First-Class State Change in Plaid

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

October 2011
CMU-ISR-11-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Objects model the world, and state is fundamental to a faithful modeling. Engineers use state machines to understand and reason about state transitions, but programming languages provide little support for building software based on state abstractions. We propose Plaid, a language in which objects are modeled not just in terms of classes, but in terms of changing abstract states. Each state may have its own representation, as well as methods that may transition the object into a new state. A formal model precisely defines the semantics of core Plaid constructs such as state transition and trait-like state composition. We evaluate Plaid through a series of examples taken from the Plaid compiler and the standard libraries of Smalltalk and Java. These examples show how Plaid can more closely model state-based designs, enhancing understandability, enhancing dynamic error checking, and providing reuse benefits.

We thank Nels Beckman and Robert Bocchino for their work on the semantics of Plaid; Manuel Mohr, Mark Hahnenberg, Aparup Banerjee, Matthew Rodriguez, and Fuyao Zhao for their work on the Plaid compiler; and the PLAID group for their helpful feedback and suggestions. This research was supported by DARPA grant #HR00110710019. Sven Stork is supported by the Portuguese Research Agency FCT, through a scholarship (SFRH/BD/33522/2008). Joshua Sunshine was supported by the Department of Defense (DoD) through the National Defense Science and Engineering Graduate Fellowship (NDSEG) Program.

Keywords: typestate, state-chart, plaid

1 Introduction

Object-oriented programming provides a rich environment for modeling real-world and conceptual objects within the computer. Fields capture attributes of objects, methods capture their behavior, and subtyping captures specialization relationships among objects. A key element missing from object-oriented programming languages, however, is abstract states and conceptual state change. State change is pervasive in the natural world; as a dramatic example, consider the state transition from egg, to caterpillar, to pupae, to butterfly. Modeling systems with abstract states and transitions between them is also common in many engineering disciplines.

In computer science, state machines are important modeling concepts in the UML. Abstract states are also a critical, though often implicit, part of many library APIs—and any client using that API must be aware of those states to use the API correctly. For example, a file may be in the open or closed state. In the open state, one may read or write to a file, or one may close it, which causes a *state transition* to the closed state. In the closed state, the only permitted operation is to (re-)open the file.

Files provide a simple example of abstract states, but there are many more. Streams may be open or closed, iterators may have elements available or not, collections may be empty or not, and even lowly exceptions can have their cause set, or not[1]. State spaces may be complex: In `ResultSet` from the Java JDBC library, we found 33 unique states dealing with different combinations of openness, direction, random access, insertions, etc [7]. States are also common: a recent study of protocols in Java suggests that almost three times as many types define protocols as define type parameters [3]. They also cause significant pain: for instance, in a study of problems developers experienced when using the ASP.NET framework, 3/4 of the issues identified involved temporal constraints such as the state of the framework in various callback functions [16]. All this raises a natural question: why not support abstract states in programming languages?

We previously proposed *Typestate-Oriented Programming* as a new programming paradigm in which programs are made up of dynamically created objects, each object has a typestate that is changeable, and each typestate has an interface, representation, and behavior [2]. The term typestate refers to a static abstract state checking methodology proposed by Strom and Yemini [25]; this paper focuses on a dynamically-typed setting, and so we will use the terms *(abstract) state* and *protocol* in place of typestate to avoid confusion.

A programming language with abstract states can have many benefits. First, in the case of stateful abstractions, the code will more clearly reflect the intended design. This in turn will make state constraints more salient to developers who need to be aware of them. If state constraints are implicitly enforced by the object model, there is no need to code up explicit checks; thus code implementing states can be more concise. Explicit state models raise the level of error messages; instead of (perhaps) silently corrupting a data structure when an inappropriate method is called, the runtime can signal an error that that method is unavailable in the current state. Finally, explicit modeling of states also exposes new concepts for widespread reuse; candidates may include open/closed resources or positioning (beginning, middle, end) of streams.

Contribution. The contribution of this paper is the concrete design and evaluation of Plaid, an object-oriented programming language that incorporates first-class state change as well as trait-like state composition. Plaid has been implemented, and has proven effective for writing a diverse set of small and medium-sized (up to 10kLOC) programs, including a self-hosted compiler. For the purposes of this paper, Plaid is dynamically typed, though there are plans to add a gradual type system following recent work on gradual typestates [28].

The most interesting aspects of Plaid's design come from the intersection of state change with support for a trait-like model of composition [12]. Central goals of the language design include supporting the

[1] E.g. in Java, the cause of an exception can only be set once.

primary state modeling constructs from statecharts [14], as well as flexible code reuse. Our design includes a hierarchical state space, so that the `open` state of a stream can be refined into `within` and `eof` substates indicating whether there is data left to be processed. Handling real designs in a modular way requires support for multi-dimensional state spaces, as in and-states from [14]; an example is a separate dimension of a stream's state indicating whether the stream has been `marked` with a location or not. Modularity further requires reasoning about dimensions separately; for example, the `mark()` method should affect the `marked` state dimension but it should not affect whether the stream is at `eof`. Dimensions also delineate natural points of reuse; we would like to specify them separately and combine them using a trait-like composition operator.

We position Plaid relative to earlier work in the next section. Plaid's design is described by example in section 3. This section validates our design, using a number of carefully chosen examples to concretely illustrate how Plaid provides the potential benefits described above. We also discuss our prototype implementation of Plaid, targeting the JVM.

In order to be precise about Plaid's semantics, Section 4 provides a formal model that includes the semantics of all of Plaid's major features. Section 5 describes how the surface Plaid language is elaborated into the core formal model. The paper concludes with a discussion of ongoing work on Plaid, together with an argument that the concrete benefits validated by example lead to higher-level benefits in software development and evolution.

2 Background and Related Work

Plaid's state constructs are inspired and guided by state modeling approaches such as Harel's statecharts [14]. Other modeling approaches include Pernici's Objects with Roles Model [20], which models objects using a set of roles, each of which can be in one of several abstract states.

Strom and Yemini proposed typestate as a compiler-checkable abstraction of the states of a data structure [25]. The Fugue system was the first to integrate typestates with an object-oriented programming language [10]. Bierhoff et al. later observed that the complexity of protocols such as the one defined by the JDBC ResultSet interface requires rich state modeling constructs like those proposed by Harel [7]. This paper considers a dynamically-typed setting, so we do not discuss static checkers further.

State-dependent behavior can be encoded using the State design pattern [13]. However, this pattern is less direct than the language support we propose, and it does not help with ensuring that a client only uses operations that are available in the current state.

Dynamic languages such as Self [27] provide the ability to add and remove methods, as supported by Plaid's state change operator. Changing a delegation slot in Self can also be used to simulate state change, as can the become method in Smalltalk [18]. We believe that Plaid's more structured and more declarative constructs for state modeling have advantages in terms of error checking, succinctness, and clear expression of design compared to these encodings. Plaid's prototype-based object model is also inspired by Self's.

Prior State-Based Languages. The Actor model [15] treats states in a first-class way, using the current state of an actor to define the response to messages in a concurrent setting.

Taivalsaari extended class-based languages with explicit definitions of logical states (modes), each with its own set of operations and corresponding implementations [26]. Plaid's object model differs in providing explicit state transitions (rather than implicit ones determined by fields) and in allowing different fields in different states.

The Ferret language [8] provides multiple classification, in which objects can be classified in one of several states in each of multiple dimensions. Ferret attaches dimensions to classes, not other states, so dimensions cannot come and go with state changes (unlike in Plaid and Statecharts).

A number of CAD tools such as iLogic Rhapsody or IBM/Rational Rose Real-Time support a programming model based even more directly on Statecharts [14]; such models benefit from many rich state modeling features but lack the dynamism of object-oriented systems. Recently Sterkin proposed embedding the principal features of Statecharts as a library within Groovy, providing a smoother integration with objects [23]. Our approach focuses on adding states to object-oriented languages, rather than libraries.

Other researchers have explored adding a class change primitive to statically-typed languages [11, 4, 6]. These systems, however, do not support the richness of state models (e.g. and-states) as provided in Statecharts and in Plaid.

Schaerli et al. proposed traits [12] as a composition mechanism that avoids some of the semantic ambiguities of multiple inheritance. Schaerli's traits did not have fields, but Plaid follows prior designs [5] to add them. Like some other recent work [22, 9], Plaid does not have the flattening property, in which the composition structure of traits is compiled away and does not affect the semantics of the resulting program. We lose the simplicity of flattening but gain the ability to model structured state spaces more directly, as described below.

An initial sketch of the Plaid language design was presented earlier [2] as an instance of the Typestate-Oriented Programming paradigm. While we recap the motivation and concept of the language from this earlier work, that paper described an unimplemented language, and neither defined the language semantics nor investigated the modeling of complex state spaces, which are the key contributions of this paper. In an earlier 4-page workshop paper, we explored the need for a modular state change operator that affects only one dimension of state change at a time [1]; this paper gives the semantics for a concrete solution to that problem. Other recent work has begun to explore a gradual, permission-based type system for Plaid [28].

3 Language

In this section we will introduce Plaid by example. These examples serve the dual purpose of explaining the language and validating the concrete benefits of Plaid.

3.1 Basics of State Change

Object protocols are rules dictating the ordering of method calls on objects. The concrete state of an object with a protocol can be abstracted into a finite number of abstract states and the object transitions dynamically between these abstract states. Therefore, clients must be aware of the abstract states in order to use the object correctly.

Most programming languages provide no direct support for protocols. Instead, protocols are encoded in the language using some combination of the state design pattern [13], conditional tests on fields, and other indirect mechanisms. In Plaid, protocols are supported directly with states, which are like classes in Java, with the crucial distinction that an object's state changes as the object evolves.

Consider the state space of files, the canonical protocol example [2], shown in Figure 1. Some files are open and some are closed. We close an open file by calling the `close` method and open a closed file by calling the `open` method. One cannot open an open file so the open file state does not include the `open` method. Similarly, one cannot read a closed file so the closed file state does not include the `read` method.

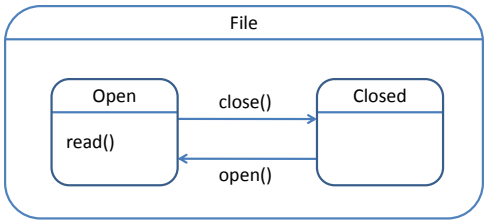

Figure 1: State space of File.

```
state File {
  val filename;
}
state OpenFile case of File = {
  val filePtr;
  method read() { ... }
  method close() { this <- ClosedFile; }
}
state ClosedFile case of File {
  method open() { this <- OpenFile; }
}
```

Listing 1: File states in Plaid

The state space of files can be encoded cleanly in Plaid as shown in Listing 1. The `state` keyword is used to define a state. The `File` state contains the fields and methods that are common between open and closed files. In this case, only the filename is shared. Fields are declared with the `val` keyword.

`OpenFile` and `ClosedFile` define the methods and fields that are specific to open and closed states. Both are substates of `File`. Specialization is declared with the `case of` keyword. In addition, `case of` implies orthogonality: files can either be open or closed, not both. Methods are defined with the `method` keyword. Open files have a `read` method, a file pointer field which is presumably used by the `read` method to read the file, and a `close` method. Closed files have the `open` method.

The `open` and `close` method bodies contain the most novel bit of syntax. An object referred to by a variable `x` can be changed to state `S` by writing `x <- S`. In the `open` method we transition the receiver, referred to as in Java by the keyword `this`, to the open state by writing `this <- OpenFile`.

An example file client is shown in Listing 2. The `readClosedFile` method takes a file as an argument, opens it, reads from it, closes it, and returns the value read from the file. All of the method calls are valid if a closed file is passed to the method. If an open file is passed instead the `open` method call will fail. The

```
method readClosedFile(f) {
  f.open();
  val x = f.read();
  f.close();
  x; //return
}
```

Listing 2: File client in Plaid

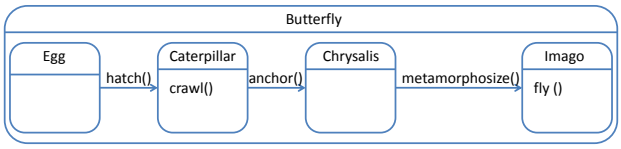

Figure 2: Buttefly life-cycle.

library writers do not need to write any special error handling code to handle this condition like they would in Java. This has the concrete benefit that Plaid code for the equivalent design is smaller.

In most programming languages, fields of an object are often null in certain abstract states. For example, Java files might contain a null `filePtr` when the file is closed. Null pointers are a frequent cause of runtime errors and their cause can be difficult to diagnose. For these reasons, Tony Hoare recently called null pointers a "billion dollar mistake," and we have not repeated this mistake in Plaid.

