A Theory of Typestate-Oriented Programming

Darpan Saini
Carnegie Mellon University
dsaini@cs.cmu.edu

Joshua Sunshine
Carnegie Mellon University
sunshine@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

ABSTRACT

Engineers in many disciplines use state machines to reason about system changes, and many object-oriented libraries require their clients to follow state machine protocols. No existing language, however, has native support for state machines, and programmers often lose productivity and introduce errors when trying to understand and follow interaction protocols. The Plaid language extends the object paradigm with explicit states and state transitions, in order to better model object state transitions. In this paper, we present $Plaid_{core}$, a core calculus for Plaid, which uses states and permissions to statically guarantee that clients use object protocols correctly.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics

General Terms

Languages, Theory

Keywords

Typestate Oriented Programming, Verification

1. INTRODUCTION

As object-oriented programming has entered the mainstream, the widespread availability of high-quality reusable libraries and frameworks has enabled an unprecedented degree of reuse. While programmers in the past often focused on algorithm and data structure details, today's developers are more often focused on stitching together components such as libraries and framework APIs with application specific logic. In order to gain maximum leverage from component reuse, it is important that programmers use components correctly.

Many reusable object-oriented components are stateful and define protocols on their usage. In previous work we proposed typestate-oriented programming [2] as an extension to the object paradigm that models objects not with classes but in terms of their changing state. An object's typestate [13] is like its class with its own interface, representation and behavior. But unlike object-oriented programming where the class never changes, in typestate oriented programming an object's typestate is allowed to change over its lifetime.

For example, a File object has states `open` and `closed`, and we can only write to or read from it when it is `open`. In the `open` state a call to close causes a state transition to `closed`, and in the `closed` state the only operation available is to (re-)open the file. In today's languages such protocols are mostly implicit and/or documented informally. However, in $Plaid_{core}$ protocols can be enforced by the type system. For the same object, methods such as read, write, close are available in the `open` state and the only method available in the `closed` state is open.

There have been various typestate-based analyses written in the recent past [8, 4]. These checkers have been successful in checking reasonably large programs [5] in languages like Java. However, there are compelling reasons to carry the idea of typestate into the programming language (summarized from [2]):

- Language influences how programmers think and go about their tasks. By explicitly including typestate in the programming language we encourage programmers to think in terms of states, which should ultimately lead to more effective designs.
- Having typestate in the programming language can lead to simplicity of reasoning. Alluding again to the file example, in a regular language an invariant of the closed state is that the file pointer must point to null. As we will see in Section 2, in $Plaid_{core}$ a file pointer simply does not exist in the ClosedFile state, thus making reasoning about programs simpler.

In this paper we present a core calculus called $Plaid_{core}$ for typestate-oriented programming. Unlike most previous work on typestate, we make states a first class element of the language and allow the state of an object to change. Depending on the current state of an object, only methods that were defined in the state can be called. Statically tracking the changing state of an object is notoriously hard in the presence of aliasing. In order to achieve this we make use of a permission [6] based type system.

The contributions of this paper are as follows:

- A novel language design called $Plaid_{core}$, which supports typestate-oriented programming, and examples

ECOOP '2010 Maribor, Slovenia EU

that illustrate its usefulness. $Plaid_{core}$ models objects as records and provides a novel state change operation.

- A type system that statically tracks the state of objects. Unlike prior type systems for typestate, our type system is structural. Like [14] we adapt a subset of access permissions [4] to the setting of the lambda calculus.

The rest of this paper is organized as follows — In section 2 we introduce the language with an example. Section 3 describes the syntax of the language, while section 4 describes its type system. We conclude with related work in section 5.

2. LANGUAGE BY EXAMPLE

In this section, we describe the features of $Plaid_{core}$ using an example. $Plaid_{core}$ is an extension of the lambda calculus with states (modeled using records), references, permissions, state change operations and has a structural type system. A structural type system provides many benefits [12], one among them being unanticipated code reuse. To provide such benefits to programmers we envision the full Plaid language to be structurally typed and hence it is a natural choice for us to model a structurally typed core calculus.

