Dynamic Casts in the Plaid Programming Language

Mark Hahnenberg
Adviser: Jonathan Aldrich

Abstract

Typestate is a new paradigm in programming language design that allows programmers to explicitly specify state transitions, which include the addition and removal of the fields and methods of objects at runtime, within their programs. Plaid, a general-purpose programming language in development at Carnegie Mellon, reifies this idea of typestate in an actual implementation. In Plaid, object aliasing complicates the static verification of state transitions by making it impossible to know with certainty the state of all other objects after a transition has been performed [1]. Plaid solves this issue with permission kinds, which help programmers as well as the compiler reason about aliasing in programs. In most languages, runtime or dynamic casts must be introduced either explicitly by the programmer or implicitly by the compiler at certain points in a program in order to ensure that the language is typesafe. The addition of aliasing information to a gradual type system raises several issues in the implementation of these casts. In order to cast something to a type with a specific permission, aliasing information must be maintained at runtime to verify that the resulting permission is compatible with all other existing permissions for that object. For my thesis I defined a static and dynamic semantics for dynamic casts in the Plaid programming language, incorporated these semantics into the Plaid compiler implementation, and examined the impact of this implementation on the overall performance of compiled Plaid programs.

Contents

Contents 2

List of Figures 3

List of Tables 4

1 Introduction 5
- 1.1 Motivation . . . 5
- 1.2 Overview . . . 7

2 Background 9
- 2.1 Gradual Typing . . . 9
- 2.2 Typestate . . . 10
- 2.3 Permissions . . . 10
- 2.4 Casts . . . 12

3 Investigation 13
- 3.1 Eager Semantics . . . 13
- 3.2 Lazy Semantics . . . 13

4 Implementation 16

5 Evaluation 18
- 5.1 Benchmarks and Implementation Limitations . . . 18
- 5.2 Future Work . . . 18

Bibliography 20

List of Figures

1.1 Code listing for an example File class. We can see that the state of the object is implicitly represented by whether or not filePtr is null. We must check the state at runtime to ensure that the client of File's API is using it correctly. 6
1.2 Code listing for the File example using typestate. 7
1.3 Client code for the File example 8

2.1 Method with potential aliasing. 11
2.2 Permission splitting judgments 12
2.3 Permission compatibility judgments 12

3.1 Two aliases to an object with the same (immutable) permission block 14
3.2 **Left**: `x` is cast to a `unique`permission, creating a new block and forwarding the old block to the new one. **Right**: `y` is accessed, causing the compatibility of the old `immutable`block to be checked against the new `unique`block, which causes an error to be thrown. 15

4.1 Snippet of Plaid code to demonstrate code generation 17
4.2 Annotated snippet of resulting Java source code generated from the earlier Plaid code in Figure 4.1 17

List of Tables

1.1 State table for File 6

2.1 A table of permissions in Plaid and their meanings 11

Chapter 1

Introduction

1.1 Motivation

One of the primary jobs of programmers has always been the manipulation of state during the execution of the programs that they write. While some forms of computation can be expressed in a stateless fashion, there will always be some programs that are inherently stateful. Until recently, programmers had to implicitly represent the states of the resources that they were manipulating within their programs.

There is no way to statically check that the objects being manipulated within the program were indeed in their expected states; rather, the programmers had to manually verify that everything was as it should be. This implicit representation of state leads to a number of bugs due to the fact that the mapping of implicit state to explicit state within a program is maintained within the minds of the programmers themselves, rather than explicitly in the code.

To demonstrate why this is a problem, let us consider the following example, to which we will refer throughout this thesis. Imagine in some programming language that we have a `File` object. This `File` object has `open()`, `close()`, and `read()` methods along with a pointer to the actual resource that represents the file in the underlying operating system. When the `File` is *open*, the pointer points to whatever operating system resource, and when the `File` is *closed*, the pointer should be `null`. When the `File` is *closed*, calling `read()` will throw a runtime error. When the `File` is *open*, calling `open()` will throw a runtime error. When the `File` is *closed*, calling `close()` will throw a runtime error.

