Featherweight Typestate

Ronald Garcia[*] **Roger Wolff**[*] **Éric Tanter**[†]
Jonathan Aldrich[*]

July 2010
CMU-ISR-10-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†] PLEIAD Laboratory, Computer Science Department (DCC), University of Chile

This research is supported by grants from the National Science Foundation and from IBM.
This work was supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project.

Keywords: access permissions, state guarantees

Abstract

Typestate oriented programming integrates notions of typestate directly into the semantics of an object-oriented programming language. This document presents the formalization of Featherweight Typestate, a typestate oriented language modeled after Featherweight Java. This language supports a classes-as-states model of typestates, and utilizes a flow-sensitive type system for checking access permissions and state guarantees, thereby enabling safe and modular typestate checking. The syntax and both static and dynamic semantics of Featherweight Typestate are presented, as well as a proof of type safety.

1 Introduction

What follows is a formalization of a system for typestate-oriented programming, with an emphasis on permission checking. This system is a statically-typed language for typestate-oriented programming. The formalization presented here is for a nominal class-oriented language modeled after Featherweight Java [Igarashi et al., 2001]. This language provides a simple model for explaining what typestate-oriented programming is about, as well as a platform for extension. We'll call it Featherweight Typestate, or FT for short.

2 Syntax

We begin with the basic syntax of the language. Since FT is modeled after Featherweight Java, it assumes a number of the same primitive notions, such as identifiers and method, field, and class names.

$$\begin{array}{rcl} x, \mathsf{this} & \in & \textsc{IdentifierNames} \\ m & \in & \textsc{MethodNames} \\ f & \in & \textsc{FieldNames} \\ C, D, E & \in & \textsc{ClassNames} \\ \mathsf{Object} & \in & \textsc{ClassNames} \end{array}$$

The this keyword is a particular identifier that is bound to the subject of a method call. The Object keyword is a special class name, indicating the top of all subtype hierarchies. Throughout the following, smallcaps (e.g. $\textsc{FieldNames}$) indicates syntactic categories, italics (e.g. C) indicates metavariables and sans serif (e.g. Object) indicates particular elements of a category.

An FT program is a list of class declarations followed by an expression (overline notation indicates sequences):

$$PG \;::=\; \langle \overline{CL}, e \rangle \quad \text{(programs)}$$

The syntax of classes is as follows:

$$\begin{array}{rcll} CL & ::= & \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F},\ \overline{M}\ \} & \text{(class declarations)} \\ F & ::= & T\ f & \text{(field declarations)} \\ M & ::= & T\ m(\overline{T \gg T\ x})\ [T \gg T]\ \{\ \mathsf{return}\ e;\ \} & \text{(method declarations)} \end{array}$$

Each class declares a superclass and contains a list of field declarations F and a list of method definitions M. In contrast to Featherweight Java, FT classes do not have an explicit constructor. For simplicity, each FT class has an implicit constructor that assigns an initial value to each field. A method declaration $T\ m(\overline{T \gg T\ x})\ [T \gg T]\ \{\ \mathsf{return}\ e;\ \}$ extends the same notion from Featherweight Java. Each method parameter is annotated with *two* type annotations, $T_1 \gg T_2\ x$, which indicate the type of the argument at the beginning (T_1) and end (T_2) of the method call. The method itself has a similar annotation in square brackets, which indicates the type of this at the beginning and end of the method call. This idea is formalized below.

The body of each method (and the top-level program) is an expression e, which is defined as follows:

$$\begin{array}{rcll} e & ::= & x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{void} \mid \mathsf{new}\ C(\overline{s}) \mid s.f & \text{(expressions)} \\ & \mid & s.m(\overline{s}) \mid s.f :=: s \mid s \leftarrow C(\overline{s}) \mid \mathsf{assert}\langle C\rangle(s) & \\ s & ::= & x & \text{(simple expressions)} \end{array}$$

Identifiers x are standard. The let expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ binds the value of the expression e_1 to the variable x for the scope of e_2. FT expressions are restricted to A-normal form [Sabry and Felleisen, 1993], so let expressions are used to sequence all operations, and every other expression in the language allows only simple expressions s as subexpressions. For now, identifier references are the only simple expressions in the language. When we discuss the runtime semantics of the language below, we add runtime object references to the set of simple expressions. The restriction to A-normal form simplifies the description of the type system, which relies on sequencing in order to track typestate. We assume throughout that variables bound by let expressions can be renamed as needed.

The void expression is a special value used to indicate operations that conceptually return no value, where we are interested in the operation's side-effects. It can be bound to a variable, but is otherwise useless. The new expression creates a new heap object based on the class C and populates its fields with the supplied values. The field reference expression $s.f$ returns the current value of the f field of s. The method call expression $s.m(\overline{s})$ executes the body of the m method with s bound to this and the arguments $\overline{s}$ to the method parameters. The field swap expression $s_0.f :=: s_1$ is a swapping assignment: it replaces the current value of the field $s_0.f$ with the value of s_1 and returns the old value as its result. Swapping semantics allows for better control over the permissions associated with object fields. The update operation $s \leftarrow C(\overline{s})$ is one of the new additions to the language specifically to support typestate. It replaces the value of s with the new object of class C, which may not be the same as s's current class. Updating an object is how FT expresses typestate change. The assert expression $\mathsf{assert}\langle C\rangle(s)$ asserts that the value of s is an object of class C. If this is true, then the expression returns a void object. If not, then the expression is erroneous. In a language like FT, an assert is more primitive than a cast that simply returns a new reference to an object with a different type. Because of typestate, the types of references already change in the language. Asserts give the programmer finer control over the permissions and class assumption individual references. The flexibility of assert is enabled by support for typestate, and a vital feature.

Types in FT extend the Java notion of class names as types. The syntax of types follows:

$$\begin{array}{rcll} T & ::= & P\ C \mid \mathsf{Void} & \text{(type descriptions)} \\ P & ::= & k(D) & \text{(permissions)} \\ k & ::= & \mathsf{full} \mid \mathsf{shared} \mid \mathsf{pure} & \text{(base permissions)} \end{array}$$

The language of types includes a Void type, which is the type of the void object. Otherwise, each FT type has two components. As with Java, each type indicates the current class of an object, which we call its *class assumption*. However, each type also indicates its *access permissions* [Bierhoff and Aldrich, 2007], which pair a *base permission* with a *state guarantee* as $k(D)$. Access permissions (or permissions for short) succinctly specify the read, write, and interference properties of a particular reference to an object. In FT, having write access permission means that a reference can swap the fields of an object, or perform an update operation within the constraints of the state guarantee. Read permission is the ability to read fields.

The full permission denotes that the given object reference has exclusive write privileges on its referred object and can also read from it and perform state change. Any other outstanding references to that object must have the pure permission. The pure permission denotes that the given object reference can read from an object. A reference with pure permissions can coexist with other reference that have write permission to the same object. The shared permission denotes that the object has non-exclusive write privileges on its referred object. Any other outstanding reference to this object must have shared or pure permissions.

A state guarantee expresses the portion of the subtype hierarchy to which a particular object's class identity is limited. Applying the update operation to an object can only change its class among the subtypes of its state guarantee class. All outstanding references to a particular object, including identifier and field bindings, must have mutually consistent state guarantees and permissions (we formalize this notion of consistency below).

Throughout the discussion of static semantics, we impose a well-formedness condition on types. The type $k(D)\ C$ is only well-formed if $C <: D$ according to the class table $\overline{CL}$ associated with the program. This ensures that the type ascribed to an object reference complies with its associated state-guarantee. From here forward, all types T are assumed to be well-formed. As a consequence, some judgments defined below are implicitly parameterized on the class table $\overline{CL}$, which determines subtyping relationships.

3 Static Semantics

As with Featherweight Java, the FT type system relies upon type contexts.

$$\Delta \ ::= \ \overline{x : T} \quad \text{(linear type contexts)}$$

A type context Δ is a list of identifier-type bindings. As is common, a well-formed type context can have only one binding for any given variable. As such, a type context is a partial function from variables to types.

Whereas Γ is the standard metavariable for type contexts, we use a different metavariable Δ to emphasize that the typing contexts are not merely lexical: they are *linear* [Girard, 1987]. One of the distinguished properties of a typestate-oriented type system is that the types of identifiers may change over the course of a program's execution. In part this reflects how the permissions to a particular object may be partitioned and shared between references as computation proceeds, but it also reflects how update operations may change the class of an object during execution.

3.1 Managing Permissions

Permissions to an object are a resource that can be consumed during execution. In particular, the permissions to an object can be split among object references in ways that can limit the operations that apply to those references. Some base permissions, like pure and shared can be infinitely split The type system must track splitting of permissions in order to check that programs are safe.

Several auxiliary judgments specify how permissions may be safely split. First, permission splitting $k_1 \Rrightarrow k_2/k_3$ describes how given a k_1 permission, permission k_2 can be acquired, leaving behind k_3 as the maximum residual.

$\boxed{k \Rrightarrow k/k}$ Base Permission Splitting

$$\text{(pure)}\frac{}{k \Rrightarrow \mathsf{pure}/k} \qquad \text{(full)}\frac{}{\mathsf{full} \Rrightarrow \mathsf{full}/\mathsf{pure}} \qquad \text{(shared)}\frac{k \in \{\mathsf{full}, \mathsf{shared}\}}{k \Rrightarrow \mathsf{shared}/\mathsf{shared}}$$

When we are only concerned that a permission k_2 can be gotten from a permission k_1 (i.e. the residual permission is irrelevant), we write $k_1 \Rrightarrow k_2$. For instance, given any permission k, $\mathsf{full} \Rrightarrow k$ and $k \Rrightarrow k$.

As is expected, the residual of a split operation is related to the part split off.

Proposition 1. *If $k_1 \Rrightarrow k_2/k_3$ then for some k_4, $k_1 \Rrightarrow k_3/k_4$ and $k_4 \Rrightarrow k_2$.*

Proof. By exhaustively checking each case. The only interesting case is where $k_1 = k_3 = k_4 = \mathsf{shared}$ and $k_2 = \mathsf{pure}$. □

Permissions play a part in determining what operations are possible, as well as when an object can be safely bound to an argument. The restrictions on permissions are formalized as a partial order on permissions, analogous to subtyping. The notation $P_1 <: P_2$ says that P_1 is a *subpermission* of P_2, which means that a reference with P_1 permissions may be used wherever an object reference with P_2 permissions is needed.