The example[1] is a simple encoding of files, where a file can either be in the open or closed state. Each state is represented with a type that contains only relevant fields[2] as shown in listing 1. An object is allowed to change its type (state) over its lifetime, and only those fields that are defined for its *current* type are available to clients.

```
1  type OpenFile =
2    state of {
3      read : (imm of) → imm int
4      close : (of ≫ ClosedFile) → unit
5      ptr : imm CFilePtr
6    }
7
8  type ClosedFile =
9    state cf {
10     open : (cf ≫ OpenFile) → unit
11   }
```

Listing 1: File States in $Plaid_{core}$

State definitions. Listing 1 declares two states `OpenFile` and `ClosedFile`. A similar File example was discussed in [2], but here we adapt it to the prototype-based, structurally-typed setting of $Plaid_{core}$. These type abbreviations abstractly capture the open and closed states of a file.

The `OpenFile` state has three fields - `read`, `close` (which have functions inside them) and `ptr`. The `read` function takes as argument an immutable `OpenFile` and returns an **int**. In the `close` function, the ≫ symbol separates the input and output types of the function's argument. In this case, the `close` function of `OpenFile` accepts its argument (conceptually the method receiver) in state *of* (the recursively bound name for `OpenFile`), but when the function returns, the object will be in state `ClosedFile`. `ptr` returns an immutable operating system level file pointer.

In addition to the type, a function also takes permissions to its arguments. We provide defaults to ease the burden of specifying permissions on the programmer. An unspecified permission means unique, except for functions, which are wrapped inside immutable permissions by default, and a missing ≫ means that the function does not change permissions to the arguments or free variables and returns them unchanged. Thus, the fully expanded type of the `open` method in `ClosedFile` is

imm (**uni** *cf* ≫ **uni** `OpenFile`) → **unit**

Access Permissions. Like the language presented in [2] $Plaid_{core}$ supports changing the state of objects and tracks state changes using access permissions. For simplicity we restrict ourselves to the unique and immutable kinds of permissions. A unique permission means that we have the only reference and we are allowed to change its state. An immutable permission means that there may be other aliases to this reference but no one is allowed to change the state of the reference.

State Change Operations. Listing 2[3] describes how states can be defined in $Plaid_{core}$. Given that the definition of `OpenFile` and `CloseFile` is mutually recursive, one way to define them is with the fixed point of a function that changes the state of a `ClosedFile` to an `OpenFile` using a `letrec`[4] like construct. We start by defining a function *openf* that takes a `ClosedFile` and changes it to an `OpenFile` using the state change operator ← (line 2). The function inside `close` recursively uses *openf* for its `open` field.

```
1  letrec openf = λthis : ClosedFile ⇒
2    this ← state of {
3      read = //use ptr to read the file
4
5      close = λthis : OpenFile ⇒
6        this ← state cf {
7          open = openf
8        }
9
10     ptr = //return low-level file pointer
11   }
```

Listing 2: Defining states in $Plaid_{core}$

Listing 3 describes how Files can be used by a client in $Plaid_{core}$. We first create a new object f and change its state to `ClosedFile`. While defining `ClosedFile` we use the previously defined *openf* variable. To actually read from the file we allude to a helper function *readFromFile* that takes an `OpenFile` and returns an integer. To call *readFromFile*, we pass an open file to it (line 19).

The call to `computeBase` in `readFromFile` presents a potential source for an error. Since f is in scope, it is possible that *computeBase* can close the file rendering the call

[1] In describing the example we take certain liberties for making it easier to understand. For instance, even though $Plaid_{core}$ does not explicitly support statement sequences or return statements, we will use them. We also use type abbreviations that are not part of $Plaid_{core}$.

[2] As a modeling choice, objects in $Plaid_{core}$ do not contain methods but only fields; methods are wrapped inside fields as functions.

[3] Type annotations for fields have been elided to reduce clutter.