Plaid objects are always consistent: in other languages a programmer might forget to check the state before performing an operation and perform the operation on an object in the wrong state. Similarly, the operation might fail, but with a less specific error message. For example, if a client calls the `read` method, implemented in Java without error handling, on a closed file, Java might throw a `NullPointerException` for a null dereference of `filePtr`.

3.2 State Transtions

The file state space is a complete directed graph, every pair of states is connected in both directions by an edge. Other kinds of objects have incomplete state spaces. Consider the life-cycle of a butterfly, which is illustrated by the state-space in Figure 2. A butterfly egg hatches to a caterpillar, but it cannot 'un-hatch'. Similarly, a butterfly never transitions directly from a caterpillar to an imago, it always transforms to a chrysalis first.

To preserve the integrity of incomplete protocols, only the method receiver (`this`), can be the target of a state change operation. If Plaid did not have this restriction it would be trivial for programmers to inadvertently violate a protocol. Consider: `val x = new Egg; x<-Caterpillar; x<-Egg`. This illegal Plaid code violates the protocol by restoring a caterpillar to an egg. Instead, in legal Plaid code, methods defined in the butterfly states perform all of the state transitions.

3.3 Dimensions of State Change

Many objects in the real world are not as simple as files or butterflies. Some objects are composed of multiple states, particularly when objects are built up from reusable components. These components may change their state independently, or orthogonally. For example, cars have both gears and brakes and when the car shifts gears it has no effect on the brakes. States that change independently are in different *dimensions*. State dimensions in programming languages were introduced in [7].

More concretely, let us say a stream is in state `unmarked` in dimension *markable*, and state `within` in dimension *position*. If the object changes to state `marked`, also in dimension *markable*, it will lose all of the fields and methods defined in `unmarked` (such as `mark`), gain those in `marked` (such as `reset`), and keep those in `within` (such as `read`).

The full power of Plaid comes when component states are themselves composed of multiple states. In such a setting the component states are gained and lost along with their parents. Many of this kind of deep hierarchies exist in the wild [3]. For example, in the Java Database Connectivity library, the ResultSet

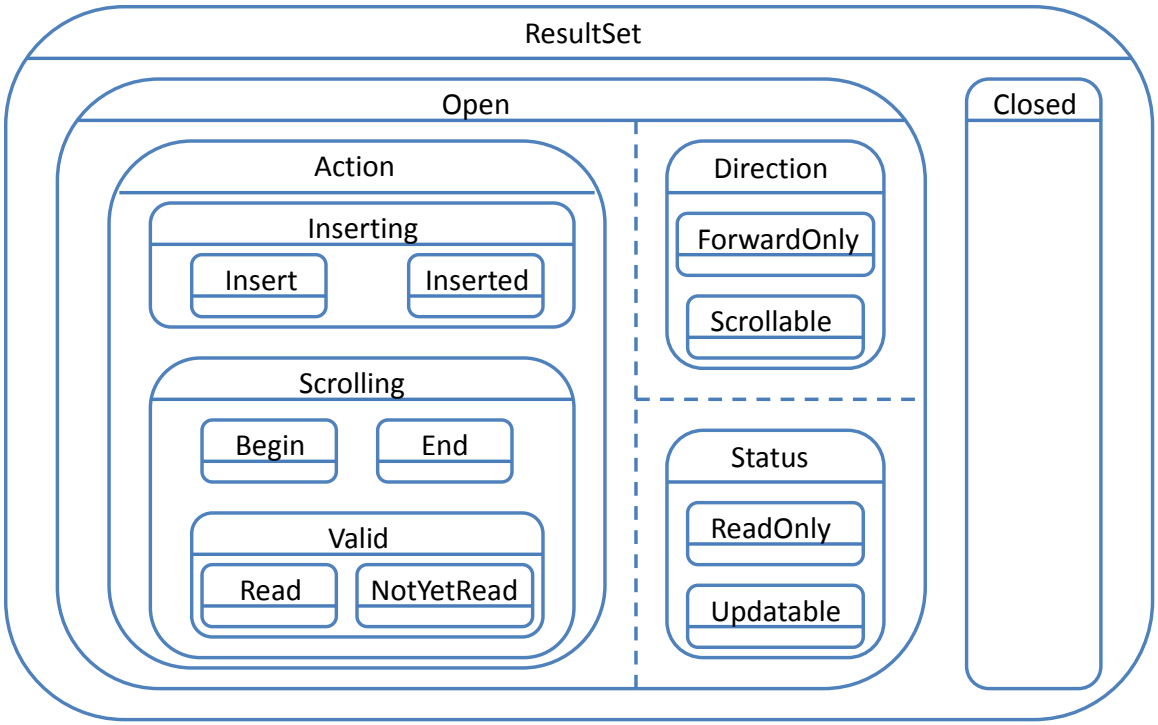

Figure 3: ResultSet state-chart.

interface is composed from a combination of 33 states, four levels of nesting, and eight dimensions. A slightly simplified schematic of the state space is shown in Figure 3.

The features of the language just described correspond directly to the 'hierarchical-states', 'and-states' and 'or-states' proposed by Harel in his seminal state-chart paper [14]. Hierarchical-states are states that are composed of other states. And-states are states that both must be present in an object—separate dimensions that are modeled using `with` composition in Plaid. Finally, or-states are states in the same dimension, and therefore only one can be present in an object—a state that is a `case of` another state. These features are the fundamental building blocks of the Harel state chart formalism (which forms the basis for UML state diagrams), and are naturally encoded in Plaid exactly in the manner we just described.

In the `ResultSet` diagram (Figure 3), or-states are separated by white space. For example, `Open` and `Closed` are states in one dimension, `ForwardOnly` and `Scrollable` are in another. Hierarchical-states are indicated by nesting of the state rectangles. For example, `Scrolling` is a child of `Open` and `Begin` of `Scrolling`. The names of states with children, like `Open` appear outside the state-rectangle, and the names of simple states without children, like `Inserted`, appear inside the state-rectangle. Finally, and-states are separated into *orthogonal regions* by dotted lines, so `Direction` and `Status` are and-states.

There is a natural one-to-one correspondence between the state rectangles in the diagram and the state declarations in Plaid code. A subset of the declarations for `ResultSet` states are shown in Listing 3. The or-states are all declared to be cases of their dimensions. For example, `ForwardOnly` and `Scrollable` are cases of the `Direction` dimension. The dimensions are themselves states in which case their or-states will inherit all of the dimension's fields and members. Sometimes, however the state is a *pure dimension* and does not contain members. In this case the state only serves to ensure that or-states do not appear together.

```
state Open case of ResultSet =
  Direction with Status with Action;
state Direction;
state ForwardOnly case of Direction;
state Scrollable case of Direction;
state Status;
state ReadOnly case of Status;
state Updatable case of Status;
state Action;
state Scrolling case of Action;
state Inserting case of Action;
state Insert case of Inserting;
state Inserted case of Inserting;
...

val myResultSet = new Open @ ForwardOnly
  with Updatable with Insert;
```

Listing 3: ResultSet state declarations and instantiation

The `Open` state contains several nested and-states. Therefore, `Open` is declared as a composition of the three nested dimensions using the `with` operator. This encoding captures the invariant that any object in the `Open` state is also in the `Direction`, `Status`, and `Action` states. Often `ResultSet` objects will be instantiated with children of the three dimensions `Direction`, `Status`, and `Action`. For example, at the end of Figure 3, `myResultSet` is assigned to an open object in the ForwardOnly, Updatable and Insert states. This object will contain the methods and fields from `Insert`, `Inserting`, `Action`, `Updatable`, `Status`, `Forwardonly`, `Direction`, `Open` and `ResultSet`. If we were to change the state of `myResultSet` to `Inserted` by calling a method that does so for us the object would have all of the same states, with the exception that `Insert` will be replaced with `Inserted`. This is because `Insert` and `Inserted` are or-states from the same dimension. When we close the object, we lose not only the `Open` state but all of the states nested inside it. We are left only with `Closed` and `ResultSet`.

The `@` operator is syntactic sugar that allows an initializer to conveniently choose nested sub-states. The `myResultSet` initializer in Figure 3 is desugared to the following code:

```
var myResultSet = new Open;
myResultSet <- ForwardOnly with Updatable
  with Insert;
```

First, an `Open` object is created. Then the object is changed to substates of the three dimensions using the state change operator. Notice that the left side of the state-change operator is not `this` in the de-sugared code which violates the restriction discussed in Section 3.2. This is okay, because the restriction only applies to Plaid source which in this case uses the `@` operator.

In this example the reader can see that the Plaid code closely reflects the design embodied in the state chart. The stateful design is salient in the state declarations. Since the mapping between the code and the state chart is so clear, a programmer reading the state declarations can easily understand the relationship between the states. In fact, our group has built a tool to automatically extract a state chart from Plural[2], a typestate checker for annotated Java code, and although we have not built such a tool for Plaid, the language

[2]http://code.google.com/p/pluralism/

```
state Position {
  state notEndState = NotEnd;
  state endState = End;
  method setToEnd();
  method setToStart();
  val vector, minPos, maxPos;
  var currPos;
}

state NotEnd case of Position {
  method setToEnd() {
    this.currPos = this.maxPos;
    this <- this.endState;
  }
  method setToStart() {
    this.currPos = this.minPos;
  }
  method nextPosition() {
    this.currPos++;
    if (this.currPos >= this.maxPos) {
      this <- this.endState;
    }
  }
}

state End case of Position {
  method setToEnd() { /* no op */}
  method setToStart() {
    this.currPos = this.minPos;
    this <- this.notEndState;
  }
}
```

Listing 4: Position code.

design clearly enables it. A second potential benefit is that code for each state can be given separately in the appropriate state declaration, potentially permitting more fine-grained reuse across multiple implementations of the ResultSet interface.

3.4 State members

As we mentioned in the introduction, Plaid combines state change with support for a trait-like model of composition [12]. We now illustrate a particularly novel feature of Plaid, namely, state members. States can have other states as members, and these state members can be customized upon composition. This allows for consistent state update, in presence of composite states.

We illustrate state members and their benefits through a Plaid version of a ReadWriteStream adapted from [12], which is in turn adapted from the Smalltalk standard library. The Plaid components mirror the trait components, except in our version the methods of a single trait are sometimes divided across multiple states.

```
1  state Reader { }
2
3  state Reading case of Reader {
4    method read() {
5      val ret = this.vector.get(this.currPos);
6      this.nextPosition();
7    }
8  }
9
10 state ReadEnd case of Reader { }
11
12 state ReadStream = Position {
13   val notEndState = Reading with NotEnd;
14   val endState = ReadEnd with End;
15 } with Reader;
```

Listing 5: ReadStream code.

The `Position` state represents the position of the pointer into a stream or collection. It has a very limited interface which therefore makes it easy to reuse throughout an input-output and collection library. The code for `Position` is shown in Listing 4. `Position` declares two abstract methods for setting the position, a reference to the underlying collection (`vector`), constant fields for minimum and maximum position, and a variable field for the current position[3].

Interestingly, `Position` contains two state members, one for the end-state and one for the not-end-state. The state members are initialized to `NotEnd` and `End`, also defined in Listing 4. These states are sub-states of `Position`, as specified by the `case of` declarations. They implement the abstract methods of `Position`. In addition, `NotEnd` has an additional method `nextPosition`, reflecting the fact that in that state, the position can be advanced. This method increments the current position, tests if the current position is at or past the maximum position, and transitions the receiver to the end state if the position is at the end. Similarly, `setToStart` in `End` transitions the receiver back to the not-end state.

The crucial part in this example is that the state transitions do not explicitly reference a specific target state, but rather reference the state members of `Position`. For instance, `nextPosition` in `NotEnd` transitions `this` to `this.endState`, not `End`. This allows for consistent and flexible reuse, composition, and extension of states, as illustrated hereafter.

Consider the code for a `ReadStream`, as shown in Listing 5. The `ReadStream` definition includes a pure dimension, `Reader`. This dimension has two children `Reading` and `ReadEnd`, which correspond to the `ReadStream` in the not-end-state and the end-state, respectively. In the not-end-state, the `ReadStream` can read, and therefore `Reading` defines the `read` method. This method reads from the underlying collection at the current position and advances the position.

