A Theory of Typestate-Oriented Programming

Darpan Saini, Joshua Sunshine and Jonathan Aldrich

July 15, 2010
CMU-ISR-10-117

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

{dsaini, sunshine, aldrich} @ cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

This research was supported by DARPA grant #HR00110710019 and NSF grant #CCF-0811592. The first author is supported by the Master of Software Engineering Fellowship 2009-10.

Keywords: Programming Languages, Typestate

Abstract

Many object-oriented libraries require their clients to follow state machine protocols, and typestate analyses have been developed to check conformance to these protocols. Prior work on typestate specification and analysis, however, has been divorced from the static and dynamic semantics of the language. This results in duplicated mechanisms in the type system and typestate checker, complicates reasoning, and prevents programmers from directly expressing their state-based designs in the language.

In this paper we present $Plaid_{core}$, a core calculus that embodies typestate-oriented programming by integrating typestate directly into the programming language and its type system. $Plaid_{core}$'s typestate change operator allows the representation of objects to change at run time, and uses a type system based on access permissions to statically guarantee that clients use object protocols correctly. We introduce the features of the calculus with examples, formally define its static and dynamic semantics, and prove type soundness. Based on this firm typing foundation, typestate-oriented programming has the potential to help engineers more easily specify and enforce protocol constraints, leading to more reliable software.

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. A recent case study on protocol usage in the wild [12] suggests that approximately 15-20% of all Java classes either define a protocol or use a protocol defined in another class. Not only is this a significant number of classes but often these protocols are unintuitive and hard to use. For example JDBC, a popular Java database connectivity library, has a state space that contains 33 states and multiple transitions which makes it difficult and error prone to use in practice.

In previous work we proposed typestate-oriented programming [1] as an extension to the object paradigm that models objects not with classes but in terms of their changing state. An object's typestate [14] 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 [7, 3]. These checkers have been successful in checking reasonably large programs [4] in languages like Java. However, there are compelling reasons to carry the idea of typestate into the programming language (summarized from [1]):

- 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[1]. 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 [5] 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 that illustrate its usefulness. $Plaid_{core}$ models objects as records and provides a novel state change operation to change the typestate of objects.

- A type system that statically tracks the typestate of objects. Unlike prior type systems for typestate, our type system is structural. Like [15] we adapt a subset of access permissions [3] to the setting of the lambda calculus.

- We prove soundness for the language using conventional progress and preservation theorems.

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 [13], 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.

2.1 File example

The first example[1] is a simple encoding of files, where a file can either be in the open or closed state (Figure 1). Each state is represented with a type that contains only relevant

[1]An earlier version of this paper was presented in the FTfJP 2010 workshop with the goal of gathering informal feedback. That version omitted important details due to space limitations, and the system presented had not been proved sound. The FTfJP workshop is high-quality but has limited visibility compared to POPL.

[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 $Plaid_{core}$.

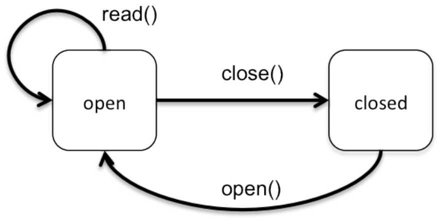

Figure 1: States of a File

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.

```
type OpenFile =
  state of {
    read : (imm of) → imm int
    close : (of ≫ ClosedFile) → unit
    ptr : imm CFilePtr
  }

type ClosedFile =
  state cf {
    open : (cf ≫ OpenFile) → unit
  }
```

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

State definitions. Listing 1 declares two states `OpenFile` and `ClosedFile`. A similar File example was discussed in [1], 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` and `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.

Access Permissions. Like the language presented in [1] $Plaid_{core}$ 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

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

permission means that there may be other aliases to this reference but no one is allowed to change the state of the reference.

When we specify the type of a function, therefore, we need to describe both the type and permission required for each of its arguments. We provide defaults to ease the burden of specifying permissions on the programmer. An unspecified permission to an object defaults to unique, while the default for function types is immutable . A missing $\gg$ means that the function does not change permissions to the arguments and returns them unchanged. Thus, the fully expanded type of the `open` method in `ClosedFile` is

$$\textbf{imm}\ (\textbf{uni}\ \mathit{cf} \gg \textbf{uni}\ \texttt{OpenFile}) \rightarrow \textbf{unit}$$

State Change Operations. Listing 2[3] describes how states can be defined in $Plaid_{core}$. Given that the definitions of `OpenFile` and `CloseFile` are mutually recursive, we must define a member function that changes the state of a `ClosedFile` to an `OpenFile` using a `letrec`-like[4] 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 `close` function 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 the `ClosedFile` state we use the previously defined *openf* function. 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 to `read` erroneous. In a Java-like language a similar situation could arise if there were 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.

[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

```
f = new
f =
  f ← state cf {
    open = openf
    }

computeBase =
  λ_ : unit [f: OpenFile ≫ ClosedFile] ⇒
    i = //...some computation
    f.close(f)
    return i

readFromFile =
  λf : OpenFile
    ⇒ i = computeBase() + f.read(f) // error!
      return i

f.open(f)
readFromFile(f)
```

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

The code in listing 3 does not typecheck 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.

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. $readFromFile2$ uses permissions to variables that occur free in its body.

```
readFromFile2 =
  λ_: unit [f: OpenFile]
    ⇒ i = f.read(f)
      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 $readFromFile2$ is

$$\textbf{imm (unit)}\ [f\colon \textbf{imm}\ \text{OpenFile} \gg f\colon \textbf{imm}\ \text{OpenFile}] \rightarrow \textbf{imm int}$$

```
computeBase =
  λ_ : unit ⇒
    i = //...some computation
    // cannot close the file!
    return i

readFromFile =
  λf : imm OpenFile
    ⇒ i = computeBase() + f.read(f)
       return i;
```

Listing 4: Fixed *readFromFile*

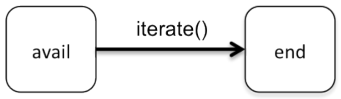

Figure 2: States of an Iterator

2.2 Iterator example

The second example is that of an iterator. In the case study we cited earlier[12], the authors identified eight different kinds of protocols that occur in Java code in the wild. Iterators fell into a category of protocols that represents 7.9% of all protocols. In this category (called *Boundary*), a method can only be called a specific number of times and this number is determined dynamically. Since iterators are used commonly in practice it we codify them in $Plaid_{core}$.

An iterator has two states available and end, signifying whether or not more elements exist in the collection to iterate over. Typically while iterating over a collection every call to the `next`() method is preceded by a call to the `hasNext`() method. This is a dynamic state test to check the state of the iterator. If `hasNext`() returns true then the state is available and it is safe to call `next`(); otherwise, the state is end and a call to `next`() now could lead to a runtime exception.

```
type Available =
  state av {
    coll : imm Collection
    iterate : (av ≫ End, int → unit) → unit
  }