[4] **letrec** is not a primitive in the language but can be encoded using recursive types

to `read` erroneous. In a regular language a similar situation could arise if there was a global reference to f. Even worse such an error would only be flagged at runtime. In $Plaid_{core}$ however, access permissions help us catch such errors at compile time.

```
1 f = new
2 f =
3   f ← state cf {
4     open = openf
5     }
6
7 computeBase =
8   λ_ : unit [f: Openfile ≫ f: CloseFile] ⇒
9     i = //...some computation
10    f.close(f)
11    return i
12
13 readFromFile =
14   λf : OpenFile
15     ⇒ i = computeBase() + f.read(f) // error!
16       return i
17
18 f.open(f)
19 readFromFile(f)
```

Listing 3: Using Files in $Plaid_{core}$

The code in listing 3 does not compile in $Plaid_{core}$ because `read` requires f to be in the `OpenFile` state, but *computeBase* closes f before `read` is called. To rectify the situation we change the signature of *readFromFile* to accept an immutable `OpenFile`. Now we are guaranteed that there is no unique permission passed to *computeBase* and hence it cannot close the file.

```
1 computeBase =
2   λ_ : unit ⇒
3     i = //...some computation
4     // cannot close the file!
5     return i
6
7 readFromFile =
8   λf : imm OpenFile
9     ⇒ i = computeBase() + f.read(f);
10      return i;
```

Listing 4: Fixed *readFromFile*

Instead of explicitly passing the parameter f to *readFromFile* we could also write a function that uses variables currently in scope (like the code for *computeBase* in Listing 3). Such a function is described in Listing 5. *readFromFile*2 uses permissions to variables that occur free in its body.

```
1 readFromFile2 =
2   λ_ : unit
3     ⇒ i = f.read(f);
4       return i;
```

Listing 5: Accessing in scope permissions

We write type of such a function with permissions to free variables in [ ]. So the type of *readFromFile*2 is

imm (unit) [f : **imm** OpenFile ≫ f : **imm** OpenFile] → **imm int**

3. FORMAL LANGUAGE

The syntax[5] of the formal language is summarized in Figure 1. In place of a simple typing context containing just the types of variables, we use a linear context Δ containing both the permission kinds *perm* and the types T (together called the *Permission*) of variables. Later while defining the dynamic semantics, we will extend the same context to contain permissions to memory locations.

Expressions. The language is restricted to A-normal form [10]. All expressions must be bound to variables using the let syntactic form, since the type system relies on sequencing to track the state of variables. Both variables and function abstractions are values and application is written using juxtaposition vv. A function abstraction is of the form $\lambda x{:}P, \Delta{\Rightarrow}e$, where P is the *Permission* to the argument x, and Δ contains permissions to other free variables in e (our concrete syntax placed this in [ ] brackets, but we omit this from the formalism). This allows e to refer to variables in scope, like curried arguments.

States are modeled using records where each record is a set of declarations of the form state s $\{\overline{D}\}$. Each field of the record is written as $f : P = v$ where f represents the name of the field, P the *Permission* and v the value. The variable s can occur free in $\overline{D}$ making the definition recursive. For simplicity, we choose to model methods by wrapping them as lambdas inside fields. The *new* expression is used to create a new record.

$v.f$ and $v!f$ are ways to deference a field depending on the permission kinds of v and f. If both v and f are unique then in order to get a unique permission to the expression result, we must remove the unique permission from the field. This is denoted with the destructive field read expression $v!f$, which removes the field f entirely from the record v, thereby changing v's *state*. In other cases, f is immutable, and the non-destructive immutable field read expression $v.f$ is used to return an immutable reference to the object the field points to. In this case the field permission within v is unaffected and hence v's state remains unchanged. In the case when v is immutable and f is unique we disallow a call to $v.f$. This is because such a case unnecessarily complicates this particular type system without providing any greater expressive power. If such a call is allowed, we have two options — on the one hand, we can remove the unique field from v, but that would be inconsistent with the semantics of immutable and on the other, we can return an immutable permission to f, but then a unique permission to it will still remain inside v. If we go with the latter, the overall invariant of the system must take this into account, which would otherwise have been that for any unique permission there must be no other permissions and for an immutable permission there must be no other unique permissions. If it is required to store the unique field inside v then it must first be made immutable (as will become evident in section 4.1, this can be done using a let-binding).