The `ReadStream` is composed from the two dimensions `Position` and `Reader`. `ReadStream` specializes `NotEnd` by overriding the two state members in `Position`. The state members in `ReadStream` are composed from two states, one from each dimension of `ReadStream`. Therefore, when the methods in `Position` and its children change state, they will change *both* dimensions of `ReadStream`. For example, when `nextPosition` advances the stream to the end, the `ReadStream` object composed of `Reading` with `NotEnd` will change to a `ReadEnd` with `End`.

[3] Abstract methods are indicated by eliding the method body; constant fields are declared with `val`, and variable fields with `var`.

```
state ReadWriteStream = Position {
  val notEndState =
    Writing with Reading with NotEnd;
  val endState =
    WriteEnd with ReadEnd with End;
} with Reader with Writer;
```

Listing 6: ReadWriteStream code.

To Initialize a `ReadStream` we need to specify the starting and-states like for `ResultSet`. The code to create a ReadStream `x` that is not at the end is `val x = new ReadStream @ ReadEnd with NotEnd;`. Here again, we use the `@` operator to begin the `ReadStream` in particular substates of the and-states nested in `ReadStream`.

Since the `Reader` dimension has the same structure as the `Position` dimension it is natural for transitions in `Position` to change `Reader` as well. In this example, there is no code in the `Reader` states that enacts the state change. Instead, the `Reader` dimension relies on the `Position` dimension to perform state changes. The state members in this example allow for this kind of dimensional reuse without extensive glue code[4]. The only code required to required to reuse the dimension is the specialization of state members in `ReadStream`.

We now illustrate a further step of consistent composition of states with the definition of `ReadWriteStream` in Listing 6. The definition uses a new dimension, `Writer`, with two substates `Writing` and `WriteEnd`, defined in the same manner as the `Reader` states.

This ReadWriteStream reuses code from all three dimensions with very little effort. The `ReadWriteStream` is the natural extension of `ReadStream`. The state members are composed from all three dimensions. The state transitions in a `ReadWriteStream` object will change all three dimension at once.

The `ReadWriteStream` example demonstrates both the power of a trait-like composition model and its novel extension to states. We reuse `ReadStream` and `WriteStream` with little effort, as we could achieve in a language with traits. In addition, we have a new unit of reuse, the `Position` dimension, which is shared with two other dimensions. This reuse eliminates duplicate code, and helps avoid bugs. Both the `Reader` and `Writer` of a `ReadWriteStream` are in the end-state or not-end state. Because the dimension is reused we can guarantee that no programmer will err and end up with an object in an inconsistent state like `WriteEnd` with `Reading`.

One important note is that the `Writer` and `Reader` contain no members in common, and therefore no conflict arises. Plaid requires explicit conflict resolution at the point of composition. This conflict resolution will be described in Section 4.2.

3.5 Validation

The introduction claims four concrete benefits of Plaid: code closely reflects design, programs are concise, error checking is implicit, and new opportunities for reuse. These benefits were illustrated in the examples in this section and they were discussed while describing the examples. We summarize the case here for emphasis. We then reflect on our experience writing mid-sized programs in Plaid, in diverse domains.

[4]State members also have a more traditional purposes. State members, like all states, can be used to create objects. They allow us to encode ML-style structures and functors. These abstraction mechanisms can be very powerful, especially in a typed version of Plaid. However, these purposes are not novel to Plaid so we do not focus on these here.

Project	Lines of Code	# Files
CodeGenerator	1205	24
AeminiumCodeGen	2610	8
Typechecker	4196	55
ASTtranslator	9506	107
PlaidApps	528	21
Standard Library	372	18
TestCompiler	2811	96
TestTypechecker	363	9
Total	21591	338

Table 1: Plaid code written for eight projects.

3.5.1 Concrete benefits

Code reflects design. Designs with stateful abstractions are clearly reflected in Plaid code. This is clear in of all three examples in this section. The implementation of the file, result set, and read-write streams all match their designs. Arbitrarily complex state-charts can be encoded in Plaid with the simple rules described alongside the result set example. Each abstract state maps to its own state in code, so the design of the abstraction and its protocol as a whole is highly *salient* in the code.

Concise programs. Since state constraints are implicitly enforced by the object model, none of our examples included any error checking code. The implementation are therefore smaller.

Error Prevention. Plaid's explicit state models make error checking more consistent, because the programmer cannot forget to check state constraints when a method is called. The level of abstraction of error messages is also thereby raised: when an inappropriate method is called, instead of triggering an internal run-time exception such as a null pointer, or (what is worse) silently corrupting data, the runtime can signal an error that a particular method is unavailable in the current state. Also, we have shown how state members can be used to enforce consistency of multiple dimensions of state at once.

Reuse. Plaid provides new reuse opportunities. Some state machines are used in many objects. For instance, the `Position` dimension was reused in both read and write streams, and it could also be reused in many IO and Collection libraries. Open and closed resources like the File and ResultSet are also very common.

3.5.2 Applicability to diverse domains

In order to gain practical experience with the language and experiment with typestate-oriented programming beyond small examples, we have written several mid-sized programs in Plaid. These programs further demonstrate the expressiveness of Plaid in a diverse set of domains including compilation, input-output, GUIs, and web. They are all available for download from the Plaid repository[5]. In total, we have written 22KLOC across 338 files. A breakdown of our implementations is in Table 1. We call out items of particular interest here.

[5] http://code.google.com/p/plaid-lang/

Compiler. Plaid is self-hosting; the CodeGenerator project compiles Plaid code into Java source. Plaid code can easily use Java libraries and many of our examples are implemented that way. In a sister project [24], we have implemented a separate compiler for parallel-by-default code, which is the AeminiumCodeGen project. We are currently working on a Plaid typechecker; the implementation is the Typechecker project. All these projects are supported by AST transformations performed by the ASTtranslator project.

GUI Library. GUI libraries often impose state constraints on their clients. We implemented Plaid wrappers for a few key Java Swing classes, including Window, Pane, and Canvas abstractions. We use states to enforce proper initialization of these abstractions. In particular, windows should have some contents added, otherwise they are created with size zero. Furthermore, windows are `Hidden` until `show()` is called, then they become `Visible`. Panes should also have contents added. Both panes and canvases must be assigned a parent window, and canvases should be given a preferred size. Our library is not comprehensive, but it is sufficient to build demonstration applications—in our case, a Turing machine that uses Plaid's states to represent the finite state control, the marks on the tape, and the illusion of an infinite tape. Both the windowing library and Turing demonstration application are in the PlaidApps project.

Miscellaneous The Plaidapps project includes the examples discussed earlier and a small web server and workflow engine. The Plaid standard library includes integers, rationals, strings, options, and standard control (e.g. `if`) and looping (e.g. `for`, `while`) structures. Finally, two testing projects include a number of smaller tests and examples.

4 Semantics

In this section we present the formal definition of the Plaid language and give it a precise semantics. At its core, Plaid is an object system with first-class generators and functions. Individual generators can be combined and specialized using composition and operators inspired by traits [12], instantiated to create objects, or used to specify the abstract state the object should change to. We start by describing the syntax and object model of a core language, which is intended to be simpler than Plaid source code yet be capable of representing all of the major semantic elements of Plaid. Then we discuss the execution semantics of the core language.

4.1 Core Syntax

The syntax of the internal representation of Plaid is given in Figure 4. In these definitions, x ranges over bound variables, while members of objects are represented by f, m, and s, which respectively range over fields, methods, and state members. We use n to represent any kind of object members when we do not distinguish between them. Abstract states are represented using tags which are generated as needed. We will introduce each syntactic category in turn, describing its purpose and motivations.

4.1.1 Expression Syntax

Plaid contains the standard expressions found in object systems, including object creation through new, field selection, and method calls. Because Plaid also has first-class functions, we include standard function definition and application as well. For sequential expressions, we include let bindings and bound variable references.

Obj Val	ov	::=	$mv \mid dv \mid mv \curlywedge ov \mid dv \curlywedge ov$
Dim Val	dv	::=	$tag\{ov\} \mid tag\{ov\} <: dv$
Mbr Val	mv	::=	$\mathsf{method}\ m(\overline{x})\{e\} \mid$
			$\mathsf{val}\ n = v$
ObjExp	oe	::=	$me \curlywedge oe \mid de \curlywedge oe \mid e \curlywedge oe \mid$
			$me \mid de \mid e$
Dim Exp	de	::=	$dv \mid tag\{oe\} \mid tag\{oe\} <: de \mid$
			$e \mid e\{\overline{to}\}$
Mbr Exp	me	::=	$mv \mid \mathsf{val}\ f \rhd x = e \mid$
			$\mathtt{recstate}\{\overline{\mathsf{val}\ s \rhd x = \mathsf{proto}\ sd}\}$
State Decl	sd	::=	$\mathsf{freshtag}\{oe\} <: de \mid$
			$\mathsf{freshtag}\{oe\} \mid oe$
Trait Op	to	::=	$\backslash n \mid n \to n' \mid me \mid (\mathsf{tagOf}\ e).me$
Val	v	::=	$\ell \mid ov \mid \mathsf{proto}\ oe \mid \mathsf{fn}(\overline{x}) \Rightarrow e$
Exp	e	::=	$x \mid v \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid$
			$e(\overline{e}) \mid e.m(\overline{e}) \mid e.n \mid$
			$e \leftarrow e \mid e \twoheadleftarrow e \mid \mathsf{new}\ e \mid$
			$\mathsf{match}(e)\{\overline{c}\} \mid$
			$\mathsf{freeze}\ e \mid \mathtt{recstate}\{\overline{mv}\}\#\mathtt{l}$
Case	c	::=	$\mathsf{case}(\mathsf{tagOf}\ e)\ \{e\} \mid \mathsf{default}\ \{e\}$

Figure 4: Internal Syntax

The rest of the expression forms are related to Plaid's encoding of abstract states and the transitions between them:

Changing state. The Plaid core has two state change operators. $\leftarrow$ represents a state update and only removes portions of the receiving object that are mutually exclusive with the incoming states. For completeness and flexibility, Plaid also includes a state replacement operator, $\twoheadleftarrow$, which wipes the receiving object clean before adding the incoming states, much like an in-place new operation. One could imagine using this operator in a situation where an object needed to be in a particular state and no other states. This cannot be guaranteed by the state update operator because state update leaves dimensions unrelated to the updating state alone.

Unlike the source language, Plaid's core does not require the target of a state change operator to be *this*. This makes the core simpler and more flexible since the restriction can be enforced at the source level.

proto values. First class instance generators are provided by proto expressions. These are values which can be stored in fields and passed as parameters. During a well-formed execution, the target of new expressions and the right-hand side of state change expressions will evaluate to a proto value. This is because they encapsulate object expressions, oe, which are uninitialized objects. The state change and new expressions cause the initialization steps specified by the object expression wrapped in the proto to be evaluated for use in creating a new object or changing the state of an existing one.

State expressions. To allow states to be chosen dynamically at runtime, we include several expression forms that can evaluate to a proto. As they are values, standard deference or bound variables could result in proto expressions. Because most states included in protocols must be defined with (mutual) recursion, proto values represented source-declared states are wrapped into a recstate. A particular proto can be selected from the recstate as from a standard record.

The freeze expression is a more novel way to get a proto. It takes the object and wraps it up in a proto allowing new instances to be generated from it. As an example of the use of freeze, consider the `myResultSet` value defined in Listing 3. Say we wanted to do some extra initialization of the `ResultSet` before using it and that over the course of a program we would create the same `ResultSet` over and over. To avoid needing to do the same initialization repeatedly, one could freeze the object the fully initialized object and then instantiated it each time a new `ResultSet` of this form was needed. freeze has already been used in the Plaid compiler to more cleanly support certain initialization paradigms, such as the transformation to let-normal form, where strings of let bindings must be concatenated together.

Matching tags. Finally, the match construct allows pattern matching based on tags. Each case tests the target object against the tagOf another expression. This expression is expected to evaluate to a proto value with a single outer tag which is grabbed by tagOf and compared with the tags of the target object. If the object contains the tag, the corresponding case is executed. Cases are evaluated in order.

An example of the use of match comes from the Plaid standard library. Plaid's syntax does not include control structures. Instead, `if` and `while` are encoded as functions that make use of match. The states `True` and `False` are each defined as a `case of Boolean`. Thus, the `if` function determines whether or not to evaluate the body based on whether the object returned by the condition matches the `True` tag.

4.1.2 Object Value Syntax

Plaid objects are collections of tags representing the states that the object is in along with fields and methods that provide the representation and operations of those states. In order to implement the desired semantics these object must be organized to formally encode the relationships between tags and members that the semantics depend on. In particular, we need to represent the following relationships between the abstract states that the tags represent:

1. *Superstates*: An object in state, S, which is defined to be a case of a superstate, T, must also be in state T. For instance, an object in the `NotEnd` state defined in figure 4 is also in the `Position` state.