Due to this behavior, a programmer using this File API should check the state of the File by examining whether or not the internal File pointer is set

```
public class File {
  private final String name;
  private RawFile filePtr;

  public File(String name) {
    this.name = name;
    this.filePtr = null;
  }

  public void open() {
    if (this.filePtr != null) {
      throw new RuntimeException("File is already open!");
    }
    filePtr = // ...
  }

  public void close() {
    if (this.filePtr == null) {
      throw new RuntimeException("File is already closed!");
    }
    // ...
    filePtr = null;
  }

  public String read() {
    if (this.filePtr == null) {
      throw new RuntimeException("Cannot closed file!");
    }
    // ...
  }
}
```

Figure 1.1: Code listing for an example File class. We can see that the state of the object is implicitly represented by whether or not filePtr is null. We must check the state at runtime to ensure that the client of File's API is using it correctly.

State of File	`open()`	`close()`	`read()`
open	BAD	(go to closed state)	
closed	(go to open state)	BAD	BAD

Table 1.1: State table for File

```
state File {
  val String name;
}
state OpenFile case of File {
  val RawFile filePtr;

  method read() {
    // ...
  }

  method close()[OpenFile>>ClosedFile] {
    // ...
    this <- ClosedFile;
  }
}
state ClosedFile case of File {
  method open()[OpenFile>>ClosedFile] {
    // ...
    this <- OpenFile;
  }
}
```

Figure 1.2: Code listing for the File example using typestate.

to null. If the programmer forgets to do so, this could introduce a subtle bug into the program that may only be found at runtime. In most programming models, there is no way to catch this error statically, which can drastically increase the cost of such errors.

1.2 Overview

The Plaid programming language aims to solve this type of problem by introducing the notion of typestate. Typestate takes the notion of the implicit state of objects at runtime and makes it explicit in the type system of the language so that the types of transitions and limitations described above in the File example can be described very naturally in a way that the compiler can verify statically. For example, one might write the File example in the manner shown in Figure 1.2.

As we can see from the client code in Figure 1.3, the typechecker can statically detect state violations. In this code, the client closes the `File` and then attempts to call `read()` on it again. The typechecker will see that the `read()` method does not exist for `f` at this point in the program and raise an

```
var f = new ClosedFile{ val String name = "foo.txt"; };
f.open()
f.read()
f.close()
f.read()  // typechecker will raise an error here
```

Figure 1.3: Client code for the File example

error.

Chapter 2

Background

The Plaid programming language is a gradually-typed typestate-oriented programming language being developed at Carnegie Mellon. To provide the necessary context of the contributions of this thesis, we will examine each of these aspects of Plaid and how they interact with one another.

2.1 Gradual Typing

Historically, one of the defining characteristics of any programming language was its type system (or lack thereof), and the first thing that anyone learning a new programming language would worry about with respect to the type system was which of the two major camps it falls into: statically-typed languages or dynamically-typed languages.

A **statically-typed** language checks the types of the expressions a programmer has entered to make sure they are consistent with the programmers stated expectations as well as with the expectations of the compiler itself. This allows the compiler to catch certain types of programmer errors at compile time, drastically reducing the cost of these errors[insert reference], but at the cost of some productivity overhead.

A **dynamically-typed** language does not perform any static checks, instead leaving these checks to be done at runtime. Obviously this removes any potential for the compiler to catch programmer mistakes at compile time, but it also provides additional flexibility and dynamism.

There is a significant amount of material detailing the pros and cons of both statically- and dynamically-typed languages. There is also a third, somewhat newer camp that is a combination of the two: gradual typing. A **Gradually-typed** programming language allows the programmer to omit some type an-

notations, causing the compiler to interpret those objects without type annotations as having a dynamic type (i.e. no type information is known or tracked for the object). This feature allows the programmer to leverage the benefits of both statically- and dynamically-typed languages. A gradually-typed language must introduce additional casts, which is why its presence in Plaid is relevant to this thesis.

