Gradual Featherweight Typestate

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

July 2010
CMU-ISR-10-116

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

Abstract

Typestate oriented programming integrates notions of typestate directly into the semantics of an object-oriented programming language. This document presents the formalization of Gradual 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.

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: gradual typing, hybrid types, access permissions, state guarantees

1 Introduction

What follows is a formalization of a system for typestate-oriented programming, with an emphasis on permission checking. This system combines static and dynamic permission checking. In another document, we formalize a purely static version of a typestate-oriented programming system.

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 Gradual Featherweight Typestate, or GFT for short.

We do not provide a runtime semantics for GFT. Instead, we provide a type-directed translation to an internal language, we call GFTIL. We provide a statics and dynamics for GFTIL, and prove type safety for that language. We also prove that a translation from GFT to GFTIL preserves typing.

2 Source Language

We now present a formal model for a language with integrated support for gradual typestate. The language is inspired by Featherweight Java (FJ) [Igarashi et al., 2001], so we call it Gradual Featherweight Typestate (GFT). Garcia et al. [2010] formalizes a fully static variant of GFT, called Featherweight Typestate.

2.1 Syntax

Figure 1 presents GFT's syntax.

As notational conventions, smallcaps (e.g. FIELDNAMES) indicate syntactic categories, italics (e.g. C) indicate metavariables, and sans serif (e.g. Object) indicates particular elements of a category. Overbars (e.g. $\overline{A}$) indicate possibly empty sequences (e.g. $A_1, ..., A_n$). GFT assumes a number of primitive notions, such as identifiers and method, field, and class names. The this keyword is a distinguished identifier that is bound to the subject of a method call. The Object keyword is a distinguished class name, indicating the top of subclass hierarchies.

A GFT program PG is a list of class declarations $\overline{CL}$ paired with an expression e. Each class declares its superclass and contains a list of field declarations F and method definitions M. Each GFT class has an implicit constructor that assigns an initial value to each field. The parameters of methods M are annotated with its input and output states, $T_1 \gg T_2\ x$. The method itself carries an annotation (in square brackets) for the receiver object this. Method signatures N are used to modularly typecheck code without the need to analyze the method bodies.

A class table CT is a mapping between class names C and classes CL. As many rules of GFT depend on the class table, for simplicity, we always assume a fixed CT.

Several helper judgments are used throughout the formalism of the language. $fields(C) = \overline{T\ f}$ yields that the types T and names f of the fields of class C. $method(m, C) = M$ yields the method m on class C, and accounts for method overloading. $mdecl(m, C) = N$ behaves equivalently, but yields the method signature N. The reflexive, antisymmetric, transitive $<:$ is the subclass relation.

Expressions The let expression let $x = e_1$ in e_2 is essentially standard. However, an optional type ascription provides fine-grained control over how permissions are distributed to the bound variable (Section 2.2). GFT expressions are restricted to A-normal form [Sabry and Felleisen, 1993], so let expressions explicitly sequence all complex operations. This restriction simplifies the description of the type system, which relies

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

$$
\begin{array}{rcll}
PG & ::= & \langle \overline{CL}, e \rangle & \text{(programs)} \\
CL & ::= & \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F};\ \overline{M}\ \} & \text{(classes)} \\
F & ::= & T\ f & \text{(fields)} \\
N & ::= & T\ m(\overline{T \gg T})\ [T \gg T] & \text{(method signatures)} \\
M & ::= & T\ m(\overline{T \gg T\ x})\ [T \gg T] & \\
 & & \{\ \mathsf{return}\ e;\ \} & \text{(methods)} \\
T & ::= & P\ C \mid \mathsf{Void} \mid \mathsf{Dyn} & \text{(types)} \\
P & ::= & k(D) & \text{(permissions)} \\
k & ::= & \mathsf{full} \mid \mathsf{shared} \mid \mathsf{pure} & \text{(access permissions)} \\
e & ::= & x \mid \mathsf{let}\ x : T = e\ \mathsf{in}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(expressions)} \\
 & \mid & \mathsf{new}\ C(\overline{x}) \mid x.f \mid x.m(\overline{x}) \mid x.f :=: x & \\
 & \mid & x \leftarrow C(\overline{x}) \mid \mathsf{hold}[x : T](e) \mid \mathsf{assert}\langle T \rangle(x) & \\
\Delta & ::= & \overline{x : T} & \text{(type contexts)}
\end{array}
$$

