An Efficient Implementation of Plaid

A novel runtime representation for abstract state.

Sarah Chasins
Swarthmore College
schasi1@cs.swarthmore.edu

May 4, 2012

Abstract

State is central to understanding objects in the real world — a moving car is very different from a parked car. Yet most object-oriented languages provide no native support for state or state change. The Plaid language introduces a new object model in which objects are not only instances of a class, but can also be in mutable states. In Plaid, a car object can have one abstract state for driving and another for not driving. Transitioning the object from the driving to the not driving state with Plaid's state change operator removes members that are available only in the driving state — for instance, speed — and adds members associated with the not driving state. Traditionally, efficient implementations of object-oriented languages have required that objects have stable members in order to facilitate fast member access. State change alters members at runtime, presenting a new challenge for efficient compilation. To address this challenge, we developed a novel representation for state at runtime that enables state change without sacrificing fast member access. We implemented our representation in a new code generator and runtime system for compiling Plaid to JavaScript. Our implementation improves program execution time by a factor of 48 over a Plaid implementation with an alternative state representation that slows member access.

1 Introduction

Maintaining implicit state information is a common practice in program design. Consider a file object. When it is open, a programmer can call a read method or a close method. When the file is closed, those operations are no longer available, but a programmer can call an open method. This file object may need some information at all times — perhaps the file name — but it has two disjoint sets of members, one for an open state, and one for a closed state. In typical object-oriented languages, this state information is never directly expressed.

The Plaid language introduces a model in which object state is made explicit. Rather than keep a variable that stores a file object's current state, a user can declare `OpenFile` and `ClosedFile` as substates of `File`. The code `file <- OpenFile` transitions the closed `file` object cleanly into the `OpenFile` state.

By introducing abstract states and explicit state change, Plaid makes these transitions salient to the programmer and facilitates code that depicts object structure more clearly. Without the need to write one's own state checks, code is neater and more compact. Where programmers might forget to write state checks, the runtime can indicate that a member is unavailable in a particular state, rather than permit continued execution and possible data corruption.

Support for abstract states is a desirable language feature from a programmer's point of view; however, implementing a language with state support is challenging. Efficient language implementation has historically relied on stable object members. But abstract state is only useful if it is accompanied by state change, and state change means altering an object's members at runtime. Transitioning `file` to `OpenFile` is not very useful if `file` does not gain a read method in the process.

This paper presents a new implementation of the Plaid language, a general-purpose language which offers language constructs for state change and for composing states. We present a code generator for com-

piling Plaid to JavaScript, and a JavaScript runtime with all the functionality necessary to support state change. Our implementation leverages JavaScript's prototype-based objects and first-class functions to create a runtime in which a Plaid object's available members are always the only members in a JavaScript object that represents that Plaid object. That is, even when a Plaid object is created by composing many states, and when it inherits members from substates, a single JavaScript object stores all the fields and methods of the Plaid object. Further, when the Plaid object undergoes state change, any new members are added to the same corresponding JavaScript object.

By maintaining a single JavaScript object with all available members, we ensure that the execution time of member accesses and method calls is independent of the number of component states that form a Plaid state, and the depth of the state hierarchy. Thus, utilizing Plaid's state abstractions to compose states should not increase the execution time of method calls, relative to hand-coding a Plaid state that contains all the methods of the component states.

With all of a Plaid object's members in a single JavaScript object, there is no self-evident procedure for enacting state change. The members associated with `File` and the members associated with `OpenFile` are not clearly distinguished. When it comes time to transition from `OpenFile` to `ClosedFile`, which members should our runtime remove from the object? To provide the information necessary to answer this question, our runtime stores a metadata object in each JavaScript representation of a Plaid object. The metadata object stores complete information about all the object's component states, their tags (their unique names), their relationships to each other, and the members associated with them. The runtime updates these metadata trees as Plaid objects transition between states.

The flat member structure and the use of a metadata object to track all state information together represent a novel way of implementing abstract state and state transitions. Also, to our knowledge no other implementation has enabled language-supported state change by adding and removing members from corresponding objects in the target language.

We used our implementation to run Plaid versions of several benchmarks from the V8 Benchmark suite[24]. A previous implementation of Plaid compiled Plaid to Java. Our new implementation produced code that was up to 48 times faster than code produced by the Java implementation.

The potential usability benefits of abstract state are significant, but before it will be widely adopted, programmers must be persuaded that code with state abstractions can be fast. This paper presents early evidence suggesting that appropriate implementation schemes can lower the overhead of supporting abstract state. This is an important step in making state support a practical option, both for language users and designers.

Our contributions are:

- A novel representation for state at runtime, which balances the demands of state change and member access.
- A code generator and runtime for compiling Plaid to JavaScript.
- A large speedup relative to a past Plaid implementation, and initial evidence that language-level support for abstract state can be implemented efficiently.

In Section 2 we introduce the Plaid language and the innovative language features it incorporates. In Section 3 we examine related research on implementing languages that allow object members to change at runtime. Section 4 outlines the central challenges to efficient compilation and offers an overview of our solution. In Section 5, we describe the implementation of our runtime. In Section 6, we describe the implementation of our code generator. In Section 7, we address some of the design issues we faced. In Section 8, we present a validation of our work with cross-language performance comparisons. In section 9, we consider possible future directions, and Section 10 concludes.

2 Background

We examine the motivation for incorporating state into high-level object-oriented languages, and how it was initially developed. We then discuss the background of the Plaid language, and briefly present the language features that pose new challenges for implementation.

2.1 The Motivation for State

State is an important part of how individuals interact with objects in the physical world. A radio is powered

on or off, and different actions are available in those different states. A set of matter can be a solid, liquid, gas, or plasma, and the current state will affect how an individual interacts with it. Object-oriented programming languages offer analogues for other features of objects — fields for characteristics, methods for actions, inheritance for specialization — but they fail to offer abstract state or state transitions. Instead, to implement a radio object, the programmer must add state checks to each method; a check to ensure the radio is on before we can ask which station is playing, or try to adjust the volume.

Many library APIs implicitly limit methods to objects in particular abstract states. For instance, the `ResultSet` class of the Java JDBC library distinguishes between objects that are open or closed, updatable or not updatable, sensitive or not sensitive, scrollable or not scrollable, and so on, the result of which is 33 distinct abstract states [6]. This is a complex state space, not easily expressed in Java, and not easily discerned by examining the library, yet proper use of a `ResultSet` object requires a full understanding of the protocols. A study of open-source Java projects revealed that protocols are pervasive even in languages that do not provide explicit support for abstract state [5].

The goals of explicitly supporting state abstractions are:

- To allow code to more clearly reflect object design.
- To make code simpler and shorter by eliminating hand-coded state checks.
- To make state change and protocols salient to the programmer, to decrease the likelihood that he or she will violate protocols.
- To prevent the execution of calls that violate protocols, if such calls are made in error.

2.2 The Development of Typestate

The concept of typestate was first introduced in a 1986 Strom and Yemini paper, in which the authors present typestate as a refinement of type [19]. They state:

> Whereas the type of a data object determines the set of operations *ever* permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.

The authors assert that type checking and static scope checking detect some errors produced by syntactically well-formed but non-meaningful statements, but that they miss others. Specifically, if all operations are type-correct, the type-checker will accept a program, even though a type-correct operation may be undefined at the time when it is called.

To illustrate this failing, we consider several brief examples. Traditional type-checking can identify the sort of nonsense statement that produces an error regardless of its positioning relative to other statements. For instance, it is never legal to assign an integer value to a variable that has been declared as a boolean, and it is never legal to use a multiplication operator on a boolean. Broadly, type-checking is sufficient when: (a) if a given statement is legal *anywhere* in the scope of the relevant variables, it is legal *everywhere* in that scope; and (b) if it is illegal *anywhere* in scope, it must also be illegal *everywhere*. However, type-checking does not identify errors such as using a variable before it has been initialized. If `a` and `b` are integers, the statements `a = 2;` and `b = a+1;` are accepted by the type-checker, because `+` and `=` are defined for the integer type. The type-checker will still accept the program even if the second statement appears before any value has been assigned to `a`. Similarly, if a pointer `p` has been declared as a pointer to some type but not yet initialized to point to an object of that type, the type-checker will allow a modification to the data to which `p` points.

Because these operations are not type-incompatible, the type-checker accepts them. As a result, the program may copy data to a "garbage" location in memory, the location represented by `p`'s contents after its declaration. These actions should not be carried out, even though the type-checker permits them. The order of statements — the statements' contexts — leads `a` and `p` to be in inappropriate *states* at the point when these operators are used. Specifically, they are not yet initialized.

While static scope rules allow compilers to identify the use of some variables outside of their scopes, they provide no such help for variables that point to heap memory. Thus, even with static scope checking in place, some nonsense statements will go unidentified.

Because this sort of error can harm even programs that did not produce the error, and because the errors can go entirely unnoticed during execution, the authors argue that typestate errors are their own class of failure, and a strongly undesirable one. They

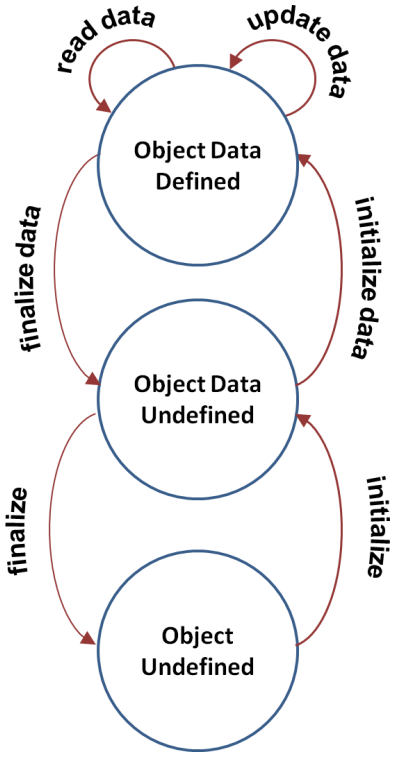


Figure 1: A visual representation of the states and transitions for a pointer to an object with one field, `data`. If a user assigns to the object's `data` field before `new` has been used to initialize the pointer, it is possible to corrupt data at an unknown location.

point out that even though their example execution sequences are considered ill-defined, prominent procedural languages did not at the time prevent them from running.

To address these issues, the authors introduce typestate. Each type has a set of typestates, and in each typestate, a number of the type's operations may be legally applied. An object of type `t` is always in one and only one of `t`'s associated typestates. For instance, an integer is always either in the initialized or the non-initialized state, but not both. An operation may change the typestate of its operands. Thus, each operation has a typestate transition for each operand, which includes the typestate precondition (which must be true in order for the operation to be applicable) and one or more postconditions (which give the operand's possible typestates after the operation). A graph in which the nodes are the typestates and these transitions are the edges produces the state machine graph for a given type.

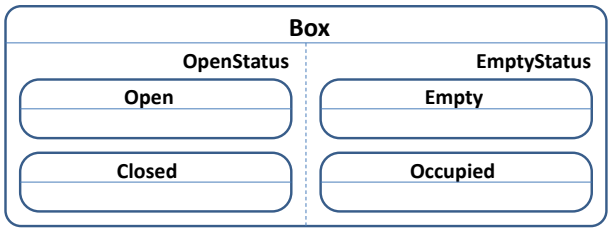


Figure 2: Statechart depicting the state space of a box object. `Box` is an and-state, composed of `OpenStatus` and `EmptyStatus`. `OpenStatus` is an or-state; it can be in either the `Open` or the `Closed` state.

For an example, see Figure 1, which displays a very simple state machine for a pointer to an object with one field. The figure indicates that the pointer must be initialized before its field can be initialized, and that the field must be initialized before it is possible to read or update the field. After the field has been finalized, it is no longer possible to read or update, and after the pointer has been finalized, it is no longer possible to initialize the field. There are thus three states, one in which both the pointer and its field are undefined, one in which the pointer is defined but its field is not, and one in which both are defined.

The authors go on to describe a scheme for tracking typestate at compile-time by making typestate a static invariant property of the program's variable names at each point in the program.

2.3 State Composition

Many objects are composed of orthogonal regions — different dimensions which may change in various ways without affecting each other. Traditional state diagrams require creating a distinct node for all acceptable combinations of these dimensions. To create a box that can be either open or closed and either empty or occupied, one would have to create four nodes: open and empty, open and occupied, closed and empty, closed and occupied. To combat the exponential explosion of states that comes from combining many dimensions, Harel introduced Harel statecharts [13], which then formed the basis for UML statecharts [17].

In a Harel statechart, one creates a single node for each orthogonal region, which can change state internally without affecting any other orthogonal regions. The box can go from open to closed without affecting whether it is empty or occupied. Although the Harel

statechart is equivalent to classic state diagrams, it reduces the number of necessary nodes and makes the state structure more easily discernible.

In the example above, a box was composed of two orthogonal regions, its open status *and* its empty status. For this reason, the box state would be called an *and-state*. An and-state may be composed of two or more independent dimensions. An object that is in an and-state is simultaneously in all the orthogonal regions that compose the and-state.

Harel and UML statecharts also modify finite state machines by introducing hierarchically nested states. For instance, open and closed would be nested states — or substates — of open status. If an object is in one of those nested states, it is also necessarily in the superstate, open status. For an object to be in both a substate (open) and superstate (open status) means that a member access on the object would first be passed to the more specialized substate. If the substate could not handle the member access, it would be passed to the superstate, facilitating easy behavior reuse, whereas the same behavior would have to be associated with many nodes in traditional state machines. A substate may itself have substates, and the nesting can be as deep as a state space requires.

A state may have many specializing substates in the hierarchical model, but an object may be in only one of those substates at any given time. That is, a box object may be in either the open *or* the closed state. For this reason, a state with specializing substates is called an *or-state*.

Figure 2 displays the state space for the box object. In UML statecharts, an or-state's specializing substates appear as state machines within the or-state's rectangle. An and-state's orthogonal regions appear within the and-state's rectangle, separated by dotted lines.

Statecharts' support for orthogonal regions provides an answer to the state and transition explosion problem. The and-state support means eliminating the need to mix behaviors from multiple dimensions, which do not affect each other. It allows a box in the closed condition to share the same open method, regardless of whether the box is full or empty, and regardless of how it may vary on many other dimensions.

2.4 The Plaid Language

Recall the file object with which the paper opened. In Listing 1, simple Plaid[20] code lays out the design of a `File` state, whose state space is depicted in

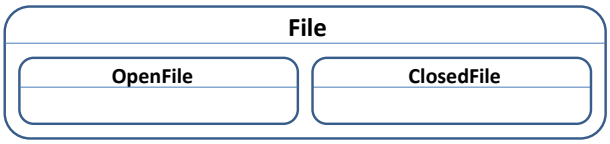


Figure 3: Statechart depicting the state space of a simple file object.