2. *Or-states*: Distinct cases of a given state, such as the `OpenFile` and `ClosedFile` case of `File`, cannot exists together in an object.

3. *And-states*: Both objects and states can be defined as a composite of other states. For example, the `Open` state from Listing 3 is defined in terms of states `Direction`, `Status`, and `Action`. Objects in the composite state are considered to be in each of the component states as well.

4. *Defining states*: Members must be associated with the state that declares them so that they can be removed from an object when their defining state is removed.

To formalize these relationships, *objects values* are organized as hierarchical collections of *dimensions*, which contain tags for the state and all of its transitive super states, and *members*.

Object values. The basic component of an object is an object value, ov, which is a list of dimension values, dv, and member values, mv. They are used to represent both top-level objects and the dimensions and members that define a given state (see dimension values below). The $\curlyvee$ operator that separates each element of the list represents composition. Object values encode and-states by allowing two dimensions to coexist together inside the definition of a state. For instance, the object value that defines a `ReadStream` would have two composed dimensions, one for the `Position` dimension, and the other for the `Reader` dimension.

Dimension values. Dimension values, $tag\{ov\}[<: dv]$, encode the structure of a state and its super states. They are represented by a tag, tag, which is a unique name for the most specialized state from the dimension. Associated with the tag is an object value which represents the collection of members that the state defines along with any other dimensions that make up the and-states of the state. A dimension value may optionally contain another dimension value encoding the superstate relationship.

By containing the representation of a given states superstates, dimension values give us a way to encode the or-state relationship as well. Two states that are the case of the same superstate would be encoded as separate dimensions with the same state at the root of the dimension. Because the tags in the dimensions partially overlap, by restricting tags to appear only once in a given object value, we can ensure that no or-states can coexist in a single object.

Concretely, we would represent an instantiated `Open` state from Listing 3 as

$$\texttt{Open}\{\texttt{Direction} \curlyvee \texttt{Status} \curlyvee \texttt{Action}\} <: \texttt{ResultSet}.$$

Here the most specific state of the represented dimension value is `Open`. This state is defined based on the three states `Direction`, `Status`, and `Action` (defining object values not shown), and specializes the `ResultSet` state, which it was defined as a `case of`.

A dimension is also Plaid's version of a trait. Multiple inheritance is achieved by allowing multiple dimensions to be composed in an object value as well as in the object values associated with the tags of a dimension. The hierarchical nature of Plaid's dimension prevent us from using all of the trait mechanisms for solving the problems of multiple inheritance. In particular, a multiple inheritance system must deal with the case when one class inherits from two classes that share a (transitive) parent. This situation is challenging because it is non-obvious how to inherit members from the common grandparent. This problem is commonly referred to as the *diamond problem* [19], because of the shape of the inheritance hierarchy diagram. The original traits proposal [12] *flattens* [6] composed traits and forces any conflicts between method names to be explicitly resolved (field were not allowed in traits). However, as Plaid's semantics depend on members being related to the tag they are defined in, we cannot use flattening. Instead, Plaid prevents the diamond problem by preventing or-states from coexisting, thereby preventing the same tag and member definition from appearing more than once (following Malayeri's no-diamonds rule [19]). Plaid's solution follows recent extensions of traits including [5, 22, 9]. Like Plaid, these system support traits with fields and work in a variety of object models including those that, like Plaid, add hierarchy and do not enforce the flattening property. As with the original trait proposal, all name conflicts across dimensions must be explicitly resolved in Plaid via the trait operators described below.

Member values. A member value is either a method, with a set of arguments and a body, or a field, val f, bound to a value, v. The member is said to be defined in the state represented by its immediately enclosing

[6] The flattening property from [12] states that members of an are treated equally regardless of what trait they were defined in.

tag. As a concrete example, an object in the ClosedFile state described in Listing 1 would be represented formally as

$$\texttt{ClosedFile}\{\mathsf{method}\ close()\{e\}\}$$
$$<: \texttt{File}\{\mathsf{val}\ filename = v\}$$

This indicates that the object is in both the `ClosedFile` and the `File` states, one of which is a substate of the other, and each of which defines a single member.

4.1.3 Uninitialized Object Syntax

Plaid has corresponding syntax for uninitialized objects organized into *object expressions*, *dimension expressions*, and *member expressions*. When compared to their value counterparts, they share the same structure but contain expressions which are not yet values. In this section, we discuss the places where execution can occur in these forms and the motivation behind them.

Object expressions. Object expressions, oe, are made up of the composition of dimension expressions, member expressions, as well as raw expressions. The purpose of unevaluated expressions in dimension and member expressions will be explained below. Raw expressions as components of object expressions allow part of an uninitialized object to be determined at the time of initialization. These expressions evaluate to proto values which are then incorporated into the initializing object. This provides for Plaid's implementation of dynamic trait composition by allowing portions of the object to be selected at runtime.

Dimension Expressions Dimension expression can contain unexecuted expressions in the object expression associated with the most specific tag as well as in tags up the hierarchy if they exist. Dimension expressions may also have associated trait operations, to, which need to be evaluated. Trait operations allow standard manipulations such as renaming, $n \to n'$, and removal, $\backslash n$. Note that these operate on the *whole* dimension, renaming or removing all members of the specified name defined directly in tags in the hierarchy (not including nested dimensions). This allows the changes to be preserved by state change in the dimension as we will see below.

Members can also be added or replaced[7]. By default, they are (re)placed in the most specific tag of the dimension expression. However, In cases where members need to be added or replaced in a particular tag, they can be qualified by a particular tag, specified as with tags in case statements by tagOf another expression. The redefinition of `Position.EndState` for the `ReadStream` in Listing 5 is an example of using qualified trait operations. This mechanism is important in Plaid because of the hierarchical nature of Plaid's object model and when and how member definitions are removed during state change.

Member expressions. Only fields can be member expressions, me, as methods do not have any initialization code. On the other hand, fields can be defined with initialization expressions that require evaluation as a part of object creation or update. In order to allow fields to refer to the initialized value of previous fields in the same state, field expressions define an internal bound variable in addition to their external name (this is a standard approach from [21], chapter 8). Fields are also generated by state declarations. Since

[7]The semantics defined here do not allow fields and states in trait operations to refer to other trait operation members. The formalism could be extended to support this, mirroring the case for declarations in states

$$
\begin{array}{rrcl}
\text{Heap} & H & ::= & [\ell \rightsquigarrow ov], H \mid \cdot \\
\text{Eval} & E & ::= & [\,] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid \\
 & & & E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid E.f \mid E \leftarrow e \mid \\
 & & & v \leftarrow E \mid v \leftarrow \mathsf{proto}\ E \mid E \twoheadleftarrow e \mid \\
 & & & v \twoheadleftarrow E \mid v \twoheadleftarrow \mathsf{proto}\ E \mid \mathsf{new}\ E \mid \\
 & & & \mathsf{new\ proto}\ E \mid \mathsf{match}(E)\{\overline{c}\} \mid \\
 & & & \mathsf{match}(v)\{\mathsf{case}(\mathsf{tagOf}\ E)\ \{e\}, \overline{c}\} \mid \\
 & & & \mathsf{freeze}\ E \mid ov \curlywedge E \mid O \curlywedge oe \mid O \\
\text{Obj} & O & ::= & \mathsf{val}\ n \rhd x = E \mid tag\{oe\} <: E \mid \\
 & & & tag\{E\} \mid tag\{E\} <: dv \mid E\{\overline{to}\} \mid \\
 & & & dv\{\overline{to}, \mathsf{val}\ n = E, \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\mathsf{tagOf}\ e).(\mathsf{val}\ n = E), \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\mathsf{tagOf}\ E).mv, \overline{to}\}
\end{array}
$$

Figure 5: Contexts

the definitions of related states, such as the `OpenFile` and `ClosedFile` from Listing 1, are typically recursive, the initialization of state members occurs in a recstate binding.

State members are also special in that when an uninitialized object containing state members is initialized, new tags may need to be generated. The proto expression encapsulates uninitialized objects as discussed above. Normally they contain object expressions, but when appearing in a recstate, they contain state declarations, sd which may contain the freshtag operations that generates a new tag when executed, resulting in an object expression. This feature means that new tags are generated for states defined inside states each time the outer state is instantiated. Because these tags can then be used to pattern match on objects, this allows Plaid to implement ML-style generative functors[8]. Functors have well recognized modularity benefits that we do not discuss here.

4.2 Dynamic Semantics

We now introduce the dynamic semantics of Plaid. We formalize the execution using a small step operational semantics. The basic evaluation judgment has the form $e@H \mapsto e'@H'$ and is read "expression e with heap H evaluates to expression e' in heap H'". We define a similar judgment $oe@H \mapsto oe'@H'$ for the evaluation of object expressions. In this section, we will define the form of the heap and the invariants that we maintain on it. We will also discuss the Plaid-specific evaluation rules, in particular those that use ancillary judgments for implementing state change. As state change is at the core of Plaid's design and is the most complicated we go into depth about the motivation and design of the rules that implement it. Finally, we describe object initialization and trait operations that may be involved.

4.2.1 Heap

A heap, H, is a mapping from locations, ℓ, to object values. We place additional well-formedness requirements on all object values stored in the heap. These restrictions prevent ambiguities from multiple inheritance.

[8]Generative functors, in contrast to applicative functors, generate new abstract types for each application of the functor. This impacts pattern matching when using these generated types in a similar way as pattern matching on freshly generated tags in Plaid.

$\boxed{e@H \mapsto e@H}$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e@H \mapsto e[v/x]@H}\text{E-LET} \qquad \frac{|\{\overline{x}\}| = |\{\overline{v}\}|}{(\mathsf{fn}\ (\overline{x}) \Rightarrow e)(\overline{v})@H \mapsto e[\overline{v/x}]@H}\text{E-APP}$$

$$\frac{\begin{array}{c}H[\ell] = ov \\ \mathtt{lookup}(m, ov) = (\mathsf{method}\ m(\overline{x})\{e\}) \\ |\{\overline{x}\}| = |\{\overline{v}\}|\end{array}}{\ell.m(\overline{v})@H \mapsto e[\ell/\mathsf{this}][\overline{v/x}]@H}\text{E-CALL} \qquad \frac{\begin{array}{c}H[\ell] = ov \\ \mathtt{lookup}(f, ov) = (\mathsf{val}\ f = v)\end{array}}{\ell.f@H \mapsto v@H}\text{E-FIELD}$$

$$\frac{\begin{array}{cc}H[\ell] = ov_1 & \mathtt{uniqueTags}(ov_2) \\ ov_1 \leftarrow ov_2 \Rightarrow ov_3 & \mathtt{uniqueMembers}(ov_3)\end{array}}{\ell \leftarrow \mathsf{proto}(ov_2)@H \mapsto \mathsf{void}@H[\ell \rightsquigarrow ov_3]}\text{E-SU}$$

$$\frac{\mathtt{uniqueTags}(ov) \qquad \mathtt{uniqueMembers}(ov)}{\ell \twoheadleftarrow \mathsf{proto}(ov)@H \mapsto \mathsf{void}@H[\ell \rightsquigarrow ov]}\text{E-REPLACE} \qquad \frac{\begin{array}{c}\ell \notin H \qquad \mathtt{uniqueTags}(ov) \\ \mathtt{uniqueMembers}(ov)\end{array}}{\mathsf{new}\ (\mathsf{proto}\ ov)@H \mapsto \ell@H[\ell \rightsquigarrow ov]}\text{E-NEW}$$

$$\frac{de = tag\{oe\}[<: de'] \qquad tag \notin \mathtt{tags}(H[\ell])}{\mathsf{match}(\ell)\{\mathsf{case}\ (\mathsf{tagOf}\ \mathsf{proto}\ de)\{e\}, \overline{C}\}@H \mapsto \mathsf{match}(\ell)\{\overline{C}\}@H}\text{E-CASENOMATCH}$$

$$\frac{de = tag\{oe\}[<: de'] \qquad tag \in \mathtt{tags}(H[\ell])}{\mathsf{match}(\ell)\{\mathsf{case}\ (\mathsf{tagOf}\ \mathsf{proto}\ de)\{e\}, \overline{C}\}@H \mapsto e@H}\text{E-CASEMATCH}$$

$$\frac{}{\mathsf{match}(\ell)\{\mathsf{default}\{e\}, \overline{C}\}@H \mapsto e@H}\text{E-CASEDEFAULT} \qquad \frac{H[\ell] = ov}{\mathsf{freeze}\ \ell@H \mapsto \mathsf{proto}\ ov@H}\text{E-FREEZE}$$

$$\frac{\mathtt{l} = \mathtt{l}_s \qquad oe = oe_{\mathtt{l}_s}[\overline{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#\mathtt{l}_i/s_i}]}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#\mathtt{l}@H \mapsto \mathsf{proto}\ oe@H}\text{E-RECSTATESELECT}$$

Figure 6: Expression Evaluation

Tag uniqueness. We require that all well-formed object values have no duplicate tags. As alluded to above, this property ensures that an object is not in two cases of a single or-state at the same time. This is because the tags representing two mutually exclusive or-states must come from the same dimension and thus must have at the least the root tag of the dimension in common. It also prevents the diamond problem of multiple inheritance by ensuring that a particular member definition does not appear multiple times in a single object. This invariant is encoded in the helper judgment `uniqueTags` also defined in Figure 10.