$v \leftarrow$ state s $\{\overline{D}\}$ is the state change operation that changes the state of v from what it was before to state s $\{\overline{TD}\}$. As we will see in section 4, state s $\{\overline{TD}\}$ can be derived from

[5]We interchangeably use uni for unique and imm for immutable to save space.

Expressions	e	::=	$\mathsf{let}\ x = e\ \mathsf{in}\ e$
			v
			vv
			new
			$v.f$
			$v!f$
			$v \leftarrow \mathsf{state}\ s\ \{\overline{D}\}$
Values	v	::=	x
			$\lambda x{:}P, \Delta \Rightarrow e$
			$()$
Types	T	::=	s
			$\mathsf{state}\ s\ \{\overline{TD}\}$
			$\Pi x.(P, \Delta \gg P', \Delta' \rightarrow P_r)$
			unit
Declarations	D	::=	$f{:}P = v$
Type Declarations	TD	::=	$f{:}P$
Permissions	P	::=	$perm\ T$
Perm. kinds	$perm$	::=	$\mathsf{unique} \mid \mathsf{immutable}$
Perm. Contexts	Δ	::=	$\Delta, x : P \mid \varnothing$

Figure 1: Syntax

$\mathsf{state}\ s\ \{\overline{D}\}$ in a straightforward manner.

Types. The type of a record is of the form $\mathsf{state}\ s\ \{\overline{TD}\}$ where the variable s can occur free in $\overline{TD}$. Each *type declaration* TD is written as $f : P$. A permission P is the combination of a permission kind and the type.

The arrow type $\Pi x.(P, \Delta \gg P', \Delta' \rightarrow P_r)$ is the type of a lambda abstraction, where P and P' are the input and output permissions of the argument, Δ and Δ' are the input and output permissions to the free variables in e (they were in brackets [ ] in the concrete syntax, which we omit here) and P_r is the permission of the return value. We use a dependent type here to facilitating currying. A curried function takes several arguments; all but the last of which come with no permissions. The dependent type allows us to bind a variable (in P_r) so that curried functions to the right of the $\rightarrow$ can refer to it in their permissions list.

Permissions. In this formalization we support two kinds of access permissions — unique and immutable. A unique permission to a variable means that there exists only one permission to that variable in the context Δ and the variable is allowed to change state. An immutable permission to a variable means that other immutable permissions to the same variable are allowed to co-exist in Δ but the variable is not allowed to change state.

In other formalizations [4], permissions such as shared, full and pure have been used. Ultimately the full Plaid language will support some or all of these permissions, but for this initial formalization effort we have decided to keep the system simple and only support two different permission kinds.

4. TYPE SYSTEM

In this section we describe the static and dynamic semantics of the language.

4.1 Static Semantics

The typing rules for the language are summarized in Figure 2. The typing judgment is of the form

$$\Delta \vdash e : P \dashv \Delta$$

The Δ to the left of the $\vdash$ represents the incoming context for typing expression e and the Δ to the right of the $\dashv$ represents the outgoing context that remains after typing e. The incoming context may contain many more permissions than are required to type e. We *thread* these unused permissions to the outgoing context for subsequent expressions.

The T-Var rule returns the permission to the variable x from the context $(\Delta, x : P)$ and threads through Δ.

The T-Abs rule is for typing a lambda abstraction. Defining a lambda requires no permissions, so the context Δ is passed through unchanged. The body of the function, however, is allowed to assume a permission for the argument as well as other permissions to free variables in the function body. Assuming $\Delta_1, x : P$, if we can type the expression e such that $e : P_r$, the outgoing context is Δ_1' and e changes the permission P to P' then we say that the function is well-typed. We also insist that all free variables in Δ_1 are present in Δ, although permissions to them at the time of definition maybe different from when the function is called. This is to prevent any accidental dynamic scoping of these permissions. Finally, we give the function an immutable permission kind because every expression or value must have a permission kind and a type.

The T-App rule is for typing a lambda application. The initial incoming context Δ is used to type the value v_2. Given the remaining context Δ' we type the value v_1 such that it has an arrow type and Δ_1 is a subset of Δ'. The output context of the rule are the permissions that the abstraction returns (Δ_1', P') in addition to the part of Δ' it didn't require (Δ_2).