type End =
  state e {}
```

Listing 6: Iterator States in $Plaid_{core}$

```
1  c = // get collection
2
3  iter = new
4  iter = iter ← state av {
5    coll = c
6
7    iterate =
8      λi : av, foo : int → unit
9        ⇒ for (int j = 0; j < coll.size(); ++j)
10             foo coll[j]
11          i ← state e {}
12 }
13
14 fun = λx : int ⇒ ...
15
16 fun2 = λx : int ⇒ ...
17
18 iter.iterate(iter, fun)
19
20 iter.iterate(iter, fun2) // error!
```

Listing 7: Iterator implementation and use in $Plaid_{core}$

To codify iterators in $Plaid_{core}$ we define two typestates, `Available` and `End`, as shown in listing 6. The `Available` typestate contains two fields: `coll`, which is the collection to be iterated over, and `iterate`, which is a two-argument function that takes the iterator and a function to be invoked on each element of the collection. We require that the field `coll` is immutable because this allows us to create more than one iterator for the collection. If it were unique an iterator would capture the unique permission precluding further creation of iterators. Also, it is safe to pass a unique permission whenever an immutable is required.

The implementation and usage of an iterator is described in listing 7[5]. The variable c contains the collection we would like to iterate over. We create a new object $iter$ and change its state to `Available`. Doing this requires that we supply values to its fields. We provide `coll` with the value c and implement `iterate`. `iterate` applies the supplied function to each element in the collection and subsequently changes the state of the iterator from `Available` to `End` (line 11). Calling `iterate` the first time with the function fun iterates over the collection and applies fun to each element, but the subsequent call to `iterate` on line 20 results in an error since the iterate is now in the state `End`.

[5]In this example we assume **for** loops in the language. How to add looping constructs to a language is well-known, hence we avoid adding unnecessary complexity to the core calculus.

3 Formal Language

The syntax[6] of the formal language is summarized in Figure 3. In place of a simple typing context containing just the types of variables, we use a linear context Δ. Later while defining the dynamic semantics, we will extend the same context to contain permissions to memory locations. The linear context differs from a conventional context in two ways:

1. As opposed to just types, Δ contains both the permission kind *perm* and the type T (together called the *Permission*) of variables

2. Each *Permission* can only be used once during typing. As typing proceeds *Permissions* are consumed and produced by typing rules.

Expressions. The language is restricted to A-normal form [9]. 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 other variables in scope, for instance curried arguments.

States are modeled using records where each record is a set of declarations of the form $\mathsf{state}\ s\ \{\overline{D}\}$ [7]. 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 empty 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, if f is immutable, the non-destructive immutable field read expression $v.f$ returns an immutable reference to the object f 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

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

[7]At different points in the formalism we use the standard convention of an overbar to represent a list of items. Also, to distinguish different lists of the same kind in a rule we use an apostrophe. If an apostrophe is absent then the lists are considered identical.

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).

$v \leftarrow \mathsf{state}\ s\ \{\overline{D}\}$ is the state change operation that changes the state of v from what it was before to $\mathsf{state}\ s\ \{\overline{TD}\}$. As we will see in section 4, $\mathsf{state}\ s\ \{\overline{TD}\}$ can be derived from $\mathsf{state}\ s\ \{\overline{D}\}$ in a straightforward manner. Essentially we drop all the fields that were part of v before the state change was initiated and replace them with $\overline{D}$.

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.

We also have the unit type for uninteresting values.

Permissions. Access permissions are a flow-sensitive mechanism to track typestate changes to aliases. They are modeled in the type system as linear resources. 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. So for instance if x was a unique `ClosedFile`, we could write a statement $x \leftarrow \mathsf{state}\ of\ \{\ldots\}$ to change its state to an `OpenFile`. On the other hand if x was an immutable `ClosedFile`, a statement such as $x \leftarrow \mathsf{state}\ of\ \{\ldots\}$ would be flagged by the type checker as erroneous.

In other formalizations [3], permissions such as shared, full and pure have been used. A real language should ideally 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 4. The typing judgment is of the form

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

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

Figure 3: Syntax

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, while P is the *Permission* (permission kind and type) of 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. This is one of the reasons why we insist our language be in the A-normal form.

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 6). 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 5). 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 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 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 permission 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 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 7). 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 $\overline{D}$.

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

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

The dynamic semantics are summarized in Figure 7. The E-App, E-Let and E-Let-Cong rules are similar to those found in 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 and o points 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. Since this is a destructive read, we must remove the field from the heap just like we do in the static semantics. This is to make sure that the type of o on the heap still matches the type of o in the context Δ. 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 Heap Invariant

To prove soundness for the simply typed lambda calculus with references it is required that all locations on the heap have exactly the same type as given by the store typing. Similarly to prove that our language is sound we need to guarantee certain consistency properties between the heap and the permission context Δ. Not only should objects on the heap have the same type as they have in Δ, but there should never be another reference to a unique object. In addition, there should never be a unique reference to immutable objects. We call this property the heap invariant and it is summarized in figure 8.

We know that each location (object) on the heap points to a set of fields, where each field contains a value. The heap invariant guarantees the following properties:

1. The heap is well formed with respect to itself.

2. The heap is well formed with respect to the permission context Δ.

Definitions.

1. $\mathsf{range}(\mu)$ is used to compute all the values and their permissions that are contained in fields of all objects that are currently on the heap (Figure 9).

2. $\mathsf{dom}(\Delta)$ is used to compute all the locations and their permissions that are in the permission context Δ (Figure 9).

For the heap to be well-formed with respect to itself, every field of every object must be consistent with its declared permission kind. For instance, if a field contains an object o and it is declared unique then there must be no other field on the heap that points to o. Similarly, if a field contains an object o and is declared immutable , then there must be no other field on the heap that points to o and is declared unique , there may however be other fields that point to o but are declared immutable .

The rules to enforce the property described above are summarized by the μ **wf** judgment in figure 9. The Heap-Wf-Empty rule says that an empty heap is well-formed. The Heap-Wf-Rec-1 rule allows us to add a new location o with no fields provided o does not already exist in the heap.

The Heap-wf-rec-uni rule allows us to add a unique field f that points to o' to object o to an existing heap given that:

- the existing heap is well-formed
- there is no other field that points to o' in the entire heap
- $o \neq o'$, to prevent self-reference

The Heap-wf-rec-imm rule allows us to add an immutable field f that points to o' to object o to an existing heap given that:

- the existing heap is well-formed
- there is no other unique $f$ield that points to o' in the entire heap
- $o \neq o'$, to prevent self-reference

Figure 8 summarizes the requirements for the heap to be consistent with respect to the permission context Δ. The rule Heap-inv-empty says that a well-formed heap μ is consistent with an empty permission context.

The rule Heap-inv-rec allows us to add a location *perm* o to Δ provided the location already exists in μ and has the same type in both Δ and μ. Also, we require that μ is well-formed with respect to itself and consistent with the existing Δ. In addition, we require that o satisfy the $check\_loc(perm\ o, \Delta, \mu)$ judgement.

Depending on the value of $perm$, $check\_loc(perm\ o, \Delta, \mu)$ specifies the requirements for adding o to Δ. If $perm$ is unique , then there must be no other permission to o in $\mathsf{dom}(\Delta)$ and $perm\ o$ cannot be in the range of μ. In other words, if a unique location exists in Δ, then no field can point to it in μ.

If on the other hand $perm$ is $\mathsf{immutable}$, then there must be no unique permission to o in $\mathsf{dom}(\Delta)$ and $\mathsf{unique}\ \ o$ cannot be in the range of μ. In other words, no unique field can point to o on the heap, but there can be other $\mathsf{immutable}$ fields that point to o.

The heap invariant above gives us enough leverage to prove soundness for the language.

4.4 Type Safety

Our proof of soundness consists of the conventional progress and preservation theorems. The theorems are as follows.

Preservation. The preservation theorem is stated as follows: If
$\Delta_1 \vdash e : P \dashv \Delta_2$
$e@\mu \longmapsto e'@\mu'$
$\Delta_1; \mu$ **wf**

then there exists Δ such that
$\Delta \vdash e' : P \dashv \Delta_2$
$\Delta; \mu'$ **wf**

Progress. The progress theorem is stated as follows: If
$\Delta_1 \vdash e : P \dashv \Delta_2$

then either e is a value or there exists μ such that $\Delta_1; \mu$ **wf**; $e@\mu \longmapsto e'@\mu'$

4.5 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 simplified the formalization of the language. Previous work on typestate included expressions for *packing* and *unpacking* objects [7], mainly to deal with reentrant methods. For every field dereference operation, an object is first unpacked, transitioning it to an invalid state until 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 [17] language an object can change the objects it delegates to (i.e. inherits from), thereby providing a way to model state changes. In addition, Self allows addition and removal of *slots* form objects at runtime; a feature that can be used to model protocols by changing representation of objects much like in our system. However, both these languages are dynamic whereas we model state changes in the type system.

Statically typed languages such as Ego [2] provide a related notion of changing the structure of objects. Ego managed change via lower-level field addition and removal operations, as well as delegation change operations; $Plaid_{core}$'s more structured state change operation, together with a more advanced permission system, simplifies the language and makes it more expressive.

In the Fickle [8] language it is possible to declare distinguished "state classes" inside a class which describe the different states objects of that class can be in. Fickle, however, is unable to accurately 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 [6]. In addition, there have been many typestate based analyses [3, 7] and related session type systems [10] in the past, but none of these allow state changes to objects in the run-time system like we do (by changing the representation). Since we change representation of objects we help simplify reasoning in many cases. Also, our setting is different in that we extend the lambda calculus and we have structural types compared to nominal.

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 [16]. 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.

Overall, our paper contributes the first sound, static type system for typestate-oriented programming. A related research project, which was begun after an initial version of our type system was developed, builds on our work to support a gradual type system for typestate-oriented programming [18]. Their work also differs in supporting a different set of permissions, supporting state guarantees, and working in a nominal setting.

6 Conclusion and Future Work

In conclusion, we presented the core calculus $Plaid_{core}$ for the Plaid programming language[8]. Our type system introduced a novel approach to adapting Bierhoff's access permissions to the lambda calculus, in addition to modeling states and state changing operations on objects. 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.

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.

8 Proof of Soundness

8.1 Preservation

The preservation theorem is stated as follows: If

(P.1) $\Delta_1 \vdash e : P \dashv \Delta_2$

(P.2) $e@\mu \longmapsto e'@\mu'$

(P.3) $\Delta_1; \mu$ **wf**

then there exists Δ such that

(P.4) $\Delta \vdash e' : P \dashv \Delta_2$

(P.5) $\Delta; \mu'$ **wf**

Proof. Proof by induction on (P.2)

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

case $\dfrac{}{\textbf{(E.1)}\ (\lambda x{:}P_0, \Delta_5 \Rightarrow e_1) v_2 @\mu \longmapsto [v_2/x] e_1 @\mu}$ E-App

By inversion for typing we have:

$$\dfrac{\textbf{(T.1)}\ \Delta_0 \vdash v_2 : P_0 \dashv \Delta_5 \qquad \textbf{(T.2)}\ \Delta_1 = \Delta_0, \Delta_3 \qquad \textbf{(T.3)}\ \Delta_3 \vdash \lambda x{:}P_0, \Delta_5 \Rightarrow e_1 : \mathsf{immutable}\ (\Pi x.(P_0, \Delta_5 \gg P_0', \Delta_5' \to P'')) \dashv \Delta_3}{\textbf{(T.4)}\ \Delta_1 \vdash (\lambda x{:}P_0, \Delta_5 \Rightarrow e_1) v_2 : P'' \dashv \Delta_5', \Delta_3, v_2 : P_0'}\ \text{T-App}$$

So,

- e is $\lambda x{:}P_1, \Delta_5 \Rightarrow e_1) v_2$
- e' is $[v_2/x] e_1$
- Δ_1 is Δ_1
- Δ_2 is $\Delta_5', \Delta_3, v_2 : P_0'$
- P is P''
- μ' is μ

Let Δ be $\Delta_3, \Delta_5, v_2 : P_0$, then to show:

- $\Delta_3, \Delta_5, v_2 : P_0 \vdash [v_2/x] e_1 : P'' \dashv \Delta_2$
- $\Delta_3, \Delta_5, v_2 : P_0; \mu$ **wf**

Proof.

(a) By case analysis on (T.3)

$$\dfrac{\textbf{(a.1)}\ \Delta_5, x : P_0 \vdash e_1 : P'' \dashv \Delta_5', x : P' \qquad \textbf{(a.2)}\ \mathsf{domain}(\Delta_1) \subset \mathsf{domain}(\Delta)}{\textbf{(a.3)}\ \Delta_3 \vdash \lambda x{:}P_0, \Delta_5 \Rightarrow e_1 : \mathsf{immutable}\ (\Pi x.(P_0, \Delta_5 \gg P_0', \Delta_5' \to P'')) \dashv \Delta_3}\ \text{T-Abs}$$

(b) $\Delta_5, v_2 : P_0 \vdash [v_2/x] e_1 : P'' \dashv \Delta_5', v_2 : P'$ by lemma (substitution)

(c) $\Delta_0 = \Delta_5, v_2 : P_0$ by set theory

(d) $\Delta_0; \mu$ **wf** by lemma (l.9) on (T.2), (P.3)

(e) $\Delta_3; \mu$ **wf** by lemma (l.9) on (T.2), (P.3)

(f) $\Delta_3, \Delta_5, v_2 : P_0 \vdash [v_2/x] e_1 : P'' \dashv \Delta_3, \Delta_5', v_2 : P'$ by lemma (l.8) on (d), (e), (b)

(g) $\Delta_1 = \Delta_3, \Delta_5, v_2 : P_0$ by set theory

(h) $\Delta_3, \Delta_5, v_2 : P_0; \mu$ **wf** since $\Delta_1; \mu$ **wf**

case $\dfrac{}{\textbf{(E.2)}\ \mathsf{let}\ x = v\ \mathsf{in}\ e_2@\mu \longmapsto [v/x]e_2@\mu}$ E-LET

By inversion for typing we have:

$$\dfrac{\textbf{(T.1)}\ \Delta_1 \vdash_\Delta \Delta_1' \qquad \textbf{(T.2)}\ \Delta_1' \vdash v : P' \dashv \Delta_3 \qquad \textbf{(T.3)}\ \Delta_3, x : P' \vdash e_2 : P \dashv \Delta_2}{\textbf{(T.4)}\ \Delta_1 \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e_2 : P \dashv \Delta_2}\text{T-LET}$$

So,

- e is $\mathsf{let}\ x = v\ \mathsf{in}\ e_2$
- e' is $[v/x]e_2$
- Δ_1 is Δ_1
- Δ_2 is Δ_2
- P is P
- μ' is μ

Let Δ be $\Delta_3, v : P'$, then to show:

- $\Delta_3, v : P' \vdash [v/x]e_2 : P \dashv \Delta_2$
- $\Delta_3, v : P'; \mu$ **wf**

Proof.

(a) $\Delta_3, v : P' \vdash [v/x]e_2 : P \dashv \Delta_2$ by lemma (substitution) on (T.3), (T.2)

(b) $\Delta_1'; \mu$ **wf** by lemma (l.6) on (T.1), (P.3)

(c) $\Delta_1' = \Delta_3, v : P'$

(d) $\Delta_3, v : P'; \mu$ **wf**

case $$\frac{\textbf{(E.1)}\ e_1@\mu \longmapsto e_1'@\mu'}{\textbf{(E.2)}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2@\mu \longmapsto \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2@\mu'}\textsc{E-Let-Cong}$$

By inversion for typing we have:

$$\frac{\textbf{(T.1)}\ \Delta_1 \vdash_\Delta \Delta_1' \qquad \textbf{(T.2)}\ \Delta_1' \vdash e_1 : P' \dashv \Delta_3 \qquad \textbf{(T.3)}\ \Delta_3, x : P' \vdash e_2 : P \dashv \Delta_2}{\textbf{(T.4)}\ \Delta_1 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : P \dashv \Delta_2}\textsc{T-Let}$$

So,

- e is $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
- e' is $\mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2$
- Δ_1 is Δ_1
- Δ_2 is Δ_2
- P is P
- μ' is μ'

Let Δ be Δ, then to show:

- $\Delta \vdash \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 : P \dashv \Delta_2$
- $\Delta; \mu'$ **wf**

Proof.

(a) $\Delta_1'; \mu$ **wf** by lemma (l.6) on (T.1), (P.3)

(b) By induction hypothesis on (T.2), (E.1), (a), we have

  (1) $\Delta \vdash e_1' : P' \dashv \Delta_3$
  (2) $\Delta; \mu'$ **wf**

(c) $\Delta \vdash_\Delta \Delta$ by rule Same

(d) $\Delta \vdash \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 : P \dashv \Delta_2$ by rule T-Let on (c), (1), (T.3)

$$\textbf{case}\quad \frac{\textbf{(E.1)}\ \overline{D} = \overline{f : P = v} \qquad \textbf{(E.2)}\ o \rightarrow \{\overline{D'}\} \in \mu \qquad \textbf{(E.3)}\ \mu' = \mu \backslash o}{\textbf{(E.4)}\ o \leftarrow \mathsf{state}\ s\ \{\overline{D}\} @ \mu \longmapsto o @ \mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}}\ \textsc{E-Update}$$

By inversion for typing we have:

$$\frac{\textbf{(T.1)}\ \Delta_1 \vdash o : \mathsf{unique}\ T \dashv \Delta_3 \qquad \textbf{(T.2)}\ \overline{D} = \overline{f : P = v} \qquad \textbf{(T.3)}\ \Delta_3 \vdash \mathsf{typecheck}(\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}) \dashv \Delta_2'}{\textbf{(T.4)}\ \Delta_1 \vdash o \leftarrow \mathsf{state}\ s\ \{\overline{D}\} : \mathsf{immutable\ unit} \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\})}\ \textsc{T-Update}$$

So,

- e is $o \leftarrow \mathsf{state}\ s\ \{\overline{D}\}$
- e' is $()$
- Δ_1 is Δ_1
- Δ_2 is $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\})$
- P is $\mathsf{immutable\ unit}$
- μ' is $\mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}$

Let Δ be $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\})$, then to show:

- $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}) \vdash () : \mathsf{immutable\ unit} \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}$
- $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}); \mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}$ **wf**

Proof.

(a) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}) \vdash () : \mathsf{immutable\ unit} \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\}$ by rule T-Unit

(b) $\Delta_3 = \Delta_1 \setminus o$ by lemma (l.3) on (T.1)

(c) $o : \mathsf{unique}\ \ \mathit{Type} \in \Delta_1$ by lemma (l.3) on (T.1)

(d) $\Delta_3; \mu'$ **wf** by lemma (l.2) on (P.3), (c), (b), (E.3)

(e) $\Delta_2'; \mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}$ **wf** by lemma (l.1) on (T.3), (d)

(f) Since $o \notin \Delta_3$ by (b) and $\Delta_2' \subseteq \Delta_3 \Rightarrow o \notin \Delta_2'$

(g) $o \notin \mathsf{range}(\mu)$ since o is unique and $\Delta_1; \mu$ **wf**

(h) $o \notin \mathsf{range}(\mu')$ since $\mu' \subset \mu$ and (h)

(i) $o \notin \overline{D[\mathsf{state}\ s\ \{\overline{f:P}\}/s]}$ by lemma (l.4) on (T.3) and (b)

(j) $o \notin \mathsf{range}(\mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f:P}\}/s]})$ by (h), (i)

(k) $check\_loc(\mathsf{unique}\ \ o, \Delta_2', (\mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f:P}\}/s]}))$ by rule Check-Loc-Uni on (f), (j)

(l) $\mu\{o\} : Type$ is vacuously true since $\overline{D}$ is same in both E-Update and T-Update

(m) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f:P}\}); \mu', o \rightarrow \{\overline{D[\mathsf{state}\ s\ \{\overline{f:P}\}/s]}\}$ **wf** by rule Heap-Inv-Rec on (k), (l), and (e)

$$\textbf{case}\ \frac{\textbf{(E.1)}\ o fresh}{\textbf{(E.2)}\ \mathsf{new}@\mu \longmapsto o@\mu, o \rightarrow \{\}}\ \text{E-New}$$

By inversion for typing we have:

$$\frac{}{\textbf{(T.1)}\ \Delta_1 \vdash \mathsf{new} : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \dashv \Delta_1}\ \text{T-new}$$

So,

- e is new
- e' is o
- Δ_1 is Δ_1
- Δ_2 is Δ_1
- P is $\mathsf{unique}\ (\mathsf{state}\ s\ \{\})$
- μ' is $\mu, o \rightarrow \{\}$ where o *fresh*

Let Δ be $\Delta_1, o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\})$, then to show:

- $\Delta_1, o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \vdash o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \dashv \Delta_1$
- $\Delta_1, o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}); \mu, o \rightarrow \{\}\ \mathbf{wf}$

Proof.

(a) $\Delta_1, o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \vdash o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \dashv \Delta_1$ by rule T-Loc

(b) $\mu\ \mathbf{wf}$ by inversion of rule Heap-Inv-Rec on (P.3)

(c) $\mu, o \rightarrow \{\}\ \mathbf{wf}$ by rule Heap-wf-Rec-1 on (b)

(d) $\Delta_1, o : \mathsf{unique\ state}\ s\ \{\}; \mu\ \mathbf{wf}$ by lemma (l.11) on (P.3), (c), where o *fresh*

case $$\dfrac{\begin{array}{c}\textbf{(E.1)}\ f : \mathsf{unique}\ T' = v \in \overline{D} \\ \textbf{(E.2)}\ o \rightarrow \overline{D} \in \mu \qquad \textbf{(E.3)}\ \mu' = \mu \backslash o \qquad \textbf{(E.4)}\ \overline{D'} = \overline{D} \setminus f\end{array}}{\textbf{(E.5)}\ o!f@\mu \longmapsto v@\mu', o \rightarrow \overline{D'}}\ \text{E-CALL-UNI}$$

By inversion for typing we have:

$$\dfrac{\begin{array}{c}\textbf{(T.1)}\ \Delta_1 \vdash o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2' \\ \textbf{(T.2)}\ (f : \mathsf{unique}\ T') \in \overline{TD} \qquad \textbf{(T.3)}\ \overline{TD}' = (\overline{TD[\mathsf{state}\ s\ \{\overline{TD}\}/s] \setminus f})\end{array}}{\textbf{(T.4)}\ \Delta_1 \vdash o!f : \mathsf{unique}\ T' \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})}\ \text{T-CALL-UNI-UNI}$$

So,

- e is $o!f$
- e' is v
- Δ_1 is Δ_1
- Δ_2 is $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})$
- P is $\mathsf{unique}\ T'$
- μ' is $\mu', o \rightarrow \overline{D'}$

Let Δ be $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), v : \mathsf{unique}\ T'$, then to show:

- $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), v : \mathsf{unique}\ T' \vdash v : \mathsf{unique}\ T' \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})$
- $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), v : \mathsf{unique}\ T'; \mu', o \rightarrow \overline{D'}$ **wf**

Proof.

(a) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), v : \mathsf{unique}\ T' \vdash v : \mathsf{unique}\ T' \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})$ by rule T-Loc-Uni (only applicable case is when v is some o')

(b) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}); \mu', o \rightarrow \overline{D}$ **wf** since $\Delta_1 = \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\})$ and $\mu = \mu', o \rightarrow \overline{D}$

(c) $\Delta_2'; \mu', o \rightarrow \overline{D}$ **wf** by lemma (1.9) on (b) since $\Delta_2' \subset (\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}))$

(d) $\Delta_2'; \mu', o \rightarrow \overline{D}'$ **wf** by lemma (1.10) on (c) since $\overline{D}' = \overline{D} \setminus f$

(e) $\mu', o \rightarrow \overline{D}'$ **wf** by inversion of rule Heap-Inv-Rec on (d)

(f) $p\ o \notin \Delta_2'$ since $\mathsf{unique}\ o \in (\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})$ and $(\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\})); (\mu', o \rightarrow \overline{D})$ **wf**, where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(g) $\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}), \mu', o \to \overline{D})$ by inversion of rule Heap-Inv-Rec on (b)

(h) $p\ o \notin \mathsf{range}(\mu', o \to \overline{D})$ by inversion of rule Check-Loc-Uni on (g), where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(i) $p\ o \notin \mathsf{range}(\mu', o \to \overline{D}')$ since $\overline{D}' \subset \overline{D}$, where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(j) $\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta_2', \mu', o \to \overline{D}')$ by rule Check-Loc-Uni on (f), (i)

(k) $\overline{TD} = \mathsf{typeof}(\overline{D})$ by (b)

(l) $\overline{TD} \setminus f = \mathsf{typeof}(\overline{D} \setminus f)$ by (k)

(m) $(\mu', o \to \overline{D}')\{o\} = \overline{D}'$ by simple lookup and $\overline{TD}' = \mathsf{typeof}(\overline{D}')$

(n) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}); \mu', o \to \overline{D}'$ **wf** by rule Check-Loc-Uni on (f), (i), (m)

(o) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}); \mu', o \to \overline{D}$ **wf** by (b)

(p) Consider the case that $p\ v \in \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\})$, then it must follow that $v \notin \mathsf{range}(\mu', o \to \overline{D})$ by rule Check-Loc-Uni, but we know that $v \in \mathsf{range}(\mu', o \to \overline{D})$ by (E.1). Hence, $p\ v \notin \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\})$ (the only applicable case is when v is some o'), where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(q) $p\ v \notin \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})$ since $p\ v \notin \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}) \subset p\ v \notin \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\})$, where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(r) $p\ v \notin \mathsf{range}(\mu', o \to \overline{D})$ by rule Heap-Wf-Rec-Uni on (e), where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(s) $p\ v \notin \mathsf{range}(\mu', o \to \overline{D}')$ since $\mu', o \to \overline{D}' \subset \mu', o \to \overline{D}$ and (r), where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(t) $\mathit{check\_loc}(\mathsf{unique}\ \ v, p\ v \notin \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), \mu', o \to \overline{D}')$ by rule Check-Loc-Uni on (q), (s), where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(u) $\Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\}), v : \mathsf{unique}\ T'; \mu', o \to \overline{D'}$ **wf** by rule Heap-Inv-Rec on (t), (n), (e)

$$\textbf{case}\ \frac{\textbf{(E.1)}\ \mu(o) = \overline{D} \qquad \textbf{(E.2)}\ f : P = v \in \overline{D}}{\textbf{(E.3)}\ o.f@\mu \longmapsto v@\mu}\text{E-CALL}$$

By inversion for typing we have:

$$\frac{\textbf{(T.1)}\ \Delta_1 \vdash v : perm\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2' \qquad \textbf{(T.2)}\ (f : \mathsf{immutable}\ T') \in \overline{TD}}{\textbf{(T.3)}\ \Delta_1 \vdash v.f : \mathsf{immutable}\ T' \dashv \Delta_1}\text{T-CALL-FIELD-IMM}$$

So,

- e is $o.f$
- e' is v
- Δ_1 is Δ_1
- Δ_2 is Δ_1
- P is $\mathsf{immutable}\ T'$
- μ' is μ

Let Δ be $\Delta_1, v : \mathsf{immutable}\ T'$, then to show:

- $\Delta_1, v : \mathsf{immutable}\ T' \vdash v : \mathsf{immutable}\ T' \dashv \Delta_1$
- $\Delta_1, v : \mathsf{immutable}\ T'; \mu$ **wf**

Proof.

(a) By case analysis on v:
case o

(b) $\Delta_1, o : \mathsf{immutable}\ T' \vdash o : \mathsf{immutable}\ T' \dashv \Delta_1$ by rule T-Loc

(c) Consider the case that $\mathsf{unique}\ o \in \Delta_1$, then it must follow that $o \notin \mathsf{range}(\mu)$ but we know that $o \in \mathsf{range}(\mu)$ by (E.2). Hence $\mathsf{unique}\ o \notin \Delta_1$

(d) μ **wf** by inversion of rule Heap-Inv-Rec on (P.3)

(e) $\mathsf{unique}\ o \notin \mathsf{range}(\mu)$ by inversion of rule Heap-Wf-Rec-Imm on (c)

(f) $check\_loc(\mathsf{immutable}\ \ o, \Delta_1, \mu)$ by rule Check-Loc-Imm on (c), (e)