Figure 3. In Plaid, every object is created as an instance of a state. A state is like a class in a more traditional object-oriented language, except that a state can change. `OpenFile` and `ClosedFile`, declared with the `case of` keyword, are substates of `File`. Method `close` makes transitioning between states — and altering the available members — simple and intuitive. It uses the `<-` operator to enact state change and shift the object on which it is called from the `OpenFile` to the `ClosedFile` state. Method `open` performs the opposite change.

```
1  state File {
2     val filename;
3  }
4  state OpenFile case of File {
5     val filePtr;
6     method read() {}
7     method close() { this <- ClosedFile;}
8  }
9  state ClosedFile case of File {
10    method open() { this <- OpenFile;}
11 }
```

Listing 1: Plaid declaration of `File` state with two case of states, `OpenFile` and `ClosedFile`.

```
1  method readClosedFile(f) {
2     f.open();
3     val x=f.read();
4     f.close();
5     x;
6  }
```

Listing 2: Using the `open` and `close` methods defined in Listing 1.

From Listing 2, it becomes clear how explicit typestate support simplifies code. No state checks or error handling appear in the method, and none are necessary. Passed a closed file, the method

`readClosedFile` will succeed. Passed an open file, the call will fail with an error, because there is no `open` method in `OpenFile`.

While such an improvement may seem trivial in the case of a file with two `case of` states, consider a larger example, such as the car object whose state space appears in Figure 4. More complex examples, such as the ResultSet interface in the Java Database Connectivity library would gain even more from the ability to clearly distribute an object's functionality between its abstract states.

Whether in the case of a simple file or in objects composed of six states simultaneously, or nested states, the practice of maintaining implicit state information is pervasive in program design. This practice motivates the design of languages that provide easy ways to model state, and the potential advantages are substantial. Introducing abstract states and explicit state change results in code that more clearly depicts object structure, code that makes the programmer's intent more obvious to a viewer. Explicitly naming states and the transitions between them makes those transitions salient to the programmer, ideally reducing the number of bugs introduced by failing to follow forgotten protocols. State support eliminates the need for hand-coded state checks, thus reducing the time spent producing boilerplate, and resulting in neater, more compact programs. Further, while a programmer might forget to write state checks for a particular field or method, all members are necessarily associated with a state in Plaid. If a member is accessed in the wrong state, the runtime automatically throws an error, preventing the data corruption that could result from a protocol violation. Recall Strom and Yemini's examples of a pointer assignment that will simply overwrite data in a "random" memory location. Without support for abstract state, this corruption occurs silently, maybe without the programmer's knowledge, and definitely without any aid to the programmer in his or her debugging efforts. With language support for state abstraction, this state-inconsistent call will instead produce a helpful error message.

2.4.1 Joining State and Composition

Plaid distinguishes itself from prominent modern languages by its facilitation of both abstract state and trait-like state composition primitives. Plaid's state abstractions, like Strom and Yemini's typestate [19], allow an object's type to reflect its current state. Plaid joins this state change support with a model of state composition that allows the language to express any state space that could be represented with statecharts, like Harel's[13] or UML state machines[17]. With state composition primitives reminiscent of trait model [11] composition constructs, any Plaid state can be used as an independent dimension in another Plaid state.

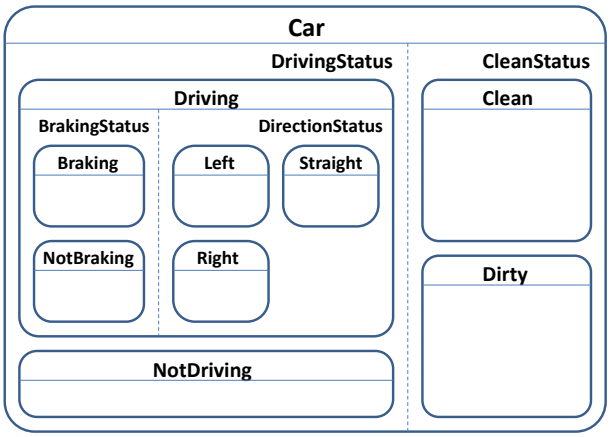


Figure 4: Statechart depicting the state space of a complex car object. In Plaid, the code `state Car = DrivingStatus with CleanStatus` would be used to generate the `Car` and-state, once `DrivingStatus` and `CleanStatus` are defined. The line `Braking case of BrakingStatus` would begin the definition of the `Braking` state, a specializing substate of `BrakingStatus`.

The initial motivation for Plaid, and an outline of possible features, were described in a work on the Typestate-Oriented design paradigm[3]. A later paper presented the motivation for a state change operator that can change one dimension at a time[2]. More recently, the creators of the language offered a solution to that problem and presented the formal semantics of the Plaid language[20].

Statecharts' and-states are created in Plaid using the `with` operator, which composes two states. A Plaid state may be declared as a specializing substate of another by using the `case of` keyword. Thus, Plaid can be used to model both and-states and or-states, and can therefore represent any state space that can be modeled with a statechart.

For example, consider the Plaid code in Listing 3 that declares the car state space pictured in Figure 4. The Plaid code in Listing 3 contains all the information necessary to construct a statechart for the `Car`

```
1  state BrakingStatus{}
2  state Braking case of BrakingStatus{
3     method stopBraking() {}
4  }
5  state NotBraking case of BrakingStatus{
6     method startBraking() {}
7  }
8
9  state DirectionStatus{}
10 state TurningLeft case of DirectionStatus{
11    method turnStraight() {}
12    method turnRight() {}
13 }
14 state Straight case of DirectionStatus{
15    method turnLeft() {}
16    method turnStraight() {}
17 }
18 state TurningRight case of DirectionStatus{
19    method turnLeft() {}
20    method turnStraight() {}
21 }
22
23 state DrivingStatus =
24       BrakingStatus with DirectionStatus
25 state Driving case of DrivingStatus{
26    var speed;
27    var acceleration;
28    method stopDriving() {}
29 }
30 state NotDriving case of DrivingStauts{
31 }
32
33 state CleanStatus{}
34 state Clean case of CleanStatus{
35    method getDirty() {}
36 }
37 state Dirty case of CleanStatus{
38    method getClean() {}
39 }
40
41 state Car = DrivingStatus with CleanStatus
42
43 method main() {
44    var car = new Car;
45    car<-Braking;
46    car<-TurningLeft;
47    car<-Clean;
48 }
```

Listing 3: The Plaid code to declare the car state depicted in Figure 4.

state. `Car` is an and-state, because it has two orthogonal regions — `DrivingStatus` and `CleanStatus` — which are defined separately, and can themselves be used as normal Plaid states. Line 41 (`state Car = DrivingStatus with CleanStatus`) uses the `with` operator to create the `Car` state from its two independent dimensions. In contrast, `DrivingStatus` and `CleanStatus` are or-states. Note that `Driving` and `NotDriving` are both declared using the `case of` keyword, indicating that they are specializing substates of `DrivingStatus`.

The next step is to establish what methods are available to each state. An object in a specializing substate can use all the methods and fields defined in its superstates. Thus, an object in state `Driving` has access to any members available to `Driving` as well as the members declared in `DrivingStatus`. All and-states have all the methods and fields of their component states. Thus an object in state `Car` will have access to all the members of both `DrivingStatus` and `CleanStatus`.

2.4.2 State Change

Consider the case where a user initializes a `Car` object in the `Driving`, `Braking`, and `TurningLeft` states. If one were to shift from the `Driving` to the `NotDriving` state, it is not only `Driving` that would disappear, but also `BrakingStatus`, `DirectionStatus`, `Braking`, and `TurningLeft`. `Driving` and `NotDriving` are substates of the same or-state, and therefore cannot coexist.

In the `main` function in Listing 3, we use the Plaid `Car` state to create a Plaid `Car` object. The code that follows uses the state change operator `<-` to make the object represent a `Car` that is `Clean`, `Braking`, `TurningLeft`, and `Driving` (implied by `Braking` and `TurningLeft`).

Plaid's formal semantics[20] offer a precise procedure for state change. See Figure 5 for a formal description of how a Plaid implementation must handle state update.

The relevant portions of Plaid's core syntax (described in [20]) are:

Obj Val $\quad ov ::= mv \mid dv \mid mv \curlywedge ov \mid dv \curlywedge ov$

Dim Val $\quad dv ::= tag\ \{\ ov\ \} \mid tag\ \{\ ov\ \} <: dv$

Mbr Val $\quad mv ::= m(\overline{x})\{e\} \mid n = v$

In this syntax and in Figure 5, *ov* represents an object value, which is a list of *mv* (member values) and *dv* (dimension values). Member values simply represent methods and fields associated with a given state. Dimension values take the form $tag\{ov\}[<: dv]$. The *tag* is the unique name for

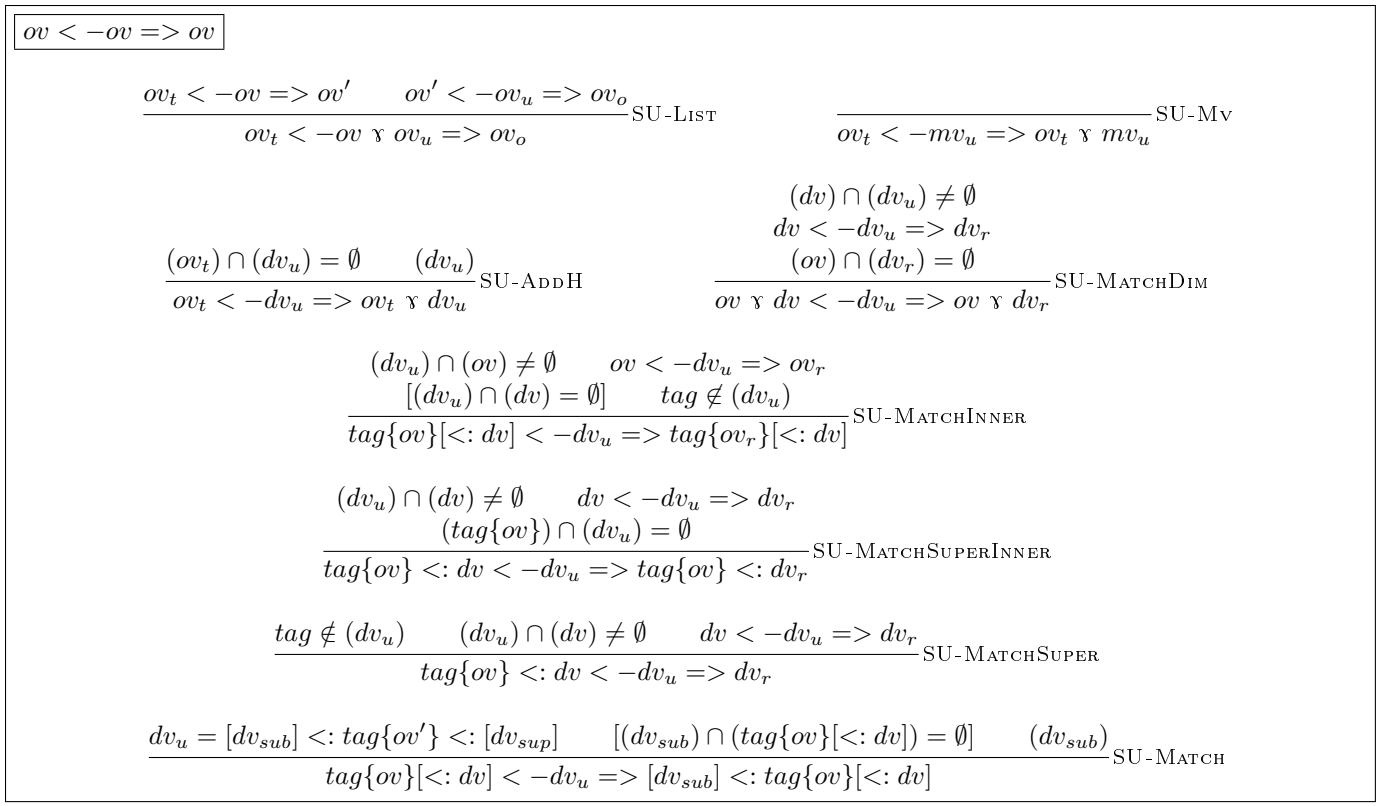


Figure 5: The formal semantics for Plaid state update, as presented in [20].

the most specialized state in the dimension. The *ov* represents all the members defined by the *tag* state, and also all the other states that compose the *tag* state — that is, all the dimensions of an and-state. The symbol $\curlyvee$ appears between these, to indicate that they are composed. A dimension value may also contain another *dv* (the $[<: dv]$ part of $tag\{ov\}[<: dv]$), which indicates that the *tag* state specializes the or-state *dv*. Thus, we would represent a `File` in the `OpenFile` state as:

`OpenFile`{ *close*(){...}, *read*(){...}} <: `File`{ *filename* = *v*}

The formal semantics in Figure 5 appear in the form of inference rules. The premises appear above each rule's line; if the premises hold, we may draw the conclusion below the line. The inference rules in Figure 5 concern the state update judgment, $ov < -ov => ov$. The judgment accepts two object values, *target* before the state update arrow (`<-`) and *update* immediately after the state update arrow. The judgment determines the object that results from enacting state change from *target*'s state to *update*'s state.

To determine the object that should result, we must first identify the dimension that is being altered. For instance, if the object `box` is in the `Open` state, updating the state of `box` to `Closed <: OpenStatus` should affect the `OpenStatus` dimension but not the `EmptyStatus` dimension. The `OpenStatus` tag is a dimension within the `Box` tag, but state update should still identify the `OpenStatus` state in object `box` and change its specializing substate from `Open` to `Closed`. Only dimensions at the top level of the *update ov* can be altered with state change. Thus, updating `box` to `NewState {Closed <: OpenStatus}` would not affect `box`'s `OpenStatus` dimension, because `OpenStatus` is a dimension of `NewState` — it is not at the top level. In fact, updating `box` to `NewState {Closed <: OpenStatus}` would fail because state update would add the new dimension `NewState`, and `OpenStatus` would appear twice in `box`'s state hierarchy, which is disallowed in Plaid's semantics.

State update maintains two important properties. First is the unique dimension property, which requires

that a state either has no supertag or is always under same supertag. That is, it can only appear in a single dimension. The other essential property is the unique tag property. This property requires that a tag appear only once in a given object's state hierarchy.

Inference rule SU-List divides the *update ov* into its component dimensions, each of which may be handled individually, since dimensions are independent.

If the state update is only adding members (for instance, in the statement `box <- {val color = "brown"}`), we compose the *target* and *update* objects to produce the output object. This is rule SU-Mv. If there is no overlap between the tags in the *target* and *update* state hierarchies, we again simply compose the two objects to produce the output object. This is SU-AddH. All other rules apply only if there is a match between the tags of *target* and the top level tags of *update*.