Member uniqueness. Even though a given definition for a member cannot appear more than once, it is still possible that multiple tags define members with the same name. To prevent ambiguities in this case we require that all members of an object are provided by exactly one dimension. Because the hierarchy of dimensions gives us a natural way to choose the visible definition (the one from the most specific tag in the

dimension) we allow a single name to be defined directly in multiple tags from a single dimension. Formally, two tags are in the same dimension is one is a transitive case of the other. This relaxation of classical traits allows, for instance, a common super state to define a default behavior for a method which can be overridden by (some of) its substates. The judgment `uniqueMembers` defined in Figure 10 captures this requirement. It uses the judgments $mv :: tag.x@ov$, which states that member value mv from tag tag defines name x in object value ov, and $tag << tag'@ov$ which asserts the property that tag tag is a transitive subtag of tag' in object value ov. Based on these helper judgments, an object value has unique members if whenever we find the same member defined in two tags, then one of these tags is a transitive subtag of the other. We prove that evaluation preserves member and tag uniqueness in appendix A.

Member lookup. As an object can contain multiple members with the same name, we need an unambiguous way to choose which one is visible. The `lookup` function also in Figure 10 defines this logic. When multiple definitions are found, we know by `uniqueMembers` that they all come from the same dimension. Since the tags of a dimension form a total order, we know that one of tags defining the member will be a transitive subtag of all other tags defining the member. The definition from this most specific tag is the one returned by `lookup`.

4.2.2 Expressions

The evaluation rules for expressions in Plaid are given in Figure 6. We only list only computation rules here, defining congruence rules using evaluation contexts shown in Figure 5. In these, each expression with a subexpression that requires evaluation defines a hole, [ ], into which any expression can be placed. Evaluation proceeds by using the computation rule that evaluates the expression in the hole.

Standard rules. The computation rules for the evaluation of the expressions from general object systems and the lambda calculus are almost all completely standard in our system. These include the rules E-LET, E-APP, E-CALL, and E-FIELD for let expressions, application, method calls and field dereferences respectively. One note is that member selection during calls and dereferences use the `lookup` judgment described above. We also use standard record evaluation rules when selecting a label from a recstate (E-RECSTATESELECT).

Match. Plaid uses a first-match semantics, so that we find the first case clause whose tag matches the target object. We find the tag to match against by grabbing the most specific tag (tagOf) from a dimension expression wrapped in a proto value. Note that in this case the dimension expression is not evaluated since we are only interested in the tag. If the tag is found in the target object, the code for this case is evaluated (E-CASEMATCH); otherwise, execution proceeds to the next case (E-CASENOTMATCH). Default cases are always executed and terminate the match if reached (E-CASEDEFAULT). Evaluation gets stuck if no matching case is found.

Freezing. To freeze a location in the heap (E-FREEZE), we simply pull the object value from the heap and wrap it in a proto expression.

Manipulating objects in the heap. The state change operators and new cause objects in the heap to be changed or allocated. Because we only allow object values to appear in the heap, we must first initialize the object that will be used to alter the heap by reducing it to an object value. Evaluation is mostly handled by the evaluation contexts: first the expression representing the object is reduced to a proto value and then the

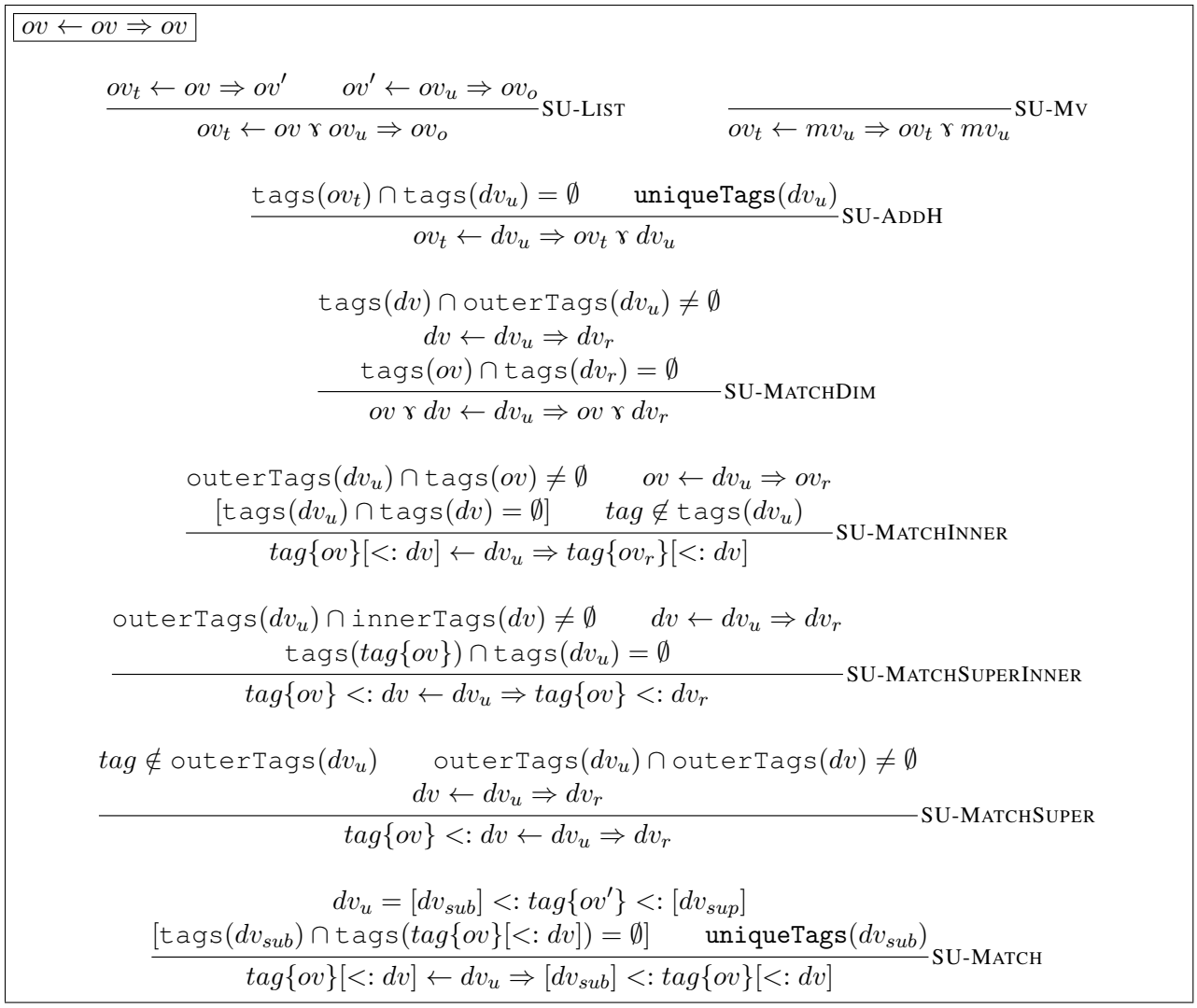

Figure 7: State Update

object expression wrapped in the proto is evaluated down to an object value. An important design decision in Plaid was to run the initializers for all members of an object expression. This happens despite the fact that not all members may end up in the object (see the explanation of state update below). In particular, any effectful initializers will always be run and update the wider context. We experimented with other possible semantics but decided that a clear and unambiguous rule for when initializers were run (always) was better than a flexible but complicated one. Furthermore, we consider it good Plaid style to avoid the use of effectful initializers and instead use other design techniques, such as factory methods, when effectful operations are required as a part of object initialization.

Once the initialization code in the proto has been run, the resulting object value can be used to update the heap. In the case of new and state replacement (↞) expressions it is clear what the object value that is inserted into the heap will be. new allocates a new location on the heap and maps it to the resulting object value. State replacement replaces the mapping of the target location on the heap with the updating object value. Since we know the precise form of the object value that is being inserted into the heap, in order to

```
1 val rs = Open {
2   Inserted <: Inserting <: Action,
3   Scrollable <: Direction,
4   Updatable <: Status }
5 <: ResultSet
```

Listing 7: Open, Inserted, Scrollable, Updatable ResultSet

maintain the heap invariants on object values, we can simply check that `uniqueTags` and `uniqueMembers` both hold on the new object value as done in the rules E-REPLACE and E-NEW. On the other hand, the semantics of updating an object on the heap using state update are much more complicated, and so we devote the next section to a discussion of its design and proof that they maintain the necessary invariants.

4.2.3 State Update

At the core of the rule E-SU which updates the heap with the result of a state update is the state update judgment, $ov \leftarrow ov \Rightarrow ov$, which is described in Figure 7. The judgment takes two object values and determines the resulting object value when the *target* object on the left side of the arrow is changed to the state given by the *update* object from the right side. The semantics of this judgement are the most complicated and important part of Plaid's dynamic semantics. Thus, before describing the semantics given by the rules, we step back and give a high-level overview of the desired behavior. We then define some general properties and assumptions of the judgment before describing the rules themselves.

Design considerations. Our goal is that the design of the state change judgment should match the semantics of stateful abstractions as modeled by state charts and similar tools. Thus, a state update should transition a target object from its current set of abstract states to a possibly new set of abstract states as specified by the update object. To do this, we need to formalize this intuition in terms of object values.

Update dimensions. Our first task is to determine which abstract state the update object is changing. That is, which dimensions of the target object need to be updated? Consider the object value (without members) of an `Open ResultSet` in the `Inserted`, `Scrollable`, `Updatable` state, stored in val rs as depicted in Listing 7. What should happen if we update rs to the `ReadOnly` state?

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

While there are clearly matches between tags in the target and update objects, since the tags are nested inside the `Open` tag of the target object, it is not clear that they should be updated. However, if we think of the state update as an transition to a new abstract state, then we can see that the nesting in the target object should not matter. This state update specifies that the `Status` dimension should transition to the `ReadOnly` substate, and thus out of the `Updatable` state.

The converse question is does nesting matter in the other direction? In other words, can a nested state trigger a change in an abstract state? Concretely, would this state update

$$rs \leftarrow \texttt{Foo}\{\texttt{ReadOnly} <: \texttt{Status}\}$$

result in an object in the `ReadOnly` state? Based on the semantics of state charts, the answer would be "no". Our definitions of object dimension indicates that the `Status` dimension of the `Foo` state is part of the

definition of `Foo`. Thus, it is brought along with the transition to the `Foo` state. The `Status` state is also a defining and-state of the `Open` state. Thus the resulting object cannot be consistent because two separate dimensions are claiming the `Status` state meaning there would need to be duplicate tags.

Therefore, we define the dimensions along which a state update occurs to be only those found at the top level of the object value that describes the update object. All other dimensions that are a part of the update object are considered definitions of these dimensions and do not induce transitions but are only added to the object with their enclosing state.

Dimension updates. Once we know which dimensions will be updated, we need to know what in those dimensions is changed. We first note that we can treat the transition in each dimension independently as dimensions are orthogonal by definition. Second, recall the file example from Listing 1. In this example, we stated that the `filename` member was shared between the `OpenFile` and `ClosedFile` states. Thus, when we transition from an `ClosedFile` to an `OpenFile` the members of the `File` state should remain constant. This is the semantics behind the restricted update semantics of state change described in [1]. We use and extend these semantics in a natural way to account for our hierarchical object model.

Properties of object and state update. With the intuition we have for the design, we can define some terminology that is used in the judgment itself.