$\boxed{P <: P}$ Subpermission

$$\text{(pure-guarantee)}\frac{E <: D}{\mathsf{pure}(E) <: \mathsf{pure}(D)} \qquad \text{(full-guarantee)}\frac{D <: E}{\mathsf{full}(E) <: \mathsf{full}(D)}$$

$$\text{(split)}\frac{k_1 \Rrightarrow k_2}{k_1(D) <: k_2(D)} \qquad \text{(trans)}\frac{P_1 <: P_2 \quad P_2 <: P_3}{P_1 <: P_3}$$

The (split) rule expresses that splitting a base permission produces a lesser (or equivalent) permission. The state-guarantee rules for pure and full capture how state guarantees affect the strength of permissions. In particular, pure permissions are covariant with their state guarantees, whereas full permissions are contravariant with their state guarantees; shared permissions are only comparable at the same state guarantee. The (trans) rule ensures that permission ordering is transitive; the (split) rule implies reflexivity since $k \Rrightarrow k$. In theory, we could define a more symmetric and nondeterministic splitting relation, but this deterministic relation suffices for the static language.

The pure permission covaries with its state guarantee because a subclass state guarantee is a stronger invariant on a type: the object that is pointed to is guaranteed to be a more specific type than the supertype. On the other hand, the full permission contravaries with its state guarantee because a state guarantee higher in the subtype hierarchy allows an object's state to be changed among a broader number of classes. The existence of full and pure references to an object restrict each others' state guarantees: A full reference cannot change an object's state to a class outside of the pure references' state guarantee, and the pure reference cannot assume that the state of the object it points to is guaranteed to be lower in the class hierarchy.

Permission splitting extends base permission splitting to also take into account the state guarantees involved.

$\boxed{P \Rrightarrow P/P}$ Permission Splitting

$$\text{(PSplit)}\ \frac{k_1 \Rrightarrow k_2/k_3 \qquad k_1(D_1) \prec: k_2(D_2) \qquad D_3 = D_1 \wedge D_2}{k_1(D_1) \Rrightarrow k_2(D_2)/k_3(D_3)}$$

This relation ensures that in addition to the compatibility of state guarantees: that the state guarantee of $k_1(D_1)$ is compatible with $k_2(D_2)$ and that the residual permission $k_3(D_3)$ has the least-restrictive state guarantee possible. The expression $D_1 \wedge D_2$ yields the lower of D_1 and D_2 in the subclass hierarchy, whichever of the two is a subclass of the other. For example, if $D_1 <: D_2$, then $D_1 \wedge D_2 = D_1$ ($\wedge$ is the greatest lower bound according to subclass ordering). It's undefined if neither is a subclass of the other, but $k_1(D_1) \prec: k_2(D_2)$ guarantees that the two classes are related. Intuitively, a lower state guarantee in the subclass hierarchy implies a stronger invariant about the class identity of the object being referenced. This ensures that the two permissions that result from the split respect each other's state guarantees.

Permission splitting in turn extends to type splitting.

$\boxed{T \Rrightarrow T/T}$ Type Splitting

$$\text{(tsplit-obj)}\ \frac{P_1 \Rrightarrow P_2/P_3 \qquad C_1 <: C_2}{P_1\ C_1 \Rrightarrow P_2\ C_2/P_3\ C_1} \qquad\qquad \text{(tsplit-void)}\ \frac{}{\mathsf{Void} \Rrightarrow \mathsf{Void}/\mathsf{Void}}$$

The (tsplit-obj) rule builds on permission splitting, but takes subclassing into account. The residual type can retain the most specific class assumption.

The (tsplit-void) rule implies that the Void type can be arbitrarily split without diminishing it in any way. In this sense, a void object is an unlimited resource.

Subtyping is defined directly in terms of type splitting.

$\boxed{T <: T}$ Subtyping

$$\frac{T_1 \Rrightarrow T_2}{T_1 <: T_2}$$

Two enlightening propositions can relate subtyping to subpermissions and subclassing:

Proposition 2.

1. $P_1 \prec: P_1$ *iff* $P_1 \Rrightarrow P_2$*;*

2. *$P_1\ C_1 <: P_2\ C_2$ iff $P_1 \prec: P_1$ and $C_1 <: C_2$.*

The first part shows that the subpermissions and permission splitting have the same relationship as subtyping and type splitting, though it's not immediately obvious from their definitions. The second part shows that subtyping for object types decomposes into subpermissioning and subclassing.

Update operations can alter the state of any number of variable references. To retain soundness in the face of these operations, it is sometimes necessary to discard previously known information in case it has been invalidated. In particular, the class assumptions for some variables must revert to the state-guarantee, which is a reliable object type after a state change. The type demotion function expresses this restricting of assumptions.

$\boxed{\downarrow : T \to T}$ Type Demotion

$$\begin{array}{rcl} (\mathsf{shared}(D)\ C)\downarrow & = & \mathsf{shared}(D)\ D \\ (\mathsf{pure}(D)\ C)\downarrow & = & \mathsf{pure}(D)\ D \\ T\downarrow & = & T \quad \text{otherwise} \end{array}$$

We write $\Delta\downarrow$ for the compatible extension of demotion to typing contexts. Since the type of objects pointed to by a pure or shared reference can be changed by another full or shared reference, the class assumption associated with one of these references must revert to the state guarantee if its class assumption might have been invalidated. If an object reference has a full permission to an object, only that reference can update the underlying object. As such, full references do not need to be demoted.

3.2 Well-typed Expressions

The type judgements for Java involve only one context that acts as an input to the typing process. In FT, however, operations like state change can alter the static type information, including both the permissions a reference has to an object as well as the object's type. For this reason, the type system threads the context through the program, updating the typing context as an abstract representation of each expression's meaning.

To capture the linearity of type associations in the FT language, the typing judgment has the form $\Delta_1 \vdash e : T \dashv \Delta_2$. This judgment means that given the typing assumptions Δ_1, the expression e can be assigned the type T and produces typing assumptions Δ_2. The assumptions in question are the permissions and class information for each object reference.

The typing judgment for void is quite simple.

$$\text{(void)}\frac{}{\Delta \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta}$$

With respect to linearity of typing contexts, this judgment does not affect the types of any other references, so it returns the input context unchanged.

In contrast, the typing rule for variable references affects the output context. If the type context binds a variable x to a type T_1 , and that variable is referenced at type T_2, then the output type context sets the type of x according to the type splitting rules defined above.

$$\text{(ctx)}\frac{T_1 \Rrightarrow T_2/T_3}{\Delta, x : T_1 \vdash x : T_2 \dashv \Delta, x : T_3}$$

If the variable had the Void type, then the reference and output type are also Void. However, if the type context binds the variable to an object reference type, the typing rule splits off some permissions to the referenced variable. Subsequent references to the variable can only use the remaining permissions since the output context associates the variable with the permissions left over after splitting.

Observe that the (ctx) rule implies what is typically presented as a subsumption rule:

Lemma 3. *If $\Delta \vdash x : T_1 \dashv \Delta'$ and $T_1 <: T_2$ Then $\Delta \vdash x : T_2 \dashv \Delta''$ for some Δ''.*

However, changing the type from T_1 to T_2 also changes the output context. For this reason we still explicitly support subsumption.

$$\text{(sub)}\frac{\Delta_0 \vdash x : T_1 \dashv \Delta_1 \qquad T_1 <: T_2}{\Delta_0 \vdash x : T_2 \dashv \Delta_1}$$

The typing judgment for let expressions is straightforward. Since the language is in A-normal form, this rule expresses how permissions flow from one expression to the next.

$$\text{(let)}\frac{\begin{array}{c}\Delta_0 \vdash e_1 : T_1 \dashv \Delta_1 \\ \Delta_1, (x : T_1) \vdash e_2 : T_2 \dashv \Delta_2, x : T\end{array}}{\Delta_0 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta_2}$$

To type a let expression, first the input context Δ_0 is used to type the bound expression e_1, yielding the context Δ_1. Then e_2 is typed in a context that augments Δ_1 by giving x the same type as e_1. The output context of the entire let expression matches the output context of typing e_2, minus any residual type assumption for x. Removing x from the context preserves lexical scoping.

The rule for creating a new object is analogous to the equivalent Java rule.

$$\text{(new)}\frac{\Delta_0 \vdash C(\overline{s}) \dashv \Delta_1}{\Delta_0 \vdash \mathsf{new}\ C(\overline{s}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta_1}$$

The difference is that a new object also has permissions associated with it. The resulting reference to the newly created object has $\mathsf{full}(\mathsf{Object})$ permissions because the new object can safely be updated to any other class without worry since there are no other references to it. This rule relies on an auxiliary judgment that captures the idea of a well-typed constructor call (technically $C(\overline{s})$ is not an FT expression, so this is a separate judgement).

$\boxed{\Delta \vdash C(\overline{s}) \dashv \Delta}$ Well-typed Constructor

$$\frac{\mathit{fields}(C) = \overline{T\ f} \qquad \Delta_0 \vdash \overline{s : T} \dashv \Delta_1}{\Delta_0 \vdash C(\overline{s}) \dashv \Delta_1}$$

Throughout, we use the notation $\Delta \vdash \overline{s : T} \dashv \Delta'$ as shorthand for iteratively typing a sequence of simple expressions, i.e.

$$\Delta = \Delta_0 \vdash s_0 : T_0 \dashv \Delta_1; \quad \Delta_1 \vdash s_1 : T_1 \dashv \Delta_2; \; \cdots \; \Delta_n \vdash s_n : T_n \dashv \Delta_{n+1} = \Delta'$$

One novel and salient feature of FT is the update operation, which can change the class of an object. Its typing rule follows.

$$\text{(update)}\frac{\begin{array}{c} k \in \{\,\mathsf{full}, \mathsf{shared}\,\} \qquad C <: E \\ \Delta_0 \vdash C(\overline{s}) \dashv \Delta_1 \\ \Delta_1 \vdash s_t : k(E)\ D \dashv \Delta_2, s_t : T \end{array}}{\Delta_0 \vdash s_t \leftarrow C(\overline{s}) : \mathsf{Void} \dashv \Delta_2{\downarrow}, s_t : k(E)\ C}$$

This expression replaces the object that s_t refers to with $C(\overline{s})$. State-change is restricted to subclasses of the object's current state-guarantee, and that guarantee is preserved after the operation. Furthermore, the classes of variables in Δ_2 are demoted to their state guarantees since state change may have invalidated those stronger assumptions. Only one object is updated by this object, but it may affect any number of outstanding references. This operation is only possible if the reference to the receiving object has shared or full permissions to the underlying object.