We apply SU-MatchDim when we have found the dimension of *target* that will be updated. Because an *update* tag has been matched in the relevant dimension of *target*, the unique dimensions property allows us to conclude that the top level tags of dv_u will not appear in *ov*. Therefore, state update can proceed on only the relevant dimension of *target* (*dv*). To ensure we do not violate the unique tags property, we must check that the tags of the state change result (dv_r) and the tags of the unmatched part of *target* (*ov*) do not intersect, because the unmatched dimensions will appear in the final object.

We apply SU-MatchInner when one of the dimensions (*ov*) of the current tag in *target* matches a top level tag in *update*. When this is the case, we can proceed with state update by enacting state change on only the matching tag in *target*. To ensure we maintain the unique tags property, we make sure there is no overlap in the tags of any of *target*'s supserstates(*dv*) and the tags being introduced by the new dimension of *update* (dv_u), because *target*'s superstates will appear in the result object.

We apply SU-MatchSuperInner when a top level tag in *update* matches a dimension of a superstate of *target*. When this is the case, we can proceed with state update by enacting state change only on the superstates of *target* (*dv*). To uphold the unique tags property, we ensure there is no overlap between the current tag of *target* (*tag*) and the tags in the relevant *update* dimension (dv_u), because the current tag of *target* will appear in the result object.

We apply SU-MatchSuper when we have identified the correct dimension, but have yet to reach the level where we find a match. The current tag (*tag*) is not in *update* (dv_u), but there is a match between tags in the superstates (*dv*) and the tags in the relevant update dimension (dv_u). This is a case where we know the current tag will not appear in the result object, since it does not appear in *update*, but a parent tag does. Thus, enacting state change on just *target*'s superstates (*dv*) will yield the result object (dv_r).

Finally, SU-Match applies when the current tag (*tag*) matches a tag in the top level tags of the relevant *update* dimension (dv_u). All child tags that did not appear in the *update* object have already been discarded by SU-MatchSuper. Thus, we identify any substates that specialize the current tag (*tag*) in *update*. These substates (dv_{sub}) are then added to the result object. No other aspect of the current tag or its superstates is altered, so to ensure the unique tags property holds, we must confirm there is no overlap between the current tag plus its superstates' tags ($tag\{ov\}[<: dv]$) and the tags of the new substates (dv_{sub}).

These inference rules lay out precisely how a Plaid implementation must execute state change. They determine exactly what information a runtime representation of state must store. Specifically, any state representation must store complete information about: every state's tag; every state's component dimensions; every state's superstates and specializing substates; and, in order to add and remove states, the members associated with each state.

Designing a representation that efficiently fulfills these requirements and implementing state change in accordance with the state update semantics are the novel challenges of compiling Plaid.

3 Related Work

In this section we consider how other work — including another implementation of Plaid — has facilitated object reclassification. We discuss the efficiency implications of using their methods to implement Plaid's state change.

3.1 Implementing State Support

When Strom and Yemini first introduced their typestate checking scheme, they embedded typestate in the NIL language [19]. NIL is not an object-oriented language, and the use of typestate was motivated by the need for safe data structures, which would

not corrupt other programs running on the same systems. As the method they developed was aimed at checking typestate statically, the new demands of incorporating state in a language were all handled in the compiler. Later, the creators of the Fugue system would incorporate typestate into their object-oriented language [9]. That language, coupled with its static typestate system, formed the basis for the Fugue tool. The tool itself was used as a typestate checker for languages that compile to .NET. As we were interested in creating an implementation for a dynamically typed version of Plaid, we do not consider the NIL or Fugue implementations further.

Some languages offer explicit state manipulation without offering any inspiration for a Plaid implementation scheme. This is most often the case in languages that implement restricted forms of object reclassification, such as Taivalsaari's modes [21]. In his class-based languages, object interfaces have modes, each of which is explicitly defined and can provide its own mode-specific versions of object members. A transition function controls shifts between modes, and updates specific mode instance variables that are used to track state. However, Taivalsaari's model does not allow for defining different members in different states, but only for changing the definitions of a stable set of members. This restriction alone prevents his model from forming the basis of an implementation of Plaid's state change.

Dynamic languages like Self [22] and Smalltalk [16] provide mechanisms for modifying an object's class at runtime. Self is a prototype-based language that permits directly adding or removing members during execution, and also allows the programmer to designate one or more slots as parent slots. Like other slots, any or all of an object's parent slots can be mutable [7]. When a message is passed to the object, the receiver first searches for a slot of that name. If it fails to find one, it passes the message to the objects in its parent slots, and so on recursively. This allows the construction of complex inheritance hierarchies. Within Self, all assignable slots (for instance, the assignable slots included to implement dynamic inheritance) are stored with each clone of a prototype. Therefore, changing an object's state by reassigning to an assignable parent slot does not alter any other object, even in its clone family. State change can be mimicked in Self by reassigning to a parent slot.

To use the Self model as a way of implementing abstract state and abstract state change, our compiler would generate an object for each state in `Car`'s state hierarchy. In the case of the `Car` object, that would mean an object for `Braking`, one for `BrakingStatus`, one for `Driving`, and so on up to the top level `Car` object. `DrivingStatus` and `CleanStatus` would be in parent slots of `Car`, `Driving` would be in a parent slot of `DrivingStatus`, `BrakingStatus` and `DirectionStatus` would be in parent slots of `Driving`, and so on down to the most specialized states. Thus, the `Car` state could inherit methods from all these other states. To access a method like `stopBraking()`, which is a member of the `Braking` state, the message would be passed first to the `Car` object, then to one of its parents — `DrivingStatus` or `CleanStatus` — then to the parents' parents, until the `Braking` object, where a match is finally found. To enact state change from the `Braking` to the `NotBraking` state, we would start at the `Car` object, search the parents until identifying the `BrakingStatus` state, then change its parent from `Braking` to `NotBraking`. While using Self parent slots to hand-code state change for a particular state space is unwieldy, it is a viable model for compiling from a language with abstract state to a language without it. This Self-inspired model of grouping state-specific behavior represents perhaps the most intuitive way to facilitate easy state change.

In Smalltalk, an object's state can be modified using the `become` method. While Smalltalk-72 used direct memory addresses and reference counting to store objects, and Smalltalk-76 used bits of the object pointer to encode class information, Smalltalk-78 introduced the use of an indexed object table. This facilitated the new `become` primitive, which simply swapped the contents of two objects' object table rows. Consider the situation where `a1` and `a2` point to the same object as `a`, where `a := 'a'`, and `b1` and `b2` point to the same object as `b`, where `b := 'b'`. The statement `a1 become: b1` would result in all of `a`, `a1`, and `a2` pointing to the string `'b'`, while all of `b`, `b1`, and `b2` point to the string `'a'`. The underlying mechanism in this case is as simple as exchanging the entries in a table.

The Smalltalk model does not lend itself well to a Plaid state change implementation. Using a class-swapping method to implement state change would mean having to use an object with exactly the intended state structure of the receiver. Because Plaid allows state composition, it is not only a matter of keeping a class for each of the four possible `Box` state combinations; it is also necessary to keep a class for a `Box` with an additional dimension — with any addi-

tional dimension, in fact. The code generator cannot feasibly generate objects with all possible classes because Plaid objects can be composed dynamically.

The State design paradigm offers a potential state change implementation scheme [12]. In the State design pattern, programmers construct multiple different classes to implement a single conceptual class. This approach means different states must copy wrapper class interfaces, leading to extensive fragile code duplication. While this is unpleasant for programmers, it would not be an obstacle for a compiler which generates those interfaces automatically. The State design pattern is also criticized for transferring consistent non-state-specific data between different objects. For instance, changing a `File` from the `OpenFile` to `ClosedFile` state in a State design implementation would mean passing `filename` to the new `ClosedFile` object. This is non-ideal even in an implementation setting, but it may not be a tremendous obstacle.

Looking to the Fickle language implementation [4], we see an approach that in fact looks very similar to using the State design paradigm to implement language support for state change. The originators of the Fickle language [10] created a translation of Fickle into Java [4]. To accomplish this, they represented each Fickle object with a pair of Java objects, where one object was a wrapper (the interface for the object), and the other was an implementor. The implementor handles member accesses, and can be changed to reflect the state at any particular time. For any wrapper-implementor pair $< w, i >$, the class of i must be a proper subclass of the class of w.

Ultimately, however, a Plaid implementation that takes its inspiration from the State design paradigm would encounter the same problem as an implementation that takes its inspiration from Smalltalk. To enact state change, the runtime would need a class that reflects exactly the intended end state of the receiver. This is possible in Fickle only because it lacks Plaid's state composition operators.

Cohen and Gil [8] promote object evolution, a restricted version of state change in which an object may gain but never lose members. They introduce the idea of an evolver which — like a constructor — has the responsibility of initializing new members. The evolver initializes members that appear in only some instances of a class.

Cohen and Gil consider three alternative methods for adding new data members to objects that are already on the heap [8]. Their first solution is to use forwarding pointers. In this scheme, there are no direct pointers; rather, each pointer stores the location of a forwarding pointer, which is itself a pointer to the object. If the object is reallocated in memory to accommodate new members, it is sufficient to modify the forwarding pointer. Their next solution is to run a full memory compaction every time an evolution is enacted. Compaction then completes the work of moving objects in memory, and the newly evolved object can be assigned to a location with sufficient space. Their final solution uses objects as proxies. In this scheme, all objects have a field called `newRef`, which is `null` until the object is evolved. When `newRef` is not null, all member accesses on the object are forwarded to the object stored in `newRef`. These implementation methods again suggest an implementation scheme like the Smalltalk and State design paradigm scheme, in which the compiled Plaid code must have access to an appropriate class for any state change or composition the programmer may want to execute. The problem of state change is again reduced to the problem of constructing a new class at runtime.

All these solutions that involve using one object rather than a tree of objects raise questions about how to identify the receiver's current state, and how to use it to identify the appropriate end state. Even in the Cohen and Gil implementations, where members are added in evolutions and could be associated with those evolutions, their solution is always to have one object with all the members. Because their object evolution never removes members, they never address the question of how to reverse an evolution, or replace one evolution's effects with another's. This issue is central to any implementation of Plaid, which must be able to identify not only which states to add, but also which states to remove — calling `open` on a closed file should add `OpenFile` state, but it should also remove the `ClosedFile` state. The Smalltalk family of solutions offers no insight into appropriate ways to identify objects' and states' current state hierarchies, or how they should be combined during state change.

In the Actor model, state can be manipulated in several ways[15][1]. Each time an actor receives a message, it determines its response to future messages. A change in response may be as simple as updating state variables. This is the same sort of state change that many object-oriented languages permit. For instance, reducing a bank account balance changes the behavior of the withdrawal function, because the function will allow less money to be with-

drawn. However, Actor-based languages also allow actors to change the procedures that will respond to future messages. To some extent, this resembles the sort of state change that can be enacted in languages with first-class functions.

The ability to alter how the same method call behaves at different points during program execution is very important in implementing Plaid's state change. It is important that `f.read()` produce a different result when it is called on an `OpenFile` and a `ClosedFile`, and changing or removing the method is one way to accomplish that. However, the Actor languages' replacement behaviors and other languages' first-class functions provide no guidance for how to group sets of behaviors that should be associated with particular states. Given the ability to alter methods, what is the best way to identify and remove all the members associated with `Braking`, then identify and add all the members associated with `NotBraking`?

From several of the last examples, it is clear that the challenge of implementing Plaid's state change is ultimately traced to the challenge of being able to create an appropriate class. Changing method behavior may not by itself suggest a fully fledged scheme for implementing state change, but it could be an important tool for building objects with all the functionality an object should have after state change.

3.1.1 The Java Implementation of Plaid

The first implementation of Plaid compiled Plaid to Java. The general approach of that implementation was to maintain a map of each object's members in the Java runtime. Each method is represented by an object with an `invoke` method that contains the body of the function. Member access in the source code produces Java code that calls the runtime's `lookup` method, passing in the name of the member, and the string that represents the relevant object or scope — that is, the Java object that represents the Plaid object.

Within the runtime `lookup` function, the object's map of members is identified. Next, the function searches for the name of the passed in member. If it finds the member, it returns it, and the `invoke` method can be called on the returned object. If it does not find the member, it identifies the parent of the current object (if one exists), and checks the parent object for a matching member that the substate inherits. This process is repeated until the member is found, or until there are no more states to check.

The result is that method calls and field reads require traversing a collection of states to check for the relevant member and searching a number of objects' member maps, because each state is represented by a collection of objects with collections of members. Each object also has its own map of members which is searched first. If a member's value is specific to the current instance, it is stored in this object map. For example, if an object is created with the Plaid code `A{val a=1;}`, the object is an instance of the state `A`, but with the value of field `a` changed to 1. The value of `a` would thus be stored in the object's own map, since it should override any assignments to `a` that appear in states composing the object's state.

State change in this implementation is accomplished by adding and removing state objects from a Java object's collection of state objects. The appropriate states to add and remove are determined by storing a hierarchy of tags, and associating each state object with its tag.

4 Implementing State and State Change

The central challenge for a Plaid compiler and runtime is to enable fast state change and fast member access for a shifting set of members. While maintaining state and permitting state change is useful from the perspective of a programmer, the challenges for implementation are significant. If an object's members do not change, there is never a need to allocate additional space for an object that has already been declared. More importantly, a stable set of members means there can be a standard, consistent way to access particular members throughout program execution, for all instances of a given class. Thus, efficient language implementation has traditionally required that every object have a stable set of members throughout its life cycle. However, state is a useful abstraction only if it is accompanied by state transition — and state transition allows members to change at runtime.

The key requirement is that given the runtime representation of a state `s` and the runtime representation of an object `o`, the runtime must be able to transition `o` into state `s` in accordance with Plaid's semantics. To accomplish this, any representation must store, in some format: the tags of all the states that compose the state of `o`, those states' relationships to each other, the members associated with each state, and the values and method bodies of those members.

These data may be accessed for method calls and field reads, and also to enact state change. The core challenge for our implementation is to find a way to store these data that both enables state change and facilitates fast member access.

4.1 Intuitive Approaches

To date, implementations of object reclassification generally suggest the same intuitive representation for state at runtime. The intuitive representation entails maintaining a target language object for each state and substate: one for `File`, another for `OpenFile`, another for `ClosedFile`. The object for each state would keep track of field values and method bodies that it defines, and another field would store the state's tag. Pointers between state objects would be sufficient to track substates' relationships to each other. State change would demand nothing more than changing a few pointers. Adding and removing states would be as simple as adding and removing nodes in a tree.

Although this pointer manipulation approach makes state change easy, consider that a file will commonly be opened once, read many times, and closed once. It is easy to bring to mind many similar examples, and it is a rare program that will require more state transitions than method calls or field reads. This being the case, fast member access is essential to a fast implementation. In the intuitive approach above, a call to `close()` would require first searching the `File` object for the desired member, then the `OpenFile` object, before finding the appropriate method. The runtime would have to traverse an entire network of state objects until it finds one that stores the desired member. While this may not greatly affect execution time in the case of a simple `File` object, consider the complex `Car` state pictured in Figure 4. The process for finding the `stopBraking()` method of `BrakingState` would require a search through five to nine objects in the state hierarchy.