The T-Let rule is for typing a let expression. Before the let binding happens, the incoming context Δ_1 can be *relaxed* through the judgment $\Delta_1 \vdash \Delta_1'$ (defined in Figure 4). This is required because a unique permission can be passed where ever we need an immutable . We perform permission relaxation in the let rule because that prevents us from having to worry about it any place else, since we can always let-bind an expression if permission splitting is required. The resulting context Δ_1' is used to type e_1 which results in Δ_2 as the output context. The let expression is well typed if assuming permissions $\Delta_2, x : P$ we can type e_2 and thread through Δ_3. We require that Δ_3 does not contain a permission for x as x goes out of scope after the let.

The T-Update rule is for typing state change operations. We first check if we have a unique permission to v. For the state change operation to work we must make sure that we have appropriate permissions to all values declared in $\overline{D}$. For this we use the helper function typecheck (Figure 3). For every declaration $f : P = v$ in $\overline{D}$, the typecheck function checks if $v : P$. The result of the state change operation is a unique permission to $\mathsf{state}\ s\ \{\overline{f : P}\}$, where $\overline{f : P}$ is straightforwardly derived from $\overline{D}$ by taking away values. Note that the input to the typecheck function is $\overline{D}$ with $\mathsf{state}\ s\ \{\overline{D}\}$ substituted for s. This is done because $\overline{D}$ can contain declaration types of the form $f : s$.

The T-New rule returns a new unique *p*ermission to an empty record without disturbing the incoming context.

The T-Call-Field-Imm rule is for typing a field deference of an immutable field of a value v. We first check if we have a permission to v in Δ, followed by checking if f is an

$\boxed{\Delta \vdash e : P \dashv \Delta}$

$$\frac{}{\Delta, x : P \vdash x : P \dashv \Delta}\text{T-Var} \qquad \frac{\Delta_1, x : P \vdash e : P_r \dashv \Delta_1', x : P' \qquad \mathsf{domain}(\Delta_1) \subset \mathsf{domain}(\Delta)}{\Delta \vdash \lambda x{:}P, \Delta_1 \Rightarrow e : \mathsf{immutable}\ (\Pi x.(P, \Delta_1 \gg P', \Delta_1' \to P_r)) \dashv \Delta}\text{T-Abs}$$

$$\frac{\Delta \vdash v_2 : P \dashv \Delta' \qquad \Delta' = \Delta_1, \Delta_2 \qquad \Delta' \vdash v_1 : \mathsf{immutable}\ (\Pi x.(P, \Delta_1 \gg P', \Delta_1' \to P'')) \dashv \Delta'}{\Delta \vdash v_1 v_2 : P'' \dashv \Delta_2, \Delta_1', v_2 : P'}\text{T-App}$$

$$\frac{\Delta_1' \vdash_\Delta \Delta_1 \qquad \Delta_1 \vdash e_1 : P \dashv \Delta_2 \qquad \Delta_2, x : P \vdash e_2 : P' \dashv \Delta_3}{\Delta_1' \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : P' \dashv \Delta_3}\text{T-Let}$$

$$\frac{\Delta_1 \vdash v : \mathsf{unique}\ T \dashv \Delta_2 \qquad \overline{D} = \overline{f : P = v} \qquad \Delta_2 \vdash \mathsf{typecheck}(\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}) \dashv \Delta_3}{\Delta_1 \vdash v \leftarrow \mathsf{state}\ s\ \{\overline{D}\} : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}) \dashv \Delta_3}\text{T-Update}$$

$$\frac{}{\Delta \vdash \mathsf{new} : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \dashv \Delta}\text{T-new}$$

$$\frac{\Delta_1 \vdash v : perm\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2 \qquad (f : \mathsf{immutable}\ T') \in \overline{TD}}{\Delta_1 \vdash v.f : \mathsf{immutable}\ T' \dashv \Delta_1}\text{T-Call-field-imm}$$

$$\frac{\begin{array}{c}\Delta_1 \vdash v : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2 \\ (f : \mathsf{unique}\ T') \in \overline{TD} \qquad \overline{TD}' = \overline{(TD[\mathsf{state}\ s\ \{\overline{TD}\}/s] \setminus f)}\end{array}}{\Delta_1 \vdash v!f : \mathsf{unique}\ T' \dashv \Delta_2, v : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})}\text{T-Call-uni-uni}$$