The type rule for method invocation extends the analogous Java rule in that method calls can change the types of their arguments, including the receiver object.

$$\text{(invoke)}\frac{\begin{array}{c} \Delta_0 \vdash s_t : P_t\ C_t, \overline{s_x : T_x} \dashv \Delta_1, s_t : T''_t, \overline{s_x : T''_x} \\ mdecl(m, C_t) = T\ m(\overline{T_x \gg T'_x})\ [P_t\ C_t \gg T'_t] \end{array}}{\Delta_0 \vdash s_t.m(\overline{s_x}) : T \dashv \Delta_1{\downarrow}, s_t : T'_t, \overline{s_x : T'_x}}$$

The static class of the receiver object s_t determines the signature of the method m. Then, if each argument to the method call has the type required by the method signature, the type of the expression is equal to the return type of the method call, and the types of the arguments and receiver object change as specified to the types $\overline{T'}$ and T'_t respectively. All other permissions are demoted since the method call may perform update operations. Because of the structure of the invoke rule, the variable reference for the receiving object and the references to all the arguments must each be distinct. While this is inconvenient for practical programming, it can easily be arranged by aliasing arguments to new variables.

The field reference expression yields a pure reference to the underlying object. The permission is downgraded because the field does not give up any of its permissions to the object.

$$\text{(ref)}\frac{\begin{array}{c} \Delta \vdash s : P_1\ C_1 \dashv \cdots \\ (k_2(D_2)\ C_2\ f) \in \mathit{fields}(C_1) \end{array}}{\Delta \vdash s.f : \mathsf{pure}(D_2)\ C_2 \dashv \Delta}$$

The variable s must refer to an object reference, so its type must not be Void. Since field reference does not consume any of s's permissions, the output context from ascertaining its type is ignored. Throughout, ellipses indicate irrelevant subterms. The resulting pure reference inherits its state guarantee and class type from the source field. If the field object is later updated, then context demotion will will properly revert the reference's class assumption to the state guarantee. Field reference expressions return the input context unchanged because field reference consumes none of the permissions to s.

To acquire the declared permissions to an object field, one must replace the field entry with another using the swap operation.

$$\text{(swap)}\frac{\begin{array}{cc} k_1 \in \{\,\textsf{full}, \textsf{shared}\,\} & \Delta_0 \vdash s_1 : k_1(D_1)\ C_1 \dashv \cdots \\ (T_2\ f) \in \mathit{fields}(C_1) & \Delta_0 \vdash s_2 : T_2 \dashv \Delta_1 \end{array}}{\Delta_0 \vdash s_1.f \mathrel{:=:} s_2 : T_2 \dashv \Delta_1}$$

The type context output by swap accounts for the consumed permissions to s_2. The reference to s_1 in this expression consumes no permissions. FT replaces Java's value-oriented class cast with a class assert operation. The assert operation $\textsf{assert}\langle C\rangle(s)$ changes class type information of the reference s.

$$\text{(assert)}\frac{}{\Delta, s : P\ C \vdash \textsf{assert}\langle D\rangle(s) : \textsf{Void} \dashv \Delta, s : P\ D}$$

Though FT types consist of more than the object's class, this operation only affects the class part of an object's type. A traditional class cast $\langle D\rangle\ s$ can be defined as syntactic sugar over class assertions.

$$\textsf{let}\ x = \langle D\rangle\ y\ \textsf{in}\ e \ \simeq \ \begin{array}{l}\textsf{let}\ x = y \\ \ \textsf{in let}\ x_1 = \textsf{assert}\langle D\rangle(x)\ \textsf{in}\ e\end{array}$$

The type rules defined above depend on a few auxiliary judgments. The $\mathit{fields}(C)$ function yields the types of all the fields of any object of class C.

$\boxed{\mathit{fields} : \textsc{ClassNames} \to \overline{T\ f}}$ Class Field Declarations

$$\text{(fields-object)}\frac{}{\mathit{fields}(\textsf{Object}) = \cdot} \qquad \text{(fields-subclass)}\frac{\begin{array}{c}\textsf{class}\ C\ \textsf{extends}\ D\ \{\ \overline{T\ f},\ \overline{M}\ \} \\ \mathit{fields}(D) = \overline{T'\ f'} \\ \overline{f'} \cap \overline{f} = \varnothing\end{array}}{\mathit{fields}(C) = \overline{T'\ f'}, \overline{T\ f}}$$

FT does not allow field overloading, so each field declared in a subclass must have a different name than any in its superclasses. The $\mathit{method}(m, C)$ yields the definition of class C's method m, and the $\mathit{mdecl}(m, C)$ function yields just its type signature.

$\boxed{\mathit{mdecl}(m, C)}$ Method Declaration
$\boxed{\mathit{method}(m, C)}$ Method Definition

$$\text{(method-override)}\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \ \{\ \overline{F},\ \overline{M}\ \} \\ T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return } e;\ \} \in \overline{M}\end{array}}{\mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return } e;\ \}}$$

$$\text{(method-super)}\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \ \{\ \overline{F},\ \overline{M}\ \} \qquad m \notin \overline{M} \\ \mathit{method}(m, D) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return } e;\ \}\end{array}}{\mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return } e;\ \}}$$

$$\text{(mdecl)}\frac{\mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return } e;\ \}}{\mathit{mdecl}(m, C) = T_r\ m(\overline{T \gg T'})\ [T_t \gg T'_t]}$$

FT methods refer to the type of the receiver object T_t, and it is guaranteed to always match the class at which it is queried.

3.3 Typing Programs

Recall that an FT program is a pair of a class table and an expression. To formalize the notion of a well-typed program, we introduce a few more judgments.

First, the method typing judgment M **ok in** C denotes that a method M is well-typed if it is defined as part of class C.

$\boxed{M \textbf{ ok in } C}$ Well-typed Method

$$\frac{\begin{array}{c}T_r\ \underline{m}(\overline{T_x \gg T'_x\ x})[P_t\ C_t \gg T'_t] \textbf{ ok in } C_t \\ \overline{x : T_x}, \textsf{this} : P_t\ C_t \vdash e : T_r \dashv \Delta_0 \\ \Delta_0 \vdash \textsf{this} : T'_t, \overline{x : T'_x} \dashv \Delta_1\end{array}}{T_r\ m(\overline{T_x \gg T'_x\ x})\ [P_t\ C_t \gg T'_t]\ \{\ \textsf{return } e;\ \} \textbf{ ok in } C_t}$$

This typing rule allows this and the arguments x to be subtypes of the output types given in the spec. The check for the output type of these variables can use subsumption to match the output specifications. Method typing relies on an analogous judgment that confirms that a method signature would be either a new method or a consistent method override.

$\boxed{M \ \textbf{ok in} \ C}$ Well-typed Method Declaration

$$\text{(new)}\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \ \{ \cdots \} \\ mdecl(D, m) \text{ undefined}\end{array}}{T_r \ m(\overline{T_x \gg T'_x})[P_t \ C \gg T'_t] \ \textbf{ok in} \ C}$$

$$\text{(override)}\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \ \{ \cdots \} \\ mdecl(D, m) = T_r \ m(\overline{T_x \gg T'_x})[P_t \ E \gg T'_t]\end{array}}{T_r \ m(\overline{T_x \gg T'_x})[P_t \ C \gg T'_t] \ \textbf{ok in} \ C}$$

For a class definition to be well-typed, all of its fields must have object reference types, all of its methods must be well-typed, and its superclass hierarchy must lead to Object. This implies that all intermediate superclasses are defined and that every chain of superclasses ends at Object, i.e. there are no inheritance cycles.

$\boxed{CL \ \textbf{ok}}$ Well-typed Class

$$\frac{C_0 <: \textsf{Object} \qquad \overline{k(D) \ E}{\downarrow} = \overline{k(D) \ E} \qquad \overline{M} \ \textbf{ok in} \ C_0}{\textsf{class } C_0 \textsf{ extends } C_1 \ \{ \ \overline{k(D) \ E \ f}; \ \overline{M} \ \} \ \textbf{ok}}$$

Furthermore we require that the permissions associated with field types be invariant under demotion (we express this by adapting context demotion to apply to type declarations). Since object field reference types do not change as a program runs, they must not be invalidated by update operations. This restriction ensures that field types remain consistent.

Finally, a program is well-typed if its class table and main expression are well-typed in turn.

$\boxed{PG \ \textbf{ok}}$ Well-typed Program

$$\frac{\overline{CL \ \textbf{ok}} \qquad \cdot \vdash e : T \dashv \cdot}{\langle \overline{CL}, e \rangle \ \textbf{ok}}$$

4 Dynamic Semantics

The runtime semantics of the language add some new syntactic notions to the language. In particular, FT is a stateful language, so most values in the language are references to heap-allocated objects.

$$\begin{array}{rcll}
o & \in & \textsc{ObjectRefs} & \\
l & \in & \textsc{IndirectRefs} & \\
C(\overline{o}) & \in & \textsc{Objects} & \\
e & ::= & \ldots \mid o \mid l & \text{(expressions)} \\
s & ::= & x \mid l & \text{(simple expressions)} \\
v & ::= & \mathsf{void} \mid o & \text{(values)} \\
\mu & \in & \textsc{ObjectRefs} \rightharpoonup \textsc{Objects} & \text{(stores)} \\
\rho & \in & \textsc{IndirectRefs} \rightharpoonup \textsc{Values} & \text{(environments)}
\end{array}$$

Ultimately, expressions in the language evaluate to values, i.e. void or an object reference o. Since the language is imperative, the value void is used as the result of operations that are only interesting for their side-effects. In other OO languages, a void object is unnecessary: imperative operations can return some arbitrary object reference. However, this language pays particular attention to how permissions to an object are allocated, so including a void object clarifies the management of permissions.

To connect object references to objects, we use stores μ, which abstract the runtime heap of a program. Stores are represented as partial functions from object references o to objects $C(\overline{o})$. A well-formedness condition is imposed on stores: only object references o in the domain of a store can occur in its range.