2.2 Typestate

Many real world programs consist of a number of objects transitioning among a set of states throughout the life the program. A method call for an object may make sense when that object is in one state while it would be considered an error to call the same method when that object is in a different state. For example, imagine a `File` object. A `File` can either be *open* or *closed*. The `open()` method, when called on a `File` that is *closed*, changes the state of that File to *open*. However, calling the `open()` method again on that now *open* `File` would not make sense with respect to the semantics of a typical file I/O API. **Typestate** seeks to make these sorts of interactions explicit in the type system of the programming language so that the compiler can statically check them.

2.3 Permissions

In Plaid, the types of objects change at runtime. Aliasing makes it impossible to statically guarantee that a particular object is in the state we think it is [1]. Take, for example, the function `foo` in Figure 2.1 that takes two arguments, `x` and `y` of type `OpenFile`. In `foo` we call `close()` on `x`, changing its type to `ClosedFile`. What is the type of `y` at this point? If `x` and `y` were aliases of the same `File` object, `y` now has the type `ClosedFile`. If they were not, then `y` is still has the type `OpenFile`. For the compiler to statically check this kind of situation it needs at least some guarantees as to the type of `y` or all bets are off. Plaid solves this problem through the use of permission kinds.

Access permissions are additional information associated with each reference to an object that give additional static information to the compiler, allowing it to resolve situations like the one described above. They have three pieces of information associated with them: how many other aliases to this object could potentially exist (many or none), what sorts of operations can be performed on this particular alias (i.e. whether or not we can change the

```
method unit foo(OpenFile x, OpenFile y) {
  x.close();
  y.close();  // is this valid?
}
```

Figure 2.1: Method with potential aliasing.

Permission	Other aliases?	Can state change?	Others can state change?
`unique`	N	Y	N
`full`	Y	Y	N
`immutable`	Y	N	N
`shared`	Y	Y	Y
`pure`	Y	N	Y
`none`	Y	N	Y

Table 2.1: A table of permissions in Plaid and their meanings

state of this object), and what sorts of operations can be performed on any other aliases. Table 2.1 shows a listing of permissions and their meanings.

Splitting/Joining

Whenever a method is called, new aliases are created for both the receiver of the method call (i.e. the o in o.foo() in the form of the `this` reference) as well as the arguments that are passed into the method. For each of these objects we have to give up some of our permission to allow method to be able to use it. The method specifies what permission it needs in order to fulfill its duties. We then split the permission into the part that the method takes and we leave the remainder behind. When the method call has completed and returns, we join the permissions back together. We define the split judgment in Figure 2.2 to explicitly specify which permissions can be split into which other permissions.

Compatibility

In order to determine when a permission cast violates any of the invariants specified above we define the compatibility judgment. Two permissions for aliases to the same object are **compatible** when they can coexist without contradicting the invariants specified in Table 2.1. These judgments are found in Figure 2.3

$$\frac{}{\texttt{unique} \Rrightarrow \texttt{unique}/\texttt{none}} \quad \frac{k \in \{\texttt{unique}, \texttt{full}\}}{k \Rrightarrow \texttt{full}/\texttt{pure}} \quad \frac{P \in \{\texttt{pure}, \texttt{none}\}}{k \Rrightarrow P/k}$$

$$\frac{k \in \{\texttt{unique}, \texttt{full}, \texttt{shared}\}}{k \Rrightarrow \texttt{shared}/\texttt{shared}} \quad \frac{k \in \{\texttt{unique}, \texttt{full}, \texttt{immutable}\}}{k \Rrightarrow \texttt{immutable}/\texttt{immutable}}$$

Figure 2.2: Permission splitting judgments

$$\frac{P_2 \leftrightarrow P_1}{P_1 \leftrightarrow P_2} \qquad \frac{}{\texttt{unique} \leftrightarrow \texttt{none}} \qquad \frac{P \in \{\texttt{pure}, \texttt{none}\}}{\texttt{full} \leftrightarrow P}$$

$$\frac{P \in \{\texttt{immutable}, \texttt{pure}, \texttt{none}\}}{\texttt{immutable} \leftrightarrow P} \qquad \frac{P \in \{\texttt{shared}, \texttt{pure}, \texttt{none}\}}{\texttt{shared} \leftrightarrow P}$$

$$\frac{P \in \{\texttt{full}, \texttt{immutable}, \texttt{pure}, \texttt{none}\}}{\texttt{pure} \leftrightarrow P}$$

$$\frac{P \in \{\texttt{unique}, \texttt{full}, \texttt{immutable}, \texttt{shared}, \texttt{pure}, \texttt{none}\}}{\texttt{none} \leftrightarrow P}$$