Inner and outer tags. In the informal description of state change, we differentiated between dimensions and tags defined at the top level of the update object and those that appear within a top-level dimension. Figure 10 defines two judgments, `outerTags` and `innerTags`, which capture this distinction. The `outerTags` of an object value, ov, are all the tags which appear as the most specific tag and any of its super tags from dimensions appearing directly in ov. For example, using rs, the `ResultSet` object from Listing 7, $\texttt{outerTags}(rs) = \{\texttt{Open}, \texttt{ResultSet}\}$. Conversely, the `innerTags` of an object value are all of the tags defined in dimensions that are recursively included in the definition of each of the outer tags. For example,

$$\texttt{innerTags}(rs) = \{\texttt{Inserted}, \texttt{Inserting}, \texttt{Action}, \\ \texttt{Scrollable}, \texttt{Direction}, \texttt{Updatable}, \texttt{Status}\}$$

Unique dimension property. Given a dimension within which to transition the target object, we need to find the location of the matching dimension within the target object value. To do this, we look for the part of the object that has tags which overlap the outer tags of the update dimension. We ignore all super-tags of the matching tag in the update dimension under the assumption that these supertags will match the tags in the target. This assumption is based on the the Unique Dimension Property which states that a single unique tag can only ever appear in a single dimension. That is a tag is either has no super tags or always appears with the same supertag. While this property is not guaranteed by the syntax and semantics of the internal language, it is enforced by the elaboration from Plaid's source syntax so we assume it in our rules.

Maintaining the `uniqueTags` property. Rule E-SU in Figure 6 does not check whether the object returned from the state update judgement has unique tags. Therefore the state update judegment must maintain this property. Formally: If $\texttt{uniqueTags}(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then $\texttt{uniqueTags}(ov_3)$. A proof of this property can be found in appendix A.

$oe@H; T \mapsto oe@H; T$

$$\frac{[oe' = oe[v/f']]}{\mathsf{val}\ f \rhd f' = v[\ \curlyvee\ oe]@H \mapsto \mathsf{val}\ f = v[\ \curlyvee\ oe']@H}\text{E-RecField}$$

$$\frac{sd = \mathsf{freshtag}\{oe\}[<: de] \qquad tag \text{ is fresh}}{r = \mathsf{recstate}\{\overline{\mathsf{val}\ s_d \rhd x_d = v}, \mathsf{val}\ s \rhd x = \mathsf{proto}\ (tag\{oe\}[<: de]), \overline{\mathsf{val}\ s_r \rhd x_r = \mathsf{proto}\ sd_r}\}[\ \curlyvee\ oe]}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_d \rhd x_d = v}, \mathsf{val}\ s \rhd x = \mathsf{proto}\ sd, \overline{\mathsf{val}\ s_r \rhd x_r = \mathsf{proto}\ sd_r}\}[\ \curlyvee\ oe]@H \mapsto r@H'}\text{E-RecState1}$$

$$\frac{\overline{oe'_i = oe_i[\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#s_i/x_i]} \quad [oe' = oe[\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe}\}\#s_i/x_i]]}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i \rhd x_i = \mathsf{proto}\ oe_i}\}[\ \curlyvee\ oe]@H \mapsto \overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe'_i}[\ \curlyvee\ oe']@H}\text{E-RecState2}$$

$$\frac{}{(tag\{oe\} <: \mathsf{proto}\ de[\{\overline{to}\}])[\ \curlyvee\ oe']@H \mapsto (tag\{oe\} <: de[\{\overline{to}\}])[\ \curlyvee\ oe']@H}\text{E-De}$$

$$\frac{}{\mathsf{proto}[ov\ \curlyvee\ ](\mathsf{proto}\ oe[\{\overline{to}\}])[\ \curlyvee\ oe']@H \mapsto \mathsf{proto}[ov\ \curlyvee\ ](oe[\{\overline{to}\}])[\ \curlyvee\ oe']@H}\text{E-Oe}$$

$$\frac{dv\{\overline{to}\} \Rightarrow dv'}{(dv\{\overline{to}\})[\ \curlyvee\ oe]@H \mapsto dv'[\ \curlyvee\ oe]@H}\text{E-TraitOps}$$

Figure 8: Object Evaluation

Inference rules. With this understanding, we can describe the rules that produce the object value after a state update operation. The rules start by breaking apart the update object ov into the individual member values and dimension values and processing the state changes for each dimension or value individually (SU-LIST). This is allowed since each dimension can be treated independently. We can assume that `uniqueTags` holds for each dimension individually since it holds for the object as a whole. For member values (SU-MV) and dimension values for which there is no overlap between the tags of the target object and update dimension (SU-ADDH), we just compose the update object with the target object. The rest of the rules assume that there is a match between the outer tags of the update object and the tags of the target object. If that is not the case, then the evaluation gets stuck.

SU-MATCHDIM covers the case where we have found a particular dimension of the target object that contains the tags that are changing. By the unique dimension property explained above, we know that the $\texttt{outerTags}(dv_u)$ will not appear in ov, so it suffices to calculate the state update on just the matched dimension. To ensure that we maintain the unique tags property, we can assume that both the result of the state update and the unmatched portion of the object have unique tags, and so it suffices to check that the tags of these two portions of the object do not intersect.

SU-MATCHINNER handles the case where there is overlap between the `innerTags` of the current tag and the `outerTags` of the update dimension. We recursively find the state update on just this matching portion and then check that the tags from the resulting object value do not intersect with the tags of the super tag, if it exists, to maintain the `uniqueTags` invariant.

In SU-MATCHSUPERINNER, we find that the matching dimension is defined somewhere inside of a super tag. Thus, we run state update on just the supertags. We then verify that the tags of the result are distinct from the tags of the subtag and its `innerTags` to maintain the `uniqueTags` invariant.

SU-MATCHSUPER represents the case where we have found the right dimension, but have not reached the level of the dimension where the tags overlap. The current tag of this dimension is not in the outer tags of the update dimension, but there is overlap somewhere in its super tags and so we find the updated state from that portion of the dimension. In this case, we know that the current tag will be removed with any of its nested tags, which means that we do not need to check if these tags would conflict with tags that enter the object with the update dimension to preserve `uniqueTags`.

The base case SU-MATCH handles the actual alteration of the target dimension. The current tag matches a specific tag in the outer tags of the update dimension, which indicates that the state update only affects states below this point in the dimension. In particular the tags below this one in the dimension in the target object are discarded, as already occurred through the SU-MATCHSUPER rule. In their place are put all the subtags of the matched tag from the incoming dimension. To make sure that we do not have duplicate tags anywhere, we only need to check that the tags added from the update dimension do not intersect with the tags that are in its new supertags.

Example. To give a specific example, consider evaluating the following state update on the object defined in Listing 7:

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

The state updates proceeds first by finding that there is tag overlap between the incoming and target objects and a match for the `Status` tag of the incoming state nested inside the `Open` state with the SU-MATCHINNER. Next it finds the correct dimension `Updateable` $<:$ `Status` using the SU-MATCHDIM rule. It discards the `Updateable` tag and recurses up the dimension in the SU-MATCHSUPER rule and finally adds the `ReadOnly` tag in its place with the SU-MATCH rule.

Reduction rule The E-SU reduction rule uses the state update judgement to determine what object value to update the target object to. The state update judgement incrementally checked that `uniqueTags` was maintained. It does not guarantee that `uniqueMembers` is satisfied and so the rule checks that the resulting object value has unique member declarations.

4.2.4 Object Evaluation

The final class of reductions that we must model is that of state expressions, including the initialization of object expressions within a proto. These rules are defined in Figure 8. Congruence rules are again taken care of by evaluation contexts from Figure 5.

- E-RECFIELD: When field members have been evaluated down to values, we propagate them forward into the rest of the declarations that need to be initialized by substituting the value in for the bound variable on the right of the $\triangleright$. This allows subsequent fields to use the values of previously declared fields during their initialization. After this propagation, we do not need to keep track of the bound variable any longer and so do not record it in the member value. Note that these semantics force us to be strict about the order in which portions of the object are initialized. In particular member declarations are initialized from left to right as specified by the evaluation contexts.
- E-RECSTATE1: If there are freshtag directives in the state declarations of a recstate, new tags are generates by picking a fresh tag not previously mentioned.

- E-RECSTATE2: After assigning new tags to all of the state declarations inside a recstate, we need to remove the recstate construct and convert it into a list of val declarations. This is done in a manner similar to the fix construct in the lambda calculus. Since our recstate is modeled as a record, we replace all references to the inner bound variable of each of the nested state vals with selections of the external name from the recstate. We do this both inside the object expressions of each proto as well as in subsequent declarations. Note again that after propagation we can remove the bound variable from the val declaration.

- E-DE and E-OE: These rules state that it is possible to unwrap a proto that is nested inside another proto. This can occur when a proto is part of an object expression inside another proto (E-OE), or when a proto is in a dimension expression, which only appear in proto expressions (E-DE). In either case, if trait operations are associated with this proto, then they are retained. Execution will continue by evaluating the wrapped object expression if needed.

- E-TRAITOPS: This rule applies only once the all of the trait operations have been fully reduced and proceeds using the trait operations judgment defined below to produce a new dimension value.

4.2.5 Trait Operations

As with state change, we define a separate judgement for trait operations that applies once all trait operations have been fully initialized, meaning that they can all be applied atomically without reduction. The rules for initialization of trait operations are all congruence rules handled by evaluation contexts (see Figure 5). Thus, the judgement, $ov\{\overline{sp}\} \Rightarrow ov$, does not require a heap. In general, trait operations follows previous work on traits. However, Plaid's object model, unlike traditional traits models, is hierarchical. Hence, trait operations other than the local member addition must take this hierarchy into account.

Local member updates are agnostic to whether the added member is already a member of the tag and simply add the new member, replacing the existing member if one exists (T-MEMBER). Updates of members in specific tags act the same, but first must recurse through the object value looking for the specified tag before performing the member update. The computation will get stuck if the tag is not found. Because each of these trait operations, as well as member renaming described below, may potentially add new members, there is the danger that the object value might no longer satisfy the `uniqueMembers` invariant. However, since the specialization must be occurring as part of object instantiation, it will be checked at the point that the object is created, so we do not make the check here.

Member removal and renaming operate on the whole object, removing or renaming instances of members with the given name throughout. This is in contrast to `lookup`, which stops at the first declaration of the member. These semantics are required in order to allow trait composition, which includes the ability to remove members from a trait and instead provide them in another trait. This would result in a conflict if some members were left in the old dimension.

5 Elaboration

The core language defined in the previous section shares much in common with the full Plaid programming language, but there are still differences. The source syntax is defined in figure 11. The semantics of the full Plaid language are defined as an elaboration into the core language defined in appendix B.

For most expressions, the elaboration proceeds structurally, without changing the construct itself. For field bindings, we add the internal variable referred to above, and replace references to the field in later field

$\boxed{ov\{\overline{to}\} \Rightarrow ov}$

$$\frac{dv\{to\} \Rightarrow dv' \qquad dv'\{\overline{to}\} \Rightarrow dv''}{dv\{to, \overline{to}\} \Rightarrow dv''}\text{T-GENERAL} \qquad \frac{ov = ov'[\curlywedge\, mv'] \qquad [\texttt{name}(mv') = x = \texttt{name}(mv)]}{(tag\{ov\}[<: dv])\{mv\} \Rightarrow tag\{ov' \curlywedge mv\}[<: dv]}\text{T-MEMBER}$$

$$\frac{ov\{\backslash n\} \Rightarrow ov' \qquad [dv\{\backslash n\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-REMOVEDV} \qquad \frac{dv\{\backslash n\} \Rightarrow dv' \qquad ov\{\backslash n\} \Rightarrow ov'}{(dv \curlywedge ov)\{\backslash n\} \Rightarrow dv' \curlywedge ov'}\text{T-REMOVEOV1}$$

$$\frac{\texttt{name}(mv) = n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlywedge ov)\{\backslash n\} \Rightarrow ov'}\text{T-REMOVEOV2} \qquad \frac{\texttt{name}(mv) \neq n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlywedge ov)\{\backslash n\} \Rightarrow mv \curlywedge ov'}\text{T-REMOVEOV3}$$

$$\frac{ov\{n \to n'\} \Rightarrow ov' \qquad [dv\{n \to n'\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-RENAMEDV}$$

$$\frac{dv\{n \to n'\} \Rightarrow dv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(dv \curlywedge ov)\{n \to n'\} \Rightarrow dv' \curlywedge ov'}\text{T-RENAMEOV1}$$

$$\frac{\texttt{name}(mv) = n \qquad \texttt{rename}(n', mv) = mv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlywedge ov)\{n \to n'\} \Rightarrow mv' \curlywedge ov'}\text{T-RENAMEOV2}$$

$$\frac{\texttt{name}(mv) \neq n \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlywedge ov)\{n \to n'\} \Rightarrow mv \curlywedge ov'}\text{T-RENAMEOV3}$$

$$\frac{de = tag\{oe\}[<: de'] \qquad ov\{tag.mv\} \Rightarrow ov'}{(ov)\{(\textsf{tagOf proto}\ de).mv\} \Rightarrow ov'}\text{T-STATEMEMBER}$$

$$\frac{tag \notin \texttt{tags}(dv) \qquad ov\{tag.mv\} \Rightarrow ov'}{(dv \curlywedge ov)\{tag.mv\} \Rightarrow dv \curlywedge ov'}\text{T-STATEMEMBEROV1}$$

$$\frac{tag \in \texttt{tags}(dv) \qquad dv\{tag.mv\} \Rightarrow dv'}{(dv[\curlywedge\, ov])\{tag.mv\} \Rightarrow dv'[\curlywedge\, ov]}\text{T-STATEMEMBEROV2}$$

$$\frac{tag \neq tag' \qquad ov\{tag'.mv\} \Rightarrow ov' \qquad [dv\{tag.mv\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{tag'.mv\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-STATEMEMBERDV1}$$

$$\frac{(tag\{ov\}[<: dv])\{mv\} \Rightarrow dv'}{(tag\{ov\}[<: dv])\{tag.mv\} \Rightarrow dv'}\text{T-STATEMEMBERDV2}$$

Figure 9: Trait Operations

initializers with the fresh variable. Sequences of state declarations are transformed into recstate blocks. Each state declaration is transformed into a val declaration which binds to a proto representing the uninitialized state, with a `freshtag` expression for generating the state's tag when the declaration is executed.