In addition to the traditional heap, the dynamic semantics uses a second heap, which we call the environment, that mediates between variable references and the object store. The environment serves a purely formal purpose: it supports the proof of type safety by keeping precise track of the outstanding permissions associated with different references to objects at runtime. In the source language, two variables could refer to the same object in the store, but each can have different permissions to that object. The environment's role is to track these differences at runtime. It maps indirect references l to object references o. Two indirect references can point to the same object, but the permissions associated with the two indirect references are kept separate. Indirect references are added to the set of simple expressions, so they will appear in subexpression position. They play the role traditionally played by object references. The environment is not needed in a practical implementation of the language. As we show later, well-typed programs can be safely run on a traditional single-heap machine where object references are simple expressions.

The dynamic semantics of FT is formalized as a structural operational semantics defined over store/environment/expression triples. Its rules follow.

$\boxed{\mu, \rho, e \rightarrow \mu', \rho', e'}$ Dynamic Semantics

$$\text{(lookup)}\frac{}{\mu, \rho, l \rightarrow \mu, \rho, \rho(l)} \qquad \text{(new)}\frac{o \notin dom(\mu)}{\mu, \rho, \textsf{new}\ C(\bar{l}) \rightarrow \mu[o \mapsto C(\overline{\rho(l)})], \rho, o}$$

$$\text{(let)}\frac{l \notin dom(\rho)}{\mu, \rho, \textsf{let}\ x = v\ \textsf{in}\ e \rightarrow \mu, \rho[l \mapsto v], [l/x]e}$$

$$\text{(update)}\frac{}{\mu, \rho, (l_t \leftarrow C(\bar{l})) \rightarrow \mu[\rho(l_t) \mapsto C(\overline{\rho(l)})], \rho, \textsf{void}}$$

$$\text{(ref)}\frac{\mu(\rho(l)) = C(\bar{o}) \qquad \mathit{fields}(C) = \overline{T\ f}}{\mu, \rho, l.f_i \rightarrow \mu, \rho, o_i}$$

$$\text{(swap)}\frac{\mu(\rho(l_1)) = C(\bar{o}) \qquad \mathit{fields}(C) = \overline{T\ f}}{\mu, \rho, l_1.f_i :=: l_2 \rightarrow \mu[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\bar{o})], \rho, o_i}$$

$$\text{(assert)}\frac{\mu(\rho(l)) = C(\cdots) \qquad C <: D}{\mu, \rho, \textsf{assert}\langle D\rangle(l) \rightarrow \mu, \rho, \textsf{void}}$$

$$\text{(invoke)}\frac{\begin{array}{c}\mu(\rho(l)) = C(\cdots) \\ \mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \textsf{return}\ e;\ \}\end{array}}{\mu, \rho, l.m(\overline{l'}) \rightarrow \mu, \rho, \overline{[l'/x]}[l/\textsf{this}]e}$$

$$\text{(congr)}\frac{\mu, \rho, e_1 \rightarrow \mu', \rho', e_1'}{\mu, \rho, \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \rightarrow \mu', \rho', \textsf{let}\ x = e_1'\ \textsf{in}\ e_2}$$

The (lookup) rule dereferences an indirect reference to get the underlying value. The (new) rule creates a new object based on the constructor expression given. The arguments to the constructor are dereferenced so that the objects in the heap contain object references. The (let) rule handles a variable binding by allocating a new indirect reference, associating the object reference in question to it in the environment and substituting the fresh reference into the body of the let expression. The (update) rule replaces a binding in the store with a newly-constructed object. The (ref) rule looks up the field of an object in the heap and returns the corresponding object reference. The (swap) rule swaps the field of an object with a new object reference and returns the old one. The (assert) rule checks that a reference points to an object with a type compatible with the assertion. If the assertion succeeds, the program returns a void value; if not, the program gets stuck. The (invoke) rule substitutes the arguments to the method invocation into the method body and continues executing. The (congr) rule ensures that the bound expression in a let is computed before the body of the let.

5 Type Safety

To type runtime programs, type contexts must be extended.

$$\begin{array}{rcll} b & \in & x \mid l \mid o & \text{(context bindings)} \\ \Delta & ::= & \overline{b : T} & \text{(linear type contexts)} \end{array}$$

Since runtime expressions may now contain indirect references l and object references o, a typing context may have entries of the form $l : T$ and $o : T$. As such, the (ctx) type rule must account for references in a runtime program, e.g.:

$$\text{(ctx)}\frac{T_1 \Rrightarrow T_2/T_3}{\Delta, b : T_1 \vdash b : T_2 \dashv \Delta, b : T_3}$$

Furthermore, context demotion $\Delta\downarrow$ must be extended to the reference entries in a context.

To prove type safety, we must account for the outstanding permissions associated with references to each object o and make sure that they are mutually consistent. To achieve this, we appeal to some helper functions.

$$\begin{array}{rcl} \mathit{types}(\mu, \Delta, \rho, o) & = & \mathit{fieldTypes}(\mu, o) +\!\!+\ \mathit{envTypes}(\Delta, \rho, o) +\!\!+\ \mathit{ctxTypes}(\Delta, o) \\ \mathit{fieldTypes}(\mu, o) & = & \underset{o' \in dom(\mu)}{+\!\!\!+\!\!\!+} \left[T_i \mid \mu(o') = C(\overline{o''}),\ \mathit{fields}(C) = \overline{T\ f},\ \text{and } o''_i = o\right] \\ \mathit{envTypes}(\Delta, \rho, o) & = & \underset{l \in dom(\rho)}{+\!\!\!+\!\!\!+} [T \mid \rho(l) = o \text{ and } (l : T) \in \Delta] \\ \mathit{ctxTypes}(\Delta, o) & = & [T \mid o : T \in \Delta] \end{array}$$

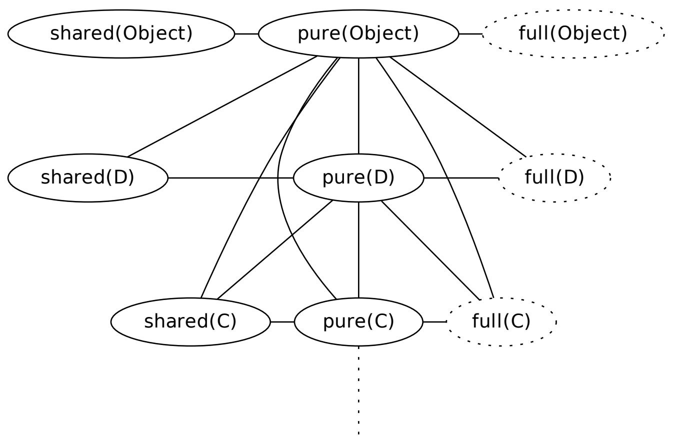
The $\mathit{fieldTypes}$ function takes a heap and an object reference in the domain of the heap and produces a list of the type declarations for every field reference to that object. This function disregards object references that are not bound to some field of some object. The $\mathit{envTypes}$ function performs the analogous operation for the indirect references in an environment that have bindings in the context. This function disregards indirect references in the environment that have no typing in the context. The $\mathit{ctxTypes}$ function does the same for object references that occur in a type context. The types function takes a heap, context, and environment, and object and yields the list of type declarations for outstanding heap, environment, and context references. These definitions use square brackets to express list comprehensions, and $+\!\!+$ to express list concatenation.

The consistent permissions relation $P_1 \leftrightarrow P_2$ says that two distinct references to the same object, one with permissions P_1 and the other with P_2 can soundly coexist at runtime.

$$\text{(pure)}\frac{E <: D}{k(E) \leftrightarrow \mathsf{pure}(D)} \qquad \text{(shared)}\frac{}{\mathsf{shared}(D) \leftrightarrow \mathsf{shared}(D)} \qquad \text{(sym)}\frac{P_1 \leftrightarrow P_2}{P_2 \leftrightarrow P_1}$$

First, a reference with pure permissions can coexist with any other permission that is bound to respect its state guarantee, meaning it could only change state among its subclasses. As a consequence, all pure permissions that are related by subclassing are consistent with each other. Also, shared permissions are consistent with each other so long as they have the same state guarantee. Observe that full permissions are not consistent with each other (or with shared permissions). Since full permissions are allowed to move their state guarantees down the subclass hierarchy, one full permission could end up out of sync with another full (or shared) permission, which might then update the referenced object in a manner that violates the state guarantees of the first full permission.

To give some more intuition, the following graph depicts consistency relations between permissions for a class C and it's immediate superclass D.



Solid lines indicate a pair of consistent permissions; the dotted line indicates that the trend continues down the inheritance hierarchy, and the dotted circles indicate that full permissions are not consistent with themselves . Observe that $pure(E)$ for some class E is connected to every permission whose state guarantee is a subclass of E, including itself. On the other hand, full is only connected to pure at the same state guarantee and superclasses, and shared is only connected with itself at the same state guarantee and pure at the same state guarantee and superclasses.

Using the *types* function and permission consistency, we can define a notion of *reference consistency* that verifies the mutual consistency of the types of all outstanding references to some object in the heap.

$$\frac{\begin{array}{c}\mu(o) = C(\overline{o'}) \qquad \left|\overline{o'}\right| = |\mathit{fields}(C)| \\ \mathit{types}(\mu, \Delta, \rho, o) = \overline{k(E)\ D} \\ C <: \overline{D} \qquad \langle\overline{k(E)}, \leftrightarrow\rangle \text{ is connected}\end{array}}{\mu, \Delta, \rho \vdash o\ \mathbf{ok}}$$

An initial consistency check for an object reference is that the object to which it refers has the proper number of fields according to the class table. Then it remains to check the consistency of references to the object. Given the list of reference types to an object o, we can ensure consistency by checking two properties. First, o's actual type must be a subtype of every reference type.

Second, the outstanding permissions to o must be pairwise consistent. This ensures that there can be at most one full reference outstanding and that shared and full permissions to o do not coexist. It also ensures that all shared permissions to o have the same state guarantee and that the state guarantee on a full(D) or shared(D) permission is compatible with all outstanding pure permissions with D or higher state guarantee.