Aside from prioritizing state change over the more common method access, this intuitive approach makes the execution time of member access dependent on how a state is composed — more component states and more substates would increase overhead. This discourages the use of Plaid's novel state abstractions. It is clear that keeping a tree of component objects will not produce an efficient implementation.

A second intuitive approach would involve simple state checks. State checks could be used to mimic Plaid's state change in almost any high-level language. While it would be unpleasant to manually code all the state checks required for a complex `Car` state, a Plaid compiler could automate the process. State variables would store the current states, and the runtime would use state checks to identify the appropriate response to a member access.

This implementation suffers from the same flaws that plagued the earlier intuitive approach: first and most importantly, it slows member access, requiring a series of state checks before each member access; second, member access time would depend on the depth of the state hierarchy.

The naive approaches above highlight a central implementation challenge: maintaining all the necessary information (all states, their relationships to each other, their associated members) and manipulating state appropriately (adding and removing states, specializing, composing), while still enabling quick member access. By packaging all of a substate's information into a node in a state hierarchy tree, an implementation scheme sacrifices fast method calls and field reads. By keeping state information in simple state variables and necessitating state checks, an implementation scheme sacrifices fast method calls and field reads. There is an unavoidable trade off between the simplicity of state mutations and the speed of member access.

4.2 An Outline of Our Solution

Our implementation offers a new representation for state at runtime. Our representation ensures fast member access and stores state information in a form that should facilitate future state change optimizations. We use this representation to implement a JavaScript compilation target, which consists of a Plaid to JavaScript compiler and a JavaScript runtime.

The essential features of our runtime representation are:

- It takes exactly *one* JavaScript object to represent each Plaid object.
- A Plaid object's one corresponding JavaScript object contains only the Plaid object's currently available members, which depend on its state.
- Each Plaid object's corresponding JavaScript object includes a metadata element, which en-

codes all the information necessary for performing state change on the object.

The goal of our representation scheme is to find a way to keep member access fast, even though members may change during execution. The need for fast method calls and field lookup motivates the central restriction we placed on our implementation: any member of a Plaid object must be a member of a single corresponding target language object.

To comply with this restriction, we developed three requirements.

All Available Members in Target Object. If target language object `f` represents `plaidF`, a Plaid `File` object, all members available to `plaidF` at a given point in the program's execution should be available to `f` at that point.

No Unavailable Members in Target Object. When attempting to access a member of `plaidF` that is not available at a given point in the program's execution, the member should also be inaccessible in `f`. That is, the runtime should not perform any state checks before allowing a field read or method call.

No Indirection. There should not be any layers of indirection to slow a field read or method call.

Taken together, these requirements mean that the members of `f` at any given point during execution should be the exact analogue of `plaidF`'s members at that point. A call to read should produce the code `f.read()` in the compiled code, rather than trigger a search through `File` and `OpenFile` objects.

To follow these requirements, it must be possible to add and remove members at runtime from an object in the target language. This motivates our use of JavaScript, which is prototype-based and supports first-class functions. Using JavaScript allows us to build a runtime that manipulates object members during execution. With JavaScript's first-class functions, it is trivial to copy all the members of an object's component states and substates to a single object.

Flattening the object representation — that is, storing all of a Plaid object's members in a single target language object — means that none of the state-related information is encoded in the object structure. With all of `File`'s members and all of `OpenFile`'s members in a single JavaScript object, there is no clear way to distinguish between `File` and `OpenFile` methods, which complicates state change. To solve this problem and store all the information needed to enact state change, we introduce a metadata item. The metadata item is a tree encoding all of a Plaid object's states, their relationships to each other, their tags (unique names) if they have any, and the state with which each member is associated. This information is sufficient for applying Plaid's state change semantics to an object. Each JavaScript representation of a Plaid object contains a metadata item.

4.2.1 JavaScript

We use JavaScript as our target language for several reasons. First, as web programming becomes more and more common, and more appealing even to non-programmers, it is essential to produce languages that novices will find natural and intuitive. To the extent that Plaid simplifies the use of state and reduces the potential for confusion surrounding implicit protocols, compiling Plaid to JavaScript should advance that goal. Our implementation makes it trivial to extend Plaid programs with JavaScript libraries, vastly expanding the immediate uses of the language. Ultimately, the ability to use Plaid for web programming could make it useful to a large population.

Second, well-optimized JavaScript engines such as V8 [23] and SpiderMonkey [18] continue to reduce JavaScript execution time. While the speed of JavaScript virtual machines in the past might have made a JavaScript compiler quite inefficient, their continual improvement means a JavaScript implementation of Plaid may be relatively fast.

Finally, JavaScript's primary appeal lies in its approach to objects. Compiling a language that supports state change to a language without state change does not necessarily require adding or removing members at runtime. However, if we add the requirement that the set of `tl`'s members match the set of `p`'s members, the ability to dynamically change object structure becomes a prerequisite. JavaScript's flexible object model allows easy addition and removal of members. Its support for first-class functions allows our runtime to manipulate an object's methods during execution. The combination yields a relatively straightforward way to build up an appropriate object from information stored in other JavaScript representations of Plaid objects and states. Creating an object from other objects' data is exactly what the runtime must accomplish to carry out state change. Thus, JavaScript offers a fairly clear and intuitive way of expressing the individual modifications required for state change.

4.2.2 Metadata and State Change Procedure

The metadata item is a JavaScript tree representing a Plaid object's state hierarchy. Each node in the tree represents one of the object's current states. The node contains the state tag, if there is one, and the names of all members associated with the state. Each node also tracks whether the node is a specializing substate of the parent state (declared using `case of`), or whether it is a component (added using `with`). To enact state change, the JavaScript runtime compares the metadata item of the object to be changed and the metadata item of the update state.

For instance, Figure 6 shows a visual representation of the metadata for a `Car` object. `Car` is an and-state, composed of `DrivingStatus` and `CleanStatus`. `DrivingStatus` is an or-state, and the object is currently in the `Driving` state, which specializes `DrivingStatus`.

Traversing the trees of the target and update objects reveals how to apply Plaid semantics to any given state change call. The trees are used to identify which states, if any, should be added or removed from the object. Following state change, the metadata is updated to reflect the new set of states. For instance, in our car example, if an object `car` is in the `NotBraking` state, the code `car<-Braking` should change the JavaScript metadata from the tree in Figure 6 to the tree in Figure 7. The `NotBraking` node is removed, and a `Braking` node is added.

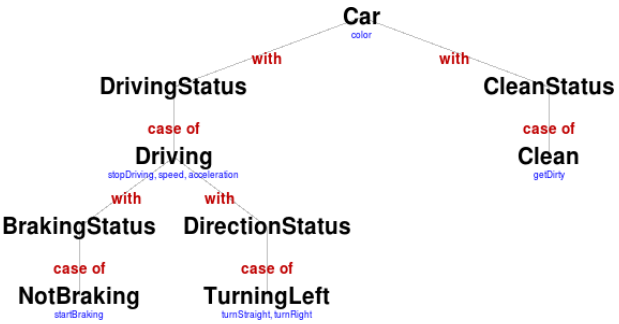


Figure 6: The metadata for a `Car` object with and-states `Car` and `Driving`, and or-states `DrivingStatus`, `BrakingStatus`, `DirectionStatus`, and `CleanStatus`.

4.2.3 Metadata and Member Access

Next, consider how the use of a metadata item affects how we can access object members. The key

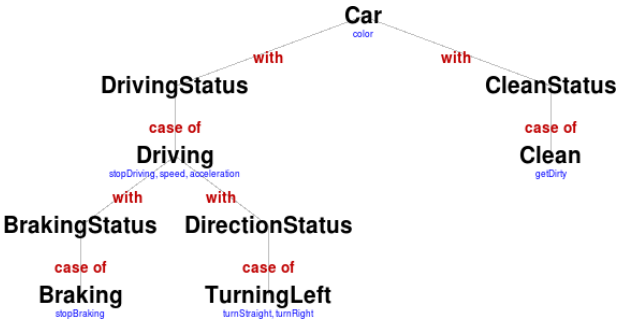


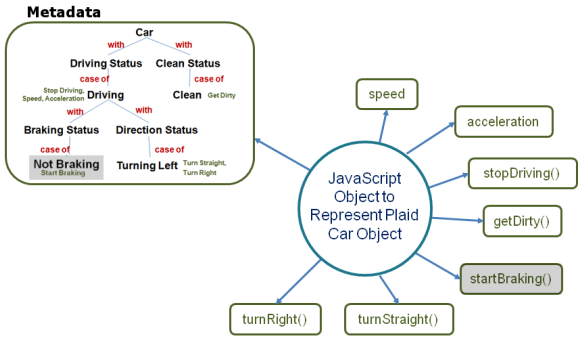
Figure 7: A visual representation of how the JavaScript metadata for a `Car` object should change when it is transitioned from the `NotBraking` state to the `Braking` state. The tree in Figure 6 shows the metadata before the state change. This figure shows the metadata after the state change. The `NotBraking` node has been removed, and the `Braking` node has been added.

reason for developing a new representation scheme for state at runtime was to permit all of a Plaid object's members to be maintained as the members of a single target language object. If the member is in fact valid, its location is immediately known without the need to search through a tree of state objects, without any form of lookup function.

To ensure fast member lookup is always possible, every field and method is added to the JavaScript object at its creation. When the process of state change reveals a set of states to be removed, the members associated with those states are identified from the object's metadata, and those members are removed from the JavaScript object. When the process of state change reveals a set of states to be added, the members associated with those states are identified by examining the metadata of the state object. The state object's values for those members are then copied over to be members of the object undergoing state change. The result is that at any point during execution, the JavaScript representation of a Plaid object has all of the fields and methods available to the Plaid object at that point, and it does not have any additional fields or members. Figure 8 shows a visualization of all the changes that would occur in the JavaScript object representing a Plaid `Car` object when it is transitioned from the `NotBraking` to the `Braking` state; this transition illustrates how the object's member set is kept up to date.

With this approach, our implementation scheme

JavaScript Object Before State Change



JavaScript Object After State Change

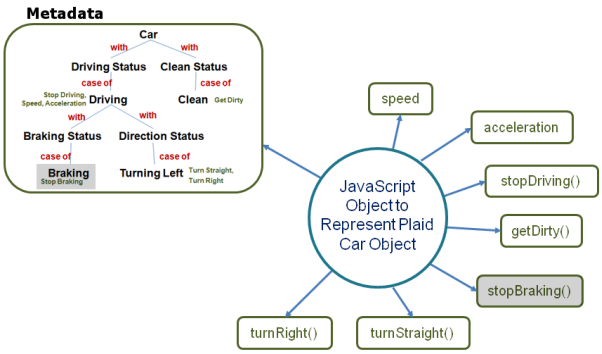


Figure 8: A depiction of the changes to the JavaScript object representing a Plaid `Car` instance when it is transitioned from the `NotBraking` to the `Braking` state. The top item shows the object before state change, and the bottom item shows the object after state change. The metadata changes as it changed in Figure 7. However, here we also see that the methods stored with the JavaScript object change. Note that the method `startBraking` is removed not only from the metadata but from the JavaScript object that represents the Plaid instance. A `stopBraking` method is added to the object.

prioritizes fast method calls and field reads. A method call in the source code produces a simple method call in the compiled code.

5 Runtime

We discuss the specifics of representing Plaid objects and states during the execution of the JavaScript code produced by the code generator. We provide specifics on the metadata structure, the distinction between objects and states, and how the methods for both objects and states are implemented. The runtime's most important role is to be able to alter objects' state structures during execution.

5.1 Objects and States

It is important to understand the distinction between objects and states in Plaid. An object is an instance of a state. Many different objects may be in the same state, and a single object may — as described in Section 2 — be in many different states, and thus have many more features than are found within a single one of its states. One creates a Plaid object from a Plaid state. In this sense, a Plaid state is like a class, and a Plaid object is like an instance.

State change does not occur with two states. Rather, the item on the left of the `<-` is an object, and the item on the right is a state. For instance, in the statement `car <- Driving`, `car` is an object, an instance of the `Car` state; `Driving` is a state. An object can be transitioned into the state of another object, but this requires first calling `freeze` on that object. The `freeze` method produces a Plaid state with all of the fields and methods of the Plaid object on which it was called.

5.2 Adding and Removing Members

To maintain the current state of an object, the runtime must add and remove object members, and change their values. JavaScript's support for first class objects makes these runtime alterations possible. To remove the member `foo` from object `o` requires the following JavaScript code: `delete o["foo"];`. To add the member `foo` to object `o` and give it the value `val` requires the following JavaScript code: `o["foo"] = val;`. The latter can also be used to change `o`'s value of `foo` to `val` from any other value. The value of `val` can be a primitive, an object, or a function. Thus the structure of the JavaScript object can be modified in any way required for our implementation.

5.3 State Metadata

Within the runtime, the metadata tree describing a state structure is stored as an array of arrays. In each node of the tree, we store the state's tag, the names of all associated members, and whether the node is a case of its parent or composed using with.

```
1 var plaidNewState_OpenFile = new PlaidState(); // create JavaScript object to represent OpenFile
2 plaidNewState_OpenFile['read'] = function read () {...} // add the appropriate methods
3 plaidNewState_OpenFile['close'] = function close () {...} // to the new JavaScript object
4 //add the array of arrays that represents the metadata
5 plaidNewState_OpenFile.tree = [['',[],'with'],
6                                [['File',[],'with'],
7                                  [['OpenFile',['close','read'],'']]]];
```

Listing 4: The JavaScript code that would be produced for creating a JavaScript representation of the state `OpenFile`.

Figure 9: A visual representation of the metadata that would be produced for an object in the `OpenFile` state. `File` has field `filename`, and it is in state `OpenFile`, which has methods `read` and `close`.

Recall the `File` example from Section 2.4, which showed in Listing 1 the Plaid code for creating a `File` and two substates, `Openfile` and `ClosedFile`. Listing 4 contains the JavaScript code that would be produced by running our code generator on the code in Listing 1 and creating a file in the `OpenFile` state. The list

```
[['',[],'with'],
  [['File',[],'with'],
    [['OpenFile',['close','read'],'']]]]
```

on lines 5 through 7 is the metadata tree for the `OpenFile` state. Figure 9 provides a pictorial representation of the list. It is easy to see how complex the metatdata tree for our `Car` state would be.

5.3.1 Members and Tags

The metadata is stored in the same format for both states and objects. Thus the same operations can be performed on the metadata of states and objects. We cover some basic information that can be extracted from metadata trees below. These will be used as the building blocks for some of the more complicated methods that must be implemented to use states and objects. Other complex methods will require their own, more specialized ways of traversing the metadata.