$\boxed{C <: C}$ Subclass

$$
\frac{}{C <: C} \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F};\ \overline{M}\ \}}{C <: D}
$$

$\boxed{\mathit{fields}(C) = \overline{T\ f}}$ Class Field Declarations

$$
\frac{}{\mathit{fields}(\mathsf{Object}) = \cdot} \qquad \frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{T\ f};\ \overline{M}\ \} \qquad \mathit{fields}(D) = \overline{T'\ f'}}{\mathit{fields}(C) = \overline{T'\ f'}, \overline{T\ f}}
$$

$\boxed{\mathit{method}(m, C) = M}$

$$
\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F};\ \overline{M}\ \} \qquad T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \} \in \overline{M}}{\mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}}
$$

$$
\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F};\ \overline{M}\ \} \qquad m \notin \overline{M} \qquad \mathit{method}(m, D) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}}{\mathit{method}(m, C) = T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}}
$$

$\boxed{\mathit{mdecl}(m, C) = N}$

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

Figure 1: Source Language Syntax and Auxiliary Functions

on sequencing to track typestate. We write $e_1; e_2$ as shorthand for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, where x does not occur in e_2. We assume throughout that variables bound by let expressions can be renamed as needed. We assume the same for parameters in method bodies.

The new expression heap-allocates an object of class C and populates its fields with the supplied values. The update operation $x_0 \leftarrow C(\overline{x_1})$ is the primary addition to the language specifically in support of typestate. It replaces the value of x_0 with the new object of class C, which may not be the same as x_0's current class. Updating an object is how GFT expresses typestate change. The field reference expression $x.f$ returns the current value of the f field of x. The expression $x_0.f :=: x_1$ is a swapping assignment: it replaces the current value of the field $x_0.f$ with the value of x_1 and returns the old value as its result. The field read expression does not give up any of the permissions to an object held by the field being read. In contrast, swapping assignment yields all permissions to the old value of the field $x_0.f$, replacing it with the new value x_1. The method invocation $x_0.m(\overline{x_1})$ executes the m method with x_0 bound to this and the arguments $\overline{x_1}$ bound to the method parameters. The expression $\mathsf{hold}[x : T](e)$ captures the amount of x's permissions denoted by T for the duration of the computation e. When e completes, these permissions are merged back into x. The expression $\mathsf{assert}\langle T\rangle(x)$ is like a cast, but rather than returning a value of the given type it changes the type of the target variable.

Types The type of a GFT object reference has two components, its permission P and its class (or *state*) C. The permission can be broken down further into its access permission k and state guarantee D. We write these *static object reference types* in the form $k(D)\ C$. Dyn is a *dynamic object reference type*, and is treated by the type system with greater leniency than the statically typed object references. Type checks on Dyn objects are deferred to runtime. The Void type classifies expressions executed purely for their effects. No source-level values have the Void type.

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 is a subclass of D. From here forward, all types T are assumed to be well-formed.

2.2 Static Semantics

The GFT type system relies upon *linear type contexts* [Girard, 1987]. In GFT's type system, the types of identifiers vary over the course of a program. 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.

Managing Permissions Before we present typing judgments for Featherweight Typestate, we must explain how permissions are treated. 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.

Figure 2 presents several auxiliary judgments that specify how permissions may be safely split, and their relation to typing. First, *access 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 residual. When we are only concerned that a permission k_2 can be split 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 $k \Rrightarrow k$.

Permissions partially determine what operations are possible, as well as when an object can be safely bound to an identifier. 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