Finally, given a store, environment, and context, we can define a notion of *type consistency* among them.

$$\frac{\begin{array}{c} ran(\rho) \subset dom(\mu) \cup \{\,\textsf{void}\,\} \\ dom(\Delta) \subset dom(\rho) \cup dom(\mu) \\ \{\,l \mid (l : \textsf{Void}) \in \Delta\,\} \subset \{\,l \mid \rho(l) = \textsf{void}\,\} \\ \{\,l \mid (l : k(D)\ C) \in \Delta\,\} \subset \{\,l \mid \rho(l) = o\,\} \\ \mu, \Delta, \rho \vdash dom(\mu)\ \textbf{ok} \end{array}}{\mu, \Delta, \rho\ \textbf{ok}}$$

The domains and ranges of the triple's components must all be in sync. First, the environment must map labels to objects that appear in the store. Next, the context must ascribe types only to indirect and direct references that appear in the environment and store respectively. Furthermore, the environment should map Void-typed and reference-typed labels respectively to void values and object references. Finally, all relevant references to each object in the store must be mutually consistent.

The definition of type consistency dictates that a type context Δ participating in the type consistency relation contains no variable bindings $x : T$, only object references and indirect references. This initial type context plays the same role that a heap signature Σ plays in formalizations type systems for mutable state [Pierce, 2002]. In those formalizations, a heap location retains the same type throughout a program's execution. Here, heap types change linearly, just as the types of variable bindings do, in response to update operations. For this reason, heap typing and bound variable typing are unified in one mechanism.

The type consistency relation allows the store to contain object references that do not appear in the environment or type context, and it allows the environment to have indirect references that do not appear in the type context. These "orphaned" references are essentially unreachable and correspond to elements that would be garbage collected in an implementation. A type context Δ serves as an upper bound on the references that are relevant to a program e that is typeable as $\Delta \vdash e : T \dashv \Delta'$.

To state progress, we need a notion of evaluation contexts

$$\mathbb{E} ::= \square \mid \textsf{let}\ x = \mathbb{E}\ \textsf{in}\ e$$

Evaluation contexts are programs with *holes*, notation $\square$, in them. An expression can be plugged into the hole to produce a program. Following the presentation of Featherweight Java by Pierce [2002], we use evaluation contexts to capture the possibility of a program getting stuck at a bad assertion. They are also used to express an invariant on object reference instances in running programs.

Theorem 4 (Progress)**.** *If e is a closed expression and $\Delta \vdash e \dashv \Delta'$, then either e is a value or for any store μ and environment ρ such that μ, Δ, ρ **ok**, either $\mu, \rho, e \rightarrow \mu', \rho', e'$ for some store μ',* *environment ρ', and expression e', or e is stuck at a bad assert, i.e., $e = \mathbb{E}[\mathsf{assert}\langle D\rangle(l)]$ where $\mu(\rho(l)) = C(\cdots)$, and $C \not<: D$.*

Our semantics has many rules that evaluate to an object reference, but they will always leave the reference about to be bound to an identifier (as in $\mathsf{let}\ x = o\ \mathsf{in}\ e$) or being the final result of the program.

The concern is that in the proof of preservation, bindings for o are added to the context Δ', but that is only a well-formed operation if that particular o is known to not already be in Δ' (Type contexts are essentially partial functions). We need to be sure that we're not in conflict with the permissions needed to type some other instance of the same object reference o in the program.

Definition 5. *An expression e is in* head reference form*, notation $hdref(e)$ iff either*

1. *e contains no object references o; or*
2. *$e = \mathbb{E}[o]$ for some $\mathbb{E}, o$ and $\mathbb{E}$ contains no object references.*

Head reference form ensures that there's at most one object reference in a runtime program and dictates where such a reference would be. With that knowledge, we can be assured that $\Delta', o : T$ is well-formed. This is significant also for ensuring that the (congr) rule preserves typing, since the body of the let binding does not depend on any object reference typings.

Definition 6. *A context Δ is* l-stronger *than a context Δ', notation $\Delta <^l \Delta'$ if and only if for all $l : T' \in \Delta'$, there is some $T <: T'$ such that $l : T \in \Delta$.*

Theorem 7 (Preservation)**.** *If e is a closed expression, $\Delta \vdash e : T \dashv \Delta''$, μ, Δ, ρ **ok**, $hdref(e)$, and $\mu, \rho, e \rightarrow \mu', \rho', e'$ then for some Δ', $\Delta' \vdash e' : T \dashv \Delta'''$, μ', Δ', ρ' **ok**, and $\Delta''' <^l \Delta''$.*

6 Single-Heap Implementation Model

As we've previously mentioned, the second heap in the FT dynamic semantics is specifically a tool for proving type safety. Here we formally show that a practical implementation of the language can use a traditional heap. The implementation semantics almost exactly matches the dynamic semantics, but leaves out the extra layer of indirection imposed by indirect references l and environments ρ.

$\boxed{\mu, e \rightarrow \mu', e'}$ Implementation Semantics

$$\text{(new)}\frac{o \notin dom(\mu)}{\mu, \textsf{new}\ C(\overline{o'}) \rightarrow \mu[o \mapsto C(\overline{o'})], o} \qquad \text{(let)}\frac{}{\mu, \textsf{let}\ x = v\ \textsf{in}\ e \rightarrow \mu, [v/x]e}$$

$$\text{(update)}\frac{}{\mu, (o_t \leftarrow C(\overline{o})) \rightarrow \mu[o_t \mapsto C(\overline{o})], \textsf{void}} \qquad \text{(ref)}\frac{\mu(o) = C(\overline{o'}) \quad \mathit{fields}(C) = \overline{T\ f}}{\mu, o.f_i \rightarrow \mu, o'_i}$$

$$\text{(swap)}\frac{\mu(o_1) = C(\overline{o'}) \quad \mathit{fields}(C) = \overline{T\ f}}{\mu, o_1.f_i :=: o_2 \rightarrow \mu[o_1 \mapsto [o_2/o'_i]C(\overline{o'})], o'_i} \qquad \text{(assert)}\frac{\mu(o) = C(\cdots) \quad C <: D}{\mu, \textsf{assert}\langle D\rangle(o) \rightarrow \mu, \textsf{void}}$$

$$\text{(invoke)}\frac{\begin{array}{c}\mu(o) = C(\cdots) \\ \mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T'_t]\ \{\ \textsf{return}\ e;\ \}\end{array}}{\mu, o.m(\overline{o'}) \rightarrow \mu, \overline{[o'/x]}[o/\textsf{this}]e}$$

$$\text{(congr)}\frac{\mu, e_1 \rightarrow \mu', e'_1}{\mu, \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \rightarrow \mu', \textsf{let}\ x = e'_1\ \textsf{in}\ e_2}$$

We define a simulation relation $\sim$ between Dynamic Semantics configurations and Implementation Semantics configurations.

$$\frac{}{\mu, \rho, e \sim \mu, \rho(e)}$$

Where $\rho(e)$ is the natural extension of $\rho(l)$ to arbitrary expressions.

Proposition 8.

1. *If e is a source program, then $\emptyset, \emptyset, e \sim \emptyset, e$.*
2. *If $\mu_1, \rho, e_1 \sim \mu_2, e_2$ and $\mu_1, \rho, e_1 \rightarrow \mu'_1, \rho', e'_1$ then $\mu_2, e_2 \rightarrow^* \mu'_2, e'_2$ and $\mu'_1, \rho', e'_1 \sim \mu'_2, e'_2$, for some store μ'_2, and some environment, e'_2.*

Proof. Part (1) is immediate. Part (2) is proven by induction on the rules of $\mu, \rho, e \rightarrow \mu', \rho', e'$. □

Appendix: Proofs of Type Safety

Lemma 9 (Context)**.** *If $\Delta \vdash b : T \dashv \Delta'$ then $\Delta = (\Delta_0, b : T')$ for some T'.*

Proof. By induction on derivations of $\Delta \vdash b : T \dashv \Delta'$. □

Theorem 10 (Progress)**.** *If e is a closed expression and $\Delta \vdash e \dashv \Delta'$, then either e is a value or for any store μ and environment ρ such that μ, Δ, ρ **ok**, either $\mu, \rho, e \rightarrow \mu', \rho', e'$ for some store μ', environment ρ', and expression e', or e is stuck at a bad assert, i.e., $e = \mathbb{E}[\textsf{assert}\langle D\rangle(l)]$ where $\mu(\rho(l)) = C(\cdots)$, and $C \not<: D$.*

Proof. By induction on the derivation of $\Delta \vdash e \dashv \Delta'$.

Case (void). Then $e = \textsf{void}$ which is a value.

Case (ctx-void). Then $e = b$ for some b. This breaks down into three cases.

1. ($e = x$): Then e is not closed. Contradiction.

2. ($e = o$): Suppose Δ, μ, ρ **ok** for some μ, ρ. The (ctx-void) rule dictates that $\Delta = (\Delta', o : \textsf{Void})$, but type consistency, particularly $\Delta, \mu, \rho \vdash o$ **ok**, requires that $o : T \in \Delta$ implies that $T = k(D)\ C$. Contradiction.

3. ($e = l$): Suppose Δ, μ, ρ **ok** for some μ, ρ. The (ctx-void) rule dictates that $\Delta = (\Delta', l : \textsf{Void})$, which combined with type consistency implies that $l \in dom(\rho)$. The (lookup) reduction rule $\mu, \rho, l \to \mu, \rho, \rho(l)$ then applies.

Case (ctx-obj). Then $e = b$ for some b. This breaks down into three cases.

1. ($e = x$): Then e is not closed. Contradiction.

2. ($e = o$): Then e is a value.

3. ($e = l$): Suppose Δ, μ, ρ **ok** for some μ, ρ. The (ctx-obj) rule dictates that $\Delta = (\Delta, l : T')$ for some T', which combined with type consistency implies that $l \in dom(\rho)$. The (lookup) reduction rule $\mu, \rho, l \to \mu, \rho, \rho(l)$ then applies.

Case (subclass). Follows immediately from the induction hypothesis on the premises.

Case (subperm). Follows immediately from the induction hypothesis on the premises.