(g) $\Delta_1, o : \mathsf{immutable}\ T'; \mu$ **wf** by rule Heap-Inv-Rec on (f), (P.3), (c)

8.2 Helper Lemmas

(l.0) **Substitution**
Given:
$\Delta, x : P \vdash e : P' \dashv \Delta'$
$v : P$
then
$\Delta, v : P \vdash [v/x]e : P' \dashv [v/x]\Delta'$

Proof by induction on the typing derivation

(l.1) Given:
$\Delta_1 \vdash \mathsf{typecheck}(\overline{D}) \dashv \Delta_2$
$\Delta_1; \mu$ **wf**
$o fresh$
then
$\Delta_2; \mu, o \rightarrow \overline{D}$ **wf**

Proof by induction on $\Delta_1 \vdash \mathsf{typecheck}(\overline{D}) \dashv \Delta_2$

(l.2) Given:
$\Delta; \mu$ **wf**
$o : \mathsf{unique}\ \ Type \in \Delta$
$\Delta' = \Delta \setminus o$
$\mu' = \mu \setminus o$
then
$\Delta'; \mu'$ **wf**

(l.3) Given:
$\Delta_1 \vdash o : \mathsf{unique}\ \ Type \dashv \Delta_2$
then
$(\Delta_2 = \Delta_1 \setminus o) \wedge (o : \mathsf{unique}\ \ Type \in \Delta_1)$

Straightforward from the definition of T-Loc

(l.4) Given:
$\Delta_1 \vdash \mathsf{typecheck}(\overline{D}) \dashv \Delta_2$
$o \notin \Delta_1$
then
$o \notin \overline{D}$

(l.5) Given:
$\Delta_1 \vdash v : P \dashv \Delta_2$
then
$\exists \Delta_1'$ such that $\Delta_1 = \Delta_1', v : P$

(l.6) Given:

(G.1) $\Delta_1 \vdash_\Delta \Delta_2$
(G.2) $\Delta_1; \mu$ **wf**
then
$\Delta_2; \mu$ **wf**
Proof by induction on $\Delta_1 \vdash_\Delta \Delta_2$

(a) **case** $\dfrac{\textbf{(T.2)}\ \Delta \vdash_\Delta \Delta'}{\textbf{(T.1)}\ \Delta, v : P \vdash_\Delta \Delta', v : P}$ Split-Rec

(b) So Δ_1 is $\Delta, v : P$ and $\Delta, v : P; \mu$ **wf** by (G.2)

(c) $\Delta; \mu$ **wf** by inversion of rule Heap-Inv-Rec on (b)

(d) $\Delta'; \mu$ **wf** by induction hypothesis on (T.2), (c)

(e) To show $\Delta', v : P; \mu$ **wf** by case analysis on v, only applicable case is when v is some o

(f) $\Delta, o : P; \mu$ **wf** by (G.2)

(g) By inversion of rule Heap-Inv-Rec on (f), we have (g.1)$check\_loc(perm\ o, \Delta, \mu)$ and (g.2)$\Delta; \mu$ **wf** and (g.3)μ **wf** and (g.4)$\mu\{o\} = \overline{f' : P' = v'}$ where P is $perm$ state s $\{\overline{f : P}\}$

(h) $check\_loc(perm\ o, \Delta', \mu)$ by lemma (l.12) on $check\_loc(perm\ o, \Delta, \mu)$, (c) and (T.2)

(i) $\Delta', o : P; \mu$ **wf** by rule Heap-Inv-Rec on (h), (d), (g.3), (g.4)

(j) **case** $\dfrac{}{\Delta, v : \mathsf{uni}\ Type \vdash_\Delta \Delta, v : \mathsf{imm}\ Type, v : \mathsf{imm}\ Type}$ Split-Uni

(k) $\Delta, v : \mathsf{unique}\ Type; \mu$ **wf** by (G.2)

(l) To show $\Delta, v : \mathsf{imm}\ Type, v : \mathsf{imm}\ Type; \mu$ **wf** by case analysis on v, only applicable case is when v is some o

(m) By inversion of rule Heap-Inv-Rec on (k), we have (k.1)$check\_loc(\mathsf{unique}\ \ o, \Delta, \mu)$ and (k.2)$\Delta; \mu$ **wf** and (k.3)μ **wf** and (k.4)$\mu\{o\} = \overline{f' : P' = v'}$ where P is $\mathsf{unique}\ \ Type$

(n) $check\_loc(\mathsf{immutable}\ \ o, \Delta, \mu)$ by lemma (l.13) on (k.1)

(o) $\Delta, o : \mathsf{imm}\ Type; \mu$ **wf** by rule Heap-Inv-Rec on (n), (k.2), (k.3), (k.4)

(p) $\Delta, o : \mathsf{imm}\ Type, o : \mathsf{imm}\ Type; \mu$ **wf** by lemma (l.14) on (o)

(q) **case** $\dfrac{}{\Delta, v : \mathsf{imm}\ Type \vdash_\Delta \Delta, v : \mathsf{imm}\ Type, v : \mathsf{imm}\ Type}$ Split-Imm

(r) straightforward application of lemma (l.14)

(s) **case** $\dfrac{}{\Delta, v : \mathit{perm\ Type} \vdash_{\Delta} \Delta, \varnothing}\textsc{Drop}$

(t) $\Delta, v : \mathit{perm\ Type}; \mu$ **wf** by (G.2)

(u) By inversion of rule Heap-Inv-Rec on (t), we have (t.1)$\mathit{check\_loc}(\mathit{perm}\ o, \Delta, \mu)$ and (t.2)$\Delta; \mu$ **wf** and (t.3)μ **wf** and (t.4)$\mu\{o\} = \overline{f' : P' = v'}$ where P is perm state s $\{\overline{f : P}\}$

(v) Proved by (t.2)

(w) **case** $\dfrac{}{\Delta \vdash_{\Delta} \Delta}\textsc{Same}$

(x) $\Delta; \mu$ **wf** by (G.2)

(l.7)

(l.8) Given:
weakening
$\Delta_1; \mu$ **wf**
$\Delta_2; \mu$ **wf**
$\Delta_1 \vdash e : P \dashv \Delta$
then
$\Delta_1, \Delta_2 \vdash e : P \dashv \Delta, \Delta_2$

(l.9) Given:
$\Delta_1 \subset \Delta_2$
$\Delta_2; \mu$ **wf**
then
$\Delta_1; \mu$ **wf**

Proven as case Drop of Lemma (l.6)

(l.10) Given:
$\Delta; \mu, o \rightarrow \overline{D}$ **wf**
$\overline{D}' = \overline{D} \setminus f$
then
$\Delta; \mu, o \rightarrow \overline{D}'$ **wf**
Proof
Straightforward induction on $\Delta; \mu, o \rightarrow \overline{D}$ **wf**

(l.11) Given:
$\Delta; \mu$ **wf**
$\mu, o \rightarrow \{\}$ **wf**
o *fresh*
then
$\Delta, o : \mathsf{unique\ state}\ s\ \{\}; \mu, o \rightarrow \{\}$ **wf**

Proof
Straightforward induction on $\Delta; \mu$ **wf**

(l.12) Given:
(G.1) $\mathit{check\_loc}(\mathit{perm}\ o, \Delta, \mu)$
(G.3) $\Delta \vdash_\Delta \Delta'$
then
$\mathit{check\_loc}(\mathit{perm}\ o, \Delta', \mu)$

Proof by induction on (G.3)

(a) **case** $\dfrac{\textbf{(T.2)}\ \Delta_1 \vdash_\Delta \Delta_2}{\textbf{(T.1)}\ \Delta_1, o' : P \vdash_\Delta \Delta_2, o' : P}$ Split-Rec

(b) $\mathit{check\_loc}(\mathit{perm}\ o, (\Delta_1, o' : P), \mu)$ by (G.1)

(c) Case analysis on *perm*
case unique

i. By inversion of rule Check-Loc-Uni on (b), we have (b.1) $p\ o \notin \mathsf{dom}(\Delta_1, o' : P)$ and (b.2) $p\ o \notin \mathsf{range}(\mu)$ where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$
ii. $p\ o \notin \mathsf{dom}(\Delta_1)$ by (b.1) where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$
iii. $p\ o \notin \mathsf{range}(\mu)$ by (b.2) where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$
iv. $\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta_1, \mu)$ by rule Check-Loc-Uni on (ii), (iii)
v. $\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta_2, \mu)$ by induction hypothesis on (iv) and (T.2)
vi. $p\ o \notin \mathsf{dom}(\Delta_2, o' : P)$ since $o \neq o'$ by (b.1) where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$
vii. $\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta_2, o' : P, \mu)$ by rule Check-Loc-Uni on (vi), (iii)

case immutable

i. By inversion of rule Check-Loc-Imm on (b), we have (b.1) $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_1, o' : P)$ and (b.2) $\mathsf{unique}\ \ o \notin \mathsf{range}(\mu)$
ii. $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_1)$ by (b.1)
iii. $\mathsf{unique}\ \ o \notin \mathsf{range}(\mu)$ by (b.2)
iv. $\mathit{check\_loc}(\mathit{imm}\ o, \Delta_1, \mu)$ by rule Check-Loc-Imm on (ii), (iii)
v. $\mathit{check\_loc}(\mathit{imm}\ o, \Delta_2, \mu)$ by induction hypothesis on (iv) and (T.2)
vi. $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_2, o' : P)$ since $o \neq o'$ by (b.1)
vii. $\mathit{check\_loc}(\mathsf{immutable}\ \ o, \Delta_2, o' : P, \mu)$ by rule Check-Loc-Imm on (vi), (iii)

(d) **case** $\dfrac{}{\textbf{(T.1)}\ \Delta_1, o' : \mathsf{uni}\ \mathit{Type} \vdash_\Delta \Delta_1, o' : \mathsf{imm}\ \mathit{Type}, o' : \mathsf{imm}\ \mathit{Type}}$ Split-Uni

(e) $\mathit{check\_loc}(\mathit{perm}\ o, \Delta_1, o' : \mathsf{uni}\ \mathit{Type}, \mu)$ by (G.1)

(f) case analysis on *perm*

case unique

i. By inversion of rule Check-Loc-Uni on (e), we have (e.1) $p\ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{uni}\ Type)$ and (e.2) $p\ o \notin \mathsf{range}(\mu)$ where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