Figure 2: Static Semantics

$\boxed{\Delta \vdash \mathsf{typecheck}(\overline{D}) \dashv \Delta}$

$$\frac{}{\Delta \vdash \mathsf{typecheck}(\varnothing) \dashv \Delta}\text{TC-emp}$$

$$\frac{\Delta_1 \vdash v : P \dashv \Delta_2 \qquad \Delta_2 \vdash \mathsf{typecheck}(\overline{D}) \dashv \Delta_3}{\Delta_1 \vdash \mathsf{typecheck}(\overline{D}, f : P = v) \dashv \Delta_3}\text{TC-Rec}$$

Figure 3: Static semantics Helpers

$$\frac{\Delta_1' \in \Delta \qquad \Delta_1' \Rrightarrow \Delta_1}{\Delta \vdash_\Delta \Delta_1}\textsc{Split}$$

$$\frac{}{x : perm\ Type \Rrightarrow x : perm\ Type}\textsc{Same}$$

$$\frac{}{x : perm\ Type \Rrightarrow \varnothing}\textsc{Drop}$$

$$\frac{}{x : \mathsf{uni}\ Type \Rrightarrow x : \mathsf{imm}\ Type, x : \mathsf{imm}\ Type}\textsc{Split-Uni}$$

$$\frac{}{x : \mathsf{imm}\ Type \Rrightarrow x : \mathsf{imm}\ Type, x : \mathsf{imm}\ Type}\textsc{Split-Imm}$$

Figure 4: Permission splitting

immutable field of v. The resulting type is an immutable permission to f. Also, the incoming context is threaded through undisturbed. As mentioned earlier, we disallow calling $v.f$ is v is immutable and f is unique .

The T-Call-Uni-Uni rule is for typing a field deference of a unique field of a unique v, and is interesting because it leads to changing the type of v. We syntactically distinguish such a dereference from the other, since it also has different dynamic semantics (rule E-Call-Uni, Figure 5). For type-checking, we first check if we have a unique permission to v and f is one of its members. The resulting type is the type of f. Note that unlike the previous rule the outgoing context is different since the permission to v is now unique state s $\{\overline{TD'}\}$, where $\overline{TD'}$ are the original type declarations minus f. Note that any occurrences of s in the signature should be replaced with the original definition of s (i.e. before f is removed); this ensures, for example, that methods which depend on the field cannot be called until the field is restored.

4.2 Dynamic Semantics

In this section we describe the dynamic semantics of the language. We augment values with references o and the linear context to hold typing information for references as well as variables. This leads to addition of a new T-Loc rule to the static semantics. We use μ to represent the heap where each reference o points to a list of declarations $\overline{D}$.

$$\begin{array}{rrcl} \textit{Values} & v & ::= & x \mid o \\ & & & \lambda x{:}P, \Delta \Rightarrow e \\ \textit{Perm. Contexts} & \Delta & ::= & \Delta, (x \mid o : P) \mid \varnothing \\ \textit{stores} & \mu & ::= & \mu, (o \to \overline{D}) \mid \varnothing \end{array}$$

$$\frac{}{\Delta, o : P \vdash o : P \dashv \Delta}\text{T-Loc}$$

Figure 5 describes the dynamic semantics. The E-App, E-Let and E-Let-Cong rules are similar to those found in