All Tags: A state's tag is its unique name. Tags are useful for ensuring the same state is not added to an object's state hierarchy in multiple places. A state's tag is also used in the Plaid pattern matching construct. To find all tags, the runtime has a function that completes a simple traversal of the whole metadata tree, identifying the tags of any non-anonymous states, and ultimately returning the set of all named states present in the Plaid state or Plaid object.

All Members: To find all methods and fields, the runtime has a function that traverses the full metadata tree, accumulating all members that appear in the metadata, including both members associated with a given tag and members associated with anonymous states. The resultant list is a list of every member for which the Plaid state or Plaid object stores a value, as well as all members that have been declared without being assigned a value.

Match (tagName): Plaid supports a convenient matching construct for executing different code based on the state of a particular object. The programmer must provide the matchable states in an order of their choice. According to Plaid's semantics, the code that should be executed is the code for the first such state that appears anywhere in the object's state hierarchy. Thus, if an object is in both state `A` and `A`'s child state `B`, but the match statement checks first for the `B` case, the code that should be executed is the code associated with state `B`. To conveniently answer this sort of demand, the runtime has a function that traverses the metadata either until the relevant tag (tagName) is found, or until it has checked the entire tree, and returns a boolean.

Members By Tag (tagName): The runtime function for identifying the members associated with a given tag (tagName) proceeds by traversing the tree until it matches the tag, if it is present. If it is present, the function returns the list of members associated with the tag. If the tag is not present, it returns an empty list.

Unique Members: An object satisfies the unique members property if: (a) no two states in an object define a member with the same name; or (b) if two states in the object do define a member with the same name, one of the defining states is a transitive specialization of the other. Plaid objects are required to satisfy the unique members property at all times. Sometimes this property must be explicitly checked.

Checking this property is complicated by the fact that a state is permitted, according to Plaid's semantics, to have a member that overrides a parent state's member. In this context, a state `A` is a parent state of state `B` if and only if (1) `A` is higher in the metadata tree and (2) all branches between `A` and `B` are `case of` branches. That is, `B` was explicitly declared as a substate of `A`. Of course, any state that appears in the path from `A` to `B` is also a parent state of `B`, since the same conditions are necessarily met by any intervening state.

To check unique members under these conditions, the runtime uses a recursive function that accepts the current portion of the metadata list being checked, a list of the tags of all of the current portion's parent states, and a list of all the members encountered thus far in the traversal. The function returns an object that encodes whether a repeat member was found, and the name of the offending member if there was one. Because JavaScript passes arrays by reference, the list of members always reflects all members found so far in the traversal, even without explicitly bubbling up the new additions.

For each list on which it is called, the `checkUniqueMembers` function first checks whether the current state has an and-state parent or an or-state parent. If the state is a `case of` its parent, the current state is added to the list of parent states. If not, the list of parent states is cleared before adding the current state.

The function next iterates through the members associated with the current state, checking whether each new member shares a name with any of the members in `memberList`, the list of all members found so far. If a member does not yet appear in `memberList`, it is added. A new entry is appended to the member list which includes both the name of the member and the tag with which it is associated. If the new member *was* found in `memberList`, there is a potential conflict. The function extracts the tag from the `memberList` object that revealed the possible conflict, and the tag is compared to all the tags in the list of parent states. If the tag does not appear in that list, it indicates that the member exists in some other part of the state hierarchy, and the current state is not permitted to overwrite this. In this case, the function constructs a return object that stores the fact that the metadata tree breaks the unique members guarantee and stores the name of the offending member, and this object is returned. The return value is then immediately propagated back up to the initial `checkUniqueMembers` call.

If none of the members of the current state revealed any conflicts, the function iterates through the state's children, calling `checkUniqueMembers` on them recursively. If none of those calls return an object indicating failure, the function constructs a return item that indicates that no violation was found.

5.4 Runtime States

This section focuses on the representation we developed for Plaid states at runtime, the JavaScript objects that correspond to Plaid states. The JavaScript "class" PlaidState contains all the necessary methods, and a PlaidState object is used to represent each state in a Plaid program. Henceforth, we call the JavaScript representation of a Plaid state a PlaidState. Similarly, we call the JavaScript representation of a Plaid object a PlaidObject.

5.4.1 Instantiate

The `instantiate` method of a PlaidState defines the procedure for creating a PlaidObject from a PlaidState. This requires first checking that the state being instantiated has unique members, then that it has unique tags. Objects that contain the same state twice (i.e. in two different parts of the state hierarchy) are not permitted in Plaid.

Next, the runtime clones the PlaidState's metadata tree and passes it into the constructor method for PlaidObject. The PlaidObject constructor simply sets that tree as the metadata of the new object. All the members of the PlaidState are copied to the PlaidObject with their associated values (also stored in the PlaidState). The resultant PlaidObject is returned.

5.4.2 State Structure Modifications

Each of the methods described below is necessary to facilitate one of the rich state modification features Plaid allows. Because a state is allowed to break the

unique members rule, none of these require checking unique members.

Remove (memberName): The `remove` function is called on a PlaidState object and passed the name of a member. It returns a copy of the PlaidState that lacks that member. The function proceeds by cloning the metadata of the original state, and traversing the clone. If a member with the relevant name is identified in the metadata, it is spliced out. If no such member is found, it throws an error. If the relevant member was found, the value of the member that is associated with the PlaidState is deleted from a new PlaidState with the new metadata. This PlaidState object is then returned.

Rename (oldName, newName): The `rename` function is called on a PlaidState object, passed the name `oldName` of a member, and a name `newName` to replace `oldName`. It returns a copy of the PlaidState in which `oldName` is named `newName`. The function first clones the metadata of the original state, then traverses the clone. If a member with the name `oldName` is found, that entry in the metadata is replaced with `newName`. If the relevant member is not found, the function throws an error. If it was found, a new PlaidState is created with the new metadata, and the value of `oldName` is copied into a new `newName` member of the PlaidState. The `oldName` member is then removed. After this, the PlaidState can be returned.

Specialize (tag, memberName, memberValue): The `specialize` function is called on a PlaidState, passed the name of a tag `tag`, the name of a member `memberName`, and the member's intended value `memberValue`. It returns a copy of the PlaidState on which it was called, in which the `memberName` member now has the value `memberValue`. The member is associated with the tag that was passed in. The function begins by cloning the metadata of the original PlaidState, then finding the portion of the tree that lists members associated with `tag`. If `memberName` is not already present, it is added to the metadata. A new PlaidState is created with the new metadata, and the PlaidState's `memberName` member is given the value `memberValue`. The new PlaidState is then returned.

With (state): For a more detailed description of how and when `with` would be called in the runtime, see Section 6.2. Within the runtime, it is sufficient to know that calling with on a PlaidState and passing in another PlaidState should produce a third PlaidState object that has all the tags and all the members of both. In the runtime function, the metadata tree of the first and second states are cloned, and a new PlaidState is created with the metadata of the first. All the members from both the first and second state are copied to the new, third state. Next, each branch at the top level of the second state's tree is pushed onto the top level of the new PlaidState's tree. That is, the new state has the metadata for both states, and every state that appeared in the top level of one of the original two states also appears in the top level of the new state, which is then returned.

WithMember (memberName, memberValue): The `withMember` function is also focused on composing states, but in this case, the second state is defined by a member declaration. Thus, the only parameters are the member's name and value. A new PlaidState is created from the metadata of the original PlaidState. The runtime creates a list to represent an anonymous state (that is, a state with no tag). This state contains the member that was passed in, and the state is added at the top level of the new PlaidState's metadata. A member with the member's name is then added to the new PlaidState and given the value that was passed in. The PlaidState is then returned.

WithMemberNoValue (memberName): A member can also be added to a state without giving it a value — it can be simply declared. To handle this case, the runtime uses the `withMemberNoValue` function, which precisely repeats the procedure explained above for `withMember`, except that no member is added to the new PlaidState that is returned. Instead, only the metadata is modified.

5.5 Runtime Objects

We discuss the representation of Plaid objects at runtime. Plaid objects are created by using the `new` keyword on a Plaid state. Plaid objects are the analogues of instances in purely class-based languages. Just as the PlaidState class was used to manipulate Plaid states at runtime, the PlaidObject class is used to manipulate Plaid objects at runtime. Henceforth, we call the JavaScript representation of a Plaid object a PlaidObject.

5.5.1 Freeze

In order to use PlaidState functionality on the state represented in a PlaidObject — for instance, to use it as a state change target, or to compose it with other states — the programmer must first freeze the object.

The `freeze` method thus very simply creates a new PlaidState object with a clone of its own metadata, copies all fields and methods to the new PlaidState, and returns it.

5.5.2 Replace

The `replace` or `wipe` function is a variant on state change. If `target` is the PlaidObject on the left-hand side of the `<-` operator and `update` is the PlaidState on the righthand side, `replace` ignores all of `target`'s state information and supplants it with `update`'s state information.

To accomplish this, the function first ensures that `update` conforms to the unique members property and also the unique tags property. If it does not, the function throws an error. Otherwise, the function continues by removing all the members associated with `target`, as revealed in its metadata, adding all the members associated with `update`, and setting `target`'s metadata to be a clone of `update`'s metadata.

5.5.3 State Change

The state change operator `<-` adds to the receiver any states that are present in the updating state but not in the receiver. It removes from the receiver only states that are mutually exclusive with the incoming states.

To informally introduce the process of implementing state change, we first look at a high level overview of the steps that must be followed to correctly enact state change given our chosen state representation. We will also examine how these steps are applied to a concrete example in the figures alongside.

The Plaid statement `target <- update` leads to the following steps:

1. Traverse `target`'s metadata until finding a state with the same tag that appears at the root of `update`'s metadata. If that tag is never found, let the top level of the tree act as the matched tag. Call this the root tag.
2. Traverse `target`'s and `update`'s metadata jointly, matching tags where possible.
3. Identify all tags below the root tag that appeared in `target`'s metadata but not in `update`'s.
4. Remove the nodes identified in the step above from `target`'s metadata.
5. Identify all tags that appeared in `update`'s metadata but not in `target`'s.
6. Add the nodes identified in the step above to `target`'s metadata, in the places corresponding to their places in `update`'s metadata.
7. From the PlaidObject itself, remove all members associated with the states that were removed from `target`.
8. From the PlaidObject itself, add all members associated with the states that were added to `target`.

For example, let `target` be an object in the `Car` state defined in Listing 3, and let `target` be currently in the `Braking` state. Then let `update` be the state `NotBraking`. The Plaid statement `target <- update` leads to the following steps:

1. In step 1 (Figure 11), the runtime identifies `Car` as the appropriate root state.
2. In step 2 (Figure 12), the runtime matches all possible tags until reaching `BrakingStatus`' children. The `BrakingStatus` children in `target` and `update` do not match.
3. In step 3 (Figure 13), the runtime determines that `Braking` does not appear in `update`'s metadata.
4. It thus removes `Braking` in step 4 (Figure 14).
5. In step 5 (Figure 15), the runtime determines that `NotBraking` does appear in `update`'s metadata, but not in `target`'s.
6. It thus adds `NotBraking` to `target`'s metadata in step 6 (Figure 16).

At this point, the metadata has been fully updated. Next, the member `stopBraking` would be removed from the *JavaScript object* that represents `target`, and the member `startBraking` would be added.

The first step in implementing state change in the Plaid runtime is to short circuit the process if `update` is not in a an acceptable state - that is, if it violates unique members or unique tags. If `update` is well-formed, then state change proceeds. To accomplish this, we use two recursive functions, the first to traverse the tree until it finds the appropriate root state in `target`'s metadata, the second to match tags once a root state has been found.

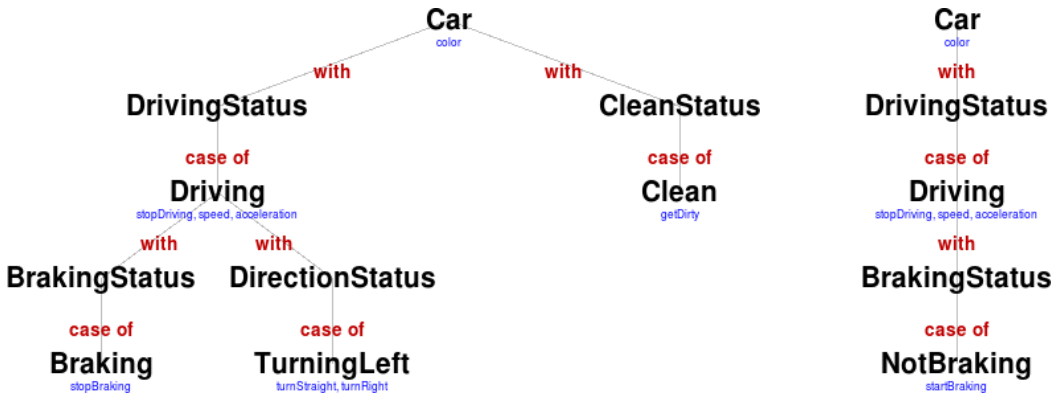


Figure 10: The Plaid object `target` (left) and the Plaid state `update` (right) before state change.

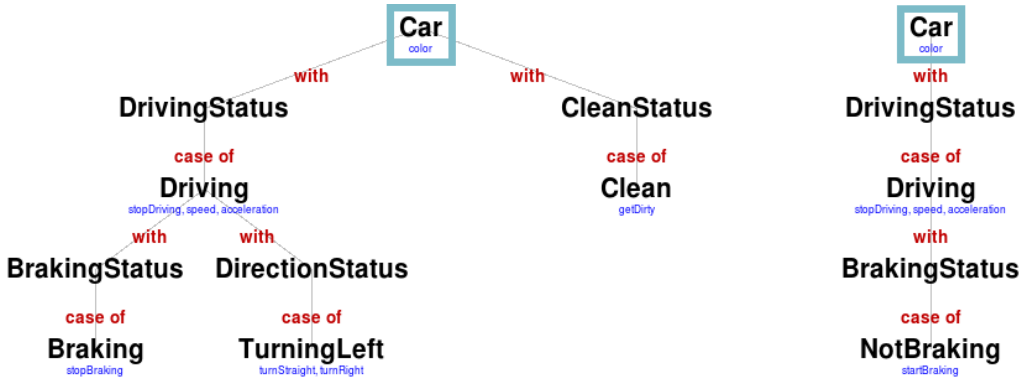


Figure 11: Step 1. Match the tag at the root of `update`'s metadata. In this case, we match `Car` immediately.