Figure 2.3: Permission compatibility judgments

2.4 Casts

Casts can be inserted in one of two ways: either the programmer inserts a cast manually or the compiler generates a cast when invoking dynamically-typed code. In the implementation, casts are checked eagerly for compatibility with all other existing permissions for that particular object. The decision of eager versus lazy semantics for casts is discussed in depth in the next chapter.

Chapter 3

Investigation

One way to broadly categorize the permission casts for Plaid is along the lines of eager semantics versus lazy semantics.

3.1 Eager Semantics

The **eager semantics** for casts in Plaid means that the compatibility of permission casts are checked immediately as soon as they are executed. This particular method has the advantage that it is obvious to the programmer where and when a program fails due to a botched cast, i.e. it is fail-fast. Additionally, no further compatibility checks must be done except for at cast sites, which should be relatively sparse.

The implementation for the eager semantics is relatively straightforward. A table of permission references counts must be maintained for each object. Whenever a cast occurs, the new permission is checked against all other permissions. If there are any incompatibilities, an error is thrown. Obviously this table must be thread-safe to prevent multiple threads from corrupting the reference counts. This makes heavy casting in a multithreaded environment potentially very costly.

3.2 Lazy Semantics

The **lazy semantics** for casts in Plaid means that the compatibility of permission casts are not necessarily checked immediately when the cast is executed. This leads to broader interpretations of when checks should be done. One natural point is whenever an object is accessed we check for any earlier bad casts. This scheme could be accomplished through pointer forwarding. Instead of a

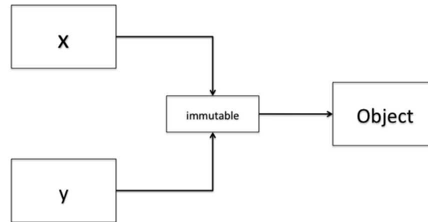

Figure 3.1: Two aliases to an object with the same (immutable) permission block

reference pointing directly to an object, it points at permission pseudo-object that keeps track of the permission associated with all the references that point to it. The block itself then points to the object. Figure 3.1 depicts this setup. Whenever a cast occurs, a new permission block is created and the old block's pointer is pointed at the new block. If, at a later time, a different alias accesses the old permission block, it sees that it has been forwarded to another block and checks itself for compatibility with the newer block. If the two are incompatible, a runtime error is thrown. Figure 3.2 demonstrates this process.

Unfortunately, the lazy semantics has several downsides. The error reporting is very non-local in that a bad cast might not be detected for quite some time, thereby making the error rather far removed from its actual source. There are remedies for this, namely blame tracking, but that adds an additional level of complexity. Another downside is that this adds an extra dereference for each object access. The upside is that casts are very fast because no compatibility checks need be made. This method also suffers from the apparent concurrency issues mentioned for the eager semantics; however, the effects are more severe in this case since locking would have to be done for each object access.

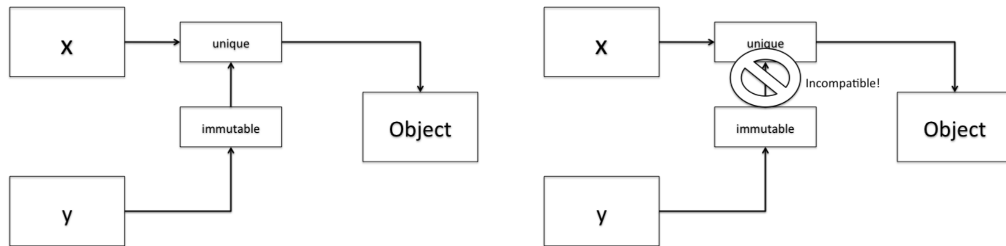

Figure 3.2: **Left**: `x` is cast to a `unique`permission, creating a new block and forwarding the old block to the new one. **Right**: `y` is accessed, causing the compatibility of the old `immutable`block to be checked against the new `unique`block, which causes an error to be thrown.