Our formal semantics defines all of the Plaid language except for module linking and cross language binding. Module linking currently follows the Java standard, including packages, imports, and a classpath

$\texttt{uniqueTags}(ov)$ $\texttt{uniqueMembers}(ov)$ $\texttt{lookup}(x, ov) = mv$ $dv \in ov$ $mv :: tag.x@ov$
$tag <<: tag@ov$ $\texttt{validTagMembers}(ov)$ $\texttt{rename}(n, mv) = mv$ $\texttt{name}(mv) = n$
$\texttt{tags}(ov)$ $\texttt{outerTags}(ov)$ $\texttt{innerTags}(ov)$

$$\frac{tag \notin \texttt{tags}(ov)\ [\ \cup \texttt{tags}(dv)] \qquad [\texttt{tags}(ov) \cap \texttt{tags}(dv) = \emptyset] \qquad \texttt{uniqueTags}(ov) \qquad [\texttt{uniqueTags}(dv)]}{\texttt{uniqueTags}(tag\{ov\}[<: dv])}\ \textsc{UniqueTagsDv}$$

$$\frac{\texttt{tags}(dv) \cap \texttt{tags}(ov) = \emptyset \qquad \texttt{uniqueTags}(dv) \qquad \texttt{uniqueTags}(ov)}{\texttt{uniqueTags}(dv\ \curlyvee\ ov)}\ \textsc{UniqueTagsOv1}$$

$$\frac{[\texttt{uniqueTags}(ov)]}{\texttt{uniqueTags}(mv[\ \curlyvee\ ov])}\ \textsc{UniqueTagsOv2} \qquad \frac{mv_1 :: tag_1.x@ov\ ...\ mv_n :: tag_n.x@ov \qquad tag_i <<: tag_1@ov\ ...\ tag_i <<: tag_n@ov}{\texttt{lookup}(x, ov) = mv_i}\ \textsc{Lookup}$$

$$\frac{\texttt{validTagMembers}(ov) \qquad \exists n(\exists tag\ mv :: tag.n@ov \wedge \exists tag'\ mv' :: tag'.n@ov) \Longrightarrow (tag <<: tag'@ov \vee tag' <<: tag@ov)}{\texttt{uniqueMembers}(ov)}\ \textsc{UniqueMembers}$$

$$\frac{}{dv \in dv}\ \textsc{Leaf1} \qquad \frac{dv \in ov'}{dv \in mv\ \curlyvee\ ov'}\ \textsc{Leaf2} \qquad \frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in dv'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\ \curlyvee\ ov'']}\ \textsc{Leaf3}$$

$$\frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in ov'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\ \curlyvee\ ov'']}\ \textsc{Leaf4} \qquad \frac{tag\{[ov_1\ \curlyvee\ ]mv[\ \curlyvee\ ov_1]\} <: dv \in ov}{mv :: tag@ov}\ \textsc{MbrInTag}$$

$$\frac{tag\{ov'\} <: dv \in ov \qquad tag' \in \texttt{outerTags}(tag\{ov'\} <: dv)}{tag <<: tag'@ov}\ \textsc{CaseOf} \qquad \frac{name(mv) \notin names \qquad [\texttt{validTagMembers}(names \cup name(mv), ov')]}{\texttt{validTagMembers}(names, mv[\ \curlyvee\ ov'])}\ \text{VTM1}$$

$$\frac{\texttt{validTagMembers}(\emptyset, ov) \qquad [\texttt{validTagMembers}(\emptyset, dv)] \qquad [\texttt{validTagMembers}(names, ov')]}{\texttt{validTagMembers}(names, (tag\{ov\}[<: dv])[\ \curlyvee\ ov'])}\ \text{VTM2}$$

$$\frac{}{n = \texttt{name}(\mathsf{val}\ n = v)}\ \textsc{Name1} \qquad \frac{}{m = \texttt{name}(\mathsf{method}\ m(\overline{x})\{e\})}\ \textsc{Name2} \qquad \frac{}{\texttt{rename}(a, \mathsf{val}\ n = v) = \mathsf{val}\ a = v}\ \textsc{Rename1}$$

$$\frac{}{\texttt{rename}(n, \mathsf{method}\ m(\overline{x})\{e\}) = \mathsf{method}\ n(\overline{x})\{e\}}\ \textsc{Rename2} \qquad \frac{}{\texttt{tags}(ov) = \texttt{innerTags}(ov) \cup \texttt{outerTags}(ov)}\ \textsc{Tags}$$

$$\frac{}{\texttt{outerTags}(tag\{ov\}[<: dv]) = \{tag\}\ [\cup \texttt{outerTags}(dv)]}\ \textsc{OuterDv}$$

$$\frac{}{\texttt{outerTags}(dv[\ \curlyvee\ ov]) = \texttt{outerTags}(dv)\ [\cup \texttt{outerTags}(ov)]}\ \textsc{OuterOv1}$$

$$\frac{}{\texttt{outerTags}(mv[\ \curlyvee\ ov]) = \emptyset\ [\cup \texttt{outerTags}(ov)]}\ \textsc{OuterOv2}$$

$$\frac{}{\texttt{innerTags}(tag\{ov\}[<: dv]) = \texttt{tags}(ov)\ [\cup \texttt{innerTags}(dv)]}\ \textsc{InnerDv}$$

$$\frac{}{\texttt{innerTags}(dv[\ \curlyvee\ ov]) = \texttt{innerTags}(dv)\ [\cup \texttt{innerTags}(ov)]}\ \textsc{InnerOv1}$$

$$\frac{}{\texttt{innerTags}(mv[\ \curlyvee\ ov]) = \emptyset\ [\cup \texttt{innerTags}(ov)]}\ \textsc{InnerOv2}$$

Figure 10: Helper Judgements

Declarations	D	::=	$SD \mid \mathsf{method}\ m(\overline{x})\{SE\} \mid$
			$\mathsf{val}\ f = SE$
State Decl.	SD	::=	$\mathsf{val}\ s = S \mid \mathsf{state}\ s = S \mid$
			$\mathsf{state}\ s\ \mathsf{case\ of}\ s\{\overline{TO}\} = S$
States	S	::=	$\mathsf{freeze}(SE) \mid \{\overline{D}\} \mid s\{\overline{T}\} \mid$
			$S\ \mathsf{with}\ S \mid SE.s \mid s$
Trait Ops	TO	::=	$\backslash n \mid n \rightarrow n' \mid$
			$\mathsf{val}\ f = SE \mid \mathsf{val}\ s = S \mid$
			$\mathsf{val}\ s.f = SE \mid \mathsf{val}\ s.t = S \mid$
			$\mathsf{method}\ m(\overline{x})\{SE\} \mid$
			$\mathsf{method}\ s.m(\overline{x})\{SE\}$
Expression	SE	::=	$x \mid \mathsf{let}\ x = SE\ \mathsf{in}\ SE \mid SE.f \mid$
			$SE(\overline{SE}) \mid SE.m(\overline{SE}) \mid$
			$SE \leftarrow S \mid SE \twoheadleftarrow S \mid \mathsf{new}\ S \mid$
			$\mathsf{match}(SE)\{\overline{C}\} \mid$
Case	C	:=	$\mathsf{case}\ SE.s\ \{SE\} \mid$
			$\mathsf{case}\ s\ \{SE\} \mid \mathsf{default}\ \{SE\}$
Compil. Unit	CU	::=	$\overline{D}$

Figure 11: Source Syntax

for loading elements. Plaid primitives are defined using Java classes and methods, which can be directly accessed in Plaid via their fully-qualified Java names. Details of both of these aspects of Plaid are discussed in more detail in the Plaid language definition [17].

6 Discussion and Future Work

The primary contribution of the Plaid language is providing a way for programmers to express state machine abstractions directly in the source code of their programs. Plaid supports the major state modeling features of Statecharts, including state hierarchy, or-states, and and-states. The explicit representation of states makes the design more salient in the code, enhancing programmer understanding. For example, the separation of members into different abstract states helps programmers quickly learn what operations are available in each state. In the future, visualization tools that leverage explicit state constructs to automatically generate statecharts from Plaid code could provide even greater benefits.

Plaid has the potential to make code more reliable. Not only do explicit states help programmers understand libraries better, avoiding errors in the first place; the runtime will also verify that the libraries are used correctly according to their state abstractions. Even a "method not available in this state" error is better than a silent corruption, but in future work, we believe we can leverage explicit states to do much better. For example, a state-related error message could be paired with a suggestion about what methods could be called in order to move the object into a correct state.

Plaid's trait-like state composition model provides a way of reusing not just fields and methods, but state abstractions. This additional layer of reuse has the potential to reduce redundancy in code and specifications, while enhancing developer productivity. The confidence that comes with the error checking in Plaid's state model may also help developers to evolve and refactor software with greater confidence.

In future work, we plan to build more programs with Plaid in order to investigate the possible benefits outlined above. We are also developing a gradual type system that can complement Plaid's dynamic state checking with static checking, where desired by programmers [28]. We believe Plaid demonstrates a new kind of language, and we are excited to explore the consequences that language may entail.

References

[1] Jonathan Aldrich, Karl Naden, and Éric Tanter. Modular composition and state update in Plaid. In *Proc. MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI, 2010.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. Onward*, 2009.

[3] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming 2011*.

[4] Andi Bejleri, Jonathan Aldrich, and Kevin Bierhoff. Ego: Controlling the Power of Simplicity. In *Proc. Foundations of Object-Oriented Languages*, 2006.

[5] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2):83–108, 2008.

[6] Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. A Mechanism for Flexible Dynamic Trait Replacement. In *Proc. Formal Techniques for Java-like Programs*, 2009.

[7] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Proc. Foundations of Software Engineering*, 2005.

[8] Bard Bloom, Paul Keyser, Ian Simmonds, and Mark Wegman. Ferret: Programming language support for multiple dynamic classification. *Computer Languages, Systems and Structures*, 35(3):306 – 321, 2009.

[9] Tom Van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang De Meuter. Adding state and visibility control to traits using lexical nesting. In *Proc. European Conference on Object-Oriented Programming*, 2009.

[10] Robert Deline and Manuel Fahndrich. Typestates for Objects. In *Proc. European Conference on Object-Oriented Programming*, 2004.

[11] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Re-classification. In *Proc. European Conference on Object-Oriented Programming*, 2001.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A.P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973.

[16] Ciera Jaspan. Proper plugin protocols: Cost-effective verification of frameworks. Technical Report CMU-ISR-11-101, Institute for Software Research, Carnegie Mellon University, April 2011. Thesis Proposal, originally accepted April 2010.

[17] Jonathan Aldrich. *The Plaid Language: Dynamic Core Specification*, 2010. `http://plaid-lang.googlecode.com/hg/docs/spec/current/current.pdf`.

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

[19] D. Malayeri and J. Aldrich. CZ: multiple inheritance without diamonds. *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[20] Barbara Pernici. Objects with Roles. In *Proc. Conference on Office Information Systems*, 1990.

[21] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[22] John Reppy and Aaron Turon. Metaprogramming with traits. In *Proc. European Conference on Object-Oriented Programming*, 2007.