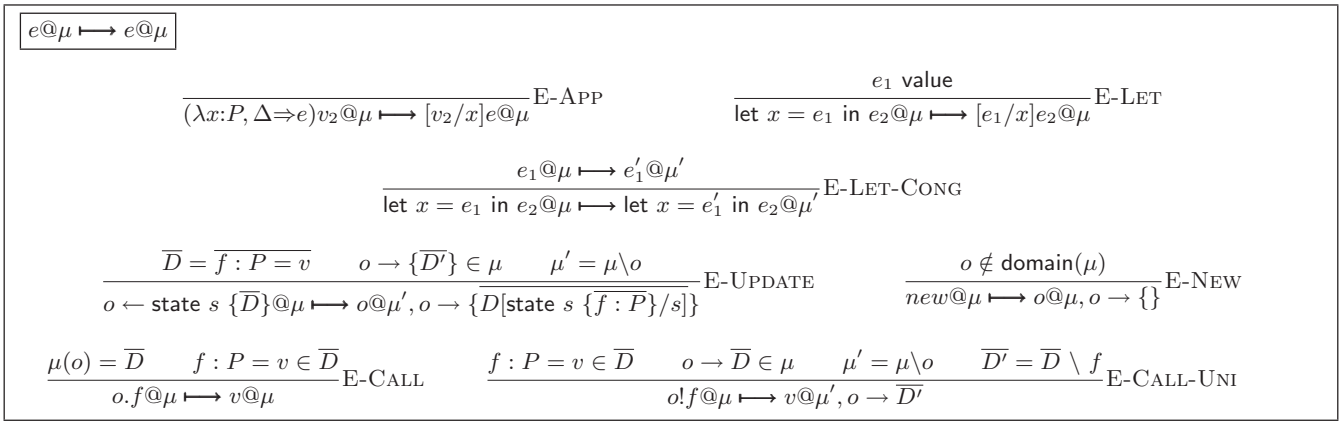

Figure 5: Dynamic semantics

the simply typed lambda calculus.

The E-Update rule is for the state change operation. Given that a reference o exists in the heap, we update its declarations to point to the new $\overline{D}$ given in the expression. Note that we substitute s in the declarations just as the declarations are added to the heap. This ensures that when we look them up later we do not encounter an undefined recursive type variable s.

The E-New rule adds a fresh o to the heap with the new location pointing to an empty set of declarations.

The E-Call rule is used for field dereferencing. We first look-up o in the heap and make sure f is a member of the resulting declarations $\overline{D}$. The rule returns the value is stored in f and leaves μ undisturbed.

The E-Call-Uni rule is used for dereferencing a unique field of a unique o. Given that a reference o exists in the heap, we update its declarations to $\overline{D}'$, which has all the original declarations but f. Note that since we already substituted s in the recursive declarations when we added them (E-Update) we don't have to worry about substitution during removal of f. The rule returns the value that is stored in f.

4.3 Type Safety

We are currently proving soundness for this language by mechanizing it in the SASyLF [1] proof checker. Our proof of soundness consists of the conventional progress and preservation theorems. For soundness to work however, we need a heap invariant that guarantees properties such as uniqueness and immutability of references on the heap. The heap invariant has been elided in this paper due to space constraints but is similar to previous work of the last author [4].

4.4 Discussion

Extending the Lambda calculus. We have envisioned the full Plaid language as multi-paradigm which supports both object-oriented and functional programming. Since $Plaid_{core}$ is the foundational basis for Plaid, it is important that it can model both first-class functions and objects in the presence of access permissions. This is the reason we chose to extend the lambda calculus with records, permissions and state change operations. Note that the state change operation subsumes assignment. To change the value of a field in a state we just change the state to one with the new value. This is a slight departure from the traditional notion of assignment, where the type of the new value must match the type of the expression it is being assigned to.

Structural Types. $Plaid_{core}$'s structural type system has lead to simplification in the formalization of this language. In previous work typestate based languages have had to include expressions for *packing* and *unpacking* objects [8]; mainly to deal with reentrant methods. For every field dereference operation, an object is first unpacked, transitioning it to an invalid state till a pack operation is called which makes it valid again. But our structural type system precludes the need for unpacking objects before dereferencing fields. Dereferencing a unique field from a unique object results in that field leaving the object and the object immediately changes its type. This way the object is never in an invalid state (though after such a "destructive field read" it may have fewer fields than it was initially declared with).