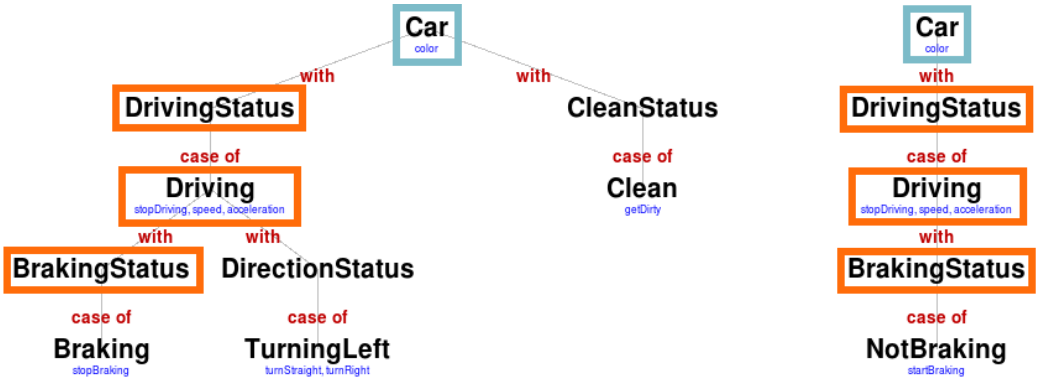


Figure 12: Step 2. Continue matching children as long as all of `update`'s children appear in `target`. We match `DrivingStatus`, `Driving`, and `BrakingStatus` before `target` fails to contain one of `update`'s children.

Let `findRootState` be the function for finding the root state. First, if the current item being examined is connected to its parent by a `with` branch, the list of parent states `parentStates` is cleared before adding the current tag to `parentStates`. Otherwise, the current tag is merely added to `parentStates`. An

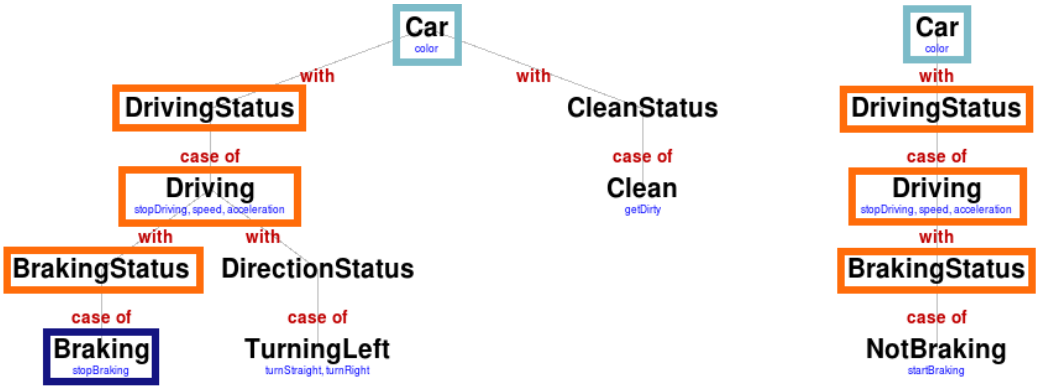


Figure 13: Step 3. Identify a `case of` state that appears in `target` but not `update`. These are mutually exclusive. In this case, the state is `Braking`.

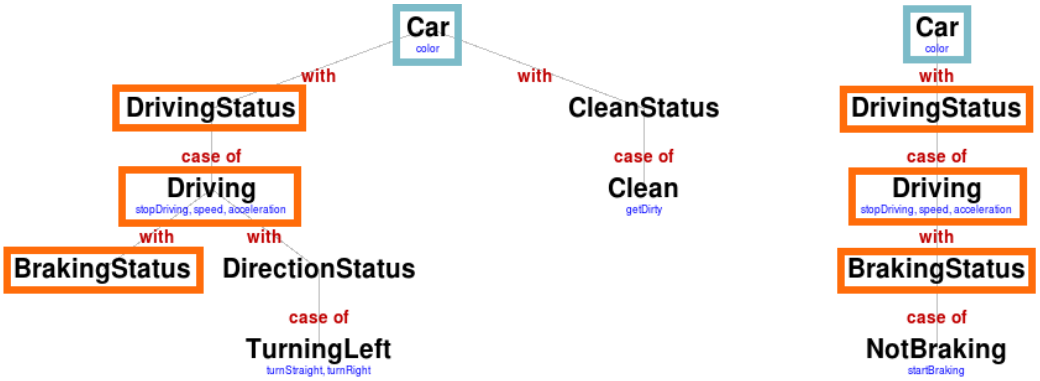


Figure 14: Step 4. Remove the `case of` state that appears in `target` but not in `update` (`Braking`).

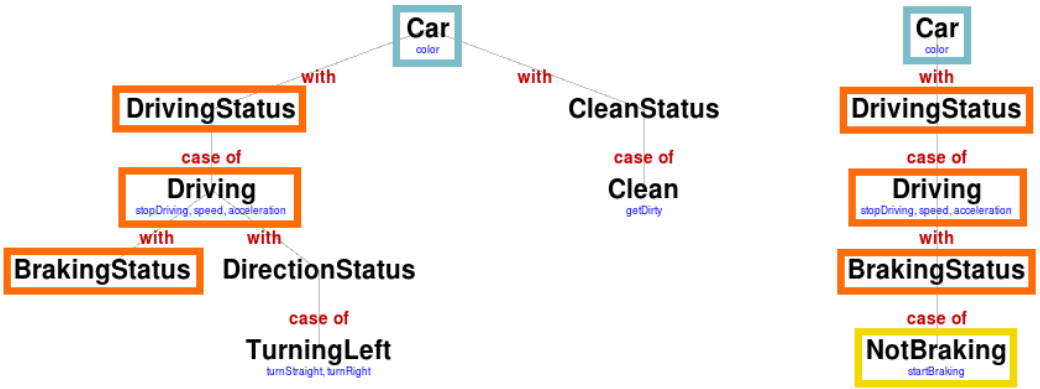


Figure 15: Step 5. Identify the state in `update`'s metadata that necessitated the removal of the `case of` state. In this case, `NotBraking` necessitated the removal of `Braking`.

object `memberData` that stores all the necessary information about members is passed to the function, and updated as necessary. If the current item in `target` is an anonymous state, any members in the current

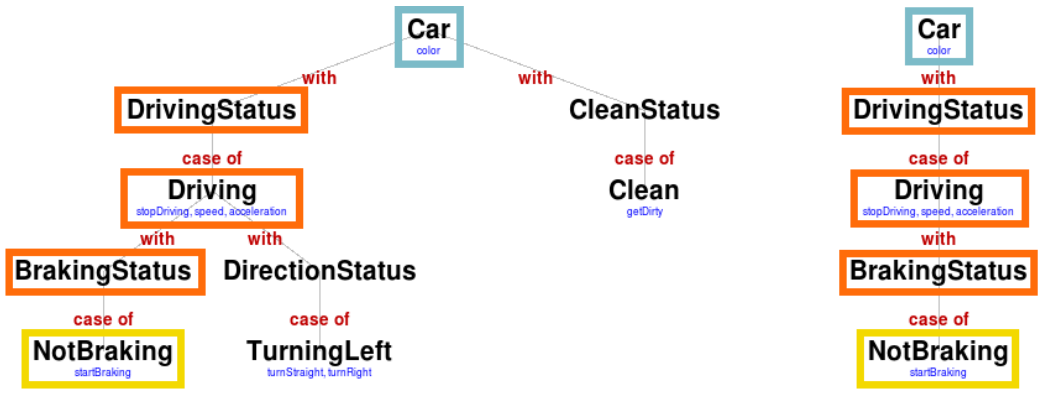


Figure 16: Step 6. The state in `update`'s metadata that was mutually exclusive with `target`'s removed state is added to `target`'s metadata. In this case, `NotBraking` is added to `target`. Metadata update is complete.

item are added to a list of `target`'s untagged members maintained in `memberData`. If the current item in `update` is an anonymous state, any members in the current item are added to a list of `update`'s untagged members maintained in `memberData`.

If `update`'s current item is an anonymous state, it indicates that we must traverse the tree to find a matchable tag. That is, although there may be members at this level, any tags that should be matched will appear as children of the current state. In this case, `findRootState` must call itself recursively on slightly modified arguments. Rather than passing in the entire tree of `update`, it passes in each child individually, while passing the same component of `target`'s metadata that it received as an argument. This allows the function to identify the correct tags in `update` to use as root states to find in `target`.

If `update`'s current item is not an anonymous state, `findRootState` checks whether the current top-level tags of `target` and `update` match. If they do, `findRootState` calls the second function, `matchStates` on the current portions of `target`'s and `update`'s metadata. If they do not, the function calls `findRootState` on each of the current `target` node's children, passing in the same portion of `update`'s metadata that it received. This results in a full traversal of `target`'s metadata, if the appropriate top-level tag of `update` is never found.

The function `matchStates` continues to track `parentStates`, as in `findRootState`. Because `matchStates` is only ever called on portions of the metadata for which the top-level tags are known to match, if both the current portion of `target`'s metadata and the current portion of `update`'s metadata have length 1, the current branch matches without any additional modification. The `memberData` object that has been built up during the execution of `findRootState` and `matchStates` calls is returned.

If the function has not yet reached a leaf, it must continue to make recursive calls to itself. Looping through all children of both current nodes reveals whether any have matching tags. If a tag match is found, `matchStates` is called on those portions of the `target` and `update` trees. During this process, the function tracks whether a match has been found for each child state from `update`.

Next the function checks whether there exists an or-state which needs to be changed to a different or-state. If any pair of left and right children is found such that they are both or-states but have a different tag, all of the left state's members are added to `memberData`'s members to remove attribute, and all of the right state's members are added to `memberData`'s members to add attribute. Recall that information about which members are associated with which tag is stored directly in the trees that this function traverses, with the tags themselves, so this information can be retrieved efficiently during the traversal. The list of members to add can be found by traversing only the current branch of `update`'s metadata, and the list of members to remove can be found by traversing only the current branch of `target`'s metadata. After gathering all the necessary information from `target`'s branch, `target`'s child state is replaced by a clone of `update`'s child state.

After this, the function checks for the addition of a new or-state — that is, for an or-state that is a child of the current top-level tag in the `update` but

not in the target. If one is found, a copy of the child state is pushed onto the current level of target's metadata tree, and all of its members are added to memberData's record of members to add.

At the end of this process — or earlier when it can be established that there is no reason to explore the current branch further — the memberData object is returned.

Because of the in-place modifications, replacing small portions of target metadata tree with copies of the corresponding portions of the update metadata tree, by the time all of these recursive calls have completed execution, target's metadata has been completely updated. It now reflects the state structure that should exist after state change is done. Further, the memberData object that was returned has a full list of all of the members that should be added or removed.

The memberData object also contains a flag to indicate whether the root tag was matched. If no root tag was found in the target's metadata, each branch of update's metadata is cloned and pushed as a new branch onto target's metadata.

To continue state change, the runtime next performs a more efficient unique members check, using the information it collected in memberData. Next, the runtime verifies the unique tags property with the standard unique tags approach. If any members that are not associated with tags in update already appear in target, the runtime throws an error. If no such error is thrown, those members are added to target's metadata, at the top level, and are also added to the list of members to add.

Finally, all the items that appear in the list of members to remove are deleted not only from the metadata but from target, the PlaidObject. All items that appear in the list of members to add are added to the target object, and assigned the values they have in update, the PlaidState. This completes the process of state change.

6 Code Generator

Our code generator implements a source-to-source translation from Plaid to JavaScript. It was implemented in Plaid. It must perform many of the same translations that other source-to-source compilers handle. However, the code generator can also play a role in some of Plaid's state-based operators, beyond simply calling the runtime functions that implement them. By generating the code for some specialize and with calls at compile-time, the code generator can reduce the execution time of the resultant code.

6.1 Specialize

Specialize is a way of creating a copy of a state that contains a different value for a particular member. For instance, if the Plaid source contained the line val s2=s1{foo=1};, s2 would have all the same tags and members as s1, but whereas s1.foo may be 5, s2.foo will be 1.

Processing a specialization call in the code generator requires first creating a record of the new state being created. The code generator must then list all of s1's "with members", then search for a member name that matches the name of the attribute to be specialized. If a match is found, the code generator emits JavaScript code to alter the value of the target member.

Next, the compiler examines all other members of s1 for a match. If a match is found, the code generator throws an error. Otherwise, if the name of the member to be added does not name a member already present in s1, the compiler emits the code to add that member to the JavaScript PlaidState. The compiler then updates its record of the new state's members, adding the new member to the tree.

This process ensures that a specialization is never mistakenly carried out, that the code for modifying the new state's member is always emitted if the specialization can occur, and that the compiler's record of the state's current members remains up to date.

The process described above is the process for the case where the state being specialized is a state about which the code generator has full information. This usually occurs if the state is defined in the file that is currently being compiled. It has the advantage of throwing any errors during compilation rather than delaying feedback until execution. It also reduces the amount of computation (unique member checking and method copying) that needs to be done during execution, thus improving the compiled code's run time.

The code generator must also emit code for the case when it does not have complete information about the state being specialized. When this case arises, the code generator retrieves the name of the JavaScript variable in which s1 is stored. It then produces a call to the runtime's specialize function, calling it on s1 and passing in the member name and member value with which the new state should be modified.

6.2 With

With is a way of creating a new state composed of two other states. The new state has all the tags and members of both component states, both of which are in the top level of the hierarchy of the new state's tree. For instance, the Plaid code `state Car = DrivingStatus with CleanStatus` would create a state `Car` that has all the tags and members of both `DrivingStatus` and `Car`.

Listings 5 and 6 show some ways in which `with` can be used. The statements in Listing 5 would appear in the state declaration portion of a program, and would store the new state being created. The statements in Listing 6 would appear in the body of the program, and would create instances of the new states.

To emit JavaScript code for a use of `with`, the code generator first checks whether any member or any tag appears in both of the states to be composed. If there is any overlap, the code generator throws an error. If there is no error, the code generator creates a record for the new state being created and associates all the new state's members with that record. It then emits code to add all the members of `S1` to a copy of `S1`, which will represent the state in the JavaScript code.

The above process is appropriate when the code generator has full information about both of the states involved. When it does not, the code generator must call the runtime `with` function of `PlaidState` objects. In this case, the code generator retrieves the names of the JavaScript variables representing the two states, and emits the code to call `with` on one of them, with the other as an argument.

It is also possible for `with` to be used when either or both of the states are simply lists of member declarations, as in the object `eObject` displayed in 6. In this case, the code generator calls the runtime method `withMember` to add each member declared in the list. It is also possible for `with`s to be chained, as in the code: `val cObject = new A with B with C`. In this case, the compiler would first generate code for the rightmost pair of states (`B` and `C`), then use the result as the argument for the next `with` (`A`), and so on. Thus, the only types of items that the code generator must combine using with are pre-defined states, lists of declarations, and the results of other calls to `with`.