ii. $p\ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{imm}\ Type)$ by (e.1) where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

iii. $p\ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{imm}\ Type), o' : \mathsf{imm}\ Type$ since $o \neq o'$ by (e.1) where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

iv. $check\_loc(\mathsf{unique}\ \ o, \Delta_1, o' : \mathsf{imm}\ Type), o' : \mathsf{imm}\ Type, \mu)$ by rule Check-Loc-Uni on (iii), (e.2)

case immutable

i. By inversion of rule Check-Loc-Imm on (e), we have (e.1) $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{uni}\ Type)$ and (e.2) $\mathsf{unique}\ \ o \notin \mathsf{range}(\mu)$

ii. $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{imm}\ Type)$ by (e.1)

iii. $\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta_1, o' : \mathsf{imm}\ Type), o' : \mathsf{imm}\ Type$ since $o \neq o'$ by (e.1)

iv. $check\_loc(\mathsf{immutable}\ \ o, \Delta_1, o' : \mathsf{imm}\ Type), o' : \mathsf{imm}\ Type, \mu)$ by rule Check-Loc-Imm on (iii), (e.2)

(g) **case** $$\frac{}{\textbf{(T.1)}\ \Delta_1, o' : \mathsf{imm}\ Type \vdash_\Delta \Delta_1, o' : \mathsf{imm}\ Type, o' : \mathsf{imm}\ Type}\ \textsc{Split-Imm}$$

(h) Similar to the case above

(i) **case** $$\frac{}{\textbf{(T.1)}\ \Delta_1, o' : P \vdash_\Delta \Delta_1}\ \textsc{Drop}$$

(j) Similar to the case above

(k) **case** $$\frac{}{\textbf{(T.1)}\ \Delta_1 \vdash_\Delta \Delta_1}\ \textsc{Same}$$

(l) Proved by (G.1)

(l.13) Given:
(G.1) $check\_loc(\mathsf{unique}\ \ o, \Delta, \mu)$
then
$check\_loc(\mathsf{immutable}\ \ o, \Delta, \mu)$

Proof.

(a) By inversion of rule Check-Loc-Uni on (G.1), we have (g.1) $p\ o \notin \Delta$ and (g.2)$p\ o \notin \mu$, where $p \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}$

(b) $\mathsf{unique}\ \ o \notin \Delta$ by (g.1)

(c) $\mathsf{unique}\ \ o \notin \mu$ by (g.2)

(d) $check\_loc(\mathsf{immutable}\ \ o, \Delta, \mu)$ by rule Check-Loc-Imm on (b), (c) (l.14) Given:

(G.1) $\Delta, o : \textsf{immutable } Type; \mu$ **wf**

then

$\Delta, o : \textsf{immutable } Type, o : \textsf{immutable } Type; \mu$ **wf**

Straightforward from inversion of rule Heap-Inv-rec on (G.1)

(l.15) Given:

$\Delta; \mu$ **wf**

$o \in \Delta$

then

$o \rightarrow \overline{D} \in \mu$

True due to form of judgment $\Delta; \mu$ **wf**

8.3 Progress

The progress theorem is stated as follows: If

(P.1) $\Delta_1 \vdash e : P \dashv \Delta_2$ (e is a closed expression)

(P.2) $\Delta_1; \mu$ **wf**

then one of the following is true:

(P.3) e is a value

(P.4) $e@\mu \longmapsto e'@\mu'$, for some e', μ'

Proof by induction on (P.1)

case T-Var does not apply since e is closed

case T-Loc, o is a value

case T-Abs, $\lambda x{:}P, \Delta \Rightarrow e'$ is a value

case $$\frac{\textbf{(T.1)}\ \Delta \vdash v_2 : P \dashv \Delta' \qquad \textbf{(T.2)}\ \Delta' = \Delta_1, \Delta_2 \qquad \textbf{(T.3)}\ \Delta' \vdash v_1 : \mathsf{immutable}\ (\Pi x.(P, \Delta_1 \gg P', \Delta_1' \to P'')) \dashv \Delta'}{\textbf{(T.4)}\ \Delta \vdash v_1 v_2 : P'' \dashv \Delta_2, \Delta_1', v_2 : P'}\ \text{T-APP}$$

Proof.

(a) By canonical forms v_1 is $\lambda x{:}P, \Delta \Rightarrow e'$

(b) $(\lambda x{:}P, \Delta \Rightarrow e')v_2@\mu \longmapsto [v_2/x]e@\mu$ by rule E-App

case $$\frac{\textbf{(T.1)}\ \Delta_1 \vdash_\Delta \Delta_1' \qquad \textbf{(T.2)}\ \Delta_1' \vdash e_1 : P \dashv \Delta_3 \qquad \textbf{(T.3)}\ \Delta_3, x : P \vdash e_2 : P' \dashv \Delta_2}{\textbf{(T.4)}\ \Delta_1 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : P' \dashv \Delta_2}\ \text{T-LET}$$

Proof.

(a) Case analysis on e_1

(b) **case** value
$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longmapsto [e_1/x]e_2$ by rule E-Let

(c) **case** e_1 takes a step

(d) $\Delta_1'; \mu$ **wf** by lemma (l.6) on (T.1) and (P.2)

(e) $e_1@\mu \longmapsto e_1'@\mu'$ by induction hypothesis on (T.2) and (d)

(f) $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longmapsto \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2$ by rule E-Let-Cong

case $$\frac{\textbf{(T.1)}\ \Delta_1 \vdash o : \mathsf{unique}\ T \dashv \Delta_3 \quad \textbf{(T.2)}\ \overline{D} = \overline{f : P = v} \quad \textbf{(T.3)}\ \Delta_3 \vdash \mathsf{typecheck}(\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}) \dashv \Delta_2'}{\textbf{(T.4)}\ \Delta_1 \vdash o \leftarrow \mathsf{state}\ s\ \{\overline{D}\} : \mathsf{immutable\ unit} \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\})}\ \text{T-Update}$$

Proof.

(a) $\Delta_1 = o : \mathsf{unique}\ P, \Delta_3$ by math

(b) $o \to \overline{D}' \in \mu$ by lemma (l.15) on (P.2), (a)

(c) $\mu'' = \mu \setminus o$ by math

(d) $o \leftarrow \mathsf{state}\ s\ \{\overline{D}\}@\mu \longmapsto ()@\mu'', o \to \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}$ by rule E-Update on (T.2), (b), (c)

case $$\frac{}{\textbf{(T.1)}\ \Delta_1 \vdash \mathsf{new} : \mathsf{unique}\ (\mathsf{state}\ s\ \{\}) \dashv \Delta_1}\ \text{T-new}$$

Proof.

(a) Let o *fresh*

(b) $\mathsf{new}@\mu \longmapsto o@\mu, o \to \{\}$ by rule E-New

case $$\frac{\textbf{(T.1)}\ \Delta_1 \vdash o : perm\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2' \quad \textbf{(T.2)}\ (f : \mathsf{immutable}\ T') \in \overline{TD}}{\textbf{(T.3)}\ \Delta_1 \vdash o.f : \mathsf{immutable}\ T' \dashv \Delta_1}\ \text{T-Call-field-imm}$$

Proof.

(a) $\Delta_1 = o : perm\ \mathsf{state}\ s\ \{\overline{TD}\}, \Delta_2'$

(b) $\Delta_1 = \Delta_2', o : perm\ \mathsf{state}\ s\ \{\overline{TD}\}$ by exchange

(c) $\Delta_2', o : perm\ \mathsf{state}\ s\ \{\overline{TD}\}; \mu\ \mathbf{wf}$ by (P.2) and (b)

(d) By inversion of rule Heap-Inv-Rec on (c), we have (c.1)$check\_loc(perm\ o, \Delta_2', \mu)$ and (c.2)$\Delta_2'; \mu\ \mathbf{wf}$ and (c.3)$\mu\ \mathbf{wf}$ and (c.4)$\mu\{o\} = \overline{f' : P' = v'}$ where P is some $perm\ \mathsf{state}\ s\ \{\overline{f' : P'}\}$ and $\overline{TD} = \overline{f' : P'}$ and $\overline{D} = \overline{f' : P' = v'}$

(e) $f : P = v \in \overline{D}$ by (T.2)

(f) $o.f@\mu \longmapsto v@\mu$ by rule E-Call on (c.4) and (e)

case $$\frac{\textbf{(T.1)}\ \Delta_1 \vdash o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}\}) \dashv \Delta_2' \quad \textbf{(T.2)}\ (f : \mathsf{unique}\ T') \in \overline{TD} \quad \textbf{(T.3)}\ \overline{TD}' = (\overline{TD[\mathsf{state}\ s\ \{\overline{TD}\}/s] \setminus f})}{\textbf{(T.4)}\ \Delta_1 \vdash o!f : \mathsf{unique}\ T' \dashv \Delta_2', o : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{TD}'\})}\ \text{T-Call-uni-uni}$$

Proof.

(a) $\Delta_1 = o : \mathsf{unique\ state}\ s\ \{\overline{TD}\}, \Delta_2'$

(b) $\Delta_1 = \Delta_2', o : \textsf{unique state } s\ \{\overline{TD}\}$ by exchange

(c) $\Delta_2', o : \textsf{unique state } s\ \{\overline{TD}\}; \mu\ \mathbf{wf}$ by (P.2) and (b)

(d) By inversion of rule Heap-Inv-Rec on (c), we have (c.1)$check\_loc(\textsf{unique}\ \ o, \Delta_2', \mu)$ and (c.2)$\Delta_2'; \mu\ \mathbf{wf}$ and (c.3)$\mu\ \mathbf{wf}$ and (c.4)$\mu\{o\} = \overline{f' : P' = v'}$ where P is some $\textsf{unique state } s\ \{\overline{f' : P'}\}$ and $\overline{TD} = \overline{f' : P'}$ and $\overline{D} = \overline{f' : P' = v'}$