5. RELATED WORK

There have been languages in the past that allowed changing the type of objects in a first-class way. State change can be modeled in Smalltalk [11] using the become method, which results in one object exchanging state and behavior with another object. Also, in the Self [16] language an object can change the objects it delegates to (i.e. inherits from), thereby providing a way to model state changes. However, both these languages are dynamic whereas we model state changes in the type system.

Statically typed languages such as Ego [3] provide a related notion of changing the class of objects. If changing classes are viewed as states, this is similar to our state change operation. Other systems such as Fickle [9] distinguish "state classes" which describe states that can change, but are unable to track the state of fields. An alternative approach to checking typestate before calls is to suspend a call until the receiver is in an appropriate state [7]. In addition, there have been many typestate based analyses [4, 8] in the past but none of these allow state changes to objects in the type system like we do. Also, as mentioned earlier they have nominal type systems compared to ours, which is structural.

From the object modeling point of view, the closest work to ours is Taivalsaari's proposal to extend class-based languages with explicit definitions of logical states (modes),

each with its own set of operations and corresponding implementations [15]. Our proposed object model differs in providing explicit state transitions (rather than implicit ones determined by fields) and in allowing different fields in different states.

Our type system has similar capabilities as the linear type system presented in [8], but our setting is different in that we extend the lambda calculus and we have structural types compared to nominal.

6. CONCLUSION AND FUTURE WORK

Our foremost priority is to prove that $Plaid_{core}$ is type safe. Mechanizing the proof in a proof checker has forced attention to detail and subsequently led to many changes in the type system. But we feel that we have now reached a semantics that is sound. Next we plan to incorporate the *share* permission kind into the system. A *share* permission suggests that there may be several aliases to a reference and anyone is allowed to change its state. A *share* permission also introduces the concept of state guarantees, where each shared reference is guaranteed to never transition out of a hierarchy of states.

In conclusion, we presented the core calculus $Plaid_{core}$ for the Plaid programming language[6]. Our type system introduced novel notions of adapting permissions to the lambda calculus, in addition to modeling states and state changing operations on objects.

7. ACKNOWLEDGMENTS

We would like to thank the Plaid group, Ronald Garcia and the anonymous reviewers for their feedback on earlier versions of this work. This research was supported by DARPA grant #HR00110710019 and NSF grant #CCF-0811592.

8. REFERENCES

[1] J. Aldrich, R. Simmons, and K. Shin. SASyLF: an educational proof assistant for language theory. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pages 31–40. ACM, 2008.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022. ACM, 2009.

[3] A. Bejleri, J. Aldrich, and K. Bierhoff. Ego: Controlling the power of simplicity. In *Proceedings of the Workshop on Foundations of Object Oriented Languages (FOOL/WOOD 2006)*. Citeseer, 2006.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, page 320. ACM, 2007.

[5] K. Bierhoff, N. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. *ECOOP 2009–Object-Oriented Programming*, pages 195–219.

[6] J. Boyland. Checking interference with fractional permissions. *Static Analysis*, pages 1075–1075.

[7] F. Damiani, E. Giachino, P. Giannini, N. Cameron, and S. Drossopoulou. A State Abstraction for Coordination in Java-like Languages. In *Proceedings of FTfJP*, 2006.

[6]http://www.plaid-lang.org

[8] R. DeLine and M. Fähndrich. Typestates for objects. *ECOOP 2004–Object-Oriented Programming*, pages 465–490.

[9] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. *ECOOP 2001 Object-Oriented Programming*, pages 130–149.

[10] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993.

[11] A. Kay. The early history of Smalltalk. *ACM SigPlan Notices*, 28:69–69, 1993.

[12] D. Malayeri. *Coding Without Your Crystal Ball: Unanticipated Object-Oriented Reuse*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

[13] 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.

[14] J. Sunshine and J. Aldrich. Dynxml: Safely programming the dynamic web. In *APLWACA '10: Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*. ACM, 2010.

[15] A. Taivalsaari. Object-oriented programming with modes. *Journal of Object Oriented Programming*, 6:25–25, 1993.

[16] D. Ungar and R. Smith. Self: The power of simplicity. *Lisp and symbolic computation*, 4(3):187–205, 1991.