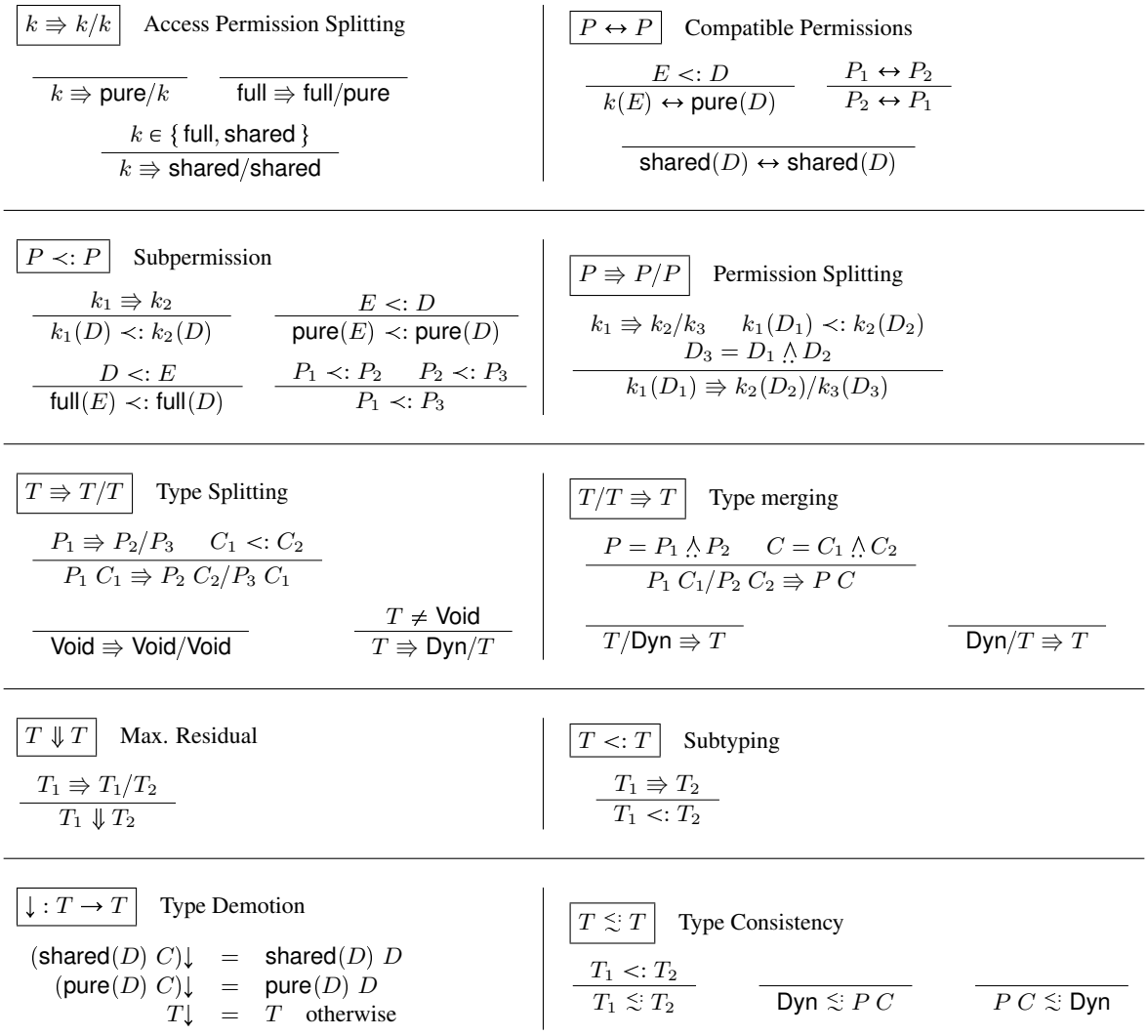

Figure 2: Permission and Type Management Relations

a reference with P_1 permissions may be used wherever an object reference with P_2 permissions is needed. As expected, the subpermission relation is reflexive and transitive. Splitting an access permission produces a lesser (or identical) permission. The rules that mention pure and full capture how state guarantees affect the strength of permissions. Pure permissions covary with their state guarantee because a pure reference with a superclass state guarantee assumes less reading capability. Full permissions contravary with their state guarantee because a full reference with a subclass state guarantee assumes less writing capability (it can update to fewer possible states).

Permission splitting extends access permission splitting by accounting for state guarantees. First, if $k_1(D_1) <: k_2(D_2)$, splitting is safe. The question is to determine the proper residual permission $k_3(D_3)$. k_3 is obtained by splitting k_2 from k_1. The resulting state guarantee D_3 is the greatest lower bound of D_1 and D_2 in the subclass hierarchy, denoted $D_1 \mathbin{\dot{\wedge}} D_2$ (it is required that either $D_1 <: D_2$ or $D_2 <: D_1$).

