The Plaid Language: Typed Core Specification (DRAFT)

version 0.3.1

The Plaid Group*

March 2010
CMU-CS-10-XXX
(TO BE PUBLISHED AS A TECH REPORT UPON THE FIRST PUBLIC BETA RELEASE OF DYNAMIC PLAID)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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 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
  dyn    dynamic  immutable   unique     none
```

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 [Type] 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.

An optional type may be given for variable declarations. If the type is omitted, then it is inferred to be the same as the initializing expression.

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]*) *[*[*Args*]*]* => *Expr*
>
> *Expr1*

A first-class function. The optional arguments surrounded with [] support specifying types and permissions for variables that are currently in scope, along with any state changes made to these variables.

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 type of the state change operation 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.

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:

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. 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. If it is determined that the variable does not have that type, or that the permission being asserted is in conflict with other permissions on the stack or in the heap, the cast may fail with an exception. Implementations are not required to detect the cast failure immediately; they may throw an exception later that "blames" the failed cast.

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:

{Modifier} **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] [*Type*] Identifier = Expr*

AbstractFieldDecl:

*Specifier [*Type*] Identifier*

MSpec:

method *[Type] IdentifierOrOperator* (*[Args]*) *[*[*[ArgSpec [***this***]], [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 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.

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. 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. 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*.

Each argument specification includes a permission, but if a different permission is returned this can be indicated with a >> and the new permission.

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:

requires

overrides

branded

PermKind

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

branded indicates that the type being defined is nominal, and is a (strict) subtype of the structural type given by its signature. If one state is a case of another, both states must be **branded**.

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.

If a *PermKind* is specified than any variables in that state have the specified permission by default. In addition, the **this** parameter to methods of this state will have the specified permission by default.

5 Types

Type:

ArgSpecs [[Args]] -> Type

QualifiedIdentifier

[PermKind] State

`none`

`dynamic`

(*Type*)

PermKind:

`unique`

`full`

`shared`

`pure`

`immutable`

ArgSpecs:

*ArgSpec { * ArgSpec }*

Function types include optional arguments in brackets [] that specify permissions to objects in scope, along with their state transitions. *ArgSpecs* include just the permission and the state, while *Args* includes the variable name as well (since for the optional arguments we are naming variables currently in scope). The arrow operator (->) is right associative. Left associativity is supported by enclosing a *Type* in parentheses.

Formally, a function that accepts multiple arguments actually accepts an argument tuple, which is written with a $*$-separated list.

State types include an optional permission kind, which defaults to the permission kind of the state, or **`dyn`** otherwise.

6 Compilation Units

CompilationUnit:

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

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

6.1 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.

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.

6.2 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.

6.3 File System Conventions

Plaid uses the Java classpath mechanism to find files. When searching for a definition for a qualified name $x_1.x_2 \ldots x_n$, where x_1 is not in scope, the system will search for a directory under the classpath named x_1 and then look for a file named x_2 there. If the file is a directory, the search proceeds with x_3 and so forth.

A compilation unit is stored in a file with extension .plaid. The file must be stored in a directory in the class path that corresponds to the package. For example, all $x_1.x_2$ package must be stored in \$CLASSPATH\$/x_1/x_2/.

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.

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 private.

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.

6.4 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.