Case (let). Then $e = \textsf{let}\ x = e_1\ \textsf{in}\ e_2$. Since e is closed, so is e_1, and the (let) rule dictates that $\Delta \vdash e_1 : T_1 \dashv \Delta_1$. The induction hypothesis applied to e_1 induces three cases:

1. If e_1 is a value, then the (let) reduction rule applies.

2. If e_1 takes a step, then the (congr) reduction rule applies.

3. If $e_1 = \mathbb{E}[\textsf{assert}\langle D\rangle(l)]$ is stuck at a bad cast, then

$$\begin{aligned} e &= \textsf{let}\ x = \mathbb{E}[\textsf{assert}\langle D\rangle(l)]\ \textsf{in}\ e_2 \\ &= \mathbb{E}'[\textsf{assert}\langle D\rangle(l)]. \end{aligned}$$

Case (new). Since e is closed, $e = \textsf{new}\ C(\bar{l})$. Suppose μ, Δ, ρ **ok**. The (new) rule dictates that $\Delta \vdash \overline{l : T} \dashv \Delta'$, from which the Context lemma implies $\bar{l} \subseteq dom(\Delta)$. Type consistency and the typing of $\bar{l}$ imply that $\overline{\rho(l)} \subset dom(\mu)$, so the (new) reduction rule applies.

Case (update). Since e is closed, $e = l \leftarrow C(\overline{l'})$. Suppose μ, Δ, ρ **ok** and let $\overline{l''} = l, \overline{l'}$ The (update) rule dictates that, $\Delta \vdash \overline{l'' : T} \dashv \Delta'$, from which the Context lemma implies $\overline{l''} \subseteq dom(\Delta)$. Type consistency then implies that $\overline{\rho(l'')} \subset dom(\mu)$, so the $(update)$ reduction rule applies.

Case (invoke). Since e is closed, $e = l.m(\overline{l'})$. Suppose μ, Δ, ρ **ok** and let $\overline{l''} = l, \overline{l'}$. The (invoke) rule dictates that $\Delta \vdash \overline{l'' : T} \dashv \Delta'$, from which the Context lemma implies $\overline{l''} \subseteq dom(\Delta)$. Type consistency then implies that $\overline{\rho(l'')} \subset dom(\mu)$, so $\mu(\rho(l)) = C(\cdots)$ for some C. Furthermore, the (invoke) rule dictates that $mdecl(m, C)$ is defined, from which it follows that $method(m, C)$ is defined, yielding a method body e. These properties suffice for the (invoke) reduction rule to apply.

Case (ref). Then since e is closed, $e = l.f$. Suppose μ, Δ, ρ **ok**. The (ref) rule dictates that $\Delta \vdash l : T_1 \dashv \Delta''$ for some Δ'', from which the Context lemma implies that $l \in dom(\Delta)$. Furthermore, $(T_2\ f) \in \mathit{fields}(C)$, so for some index i, $f = f_i \in \mathit{fields}(C)$. Finally, type consistency means implies that the object to which l refers, $\mu(\rho(l)) = C(\overline{o})$ has an ith element. These properties ensure that the (ref) reduction rule applies.

Case (swap). Since e is closed, $e = l_0.f :=: l_1$.

Case (assert). Since e is closed, $e = \mathsf{assert}\langle D\rangle(l)$. Suppose μ, Δ, ρ **ok**. The (assert) rule dictates that $l : P\ C \in \Delta$, so by type consistency
$\mu(\rho(l)) = C(\cdots)$ for some C. If $C <: D$ then the (assert) rule applies. If not, then e is stuck and e = $\square[\mathsf{assert}\langle D\rangle(l)]$.

$\square$

Lemma 11 (Inversion)**.**

1. *If* $\Delta \vdash \mathsf{void} : T \dashv \Delta'$ *then* $T = \mathsf{Void}$ *and* $\Delta' = \Delta$.

2. *If* $\Delta \vdash b : T \dashv \Delta'$ *then*

   (a) $\Delta = \Delta_0, b : T_0$*;*

   (b) $T_0 \Rrightarrow T_1/T_2$*;*

   (c) $\Delta' = \Delta_0, b : T_2$*; and*

   (d) $T_1 <: T$.

3. *If* $\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T \dashv \Delta'$ *then* $\Delta \vdash e_1 : T_1 \dashv \Delta_0$ *and*
   $\Delta, x : T_1 \vdash e_2 : T \dashv \Delta', x : T_1'$.

4. *If* $\Delta \vdash \mathsf{new}\ C(\overline{s}) : T \dashv \Delta'$ *then* $\Delta \vdash \overline{s : T'} \dashv \Delta'$ *for* $\overline{T'\ f} = \mathit{fields}(C)$*, and* $T = \mathsf{full}(C)\ C$.

5. *If* $\Delta \vdash s \leftarrow C(\overline{s}) : T \dashv \Delta'$ *then* $\Delta \vdash \overline{s : T'} \dashv \Delta_1$ *for* $\overline{T'\ f} = \mathit{fields}(C)$*,* $\Delta_1 \vdash s_t : k(E)\ D \dashv \Delta_2, s_t : T_t$ *where* $C <: E$*,* $k \in \{\mathsf{full}, \mathsf{shared}\}$*,*
   $\Delta' = (\Delta_2)\!\downarrow, s_t : k(E)\ C$*, and* $T = \mathsf{Void}$*.*

6. *If* $\Delta \vdash s_t.m(\overline{s_x}) : T \dashv \Delta'$ *then*
   $\Delta \vdash s_t : P_t\ C_t, \overline{s_x : T_x} \dashv \Delta_0, s_t : T_t'', \overline{s_x : T_x''}$,
   $mdecl(m, C_t) = T_r\ m(\overline{T_x \gg T_x'\ x})\ [P_t\ C_t \gg T_t']$,
   $\Delta' = \Delta_0\!\downarrow, s_t : T_t', \overline{s_x : T_x'}$*, and* $T = T_r$*.*

7. *If* $\Delta \vdash s.f : T \dashv \Delta'$ *then* $\Delta \vdash s : P\ C \dashv \Delta_0$*,* $k'(D')\ C'\ f \in \mathit{fields}(C)$*,* $T = \textsf{pure}(D')\ C'$*, and* $\Delta' = \Delta$*.*

8. *If* $\Delta \vdash s_0.f :=: s_1 : T \dashv \Delta'$ *then* $\Delta \vdash s_0 : k(D)\ C \dashv \Delta_0$ *for* $k \in \{\ \textsf{full}, \textsf{shared}\}$*,* $T'\ f \in \mathit{fields}(C)$*,* $\Delta \vdash s_1 : T' \dashv \Delta'$*, and* $T = T'$*.*

9. *If* $\Delta \vdash \textsf{assert}\langle D\rangle(s) : T \dashv \Delta'$ *then* $\Delta = \Delta_0, s : P_0\ C$*,* $\Delta' = \Delta_0, s : P_0\ D$*, and* $T = \textsf{Void}$*.*

Proof. Immediate from the specification of $\Delta \vdash e : T \dashv \Delta$. □

Lemma 12 (Weakening)**.**

1. *If* $\Delta \vdash e : T \dashv \Delta'$ *then* $\Delta, b : T_0 \vdash e : T \dashv \Delta', b : T_1$ *where* $T_1 <: T_0{\downarrow}$*.*

2. *If* $\Delta, b : T_0 \vdash e : T \dashv \Delta', b : T_2$ *and* $T_1 <: T_0$ *then* $\Delta, b : T_1 \vdash e : T \dashv \Delta', b : T_3$ *for* $T_3 <: T_2$

Proof. By induction on derivations of $\Delta \vdash e : T \dashv \Delta'$ □

Lemma 13 (Strengthening)**.**
If $\Delta, b : T_0 \vdash e : T \dashv \Delta', b : T_1$ *and b does not occur in e, then* $\Delta \vdash e : T \dashv \Delta'$

Proof. By induction on derivations of $\Delta \vdash e : T \dashv \Delta'$ □

Lemma 14 (Substitution)**.** *If* $\Delta, x : T_1 \vdash e : T_2 \dashv \Delta'$ *then* $\Delta, l : T_1 \vdash [l/x]e : T_2 \dashv [l/x]\Delta'$ *for l fresh.*

Proof. Substitute l for x throughout the proof of $\Delta, x : T_1 \vdash e : T_2 \dashv \Delta'$. □

Lemma 15 (Split Consistency)**.** *If* $k_0 \Rrightarrow k_1/k_2$ *then* $k_1(C) \leftrightarrow k_2(C)$*.*
Furthermore, if $k_0(C_0) \leftrightarrow k_1(C_1)$ *then*

1. *if* $k_0 \Rrightarrow k_0'$ *then* $k_0'(C_0) \leftrightarrow k_1(C_1)$*; and*

2. *if* $k_1 \Rrightarrow k_1'$ *then* $k_1'(C_1) \leftrightarrow k_0(C_0)$*.*

Proof. The first part is easily shown by cases analysis of $k_0 \Rrightarrow k_1/k_2$ derivations. The second part is proven by induction on derivations of $k_0(C_0) \leftrightarrow k_1(C_1)$.
Case (pure). Then $k_0(C_0) \leftrightarrow \textsf{pure}(C_1)$ and $C_0 <: C_1$.

1. if $k_0 \Rrightarrow k_0'$ then $k_0'(C_0) \leftrightarrow \textsf{pure}(C_1)$ by (pure).

2. then $\textsf{pure} \Rrightarrow \textsf{pure}$, and (pure) applies.

Case (shared). Then $\textsf{shared}(C_0) \leftrightarrow \textsf{shared}(C_0)$.

1. Suppose $\textsf{shared} \Rrightarrow k$. Then proceed by cases.

   (a) If $\textsf{shared} \Rrightarrow \textsf{shared}$ then (shared) applies.

(b) If $\mathsf{shared} \Rrightarrow \mathsf{pure}$ then $\mathsf{pure}(C_0) \leftrightarrow \mathsf{shared}(C_0)$ by (pure) then (sym).

2. Symmetric to the preceding case.

Case (sym). Follows immediately from the inductive case.

□

Corollary 16. *If $k(D)\ C \leftrightarrow P'\ C'$ and $k \Rrightarrow k_1/k_2$ then $\langle(k_1(D), k_2(D), P'), \leftrightarrow\rangle$ is connected.*

Lemma 17 (Subpermission Consistency)**.** *If $P_0 \leftrightarrow P_1$ then*

1. *if $P_0 <: P'$ then $P' \leftrightarrow P_1$; and*
2. *if $P_1 <: P''$ then $P'' \leftrightarrow P_0$.*

Proof. By induction on derivations of $P_0 \leftrightarrow P_1$
Case (pure). Then $P_0 = k(C) \leftrightarrow \mathsf{pure}(D) = P_1$ for $C <: D$.