Permission splitting in turn extends to *type splitting* $T \Rrightarrow T/T$, taking subclasses into account for object references. The Void type can be arbitrarily split into multiple Void types. Any object reference type may split off a Dyn while retaining the original type as residual. We use type splitting to define the notion of subtyping $T <: T$ used in GFT. As with base permission splitting, we write $P_1 \Rrightarrow P_2$ or $T_1 \Rrightarrow T_2$ to express that P_2 or T_2 can be split from P_1 or T_1 respectively.

The *maximum residual* relation $T_1 \Downarrow T_2$ specializes type splitting for the case where all the permissions to an object are acquired. The result type T_2 is what is leftover; for instance,
$\mathsf{full}(D)\ C \Downarrow \mathsf{pure}(D)\ C$ and $\mathsf{shared}(D)\ C \Downarrow \mathsf{shared}(D)\ C$.

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 these cases, an object reference's class must revert to its state guarantee, which is a trusted state after an update. The *type demotion* function $T{\downarrow}$ expresses this restricting of assumptions. Note that full references need not be demoted since no other reference could have changed their states. We write $\Delta{\downarrow}$ for the compatible extension of demotion to typing contexts.

Type merging $T/T \Rrightarrow T$ describes how two separate permissions to an object may be combined. It is used to specify hold's semantics. Type merging is defined in terms of the $\dot{\wedge}$ and $\mathbin{\dot{\wedge}}$ relations, where $\dot{\wedge}$ is the analogue of $\mathbin{\dot{\wedge}}$ for subpermissions.

The *compatible 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. A reference with pure permissions is compatible with any other permission that respects its state guarantee, meaning it could only change state among its subclasses. Shared permissions are only compatible when they have the same state guarantee. A full permission is only compatible with pure permissions that respect its state guarantee.

Well-typed Expressions In contrast to a traditional type system, the GFT typing judgments are quaternary relations roughly of the form $\Delta_1 \vdash e : T \dashv \Delta_2$: given the typing assumptions Δ_1, the expression e can be assigned the type T and produces typing assumptions Δ_2 as its output. The assumptions in question are the types of each reference. Threading typing contexts through the typing judgment captures the flow-sensitivity of the type assumptions.

The type system specification is designed to both ensure determinism of our type system and also retain flexibility. Consider a candidate typing judgment for variable references.

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

It states that if x is assumed to have type T_1, and T_1 can be split into T_2 and T_3, then the expression x can be typed at T_2. Because T_2 may not be unique, a source program may be well-typed according to multiple

$\boxed{\Delta \vdash e \Leftrightarrow T \dashv \Delta}$ Source Expression Typing

$$\text{(ctx}\Rightarrow\text{)}\frac{T_1 \Downarrow T_2}{\Delta, x : T_1 \vdash x \Rightarrow T_1 \dashv \Delta, x : T_2} \qquad \text{(new}\Rightarrow\text{)}\frac{\overline{fields(C) = \overline{T'\ f}} \quad \overline{x : T \vdash x \Leftarrow T' \dashv x : T''}}{\Delta, \overline{x : T} \vdash \mathsf{new}\ C(\overline{x}) \Rightarrow \mathsf{full}(\mathsf{Object})\ C \dashv \Delta, \overline{x : T''}}$$

$$\text{(ctx}\Leftarrow\text{)}\frac{T_1 \Rrightarrow T_2/T_3}{\Delta, x : T_1 \vdash x \Leftarrow T_2 \dashv \Delta, x : T_3} \qquad \text{(invoke}\Rightarrow\text{)}\frac{\begin{array}{c}mdecl(m, C_1) = T\ m(\overline{T_x \gg T'_x})[T_t \gg T'_t] \\ P_1\ C_1 \lesssim T_t \qquad \overline{T_2 \lesssim T_x}\end{array}}{\begin{array}{l}\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash \\ \quad x_1.m(\overline{x_2}) \Rightarrow T \dashv \Delta\downarrow, x_1 : T'_t, \overline{x_2 : T'_x}\end{array}}$$

$$\text{(ctx}_d\Leftarrow\text{)}\frac{}{\Delta, x : \mathsf{Dyn} \vdash x \Leftarrow P\ C \dashv \Delta, x : \mathsf{Dyn}} \qquad \text{(invoke}_d\Rightarrow\text{)}\frac{\overline{T_2 \lesssim \mathsf{Dyn}}}{\begin{array}{l}\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash \\ \quad x_1.m(\overline{x_2}) \Rightarrow \mathsf{Dyn} \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}\end{array}}$$

