

The **case of** keyword assigns a superstate. States have all of the members of a superstate. Different cases of the same superstate are orthogonal; no object may ever be tagged with two cases of the same superstate.

The final two declarations 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 must be filled in by sub-states or when the state is instantiated.

Fields and state operators are discussed in more detail below.

[1] Extending polymorphism to types should be straight forward.

StateBinding:

= *State*

{ *{Decl}* }

State:

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

StatePrim:

SimpleExpr1 [{ {DecOrStateOp} }]

{ *{Decl}* }

freeze *SimpleExpr1*

A state is a composition of primitive states separated by the **with** keyword. These primitive states include literal blocks with a series of declarations and references to some previous object or state definition. In addition, objects can be transformed into primitive states with the **freeze** keyword.

Composition is in general symmetric, as in traits. It is an error if two states are composed with a member in common. The conflict can be resolved manually with state operators, remove members from, and rename members in a state.

GroupDecl:

group *Identifier* = **new group** ;

FieldDecl:

ConcreteFieldDecl

AbstractFieldDecl

ConcreteFieldDecl:

*[Specifier] [*Type*] Identifier = Expr*

AbstractFieldDecl:

Specifier Identifier

*[*Specifier*] Type Identifier*

MSpec:

method *[Type] IdentifierOrOperator [MetaParams]* (*[Args]*) *[*[*ThisArgs*]*]*

ThisArgs:

ArgSpec

ArgSpec, *Args*

Args

Args:

[ArgSpec] Identifier { , [ArgSpec] Identifier}

ArgSpec:

Type [>> Type]

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 may be omitted. **`var`** fields are assignable, **`val`** fields are not.

If a type is missing and an expression is given for a **`val`** field, then the type of the field is inferred from the expression as in variable declaration statements. If the type is missing and either no expression is given or it is a **`var`** field, then the type is **`dynamic`**.

The method header *MSpec* also has a standard format. In addition to the types of the parameters, programmers declare permission and state changes to variables in scope, including the receiver, with the *ThisArgs* in brackets. The *ArgSpec* associated with the receiver must appear without an associated identifier and as the first *ArgSpec* in the list.

Each argument specification includes the required type at the time of the method call. If the parameter ends the call with a different type this is indicated with a >> and the resulting type.

Modifier:

`requires`

`override`

DefaultPermission:

`immutable`

ModifierOrDefaultPermission:

Modifier

DefaultPermission

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

`requires` is similar to **`abstract`** in Java. However, things are more interesting in Plaid, because one can pass around an object that has abstract/required members. It is not necessary to use the **`requires`** modifier in state definitions; one can simply leave off the definition of a function. **`requires`** is necessary in types, however, to distinguish the presence vs. absence of a member in that type. Unlike in Java, methods may be called on an object that has a required member, but only if the type given to the method's receiver does not expect that member to be present.

`immutable` means a state is immutable and that any fields, local variables, or parameters declared with that state but without a permission default to **`immutable`**. If the state is not immutable then the default permission when not specified is **`unique`**.

6 Types

Type:

void

*[***immutable***] LambdaStructure*

[Permission] NominalStructure

(*Type*)

Permission:

unique

SymmetricPermission

LocalPermission

SymmetricPermission:

shared <*SimpleExpr1*>

immutable

LocalPermission:

local *SymmetricPermission*

LambdaStructure:

[MetaParams] (*ArgSpecs*) *[*[*Args*]*] -> Type*

NominalStructure:

top

*QualifiedIdentifier [*MetaArgs*]*

ArgSpecs:

*ArgSpec { * ArgSpec }*

void is the most general type in Plaid. It represents the weakest and unwritable permission **none** with structure **top** which gives no information about the object's abilities. Parameters and some variables without type annotations are given the type **dynamic**. Values of type **dynamic** must be cast to another type to be used by the type system. All other types include an optional *permission* and a structure.

LambdaStructures represent function types. They optionally include the initial and resulting types of references in scope during a function call list in [ ] as in function and method declarations. Formally, a function that accepts multiple arguments actually accepts an argument tuple, which is written with a $\star$-separated list.

NominalStructures represent the types of declared states and the special **top** structure which is a superstructure of all structures.

If the permission for a structure is not given, then a default is applied. *LambdaStructures* can only be declared **immutable** and also default to **immutable**. The **top** structure defaults to **none** and a *NominalStructures* defaults to **immutable** if it represents an **immutable** state and **unique** otherwise.

The **unique** permission indicates that there are no usable aliases to the same object. There may be other references to the object with the **none** permission which does not allow the object to be used in any way.

SymmetricPermissions allow new aliases to be created with the same permission. **immutable** references cannot update the object but can assume that it never changes. **shared** references can make changes, but must assume that other references may have changed the object.

local permissions give the same abilities and guarantees as their underlying *SymmetricPermission*, but are restricted to local variables. **local** references cannot be assigned into fields. This restriction allows **local** permissions to be returned to their original location to regain a stronger permission. For example, a **unique** reference passed to a function that requires and results in a **local** permission will still be **unique** after the function call.

7 Compilation Units

CompilationUnit:

package *QualifiedIdentifier* **;** *{Import} {Decl}*

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

7.1 Imports

Import:

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.

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.

7.2 Java Interoperability

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. A Plaid state can be declared to be a case of 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.

7.3 File System Conventions

A compilation unit is stored in a file with extension .plaid. For each top-level declaration in the file, a Java class in the package declared is created with the name of the top-level declaration. The Java class implementing a declaration is found at run time using Java's normal classpath-based lookup mechanism.

7.4 Applications

An application is any globally-visible function that takes no arguments.
If the user types at the command line:

```
plaid Name
```

where `Name` is a qualified name, the `plaid` executable will search the classpath for a declaration of the named function and will try to execute it.