6.3 State Change

The code generator does not perform any part of the computation for state change, but rather produces the code to call the runtime's state change methods, distinguishing between the situations in which each state change method is appropriate. The code generator first processes the object on which state change will be called. Next, if the target state is a predefined state, it emits a call to the basic `stateChange` method. In the case that the target state is a list of declarations, the code generator emits code to repeatedly call `stateChangeMember`, adding the members to the object in the order in which they are defined. This is the extent of the code generator's involvement in state change.

7 Design Issues

We highlight a few of the most important design questions we faced in the development of our compiler and runtime system. We also consider alternative designs and explore the preferred solutions we implemented.

7.1 How to Store the Metadata

The central focus of our work is to produce a new, efficient runtime representation for objects with state. We have already discussed at length the reasons for using a metadata object to accomplish this. Next, we turn to the design of the metadata object itself.

States that are connected to their parents with `case of` relationships are easily visualized as part of a tree that allows one child per parent. While states that are composed from other states are not as obviously part of a tree structure, the same member rules apply as in the `case of` situation, if the composing states are treated as the children of the composed state. For instance, just as a `File` in the `OpenFile` state should be able to use all the members of `OpenFile`, a `Table = Surface with Legs` should be able to use all the members of the `Surface` and `Legs` states. This suggests the representation of state data in tree form, where each branch from a node `A` represents another state that is either a `case of` state `A` or a state used to compose state `A`. In this scheme, every state begins as a tree with only a root, and all composing states added as children of the root.

With this tree structure in mind, and rules in place for linking states and substates, the next question to address is what information to store in each node. This is answered simply by examining all the information necessary for state change:

1. The state's tag

```
state A { val a = 1; method y() { 1 } }
state B { val b = 2; method z() { 2 } }

state C = A with B // state with state
state D = A with { method f() {6} } //state with member
state E = { method g() {8} } with {val seven = 7; } // member with member
```

Listing 5: Several examples of the use of `with` to compose states, in state declarations.

```
val cObject = new A with B; //state with state
val dObject = new A with {method f() {6}}; //state with member
val eObject = new {method g() {8}} with {val seven = 7;}; //member with member
```

Listing 6: Several examples of the use of `with` to compose states, in program body.

2. The members declared within the state
3. Whether the state is a component of the parent state, or a `case of` the parent state

For each state in an object's state structure, there exists a node in the metadata that contains this information.

Although the metadata would have been most intuitively represented with a tree data structure, our implementation uses an equivalent deeply nested array of arrays to represent the tree. A Plaid programmer never needs to interact with the metadata representation, so ease of user programming was not relevant to the choice of representation. Further, using an array of arrays means being able to change the tree structure with simple array manipulations, and accessing items in an array, rather than attributes of an object. Although all JavaScript objects can be understood as associative arrays — with the result that using arrays instead of an object model should not change performance — microbenchmarks reveal that some JavaScript engines perform better on array access than on associative array access [14]. Since the metadata node class would have a fixed set of three members (corresponding to the three pieces of information stored with each node), simple array indexes can be used to identify the members.

```
method main() {
   var file = new OpenFile;
}
```

Listing 7: Plaid code for creating a file state with a shallow metadata tree. Assumes the `FileObject` state and its substates have already been defined.

```
[['', [], 'with'],
  [['File', ['filename'], 'with'],
    [['OpenFile', ['close', 'read'], '']]]]
```

Listing 8: The metadata for the `File` object created in 7. The root state is `File`, which has one member, `filename`. Its child `OpenFile` is a `case of File`, and has members `close` and `read`.

```
method main() {
   var car = new Car;
   car<-Braking;
   car<-TurningLeft;
   car<-Clean;
}
```

Listing 9: Plaid code for a car state with a deep metadata tree. Assumes the `Car` state and its substates have already been defined.

As an example, Listing 7 creates a simple `File` object with a shallow metadata tree, shown in Listing 8. A pictorial representation of the tree appears in Figure 9. Listing 9 creates a complex `Car` object with a deeper metadata tree, shown in Listing 10. A pictorial representation of the tree appears in Figure 6.

7.2 When to Create Metadata and Add Members

The runtime includes functions that can carry out all permissible forms of state composition and mutation. These functions can create and modify the metadata, and add state members. This permits a compilation

```
[['', [], 'with'],
   [['Car', [], 'with'],
      [['CleanStatus', [], 'with'],
         [['Clean', ['getDirty'], '']]],
      [['DrivingStatus', [], 'with'],
         [['BrakingStatus', [], 'with'],
            [['Braking', ['stopBraking'], '']]],
         [['DirectionStatus', [], 'with'],
            [['TurningLeft', ['turnRight', 'turnStraight'], '']]]]]]
```

Listing 10: The metadata for the `Car` object created in Listing 9.

scheme in which any state built up from other states is composed at run time. Although this significantly simplifies the process of code generation, it is important to keep in mind that very complex states can be constructed in the state declaration portion of a program, using `with` and `specialize`. Thus at runtime, the execution of `main` may be delayed until the completion of many `with` and `specialize` calls. This means waiting until the runtime functions create and combine many metadata trees; until they identify the members of component states; until they copy those members to the new composite states.

We chose a route that mitigates early state composition costs. In our implementation, the code generator uses a `Tree` object to accumulate metadata. The Plaid tree is first created when a new state `A` is declared using a list of declarations. A function converts this `Tree` object into the JavaScript array of arrays that is used to construct a PlaidState. After this point, if a new state `B` is declared using state `A`, the code generator creates a new Plaid `Tree` for `B`. The data stored in `A`'s `Tree` can be used to identify all of `A`'s members that must be copied to `B`, and the code generator emits the code to perform each copy, rather than relying on the runtime to traverse the metadata and identify all members to copy. Having `A`'s metadata available also means that `B`'s Plaid `Tree` can contain all the information relevant to constructing the JavaScript array of arrays for `B`. Thus, rather than emitting a line like:

```
var plaidNewState_C =
plaidNewState_B.with(plaidNewState_A);
```

which requires a fair amount of runtime computation to both create the tree and identify the appropriate members to add, the code generator can emit the code in Listing 11, which creates the tree and adds the members. The work of the `with` computation has essentially been shifted from the runtime to the code generator, a clearly preferable balance of work, given that a program will be compiled once and may be run many times.

There are limitations, however, on how much of the state-related computation can be shifted to the code generator. First, the code generator emits calls to the runtime for state compositions that take place during program execution. The Plaid `Trees` are only used for declarations of states that occur outside of executable code, in the state declaration phase of the program being compiled. Second, in the current implementation, the compile-time approach cannot be used if one of the component states is defined in another source file or a library. In a more advanced compiler, it might be desirable to associate some information about declared states with any compiled program, in a Plaid-readable format. This information could be used by the code generator to generate full Plaid `Tree` objects even for states composed using externally declared states.

7.3 Overrides

Overriding behavior has been briefly discussed so far, but supporting this feature does require special steps that store some members even if they are not currently callable.

Plaid allows a child state to override a parent's value for a field or method, if the child is a `case of` the parent. The runtime must be able to determine which of these values to assign to the PlaidObject's member of that name. It also means that using the simple state change scheme described above, the runtime might remove a member entirely even though the member should still exist, but must store the value provided by another state.

One question is how to keep track of a single object's different substates' values for the same member. Substate information must be stored with the object, in case the value is specific to a particular in-

```
var plaidNewState_C = new PlaidState(); // create the new JavaScript object to represent state C
plaidNewState_C['a'] = plaidNewState_A['a']; // add all of A's members to the new JavaScript object
plaidNewState_C['b'] = plaidNewState_B['b'];} // add all of B's members to the new JavaScript object
// add the appropriate metadata tree to the new JavaScript object
plaidNewState_C['tree'] = [['', [], 'with'],
                           [['C', [], 'with'],
                             [['B', ['b'], 'with']],
                             [['A', ['a'], 'with']]]];
```

Listing 11: The JavaScript code that would be emitted by the code generator to complete the `with` operation that composes States `A` and `B` to form a new state `C`. State `A` has member `a`, and State `B` has member `b`.

stance. Our usual approach would simply overwrite old values. Let state `B` be a case of `A`, and let `A` define `x` to be 'a', and `B` define it to be 'b.' If we start with a PlaidObject `plaidObjA` created by calling instantiate on `A`, `plaidObjA['x']` will be 'a.' If we then transition the object to state `B`, `plaidObjA['x']` will be 'b,' and there will no longer be any record of what `x` should be if the object is not in state `B`. In fact, any state change that would remove state `B` without adding a new value for `x` would remove `plaidObjA['x']` entirely.

The first step is to store all values for an overridden member during code generation. If a state `B` is declared as a `case of` another state `A`, the tree of state `A` is checked for any overlapping members. If an overlapping member `x` is found, the code generator checks whether `A`'s tree contains a member named `Plaid$A_x`. If it does, no change is made to `A`. If it does not, the member is added to the code generator's `A` metadata and then to the JavaScript PlaidState, by emitting code to add the member to the PlaidState's metadata, then emitting JavaScript code to set the object's value for `Plaid$A_x` equal to its value for `x`. The value of `B`'s `x` member is set in the usual way, but the member `Plaid$B_x` is also added to its metadata and the corresponding PlaidState.

Whenever an object is created or changed in the runtime, it is first checked for unique members. This process reveals any members whose names appear twice in the metadata. When a member does appear twice, the parent states of the instance at the greater depth are checked to ensure that it is one of those states that declares the member. When the runtime finds that a state `B` and its parent `A` both define a member `x` in PlaidObject `obj`, the runtime checks whether `obj['Plaid$A_x']` is defined. If it is not, it sets `obj['Plaid$A_x'] = obj['x']`, if `A` was already present in the state. If `A` was added in the current function, it sets `obj['Plaid$A_x']` to the value of `Plaid$A_x` in the source state. Next, it sets `obj['Plaid$B_x']` equal to the source state's value for `Plaid$B_x`, or to the source state's value for `x` if `Plaid$B_x` is not defined[1].

The process detailed above ensures that all possible values for an overridden member are available within the object. It does not ensure that the object's value for that member (e.g. `obj['x']`) is up to date. To achieve this, we process the unique member checks in order of increasing depth, tracking which states define `x`. If the last state to do so is `X`, `obj['x']` is set equal to `obj['Plaid$X_x']`.

The procedure detailed above ensures that the correct value of `x` is in force when the same member name appears twice in an object's metadata. However, it does not cover the case where there is only one remaining state that defines the member. For instance, consider the state `B` described above. If state change removed the substate `B` from an object in state `B` without adding a new overriding value, `A`'s value of `x` should be used as the object's value for `x`. Instead, `x` would be deleted from the object, and no unique members check would add a new value. To handle this case, the runtime also accumulates the list of all members whose names begin with `Plaid$`, removing items as they are handled in unique member checks. Any items that remain in the list after the verification of the unique members property represent backup values for members that should not be deleted but overwritten. The runtime checks whether

[1]This scenario, where `Plaid$B_x` is undefined is unlikely, because in order for `B` to be a `case of A` in the PlaidObject, it must also be a `case of A` in the source state. However, it is possible. Consider a case where a PlaidObject `obj` is created from `A {var x=5;}`, and then a new PlaidState `state` is created from `B {var x=6;}`. Because no duplicate member exists in either the PlaidObject or the PlaidState, neither would have backup values for `x`. However, the statement `obj<-state` must still result in values for both `obj['Plaid$A_x']` and `obj['Plaid$B_x']`.

those members are on the list of members to remove. If they are not, the value remains untouched. If they are on the list of members to eliminate, they are removed from that list, and the backup value replaces the old value. For instance, if `Plaid$A_x` remains in the list, the runtime checks whether `x` is on the list of members to remove. In the case discussed above, `x` would be on that list, so `obj['x']` is set to `obj['Plaid$A_x']`.

8 Results and Discussion

To evaluate the performance of this implementation, we tested on several benchmarks from the V8 benchmark suite [24]. The JavaScript programs that constitute this suite are used to tune V8, Chrome's JavaScript engine [23]. We translated the suite's Splay, Richards, and DeltaBlue benchmarks from JavaScript into Plaid to form a small suite of Plaid benchmarks.

We tested our implementation's performance on these Plaid benchmarks in two ways. We compiled the benchmarks to Java with the old Plaid implementation (discussed in Section 3.1.1), and to JavaScript with our new implementation, and compared execution times. We also compared the execution time of the JavaScript code produced by our Plaid-to-JavaScript compiler with the execution time of the original JavaScript versions of the benchmarks. All JavaScript code was run in SpiderMonkey [18].

First, we compare the old Java and new JavaScript implementations of Plaid. Even with the current non-optimized compilation strategy, the JavaScript compiler far surpasses the performance of the pre-existing Java compiler. Table 1 displays the results of running the two Plaid compilers on the Plaid Richards benchmark. Execution time for the Java code was 48 times slower than the execution time of the code produced by our JavaScript compiler (7,600ms vs. 157ms in Table1). As planned optimizations move forward, the performance of the JavaScript compiler should compare even more favorably.

Next, we compared the JavaScript code produced by our compiler to the original V8 JavaScript programs, the results of which appear in Tables 2, 3, and 4. These preliminary results reveal that the run time for code compiled from Plaid ranged from 2.6 times slower than the run time of the original JavaScript implementation (760ms vs. 1950ms for the Splay benchmark in Table 2) to 8.7 times slower than the run time of the original JavaScript implementation (19ms vs. 166ms for the Richards benchmark in Table 4).

Java and JavaScript Compilation		
	Plaid to Java	Plaid to JavaScript
Mean Run Time (100 μs)	76,000	1,570
Standard Deviation	2,000	40

Table 1: A comparison of the run time for the Richards benchmark, executed using the Plaid to Java compiler and the Plaid to JavaScript compiler. The values are in units of 100μs and represent the results of

Splay		
	JavaScript	Plaid to JavaScript
Mean Run Time (ms)	760	1950
Standard Deviation	30	70

Table 2: Execution time for the Splay benchmark. A comparison of the original JavaScript program and the same program translated to Plaid and compiled to JavaScript. The values are in ms and represent the results of 300 runs. The original JavaScript version is 2.6 times faster than our Plaid version.

DeltaBlue		
	JavaScript	Plaid to JavaScript
Mean Run Time (ms)	40	170
Standard Deviation	10	20

Table 3: Execution time for the DeltaBlue benchmark. A comparison of the original JavaScript program and the same program translated to Plaid and compiled to JavaScript. The values are in ms and represent the results of 300 runs. The original JavaScript version is 4.2 times faster than our Plaid version.

The size of the slowdowns of our Plaid versions over the original JavaScript versions appears to be related to the execution time of the benchmarks. Our Plaid implementation exhibits better relative performance on longer-running benchmarks. Although Splay is also the shortest source file, the DeltaBlue benchmark has substantially more lines of code than the Richards benchmark, so we conclude that run time rather than program length accounts for this effect.