[23] Asher Sterkin. State[chart]-Oriented Programming. In *Proc. Multiparadigm Programming with Object-Oriented Languages*, 2008.

[24] S. Stork, P. Marques, and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 933–940. ACM, 2009.

[25] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

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

[27] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 1987.

[28] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming 2011*.

Appendices

A Unique members proof

Theorem 1: If $\texttt{uniqueTags}(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then $\texttt{uniqueTags}(ov_3)$.
Proof: By induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$.
Case SU-List:
$\texttt{uniqueTags}(ov_o)$ by the induction hypothesis.
Case SU-MV:
$\texttt{uniqueTags}(ov_t)$ by assumption.
$\texttt{uniqueTags}(ov_t \curlywedge mv_u)$ by rule UniqueTagsOV2.
Case SU-AddH:
$\texttt{uniqueTags}(ov_t)$ by assumption.
$\texttt{uniqueTags}(ov_t \curlywedge mv_u)$ by rule UniqueTagsOV1.
Case SU-MatchDim:
$\texttt{uniqueTags}(ov_t)$ by assumption.
$\texttt{uniqueTags}(dv_r)$ by the induction hypothesis.
$\texttt{uniqueTags}(ov_t \curlywedge dv_r)$ by rule UniqueTagsOV1.
Case SU-MatchInner:
$\texttt{uniqueTags}(tag\{ov\} <: dv)$ by assumption.
$\texttt{uniqueTags}(ov_r)$ by the induction hypothesis.
$tag \notin \texttt{tags}(dv) \wedge tag \notin \texttt{tags}(ov) \wedge \texttt{uniqueTags}(dv) \wedge$
$\texttt{tags}(ov) \cap \texttt{tags}(dv) = \emptyset$ by inversion of
$\texttt{uniqueTags}(tag\{ov\} <: dv)$.
$tag \notin \texttt{tags}(ov_r)$ by Lemma 3.
$\texttt{tags}(ov_r) \cap \texttt{tags}(dv) = \emptyset$ by Lemma 2.
$\texttt{uniqueTags}(tag\{ov_r\} <: dv)$ by rule UniqueTagsDv.
Case SU-MatchSuperInner:
$tag \notin \texttt{tags}(dv_r)$ by Lemma 3.
$\texttt{tags}(ov) \cap \texttt{tags}(dv_r) = \emptyset$ by Lemma 2.
$\texttt{uniqueTags}(tag\{ov\} <: dv_r)$ by UniqueTagsDv.
Case SU-MatchSuper:
$\texttt{uniqueTags}(dv_r)$ by the induction hypothesis.
Case SU-Match:
$\texttt{uniqueTags}(dv_{sub} <: tag\{ov\} <: dv)$ by rule UniqueTagsDv.
□

Lemma 1: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3$ then
$\texttt{tags}(ov_3) \subseteq \texttt{tags}(ov_1) \cup \texttt{tags}(ov_2)$
Proof: By easy induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$
□

Lemma 2: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge$
$\texttt{tags}(ov) \cap \texttt{tags}(ov_1) = \emptyset \wedge \texttt{tags}(ov) \cap \texttt{tags}(ov_2) = \emptyset$ then $\texttt{tags}(ov) \cap \texttt{tags}(ov_3) = \emptyset$
Proof:

$\texttt{tags}(ov) \cap (\texttt{tags}(ov_1) \cup \texttt{tags}(ov_2)) = \emptyset$
$\texttt{tags}(ov_3) \subseteq \texttt{tags}(ov_1) \cup \texttt{tags}(ov_2)$ by Lemma 1.
$\texttt{tags}(ov) \cap \texttt{tags}(ov_3) = \emptyset$
□

Lemma 3: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge tag \notin \texttt{tags}(ov_1) \wedge$
$tag \notin \texttt{tags}(ov_2)$ then $tag \notin \texttt{tags}(ov_3)$
Proof:
$tag \notin (\texttt{tags}(ov_1) \cup \texttt{tags}(ov_2)) = \emptyset$
$\texttt{tags}(ov_3) \subseteq \texttt{tags}(ov_1) \cup \texttt{tags}(ov_2)$ by Lemma 1.
$tag \notin \texttt{tags}(ov_3)$
□

Theorem 2: If we $e@H \mapsto e'@H'$ and $\forall \ell \in H.\texttt{uniqueTags}(H[\ell])$, then $\forall \ell \in H'.\texttt{uniqueTags}(H'[\ell])$
Proof: By induction on $e@H \mapsto e'@H'$
Case E-New and E-Replace:
$\forall \ell' \in H[\ell \rightsquigarrow ov].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis and rule premise.
Case E-Su:
$\texttt{uniqueTags}(ov_3)$ by Theorem 1.
$\forall \ell' \in H[\ell \rightsquigarrow ov_3].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis.
All Other Rules:
Heap does not change.

B Source Translation Rules

The rules in figures 12 and 13 below describe how to translate a program written in the Plaid source language given in figure 11 to the internal language defined in figure 4.

$$\boxed{CU \rightsquigarrow e \qquad \overline{D} \rightsquigarrow oe \qquad SD \rightsquigarrow \mathsf{val}\ s \rhd s' = se \qquad S \rightsquigarrow oe \qquad \overline{TO} \rightsquigarrow \overline{to} \qquad \texttt{groupStates}(\overline{D})}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{CU \rightsquigarrow \mathsf{let}\ top = \mathsf{new}\ (\mathsf{proto}\ oe)\ \mathsf{in}\ top.main()}\text{TR-CU}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE_f \rightsquigarrow e_f \qquad f' = freshname \qquad oe'_d = oe_d[f'/f]}{\mathsf{val}\ f = SE_f, \overline{D} \rightsquigarrow \mathsf{val}\ f \rhd f' = e_f \curlyvee oe'_d}\text{TR-DeclField}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE \rightsquigarrow e}{\mathsf{method}\ m(\overline{x})\{SE\}, \overline{D} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\} \curlyvee oe_d}\text{TR-DeclMethod}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad \overline{SD \rightsquigarrow \mathsf{val}\ s \rhd s' = sd}}{\{\overline{SD}\}, \overline{D} \rightsquigarrow (\mathsf{recstate}\{\overline{\mathsf{val}\ s \rhd s' = \mathsf{proto}\ sd}\} \curlyvee oe_d)[\overline{s'/s}]}\text{TR-DeclStates}$$

$$\frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto}\ \mathsf{freshtag}\{oe\})}\text{TR-StateTag} \qquad \frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{val}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = oe}\text{TR-StateVal}$$

$$\frac{s_s\{\overline{TO}\} \rightsquigarrow de \qquad S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s\ \mathsf{case\ of}\ s_s\{\overline{TO}\} = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto}\ \mathsf{freshtag}\{oe\} <: de)}\text{TR-StateCase}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{\{\overline{D}\} \rightsquigarrow oe}\text{TR-StateDecl} \qquad \frac{S_1 \rightsquigarrow oe_1 \qquad S_2 \rightsquigarrow oe_2}{S_1\ \mathsf{with}\ S_2 \rightsquigarrow oe_1 \curlyvee oe_2}\text{TR-StateWith}$$

$$\frac{SE \rightsquigarrow e}{\mathsf{freeze}(SE) \rightsquigarrow \mathsf{freeze}(e')}\text{TR-StateFreeze} \qquad \frac{SE \rightsquigarrow e}{SE.s \rightsquigarrow e.s}\text{TR-StateSelect} \qquad \frac{}{s \rightsquigarrow s}\text{TR-StateName}$$

$$\frac{\overline{TO} \rightsquigarrow \{\overline{to}\}}{s\{\overline{TO}\} \rightsquigarrow s\{\overline{to}\}}\text{TR-StateInit} \qquad \frac{TO \rightsquigarrow to \qquad \overline{TO} \rightsquigarrow \overline{to}}{TO, \overline{TO} \rightsquigarrow to, \overline{to}}\text{TR-SpecGeneral}$$

$$\frac{\mathsf{val}\ f = SE \rightsquigarrow \mathsf{val}\ f \rhd f' = e}{\mathsf{val}\ [s.]f = SE \rightsquigarrow [s.](\mathsf{val}\ f = e)}\text{TR-SpecField} \qquad \frac{\mathsf{method}\ m(\overline{x})\{E\} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\}}{\mathsf{method}\ [s.]m(\overline{x})\{E\} \rightsquigarrow [s.](\mathsf{method}\ m(\overline{x})\{e\})}\text{TR-SpecMethod}$$

$$\frac{S \rightsquigarrow oe}{\mathsf{val}\ [s.]t = S \rightsquigarrow [s.](\mathsf{val}\ t = oe)}\text{TR-SpecState} \qquad \frac{}{\backslash n \rightsquigarrow \backslash n}\text{TR-SpecRemove}$$

$$\frac{}{n \to n' \rightsquigarrow n \to n'}\text{TR-SpecRename} \qquad \frac{\texttt{groupStates}(SD', \overline{D}) = \{\overline{sd_i = SD_i}\}, \overline{D'} \qquad s = \texttt{name}(SD)}{\texttt{groupStates}(SD, SD', \overline{D}) = \{s = SD, \overline{sd_i = SD_i}\}, \overline{D'}}\text{TR-GStAdd}$$

$$\frac{s = \texttt{name}(SD) \qquad \overline{D} = \cdot \vee (\overline{D} = (D, \overline{D'}) \wedge D \neq SD)}{\texttt{groupStates}(SD, \overline{D}) = \{s = SD\}, \texttt{groupStates}(\overline{D})}\text{TR-GStStart}$$

$$\frac{D \neq SD}{\texttt{groupStates}(D, \overline{D}) = D, \texttt{groupStates}(\overline{D})}\text{TR-GMember}$$

Figure 12: Translate Declarations

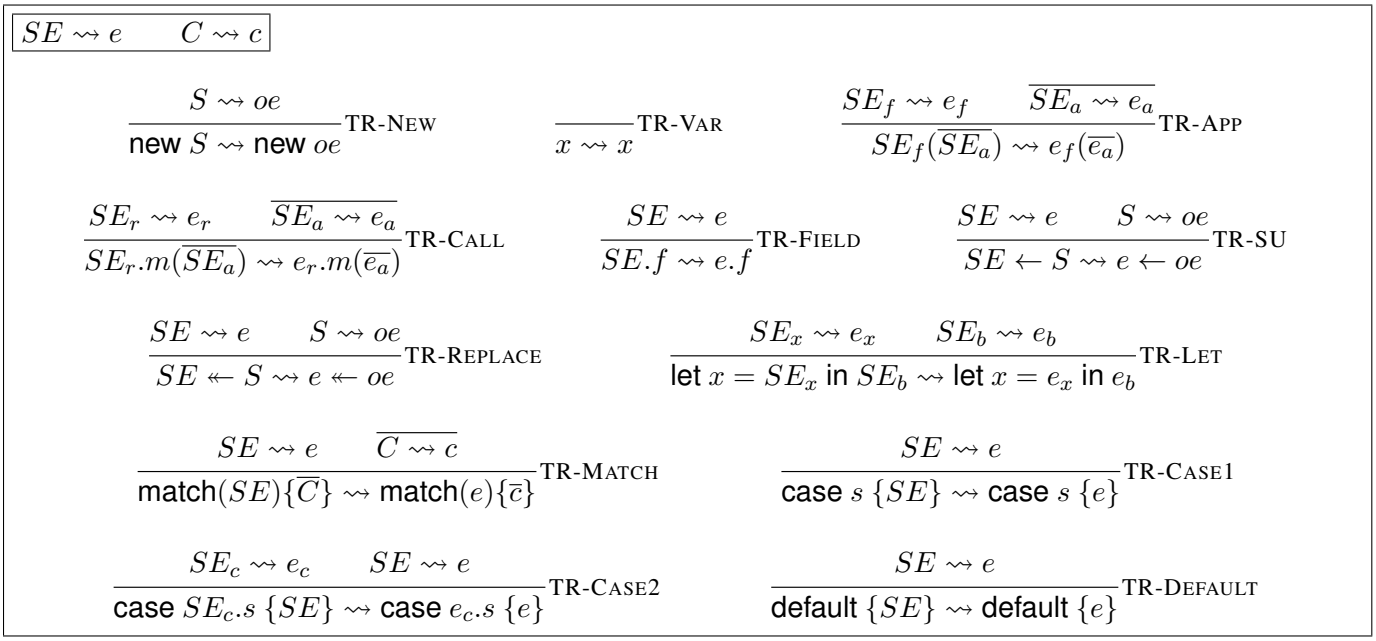

Figure 13: Translate Expressions