Dynamic Casts in the Plaid Programming Language

Mark Hahnenberg, advised by Jonathan Aldrich

April 11, 2011

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.

1 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.

1.1 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 annotations, 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.

1.2 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.

```
public class File {
  private String fileName;
  private OSFilePtr rawFile;

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

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

  public void read() {
    if (rawFile == null) {
      throw new RuntimeException("Cannot read from file: file is closed!");
    }
    // ...
  }
}
```

Figure 1.2.a - A File I/O Implementation without Typestate

```
state File { val String fileName; }
state OpenFile case of File {
  val OSFilePtr rawFile;

  method unit close()[OpenFile>>ClosedFile] { /* ... */ }
```

```
  method String read() { /* ... */ }
}
state ClosedFile case of File {
  method unit open()[ClosedFile>>OpenFile] {
    // ...
  }
}
```

Figure 1.2.b - A File I/O Implementation without Typestate

1.3 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]. Imagine a function `foo` 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.

```
method unit foo(OpenFile x, OpenFile y) {
  x.close();
  // What is the type of y now?
  y.read(); // Is this read() valid?
  // ...
}
```

Figure 1.2.b - A File I/O Implementation without Typestate

Aliasing 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 state of this object), and what sorts of operations can be performed on any other aliases.

2 Problem

In order to be able to statically check programs that use typestate and typestate transitions, we must track aliasing information using permissions. These permissions are incorporated into the type of each alias in the program. In order for our language to be typesafe, the compiler will need to insert some dynamic casts at the boundaries between statically typed and dynamically typed code as well as at other locations in the code, for example, where permissions are split and joined (i.e. at method call boundaries). There are two broad categories of the type of semantics that could be used for these permission casts: lazy or eager.

3 Lazy vs. Eager Semantics

Lazy permission casts delay the checking of aliasing information for consistency until after the cast itself (the point at which it eventually does occur is dependent on the exact semantics). Eager permission casts, on the other hand, check each permission cast for consistency the moment it occurs.

During the first part of this thesis, the pros and cons of the lazy and eager semantics were investigated and weighed against one another to decide which should be implemented. The eager semantics are relatively straightforward to implement, while the lazy semantics are a bit more complex. Additionally, the eager semantics fails right when a bad cast occurs, making it easier to see what the program was doing when that bad cast occurred. The lazy semantics might not catch this bad cast until much later. Blame tracking can partially alleviate this problem, but it increases the complexity and removes some of the observational locality of what the program was doing when the bad cast occurred. Unfortunately, the eager semantics can fail on a cast that actually would not be a problemif there exist any aliases that conflict with the one being cast but they would not be used again, then the eager semantics will still fail where the lazy would succeed.

The lazy semantics were thought to maybe provide some benefit in a concurrent setting, although the implementation designs for the lazy semantics turned out to be just as bad and in some cases even worse with respect to the number of compatibility checks and the amount of locking required when performing casts.

4 Implementation

The eager semantics were implemented by maintaining with each object a reference count of each kind of alias currently present in the program. Whenever a cast is performed these reference counts are checked for consistency with the permission to which the alias is being cast.

The implementation of the Plaid language type system also includes special splits and joins for method calls. Whenever we pass objects to method calls, we are creating new aliases for those objects within those methods, so we need to split off permissions from the ones we currently are holding to give to the method. When that method returns, we need to join the permissions of those aliases we created with the ones we left behind when we called the method. These are where most of the casts in the language come from since the compiler inserts them automatically for every method call.

5 Future Work

At the time of writing this extended abstract, benchmarks still need to be run on the eager implementation.

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.

References

[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.