Chapter 4

Implementation

The Plaid compiler is self-hosting and targets the Java Virtual Machine through the generation of Java source code. The parser is written using JavaCC and the runtime is written in Java, while everything else, from the typechecker to the code generator, is written in Plaid itself. As of now, the Plaid implementation only includes three permissions: `unique`, `immutable`, and `none`.

I had to modify both the runtime and the code generator to incorporate casts into the compiler implementation. Since I was implementing the eager cast semantics, I added a thread-safe reference counting table to each object created at runtime. I also modified the code generator to insert the correct calls into the runtime at the various splits, joins, and casts to both maintain correct reference counts and to do compatibility checks. Figure 4 contains an example of the corresponding Java code that would be generated for the given snippet of Plaid code given in Figure 4.1

```
state Foo {
  method unit test() {
    java.lang.System.out.println("Hello, World!");
  }
}

method unit main() {
  var f = (immutable) new Foo;
  // Java source snippet starts here
  f.test();
  // Java source snippet ends here
}
```

Figure 4.1: Snippet of Plaid code to demonstrate code generation

```
// lookup the test() method in f (defined earlier)
final PlaidObject vAr1915$plaid;
vAr1915$plaid = PlaidRuntime.getRuntime().getClassLoader().lookup("test", f);

// get a unit value to apply the method to
final PlaidObject vAr1917$plaid;
vAr1917$plaid = plaid.runtime.PlaidRuntime.getRuntime().getClassLoader().unit();

// split the types for the method call
PlaidPermType start = PlaidPermType(immutable(), nominalType("f"));
PlaidPermType end = PlaidPermType(immutable(), nominalType("f"));
PlaidPermType leftover = PlaidPermType(immutable(), nominalType("f"));
vAr1917$plaid.split(start, end, leftover);

// call test()
PlaidObject vAr1918$plaid;
vAr1918$plaid = plaid.runtime.Util.call(vAr1916$plaid, vAr1917$plaid);

// join the types back together
vAr1917$plaid.join(leftover, end, start);
```

Figure 4.2: Annotated snippet of resulting Java source code generated from the earlier Plaid code in Figure 4.1

Chapter 5

Evaluation

5.1 Benchmarks and Implementation Limitations

Several benchmarks were ported from Google's V8 JavaScript benchmark suite. JavaScript shares many features with Plaid, including the ability of objects to change types at runtime through the addition and removal of fields and methods. While the type system of JavaScript provides far fewer guarantees than that of Plaid, the dynamism of the language makes its benchmarks suitable for testing Plaid.

Unfortunately, as of the writing of this thesis, the Plaid implementation is not quite mature enough to be able to distinguish any performance difference between Plaid with casts and Plaid without casts. In fact, the benchmarks uncovered several deeper issues in the Plaid implementation–its heavy, perhaps wasteful use of memory and its rather low recursion limit. Each method call in Plaid translates into three to four Java function calls, making it quite easy to overflow the stack. These problems are probably simply a limitation of the JVM, as it was not constructed with such a dynamic language in mind.

5.2 Future Work

There is some additional information that can be tracked by the aliasing permissions, namely state guarantees. They guarantee a ceiling state (i.e. the most general supertype possible) for a particular alias. This extra information allows the compiler to more accurately track the state of a particular object, which in turn allows the type system to catch more errors than it would without state guarantees.

While they have been included in past publications on gradual typestate [2], state guarantees are not currently part of the official Plaid language specification. When the specification matures a little to the point where these are included, dynamic casts with state guarantees could be examined further. Additionally, the remaining permissions (`full`, `shared`, and `pure`) need to be added to the implementation as soon as they are incorporated into the official language specification.

Bibliography

[1] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *OOPSLA*, 2007.

[2] Roger Wolff, Ronald Garcia, Eric Tanter, and Jonathan Aldrich. Gradual featherweight typestate. *CMU-ISR-10-116R*, 2010.