1. Suppose $P_0 = k(C) <: P'$. Then $P' \leftrightarrow \mathsf{pure}(D) = P_1$. Proof is by induction on derivations of $k(C) <: P'$.

   Case (pure-guarantee). Then $P_0 = \mathsf{pure}(C) <: \mathsf{pure}(E) = P'$ for $C <: E$. Since $C <: E$ and $C <: D$, then D and E are related by subclassing. Therefore, $P' = \mathsf{pure}(E) \leftrightarrow \mathsf{pure}(D) = P_1$, either by (pure) or by (pure) followed by (sym).

   Case (full-guarantee). Then $P_0 = \mathsf{full}(C) <: \mathsf{full}(E) = P'$ for $E <: C$. Since $E <: C <: D$ then $P' = \mathsf{full}(E) \leftrightarrow \mathsf{pure}(D) = P_1$ by (pure).

   Case (split). Follows from the Split Consistency Lemma.

   Case (trans). Follows directly from the inductive cases: $P_1 = k(C) <: P_2$ and $P_2 <: P'$ for some P_2. By the induction hypothesis, $P_2 \leftrightarrow P_1$, and again by the induction hypothesis, $P' \leftrightarrow P_1$.

2. Suppose $P_1 = \mathsf{pure}(D) <: P''$. Then $P'' \leftrightarrow k(C) = P_0$. Proof is by induction on derivations of $\mathsf{pure}(D) <: P''$.

   Case (pure-guarantee). Then $P_1 = \mathsf{pure}(D) <: \mathsf{pure}(F) = P''$ for $D <: F$. Since $C <: D <: F$, $P_0 = \mathsf{pure}(C) \leftrightarrow \mathsf{pure}(F) = P''$ by (pure), so $P'' \leftrightarrow P_0$ by (sym).

   Case (full-guarantee). Does not apply.

   Case (split). Follows from the Split Consistency Lemma.

   Case (trans). Follows directly from the inductive cases.

Case (shared). Then $P_0 = \mathsf{shared}(C) \leftrightarrow \mathsf{shared}(C) = P_1$.

1. Suppose $P_0 = \mathsf{shared}(C) <: P'$. then $P' \leftrightarrow \mathsf{shared}(C) = P_1$. Proof is by induction on derivations of $\mathsf{shared}(C) <: P'$.

   Case (pure-guarantee). Does not apply

Case (full-guarantee). Does not apply

Case (split). Follows from the Split Consistency Lemma.

Case (trans). Follows directly from the inductive cases.

2. Identical.

Case (sym). Follows immediately from the induction hypothesis.

□

Lemma 18 (Reference Consistency)**.** *If* $\Delta \vdash b : P\ C \dashv \Delta'$ *then*

- $\Delta = \Delta_0, b : P_0\ C_0$,
- $\Delta' = \Delta_0, b : P_1\ C_1$, *and*
- *if* $P_0 \leftrightarrow P'$ *for any* P' *then* $\langle (P, P_1, P'), \leftrightarrow \rangle$ *is connected.*

Proof. By inversion, we can show that $P_0\ C_0 <: P\ C$ and $P_0\ C_0 <: P_1\ C_1$ from which it follows that $P' \leftrightarrow P$ and $P' \leftrightarrow P_1$. It remains to show that $P \leftrightarrow P_1$, which is straightforward. □

Corollary 19. *Let* $\overline{b}$ *be a sequence on* $\{\, \overline{b_i}\,\}$.
If $\Delta \vdash \overline{b : T} \dashv \Delta'$.
then $\Delta = \Delta_0, \overline{b_i : T_i}$, $\Delta' = \Delta_0, \overline{b_i : T_i'}$, *and for each* $b_i \in \{\, \overline{b_i}\,\}$ *either:*

1. $T_i = T_i' = T = \textsf{Void}$; *or*
2. $T = P\ C$, $T_i = P_i\ C_i$, $T_i' = P_i'\ C_i'$ *and collecting all typings* $b_i : \overline{P''\ C''}$ *in* $\overline{b : P\ C}$*, If* $P_i \leftrightarrow P$ *for any* P*, then* $\langle (P_i', \overline{P''}), \leftrightarrow \rangle$ *is connected.*

Proof. By induction on the length of $\overline{l}$. □

Lemma 20 (Demoted Class Consistency)**.** *If* $k(D) \leftrightarrow k'(D')$ *and* $k \in \{\,\textsf{full}, \textsf{shared}\,\}$ *then If* $C <: D$ *then* $C <: D'$.

Proof. By induction on derivations of $k(D) \leftrightarrow k'(D')$ (with induction strengthening to account for symmetry). □

Theorem 21 (HDRef)**.** *If* e *is a closed expression, hdref*(e)*, and* $\mu, \rho, e \to \mu', \rho', e'$*, then hdref*$(e')$*,*

Proof. By induction on $\mu, \rho, e \to \mu', \rho', e'$

Case (lookup). Then $\mu, \rho, l \to \mu, \rho, \rho(l)$. Then $\rho(l)$ is either $\textsf{void}$ or o, both of which are head references.

Case (new). Then $\mu, \rho, \textsf{new}\ C(\overline{l}) \to \mu[o \mapsto C(\overline{\rho(l)})], \rho, o$, and o is a head reference.

Case (update). Then $\mu, \rho, (l \leftarrow C(\overline{l'})) \to \mu[\rho(l) \mapsto C(\overline{\rho(l')})], \rho, \textsf{void}$, and $\textsf{void}$ is a head reference.

Case (ref). Then $\mu, \rho, l.f \to \mu, \rho, o$, and o is a head reference.

Case (assert). Then $\mu, \rho, \textsf{assert}\langle C \rangle(l) \to \mu, \rho, \textsf{void}$ and $\textsf{void}$ is a head reference.

Case (swap). Then $\mu, \rho, l_0.f :=: l_1 \to \mu[\rho(l_0) \mapsto [\rho(l_1)/o'_i]C(\overline{o'})], \rho, o'_i$ and o'_i is a head reference.

Case (let). Then $\mu, \rho, \textsf{let } x = v \textsf{ in } e_1 \to \mu, \rho[l \mapsto v], [l/x]e_1$ for fresh l. From $hdref(e)$ we have that there are no object references o in e_1, so the same follows for $[l/x]e_1$.

Case (invoke). Then $\mu(\rho(l)) = C_o(\cdots)$,
$method(m, C_o) = T_r\ m(\overline{T' \gg T''\ x})\ [P_t\ C_s \gg T'_t]\{\ \textsf{return } e'\ \}$,
and $\mu, \rho, e \to \mu, \rho, [l/\textsf{this}][\overline{l'/x}]e'$. The method body e' by definition contains no object references o, so the same is true for $[l/\textsf{this}][\overline{l'/x}]e'$

Case (congr). Then $\mu, \rho, e \to \mu', \rho', \textsf{let } x = e'_1 \textsf{ in } e_2$ where $\mu, \rho, e_1 \to \mu', \rho', e'_1$. By assumption, the initial $\textsf{let}$ expression is in head reference form, so e_2 contains no object references. Furthermore, by the induction hypothesis, e'_1 is in head reference form. It follows then that e' is also in head reference form.

□

Theorem 22 (Preservation). *If e is a closed expression, $\Delta \vdash e : T \dashv \Delta''$, μ, Δ, ρ **ok**, $hdref(e)$, and $\mu, \rho, e \to \mu', \rho', e'$ then for some Δ', $\Delta' \vdash e' : T \dashv \Delta'''$, μ', Δ', ρ' **ok**, and $\Delta''' <^l \Delta''$.*

Proof. By induction on $\mu, \rho, e \to \mu', \rho', e'$.

Case (lookup). Since e is closed, $e = l$. Suppose μ, Δ, ρ **ok**. Then there are two cases to consider:

1. If $T = \textsf{Void}$ then by inversion, $l : \textsf{Void} \in \Delta$ and $\Delta = \Delta''$. Type consistency then dictates that $\rho(l) = \textsf{void}$. Thus $\mu, \rho, l \to \mu, \rho, \textsf{void}$ and
   $\Delta \vdash \textsf{void} : \textsf{Void} \dashv \Delta$.

2. If $T = k(D)\ C$ then by inversion,

   $\Delta = \Delta_0, l : T_0, T_0 \Rrightarrow T_1/T_2, \Delta'' = \Delta_0, l : T_2$; and $T_1 <: T$.

   The (lookup) rule dictates that $\mu, \rho, l \to \mu, \rho, v$ where $v = \rho(l)$.

   If $T = \textsf{Void}$, then so do T_0, T_1, and T_2, and by type consistency, $v = \textsf{Void}$ so $\Delta \vdash \textsf{void} : \textsf{Void} \dashv \Delta$.

   If $T = k(D)\ C$, then $T_i = k_i(D_i)\ C_i$ and by type consistency $v = o$.

   Since l contains no os, we can assume without loss of generality that Δ_0 does not bind o. Let $\Delta' = \Delta'', o : T$. Then
   $$\Delta'', o : T \vdash o : T \dashv \Delta'', o : T'$$
   for some T', by the (ctx-obj) rule.

   Furthermore, type consistency dictates that $k_0(D_0)$ be a consistent permission to o, and by the Reference Consistency Lemma, $k(D), k_1(D_0)$ can replace it. Thus, μ, Δ', ρ **ok**.

Case (new). Since e is closed $e = \mathsf{new}\ C(\overline{l})$. Suppose μ, Δ, ρ **ok**. By inversion, $T = \mathsf{full}(\mathsf{Object})\ C$ and $\Delta = \Delta_0, \overline{l_i : T_i}$ and $\Delta'' = \Delta_1, \overline{l_i : T_i'}$ for $l_i \in \overline{l}$; and $\Delta \vdash \overline{l : T_f} \dashv \Delta''$ where $\mathit{fields}(C) = \overline{T_f\ f}$.

The (new) rule dictates that $\mu, \rho, e \to \mu[o \mapsto C(\overline{\rho(l)})], \rho, o$ for $o \notin \mathit{dom}(\mu)$.