Richards		
	JavaScript	Plaid to JavaScript
Mean Run Time (ms)	19	166
Standard Deviation	2	5

Table 4: Execution time for the Richards benchmark. A comparison of the original JavaScript program and the same program translated to Plaid and compiled to JavaScript. The values are in ms and represent the results of 300 runs. The original JavaScript version is 8.7 times faster than our Plaid version.

This leads us to believe that SpiderMonkey's frequent path optimizations may be the cause. With this in mind, we predict that most long-running programs with frequently repeated code may have slowdowns in approximately the range of Splay's 2.6 factor.

Plaid Richards and Plaid′ Richards		
	Plaid	Plaid′
Mean Run Time (ms)	290	2,460
Standard Deviation	70	80

Table 5: A comparison of run times for the Plaid and Plaid′ versions of the Richards benchmark, compiled with our JavaScript compiler. The Plaid′ version of the benchmark uses state change in the inner loop.

Although these results are encouraging given that they are produced with a still non-optimizing compiler, it is important to keep in mind that the Splay, DeltaBlue, and Richards benchmarks were developed for traditional languages, without Plaid's new features. Most importantly, although they make use of other aspects of Plaid's state model, these benchmarks do not use state change. However, we did discover that the Richards benchmark could benefit from explicit state transition. In fact, the Richards program uses de facto state change within an inner loop. To test the efficiency of our state change implementation, we wrote a Plaid′ version of the Richards benchmark. The Plaid′ version of Richards is a variation on our first Plaid Richards benchmark, a variation that leverages Plaid's novel state change functionality. Table 5 shows the execution time of Plaid′ Richards and Plaid Richards, compiled to JavaScript with our new implementation. We see an 8.4 times slowdown of Plaid′ Richards over Plaid Richards (290ms vs. 2,460ms in Table 5).

This slowdown is far above what we consider an

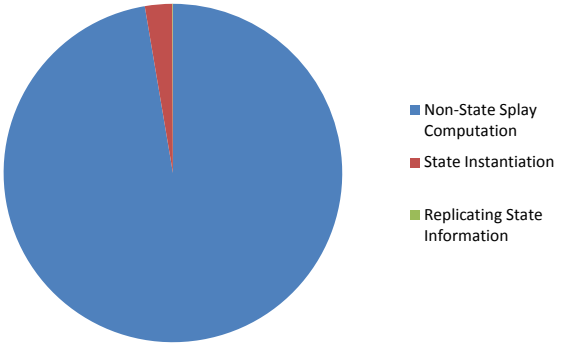


Figure 17: Splay: 2.7% of execution time is spent on state-related computation.

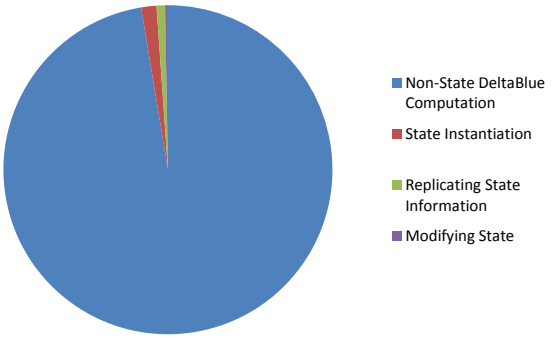


Figure 18: DeltaBlue: 2.6% of execution time is spent on state-related computation.

acceptable level. However, it is important to keep in mind that we developed our implementation to optimize for frequent member access, not frequent state change. In fact, we proposed that in most programs, member accesses will far outnumber state transitions. In the Plaid′ Richards benchmark, state transition operations are proportional to member accesses. We believe this is far from the common case. Nevertheless, programs like this are more naturally expressed in a language with abstract state support, and they should be efficient in Plaid. Our Plaid′ findings therefore inspire the state change optimization we propose, which we believe is the next crucial step in implementing fast state support.

To determine how effectively we minimized state-related computation, and to identify appropriate targets for optimization, we profiled all four Plaid language benchmarks. Casual examination of the Splay chart in Figure 17 and the DeltaBlue chart in Figure 18 reveals that state-related computation requires

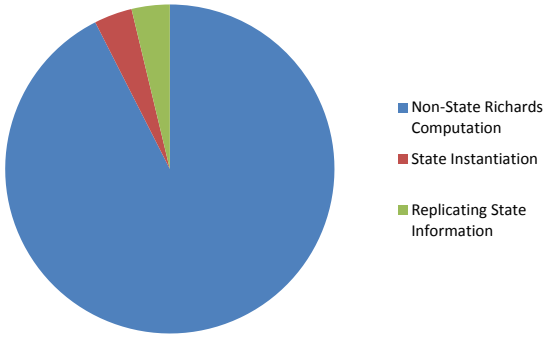


Figure 19: Richards: 7.5% of execution time is spent on state-related computation.

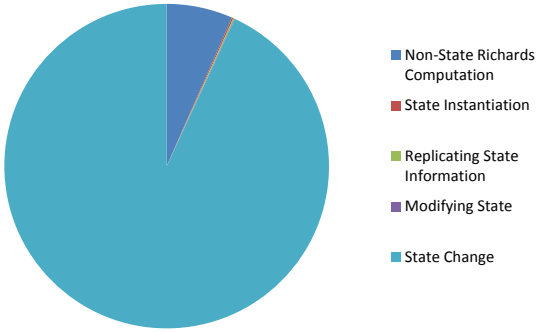


Figure 20: Plaid′ Richards: 93.4% of execution time is spent on state-related computation.

only a very small portion (less than %3) of the execution time of those benchmarks. As we see in Figure 19, the percentage is a little higher for Richards, but is still only 7.5%. These charts also indicate that state instantiation is the most computationally expensive portion of the runtime, accounting for 2.6% of Splay run time, 1.5% of DeltaBlue run time, and 3.7% of Richards run time. Crucially, the balance of state computation to non-state computation in these benchmarks suggests that most of the slowdown in our execution times is not a result of our state representation, but of our simple scheme for compiling ordinary code.

Next we turn to the breakdown of execution time for the Plaid′ version of the Richards benchmark, shown in Figure 20. For this benchmark, state-related computation constitutes 93.4% of the run time. State change alone accounts for 93.1% of the run time.

These results indicate that state change must be the primary target for future optimizations of our implementation. Because we prioritize member access over state change, our current implementation scheme has yet to address the challenge of making state change efficient. Undertaking that task is a critical next step.

Preliminary results show that we have succeeded in our task of supporting state without slowing member access. Although we see substantial slowdowns when we compile Plaid benchmarks to JavaScript and run them alongside the original JavaScript benchmarks, we find that only small portions of the additional execution time stem from features of our state representation.

9 Future Work

There remain many pressing questions on the topic of how to efficiently compile a language that supports state change. This research will go on to investigate further refinements of the JavaScript implementation. To start, we hope to implement several state-related optimizations. However, we also consider alternative implementations for languages without the flexible object model JavaScript offers.

9.1 Reducing Unique Member Checks

State instantiation is the creation of a PlaidObject from the information in a PlaidState object. It is one of the more computationally expensive procedures that the runtime must conduct. The process of instantiation as we laid it out before always checks the unique members property, even if the state being instantiated has been instantiated in the past without errors.

Because each new PlaidObject must have its own copy of all its state information, and its own members, to speed up member access at runtime, reducing object creation time was unlikely to be a fruitful target for optimizations. However, the time spent rechecking a single state for compliance with Plaid's semantics is redundant. To optimize state instantiation, the unique members check need only ever be performed once on any given state.

To accomplish this, we could store a flag with each JavaScript representation of a PlaidState, indicating whether or not it has been instantiated in the past. If it has been, and no error was thrown, the runtime can conclude that the state satisfies the unique mem-

bers check, and the check need not be repeated. The runtime will proceed with creating the appropriate PlaidObject. If the state has never been instantiated, the runtime will complete the unique members check, then use the flag to indicate that the state has been checked if the object satisfies the unique members property.

9.2 State Change Caching

Our implementation prioritizes fast member access by sacrificing fast state change, with its complex metadata processing. Given the relative frequency of state change and member access in most programs, it is appropriate speed up member access at the cost of slowing state change. However, it is still possible to change state fairly frequently, and it is easy to imagine programs in which there is one state transition for each member access, in which case there is an incentive to reduce the time spent on state change.

State change is computationally expensive largely because of the tree traversal process. If it were possible to identify the appropriate members to add and remove without the joint metadata traversal, there would be a substantial improvement in computation time of state change.

It is fairly common to use objects that will repeat transitions between the same pairs of states. Whether it is a stack model that shifts back and forth from empty to non-empty; a carousel that repeats the transitions from loading to running, running to unloading, and unloading to loading; or a pace model that repeatedly executes one of the six transitions between standing, walking, and running. This drives the need for storing the results of state change between a given state and object. If we choose to trade off space for execution time, it makes sense to store all the modifications that should be made for a particular transition, so that those modifications need not be recalculated when a specific state transition is enacted multiple times.

This rationale motivates a state change caching optimization, which proceeds by populating a set of hash tables in the runtime. For each PlaidObject on which a state transition is enacted, the runtime would add an entry in the transition cache for the state to which it is transitioned. The entries also have a direction. The information used for the transition `a<-b` is not the same information used for the transition `b<-a`.

When a new state transition occurs, that does not yet appear in the table, the runtime would execute the normal state change calculations described in 4. In the state change cache entry, the runtime records the members that were added and removed as a result of the state transition. This process adds very little to the run time of a new state change, since it simply requires storing two lists that the process already created. As a result, even in programs where most state transitions are new transitions, this modification should not significantly slow execution. Alternatively, rather than store the lists of members, the runtime could generate the code to transition the object, then cache that code. Again, because this code must be generated in any case, this should not add significantly to the execution time.

When any state transition occurs, the runtime would first check whether the transition already exists in the state change cache. If it does not, it would execute the steps detailed above. If, however, the state transition has occurred in the past, the cached entry would be retrieved. Rather than traverse the two trees again, the runtime would add the members that were added last time and remove the members that were removed last time. Or in the alternative scheme, it would call `eval` on the cached transition code. The result is that state change between the same two metadata trees in the same direction only requires metadata traversal once.

9.3 Hash Consing Metadata

The state representation that facilitates our implementation is built on metadata trees with very consistent structures. Because all objects in a given state have the same tag, the same relationship with the parent state, and often the same members, there is significant overlap between PlaidObjects' metadata trees. The current implementation does not take advantage of this repetition. In future, it should be possible to improve performance by hash consing the metadata, thus saving space and avoiding time consuming memory allocation operations.

9.4 Other Runtime State Representations

While adding and removing members from a single target language object is an option in JavaScript and some other potential compilation targets, it would be ideal to have an efficient implementation scheme for languages without prototype support. For this reason, additional future work should focus on efficient

state representation for class-based, more conventionally object-oriented languages like Java. With classes able to inherit members from only a single superclass, how can a Plaid state be created as a composition of two other states? How can states be allowed to transition, and their members with them? While slow solutions come readily to mind, efficient ones do not.

The creators of Plaid are currently developing a new Plaid to Java compiler inspired in part by our work, but the details of its new implementation scheme are still being determined. With a faster target language, it may well be possible to outperform our JavaScript implementation, even with a scheme that delays member access to some extent. Perhaps the solution will be something as simple as emulating the State design paradigm in the compiled code. Maybe the best approach will be keeping state variables and generating appropriate state checks. Or perhaps there is a way to build a tree of component states, but more efficiently than it has been done in the past. These approaches and many others will provide fertile ground for continued research.

10 Conclusion

Substantial work has gone into developing efficient compilation methods for dynamically typed languages. As languages like Python and Ruby meet with steadily greater success, the importance of such work is only growing.

It is in joining state change and dynamic state composition that Plaid introduces a new challenge for implementation, one that has not yet been addressed in the field. Languages with abstract state and composable traits have not yet been efficiently compiled. This work represents a first step towards that goal. Our implementation — even in its non-optimized form — produces code that is 48 times faster than code produced with the implementation scheme that past works on object reclassification suggest.

Our implementation offers a novel representation for state at runtime, a representation shaped by the need to facilitate efficient member access. Other approaches will have to be developed for non-prototype languages; but for languages that allow the addition and removal of members at runtime, the use of a flat member structure and metadata trees appears to be a viable starting point.

There remains significant room for improvement on our implementation. Crucially, we believe our state representation lends itself well to state-related optimizations. The maintenance of a metadata item, a central store of all an object's state information, is key to the drive to reduce state change execution times. Further, we believe the optimizations we have already laid out could significantly improve the run time of code produced by our implementation.

Ultimately, abstract state is a useful language feature, a feature which may reduce programmers' errors, improve their awareness of objects' state spaces, and reduce the time spent on boilerplate state checks. An efficient implementation of abstract state will be an important step in establishing state-based languages' usefulness for practical purposes, and in making state abstraction a viable option for language users and designers.

References

[1] Gul Agha, Christopher R. Houck, and Rajendra Panwar. Distributed execution of actor programs. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 1–17, London, UK, 1992. Springer-Verlag.

[2] Jonathan Aldrich, Karl Naden, and Éric Tanter. Modular composition and state update in plaid. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 4:1–4:4, New York, NY, USA, 2010. ACM.

[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, Onward! '09, pages 1015–1022, New York, NY, USA, 2009. ACM.

[4] Davide Ancona, Christopher Anderson, Ferruccio Damiani, Sophia Drossopoulou, Paola Giannini, and Elena Zucca. A provenly correct translation of fickle into java. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[5] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'11, pages 2–26, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 217–226, New York, NY, USA, 2005. ACM.

[7] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM.

[8] Tal Cohen and Joseph (Yossi) Gil. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 57–66, New York, NY, USA, 2009. ACM.

[9] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *ECOOP '04*, pages 465–490. Springer, 2004.

[10] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, UK, 2001. Springer-Verlag.

[11] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388, 2006.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, 1987.

[14] Javascript microbenchmarks. http://jsperf.com/javascript-associative-vs-non-associative-arrays.

[15] Dennis G. Kafura and Manibrata Mukherji. The design and implementation of concurrent input/output facilities in act++ 2.0. Technical Report TR-92-46, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 1992.

[16] Alan C. Kay. The Early History of Smalltalk. *SIGPLAN Notices*, 28(3), 1993.

[17] Johan Lilius and Iván Porres Paltor. Formalising uml state machines for model checking. In *Proceedings of the 2nd international conference on The Unified Modeling Language: beyond the standard*, UML'99, pages 430–444, Berlin, Heidelberg, 1999. Springer-Verlag.

[18] Spidermonkey. https://developer.mozilla.org/en/SpiderMonkey.

[19] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, 1986.

[20] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM.

[21] Antero Taivalsaari. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming*, 6(3):25–32, 1993.

[22] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[23] V8 javascript engine. http://code.google.com/p/v8/.

[24] V8 benchmarks. http://v8.googlecode.com/svn/data/benchmarks/v7/.