$$\text{(let}\Leftrightarrow\text{)}\frac{\Delta \vdash e_1 \Rightarrow T_1 \dashv \Delta_1 \quad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \dashv \Delta', x : T'_1}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \dashv \Delta'} \qquad \text{(ref}\Rightarrow\text{)}\frac{T_2\ f \in fields(C_1) \qquad T_2 \Downarrow T'_2}{\Delta, x : P_1\ C_1 \vdash x.f \Rightarrow T'_2 \dashv \Delta, x : P_1\ C_1}$$

$$\text{(letT}\Leftrightarrow\text{)}\frac{\Delta \vdash e_1 \Leftarrow T_1 \dashv \Delta_1 \quad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \dashv \Delta', x : T'_1}{\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \dashv \Delta'} \qquad \text{(ref}_d\Rightarrow\text{)}\frac{}{\Delta, x : \mathsf{Dyn} \vdash x.f \Rightarrow \mathsf{Dyn} \dashv \Delta, x : \mathsf{Dyn}}$$

$$\text{(}\hat{e}\Leftarrow\text{)}\frac{\Delta \vdash \hat{e} \Rightarrow T_1 \dashv \Delta' \quad T_1 \lesssim T_2}{\Delta \vdash \hat{e} \Leftarrow T_2 \dashv \Delta'} \qquad \text{(hold}\Rightarrow\text{)}\frac{T_1 \Rrightarrow T_2/T_3 \qquad T_2\downarrow / T'_3 \Rrightarrow T'_1 \quad \Delta, x : T_3 \vdash e \Rightarrow T \dashv \Delta', x : T'_3}{\Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) \Rightarrow T \dashv \Delta', x : T'_1}$$

$$\text{(update}\Rightarrow\text{)}\frac{\begin{array}{c}k \in \{\mathsf{full}, \mathsf{shared}\} \\ C <: E \qquad \overline{fields(C) = \overline{T'_2\ f}} \\ \overline{x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \dashv x_2 : T''_2}\end{array}}{\begin{array}{l}\Delta, x_1 : k(E)\ D, \overline{x_2 : T_2} \vdash \\ \quad x_1 \leftarrow C(\overline{x_2}) \Rightarrow \mathsf{Void} \\ \quad \dashv \Delta\downarrow, x_1 : k(E)\ C, \overline{x_2 : T''_2\downarrow}\end{array}} \qquad \text{(swap}\Rightarrow\text{)}\frac{\begin{array}{c}k_1 \in \{\mathsf{full}, \mathsf{shared}\} \\ T'_2\ f \in fields(C_1) \\ x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \dashv x_2 : T''_2\end{array}}{\begin{array}{l}\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash \\ \quad x_1.f\ :=:\ x_2 \Rightarrow T''_2 \\ \quad \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T''_2\end{array}}$$

$$\text{(update}_d\Rightarrow\text{)}\frac{\overline{fields(C) = \overline{T'_2\ f}} \quad \overline{x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \dashv x_2 : T''_2}}{\begin{array}{l}\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash \\ \quad x_1 \leftarrow C(\overline{x_2}) \Rightarrow \mathsf{Void} \\ \quad \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : T''_2\downarrow}\end{array}} \qquad \text{(swap}_d\Rightarrow\text{)}\frac{T_2 \lesssim \mathsf{Dyn}}{\begin{array}{l}\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash \\ \quad x_1.f\ :=:\ x_2 \Rightarrow \\ \quad \mathsf{Dyn} \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2\end{array}}$$

$$\text{(assert}\Rightarrow\text{)}\frac{T \Rrightarrow T'}{\begin{array}{l}\Delta, x : T \vdash \\ \quad \mathsf{assert}\langle T'\rangle(x) \Rightarrow \mathsf{Void} \dashv \Delta, x : T'\end{array}} \qquad \text{(assert}_d\Rightarrow\text{)}\frac{T \neq \mathsf{Void} \qquad T' \neq \mathsf{Void}}{\begin{array}{l}\Delta, x : T \vdash \\ \quad \mathsf{assert}\langle T'\rangle(x) \Rightarrow \mathsf{Void} \dashv \Delta, x : T'\end{array}}$$

Figure 3: Source Language Static Typing Rules

derivations, with each derivation representing a different split of permissions between this particular variable reference and the remainder of the program. Although determinism is not important for the fully static case[1], nondeterminism is incompatible with dynamic permission assertions: such a system could succeed sometimes and fail other times if permissions could be transferred more than one way for the same code.