Let $\Delta' = \Delta'', o : \mathsf{full}(\mathsf{Object})\ C$. Then

$$\Delta'', o : \mathsf{full}(\mathsf{Object})\ C \vdash o : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta'', o : \mathsf{pure}(\mathsf{Object})\ C.$$

By the Reference Consistency lemma, the permissions to $\overline{l_i}$ are consistently split between the fields $\overline{T_f\ f}$ of object $C(\overline{\rho(l)})$ and the indirect references $\overline{l_i : T_i'}$. Furthermore, $o \notin \mathit{dom}(\mu)$ combined with type consistency means that neither Δ_0 nor ρ refer to o, so its reference consistency is immediate. Finally, $o \in \mathit{dom}(\mu')$; so μ', Δ', ρ **ok**.

Case (update). Since e is closed, $e = l_t \leftarrow C(\overline{l'})$. Suppose μ, Δ, ρ **ok**.

By inversion, $\Delta \vdash \overline{l' : T_f}, l_t : k_t(D_t)\ C_t \dashv \Delta_2$ where $k_t \in \{\mathsf{full}, \mathsf{shared}\}$, $\Delta = \Delta_0, l_t : T_t, \overline{l' : T'}$ for some $T_t, \overline{T'}$, $\Delta_2 = \Delta_0, l_t : T', \overline{l' : T''}$ for some $T', \overline{T''}$, $\mathit{fields}(C) = \overline{T_f\ f}$, and $C <: D_t$. Furthermore, $\Delta'' = \Delta_2\!\downarrow, l_t : k_t(D_t)\ C$.

The (update) rule dictates that $\mu, \rho, (l \leftarrow C(\overline{l'})) \to \mu[\rho(l) \mapsto C(\overline{\rho(l')})], \rho, \mathsf{void}$.

Let $\Delta' = \Delta''$. Then $\Delta' \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta'$ is well-typed. By the Reference Consistency lemma, the permissions to $\overline{l'}$ are consistently split between the fields $\overline{T_f\ f}$ of object $C(\overline{l'})$ and the indirect references $\overline{l' : T_i'}$. The reference to l_t also retains consistent permissions.

The update operation changes $\mu(\rho(l_t))$'s runtime class identity to C, but the Demoted Class Consistency lemma ensures that all context references to $\mu(\rho(l_t))$ have class designations that are consistent with the new class, and since field references are invariant under demotion, they too are consistent with the new class identity.

Case (ref). Since e is closed, $e = l.f$. Suppose μ, Δ, ρ **ok**. By inversion, $\Delta = \Delta''$, $\Delta \vdash l : k(D)\ C \dashv \Delta_0$, and $k'(D')\ C'\ f \in \mathit{fields}(C)$. The (ref) rule dictates that $\mu, \rho, e \to \mu, \rho, o_i$ for $\mu(\rho(l)) = C(\overline{o})$.

Since $l.f$ contains no object refs o, we can assume without loss of generality that $o_i \notin \mathit{dom}(\Delta)$. Let $\Delta' = \Delta, o_i : \mathsf{pure}(D_i)\ C_i$. Then
$\Delta' \vdash o_i : \mathsf{pure}(D_i)\ C_i \dashv \Delta'$.

Since $k'(D')\ C'$ is a consistent reference to o, so is $\mathsf{pure}(D')\ C'$ by the Split Consistency lemma. So μ, Δ', ρ **ok**

Case (assert). Since e is closed, $e = \mathsf{assert}\langle C\rangle(l)$ Suppose μ, Δ, ρ **ok**. By inversion, $\Delta = \Delta_0, l : P\ C_0$, $\Delta'' = \Delta_0, l : P\ C$, and $T = \mathsf{Void}$.

The (assert) reduction rule dictates that $\mu, \rho, e \to \mu, \rho, \mathsf{void}$ and that $\mu(\rho(l))$ is an object of a subclass of C.

Let $\Delta' = \Delta''$. Then $\Delta' \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta'$. The subclass requirement for reducing assert ensures that μ, Δ', ρ **ok**.

Case (swap). Since e is closed, $e = l_0.f :=: l_1$. Suppose μ, Δ, ρ **ok**. By inversion, $\Delta \vdash l_0 : k(D)\ C \dashv \Delta_0$ for $k \in \{\mathsf{full}, \mathsf{shared}\}$, $T'\ f \in \mathit{fields}(C)$, $\Delta \vdash l_1 : T' \dashv \Delta''$, and $T = T'$. By inversion again, $\Delta = \Delta_1, l_1 : T_0$, $\Delta'' = \Delta_1, l_1 : T''$ for some T''.

The (swap) reduction rule says that $\mu, \rho, e \to \mu[\rho(l_0) \mapsto [\rho(l_1)/o_i']C(\overline{o'})], \rho, o_i'$ for $\mu(\rho(l_0)) = C(\overline{o'})$.

Let $\Delta' = \Delta'', o'_i : T'$ and then $\Delta' \vdash o'_i : T \dashv \Delta_1, l_1 : T'', o'_i : T_o$ for some T_o. So $\Delta''' = \Delta_1, l_1 : T'', o'_i : T_o$. The reference to o'_i in Δ' replaces the field reference that was in $\mu(\rho(l_0))$. Furthermore, the reference to l_1 in Δ' is compatible with the new field reference to the same object by the Split Consistency Lemma. Thus μ', Δ', ρ **ok**.

Case (let). Since e is closed, $e = \mathsf{let}\ x = v\ \mathsf{in}\ e_1$. Suppose μ, Δ, ρ **ok**. By inversion, $\Delta \vdash v : T_1 \dashv \Delta_1$ and $\Delta_1, x : T_1 \vdash e_1 : T \dashv \Delta'', x : T'_1$.

By the (let) reduction rule, $\mu, \rho, e \rightarrow \mu, \rho[l \mapsto v], [l/x]e_1$ for fresh l.

There are two cases for v.

1. If $v = \mathsf{void}$ then $\Delta_1 = \Delta$ and $T_1 = \mathsf{Void}$.

   Let $\Delta' = \Delta, l : \mathsf{Void}$, Then by Substitution, $\Delta' \vdash [l/x]e_1 : T \dashv \Delta'', l : T'_1$. Then the extensions to both Δ and ρ are consistent and do not affect permissions, so μ, Δ', ρ' **ok**.

   Furthermore, since $\mathit{hdref}(e)$, and $e = E[\mathsf{void}]$ it follows that there are no object refs in $[l/x]e_1$, so $\mathit{hdref}([l/x]e_1)$.

2. If $v = o$ then by inversion on the typing of o and reference consistency, $\Delta = \Delta_0, o : T_0$ and T_1's permissions are a consistent replacement for T_0.

   Let $\Delta' = \Delta_0, l : T_1$. Then by Substitution, $\Delta' \vdash [l/x]e_1 : T \dashv \Delta'', l : T'_1$. Since T_1 is a consistent replacement for T_0, μ, Δ', ρ' **ok**.

Case (invoke). Since e is closed, $e = l.m(\overline{l'})$. Suppose μ, Δ, ρ **ok**. By inversion, $\Delta \vdash l : P_t\ C_t, \overline{l' : T'} \dashv \Delta_0, l : T''_t, \overline{l' : T''}$,
$\mathit{mdecl}(m, C_t) = T_r\ m(\overline{T' \gg T''\ x})\ [P_t\ C_t \gg T'_t]$,
$\Delta'' = \Delta_0{\downarrow}, l : T'_t, \overline{l' : T''}$,
and $T = T_r$.

By the (invoke) reduction rule, $\mu(\rho(l)) = C_o(\cdots)$,
$\mathit{method}(m, C_o) = T_r\ m(\overline{T' \gg T''\ x})\ [P_t\ C_s \gg T'_t]\{\ \mathsf{return}\ e'\ \}$,
and $\mu, \rho, e \rightarrow \mu, \rho, [l/\mathsf{this}][\overline{l'/x}]e'$.

By the definition of method that the method to be called was defined in some superclass C_s of C_o (easy proof by induction). Also, we know by the well-formedness of the class table that the method definition is ok in C_s.

Since the method is ok in C_s, it follows that
$\overline{x : T'}, \mathsf{this} : P_t\ C_s \vdash e' : T_r \dashv \Delta_m$, and $\Delta_m \vdash \mathsf{this} : T'_t, \overline{x : T''} \dashv \Delta'_m$.

Since $C_o <: C_s$, $\overline{x : T'}, \mathsf{this} : P_t\ C_o \vdash e' : T_r \dashv \Delta_m$ (another easy induction). By substitution we can type $[l/\mathsf{this}][\overline{l'/x}]e'$ at these types: $\overline{l' : T'}, l : P_t\ C_o \vdash [l/\mathsf{this}][\overline{l'/x}]e' : T_r \dashv [l/\mathsf{this}][\overline{l'/x}]\Delta_m$.

Since $\mu(\rho(l)) = C_o(\cdots)\ l : P_t\ C_o$ is a consistent replacement for $l : P_t\ C_t$.

Then we can weaken the input context to $\Delta' = \Delta''$. It follows that μ, Δ', ρ **ok**.

Furthermore, the output context Δ''' will preserve the types of the bindings $\Delta_0{\downarrow}$ since they do not occur in e'

Case (congr). Since e is closed, $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ where e_1 is closed. Suppose μ, Δ, ρ **ok**. By inversion, $\Delta \vdash e_1 : T_1 \dashv \Delta_1$ and $\Delta_1, x : T_1 \vdash e_2 : T \dashv \Delta'', x : T'_1$.

By the (congr) reduction rule, $\mu, \rho, e \rightarrow \mu', \rho', \textsf{let}\ x = e_1'\ \textsf{in}\ e_2$ where $\mu, \rho, e_1 \rightarrow \mu', \rho', e_1'$.

By the induction hypothesis, $\Delta' \vdash e_1' : T_1 \dashv \Delta_1'$ and μ', Δ', ρ' **ok**, That $\Delta_1' <^l \Delta_1$, and $\mathit{hdref}(e_1')$. It follows by Weakening that $\Delta_1', x : T_1 \vdash e_2 : T \dashv \Delta''', x : T_1'$ for some $\Delta''' <^l \Delta''$.

The induction hypothesis says nothing about o bindings in Δ_1 or Δ_1'. Nonetheless, by $\mathit{hdref}(e)$ there are no object references o in e_2, so by the Strengthening Lemma, any o bindings in Δ_1 and Δ_1' can be ignored.

□

References

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 301–320, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297050.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(87)90045-4.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/503502.503505.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp Symb. Comput.*, 6(3-4):289–360, 1993. ISSN 0892-4635. doi: http://dx.doi.org/10.1007/BF01019462.