(e) let $\mu'' = \mu \setminus o$ by math

(f) let $\overline{D'} = \overline{D} \setminus f$ by set theory

(g) $f : P = v \in \overline{D}$ by (T.2)

(h) $o!f@\mu \longmapsto v@\mu'', o \rightarrow \overline{D'}$ by rule E-Call-Uni on (g), (c.4), (e), (f)

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

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

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

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

[10] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *Symposium on Principles of programming languages*, pages 299–312. ACM, 2010.

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

[12] D. Kim. An empirical study on the frequency and classification of object protocols in java. *Master's thesis, Korea Advanced Institute of Science and Technology*, 2010.

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

[14] 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.

[15] 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*, New York, NY, USA, 2010. ACM.

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

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

[18] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual Typestate. In *Submitted to POPL*, 2011.

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

$$\frac{}{\Delta, x : P \vdash x : P \dashv \Delta}\,\textsc{T-Var} \qquad\qquad \frac{}{\Delta, o : P \vdash o : P \dashv \Delta}\,\textsc{T-Loc}$$

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

$$\frac{\begin{array}{c}\Delta \vdash v_2 : P \dashv \Delta' \\ \Delta' = \Delta_1, \Delta_2 \qquad \Delta' \vdash v_1 : \mathsf{immutable}\ (\Pi x.(P, \Delta_1 \gg P', \Delta_1' \rightarrow P'')) \dashv \Delta'\end{array}}{\Delta \vdash v_1 v_2 : P'' \dashv \Delta_2, \Delta_1', v_2 : P'}\,\textsc{T-App}$$

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

$$\frac{\begin{array}{c}\Delta_1 \vdash v : \mathsf{unique}\ T \dashv \Delta_2 \\ \overline{D} = \overline{f : P = v} \qquad \Delta_2 \vdash \mathsf{typecheck}(\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}) \dashv \Delta_3\end{array}}{\Delta_1 \vdash v \leftarrow \mathsf{state}\ s\ \{\overline{D}\} : \mathsf{immutable\ unit} \dashv \Delta_3, v : \mathsf{unique}\ (\mathsf{state}\ s\ \{\overline{f : P}\})}\,\textsc{T-Update}$$

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

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

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

Figure 4: Static Semantics

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

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

$$\boxed{\mathsf{domian}(\Delta) = \{\overline{x}\}}$$

$$\frac{}{\mathsf{domain}(\varnothing) = \varnothing}\text{DOMAIN-EMPTY} \qquad \frac{}{\mathsf{domain}(\Delta, x : P) = x, \mathsf{domain}(\Delta)}\text{DOMAIN-REC}$$

Figure 5: Static semantics Helpers

$$\boxed{\Delta \vdash_\Delta \Delta}$$

$$\frac{}{\Delta \vdash_\Delta \Delta}\text{SAME} \qquad \frac{}{\Delta, x : \mathit{perm}\ \mathit{Type} \vdash_\Delta \Delta, \varnothing}\text{DROP}$$

$$\frac{}{\Delta, x : \mathsf{uni}\ \mathit{Type} \vdash_\Delta \Delta, x : \mathsf{imm}\ \mathit{Type}, x : \mathsf{imm}\ \mathit{Type}}\text{SPLIT-UNI}$$

$$\frac{}{\Delta, x : \mathsf{imm}\ \mathit{Type} \vdash_\Delta \Delta, x : \mathsf{imm}\ \mathit{Type}, x : \mathsf{imm}\ \mathit{Type}}\text{SPLIT-IMM}$$

$$\frac{\Delta \vdash_\Delta \Delta'}{\Delta, x : P \vdash_\Delta \Delta', x : P}\text{SPLIT-REC}$$

Figure 6: Permission splitting

$\boxed{e@\mu \longmapsto e@\mu}$

$$\frac{}{(\lambda x{:}P, \Delta \Rightarrow e)v_2@\mu \longmapsto [v_2/x]e@\mu}\textsc{E-App} \qquad \frac{e_1 \ \mathsf{value}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2@\mu \longmapsto [e_1/x]e_2@\mu}\textsc{E-Let}$$

$$\frac{e_1@\mu \longmapsto e_1'@\mu'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2@\mu \longmapsto \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2@\mu'}\textsc{E-Let-Cong}$$

$$\frac{\overline{D} = \overline{f : P = v} \qquad o \to \{\overline{D'}\} \in \mu \qquad \mu' = \mu \backslash o}{o \leftarrow \mathsf{state}\ s\ \{\overline{D}\}@\mu \longmapsto ()@\mu', o \to \{\overline{D[\mathsf{state}\ s\ \{\overline{f : P}\}/s]}\}}\textsc{E-Update}$$

$$\frac{o\ \mathit{fresh}}{\mathsf{new}@\mu \longmapsto o@\mu, o \to \{\}}\textsc{E-New} \qquad \frac{\mu(o) = \overline{D} \qquad f : P = v \in \overline{D}}{o.f@\mu \longmapsto v@\mu}\textsc{E-Call}$$

$$\frac{f : P = v \in \overline{D} \qquad o \to \overline{D} \in \mu \qquad \mu' = \mu \backslash o \qquad \overline{D'} = \overline{D} \setminus f}{o!f@\mu \longmapsto v@\mu', o \to \overline{D'}}\textsc{E-Call-Uni}$$

Figure 7: Dynamic semantics

$\boxed{\Delta; \mu\ \mathbf{wf}}$

$$\frac{\mu\ \mathbf{wf}}{\varnothing; \mu\ \mathbf{wf}}\textsc{Heap-Inv-Empty}$$

$$\frac{\mathit{check\_loc}(\mathit{perm}\ o, \Delta, \mu) \qquad \Delta; \mu\ \mathbf{wf} \qquad \mu\ \mathbf{wf} \qquad \mu\{o\} = \overline{f : P = v}}{\Delta, o : \mathit{perm}\ \mathsf{state}\ s\ \{\overline{f : P}\}; \mu\ \mathbf{wf}}\textsc{Heap-Inv-Rec}$$

Figure 8: Heap Invariant

$$\boxed{\mu\ \mathbf{wf}}$$

$$\frac{}{\varnothing\ \mathbf{wf}}\textsc{Heap-wf-empty} \qquad \frac{\mu\ \mathbf{wf} \qquad o\ \mathit{fresh}}{\mu, o \rightarrow \varnothing\ wf}\textsc{Heap-wf-rec-1}$$

$$\frac{\mu, o \rightarrow \overline{D}\ \mathbf{wf} \qquad o \neq o' \qquad \mathit{perm}\ o' \notin \mathsf{range}(\mu, o \rightarrow \overline{D}) \qquad \mathit{perm} \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}}{\mu, o \rightarrow (\overline{D}, f : \mathsf{unique}\ \ \mathit{Type} = o')\ \mathbf{wf}}\textsc{Heap-wf-rec-un}$$

$$\frac{\mu, o \rightarrow \overline{D}\ \mathbf{wf} \qquad o \neq o' \qquad \mathsf{unique}\ o' \notin \mathsf{range}(\mu, o \rightarrow \overline{D})}{\mu, o \rightarrow (\overline{D}, f : \mathsf{immutable}\ \ \mathit{Type} = o')\ \mathbf{wf}}\textsc{Heap-wf-rec-imm}$$

$$\boxed{\mathit{check\_loc}(\mathit{perm}\ o, \Delta, \mu)}$$

$$\frac{\mathit{perm}\ o \notin \mathsf{dom}(\Delta) \qquad \mathit{perm}\ o \notin \mathsf{range}(\mu) \qquad \mathit{perm} \in \{\mathsf{unique}\ , \mathsf{immutable}\ \}}{\mathit{check\_loc}(\mathsf{unique}\ \ o, \Delta, \mu)}\textsc{Check-loc-uni}$$

$$\frac{\mathsf{unique}\ \ o \notin \mathsf{dom}(\Delta) \qquad \mathsf{unique}\ \ o \notin \mathsf{range}(\mu)}{\mathit{check\_loc}(\mathsf{immutable}\ \ o, \Delta, \mu)}\textsc{Check-loc-imm}$$

$$\boxed{\mathsf{range}(\mu) = \{\overline{\mathit{perm}\ o}\}}$$

$$\frac{}{\mathsf{range}(\varnothing) = \varnothing}\textsc{range-empty}$$

$$\frac{v\ \mathsf{is\ a\ location}}{\mathsf{range}(\mu, o \rightarrow (\overline{D}, f : \mathit{perm}\ \mathit{Type} = v)) = \mathit{perm}\ v, \mathsf{range}(\mu, o \rightarrow \overline{D})}\textsc{range-rec}$$

$$\boxed{\mathsf{dom}(\Delta) = \{\overline{\mathit{perm}\ o}\}}$$

$$\frac{}{\mathsf{dom}(\varnothing) = \varnothing}\textsc{dom-empty} \qquad \frac{}{\mathsf{dom}(\Delta, o : \mathit{perm}\ \mathit{Type}) = \mathit{perm}\ o, \mathsf{dom}(\Delta)}\textsc{dom-rec}$$

Figure 9: Heap Invariant Helpers