Rather than requiring type annotations for all variable references, we use the bidirectional typing approach of Pierce and Turner [2000] to structure the type system so that permission transfer is deterministic, and so type annotations can be used to selectively tune how permissions are split.

The type system is therefore structured as two mutually recursive judgments. The *type synthesis* judgment $\Delta_1 \vdash e \Rightarrow T \dashv \Delta_2$ conceptually analyzes the expression e in the context Δ_1 and synthesizes a type T for it; the type T is an output of the judgment, along with the output context Δ_2. The *type checking* judgment $\Delta_1 \vdash e \Leftarrow T \dashv \Delta_2$ checks that the expression e under the type context Δ_1 can be given the type T. The type T is conceptually an input to the judgment, and the only output is the context Δ_2. By convention, the synthesis rule names have a $\Rightarrow$ suffix, while the checking rule names have a $\Leftarrow$ suffix.

Figure 3 presents the typing rules for GFT expressions.

A variable reference is typed differently depending on whether its type is synthesized or checked. The synthesis rule (ctx$\Rightarrow$) yields maximal permissions to the referenced object. Its output context associates the maximum residual permissions to the variable. In contrast, the checking rule (ctx$\Leftarrow$) just ensures that the desired type can be split from the starting type, and leaves the corresponding residual in the output context. (ctx$_d$ $\Leftarrow$) states that dynamic object references may be typed as any static object reference type. Safety checks will be deferred until runtime.

Each of the typing rules for let represents both a checking and synthesis rule. Replacing the $\Leftrightarrow$ with $\Leftarrow$ gives the checking rule, which checks the type of e_2, and $\Rightarrow$ gives the synthesis rule, which synthesizes the type of e_2. The crucial difference between the (let$\Leftrightarrow$) and (letT$\Leftrightarrow$) is whether the bound expression's type is checked or synthesized. When the bound variable has a type ascription, $x : T_1$, the expression e_1 is checked against that type. If there is no type ascription, the type of e_1 is synthesized.

The typing rules for let and variable references combine to determine how permissions transfer between references. When a variable reference is bound to another variable, the new variable by default acquires maximal permissions to the referenced object; A type annotation on the let-bound variable can tune how permissions are transferred to a binding. For instance, assume x has type $\mathsf{full}(D)\ C$ and consider the two expressions:

(1) $\mathsf{let}\ y = x\ \mathsf{in}\ e$
(2) $\mathsf{let}\ y : \mathsf{shared}(D)\ C = x\ \mathsf{in}\ e$

In expression (1) y has $\mathsf{full}(D)\ C$ type and x has $\mathsf{pure}(D)\ C$ type in e, but in expression (2) both x and y have $\mathsf{shared}(D)\ C$ type.

Type checking can be treated uniformly for all other expressions. The ($\hat{e}$ $\Leftarrow$) rule schematically expresses checking for those expressions, which we indicate with $\hat{e}$. For all of them, type checking can be characterized simply in terms of type synthesis: an expression checks at type T_1 if its type synthesizes to some subtype T_2 of T_1. The rest of the expressions in the language only require type synthesis rules.

(update$\Rightarrow$) states that a variable reference can only be used to update an object if it has a write permission. Also, the target class of the update must respect the reference's state guarantee. The arguments to the constructor are type checked against the types of the target class's fields. The update operation is performed solely for its effect on the heap, so the type of the overall expression is Void. Finally, type assumptions from the input context are demoted (i.e. $\Delta\downarrow$) in the output context to ensure that any aliases to the updated object

[1]For instance, Featherweight Typestate uses non-deterministic typing rules [Garcia et al., 2010].

retain a conservative approximation of the object's current class. The output type of the updated object reflects its new class. (update$_d$ $\Rightarrow$) types the update expression when the target of the update is typed Dyn. Safety checks on the target are deferred until runtime.

(new$\Rightarrow$) expressions are given full permission with a maximally lenient state guarantee Object to a newly constructed object of class C. The arguments to the constructor are checked against the fields of C. The output type of the arguments is the residual type after permissions that are needed to be stored in the fields are split.

(invoke$\Rightarrow$) is typed according to the method signature $mdecl(m, C_1)$, as found in the class table. Parameters must be consistent with the declarations of the signature. The resulting expression, and output types of the parameters are also taken from the signature.

(invoke$_d$ $\Rightarrow$) is a method invocation of a dynamically typed receiver. All checks, such as the existence and arity of the method m and the types of the parameters, are deferred until runtime.

(ref$\Rightarrow$) is a field read typed at the maximum residual of the static object reference's field.

(ref$_d$ $\Rightarrow$) is a field read of a dynamically typed object. The existence of field f is deferred until runtime.

(hold$\Rightarrow$) is typed by typing the subexpression e after splitting T_2 from variable x. The resulting type of x is the merge of the demotion of the T_2 (the type being being held) and T_3', the resulting output type of x after evaluation of e.

(swap$\Rightarrow$) is a field swapping assignment. x_2 is checked against the type of the field f of the class of the statically typed object reference (C_1). The resulting type of the expression, T_2' is the type of the field (the old value will be returned). The output type of x_2 is the residual type after T_2' has been split.

(swap$_d$ $\Rightarrow$) a field swapping assignment for a dynamically typed object reference. The existence of the field, and requisite permission checking is deferred until runtime.

(assert$\Rightarrow$) and (assert$_d$ $\Rightarrow$) are asserts that are used purely for their effects of changing the argument's type. (assert$\Rightarrow$) expresses a statically safe assert (analogous to an upcast). (assert$_d$ $\Rightarrow$) is a statically safe assert (analogous to upcasts) and is used modify the type of a variable reference.

(assert$_d$ $\Rightarrow$) is similar, but requires a runtime check to acquire the requested permission associated with T'. The two separate rules are not required, but are done for clarity when we define the translation to the internal language.

Well-typed Programs Now that we have defined what it means for an expression to be well-typed, we can define a well-typed program. Figure 4 describes the relevant judgments.

A well-typed method signature N in class C must have the current class of the receiver match the class that it is defined in. An overridden method must match its overridden signature everywhere else. GFT does not support method overloading.

The body of a well-typed method M in class C must be typechecked against the return value of its signature. Types of parameters of the output context must be consistent with the output types declared in the signature.

The types of class fields have an interesting restriction: they must be invariant under demotion (i.e. $T{\downarrow} = T$). Since the types of fields do not change as a program runs, they must not be invalidated by update operations. This restriction ensures that field types remain compatible with other aliases to their objects. GFT does not support field overloading.

A well-typed class must consist of well-typed methods and fields. And a well-typed program must consist of well-typed classes and a well-typed main expression.

Not expressed in these rules, but implicitly understood are some other sanity conditions (the same as in Featherweight Java) regarding the typing of programs. All classes mentioned in the program must be

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

$$\frac{\textsf{class } C \textsf{ extends } D \textsf{ } \{ \textsf{ } \overline{F}, \overline{M} \textsf{ } \} \quad mdecl(D, m) = T_r \; m(\overline{T_x \gg T'_x})[P_t \; E \gg T'_t]}{T_r \; m(\overline{T_x \gg T'_x})[P_t \; C \gg T'_t] \textbf{ ok in } C}$$

$$\frac{\textsf{class } C \textsf{ extends } D \textsf{ } \{ \textsf{ } \overline{F}, \overline{M} \textsf{ } \} \quad mdecl(D, m) \text{ undefined}}{T_r \; m(\overline{T_x \gg T'_x})[P_t \; C \gg T'_t] \textbf{ ok in } C}$$

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

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

$\boxed{F \textbf{ ok}}$ Well-typed Field

$$\frac{T{\downarrow} = T}{T \; f \textbf{ ok}}$$

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

$$\frac{\overline{F \textbf{ ok}} \qquad \overline{M} \textbf{ ok in } C_0}{\textsf{class } C_0 \textsf{ extends } C_1 \textsf{ } \{ \textsf{ } \overline{F}; \; \overline{M} \textsf{ } \} \textbf{ ok}}$$

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

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

Figure 4: Source Language Program Typing Rules

defined (or Object). There are no cycles in the subclass hierarchy: therefore Object must be the superclass of all defined classes.

3 Internal Language

Gradual Featherweight Typestate leaves many aspects of dynamic permission management implicit. This section introduces an internal language, GFTIL, that makes these details explicit. GFT's semantics are then defined by type-directed translation to GFTIL.

3.1 Syntax

GFTIL is structured much like GFT but elaborates several concepts (Figure 5). First, the internal language introduces explicitly dynamic variants e_d of some operations from the source language. A dynamic operator takes a dynamically typed reference in its primary position (e.g. as receiver of an object method). Static operators require statically typed references.

Second, many expressions in the language carry explicit type information. This information is used to dynamically account for the flow of permissions as the program is evaluated. As shown below, these type annotations play a role in both the type system and the dynamic semantics.

Finally, it adds several constructs that only occur at runtime. Object references and indirect references point to runtime objects. Object references correspond to heap pointers; indirect references are an artifact that facilitates the type-safety proof. GFTIL is also in A-normal form, though at runtime the arguments to expressions are generalized to *simple expressions*: variable names or indirect references. The merge expression is used to specify the dynamic semantics of hold. The void value is the runtime result of expressions that return no value. Reference expressions come in two forms. A bare reference b signifies a variable or reference that is never used again. In contrast, a splitting reference $s[T \Rrightarrow T/T]$ explicitly specifies the

Language Syntax

x, this	$\in$	IDENTIFIERNAMES	
m	$\in$	METHODNAMES	
f	$\in$	FIELDNAMES	
C, D, E	$\in$	CLASSNAMES	
Object	$\in$	CLASSNAMES	
o	$\in$	OBJECTREFS	
PG	$::=$	$\langle \overline{CL}, e \rangle$	(programs)
CL	$::=$	class C extends D { $\overline{F}$, $\overline{M}$ }	(class declarations)
F	$::=$	$T\ f$	(fields)
M	$::=$	$T\ m(\overline{T \gg T\ x})\ [T \gg T]$ { return e; }	(methods)
N	$::=$	$T\ m(\overline{T \gg T})\ [T \gg T]$	(method signatures)
e	$::=$	$e_s \mid e_d \mid e_i$	(expressions)
e_s	$::=$	$b \mid$ void $\mid s[T \Rrightarrow T/T] \mid$ new $C(\overline{s})$	(static expressions)
	$\mid$	let $x = e$ in $e \mid$ release$[T](s)$	
	$\mid$	hold$[s : T \Rrightarrow T/T \gg T \Rrightarrow T](e)$	
	$\mid$	$s.f \mid s.m(\overline{s}) \mid s.f\ :=:\ s$	
	$\mid$	$s \leftarrow C(\overline{s}) \mid$ assert$\langle T \gg T \rangle(s)$	
e_d	$::=$	$s._d f \mid s._d m(\overline{s}) \mid s.f\ :=:_d\ s$	(dynamic expressions)
	$\mid$	$s \leftarrow_d C(\overline{s}) \mid$ assert$_\mathsf{d}\langle T \gg T \rangle(s)$	
e_i	$::=$	merge$[l : T/l : T \Rrightarrow T](e)$	(internal expressions)
s	$::=$	$x \mid l$	(simple expressions)
T	$::=$	$P\ C \mid$ Void $\mid$ Dyn	(types)
P	$::=$	$k(C)$	(permission and state guarantee)
k	$::=$	full $\mid$ shared $\mid$ pure	(permissions)
Δ	$::=$	$\overline{b : T}$	(linear type context)
b	$::=$	$x \mid l \mid o$	(context bindings)
l	$\in$	INDIRECTREFS	

Figure 5: Syntax of the internal language.

starting type, result type, and the residual type of the reference. The release$[T](s)$ expression explicitly releases a reference and its permissions, after which it can no longer be used.

3.2 Static Semantics

Because of GFTIL's explicit form, its type judgement
$\Delta \vdash e : T \dashv \Delta'$ does not need to be bidirectional. Furthermore, its type rules use the same permission and type management relations from the source language.

Well-typed Expressions Figure 6 presents GFTIL's typing rules. These rules exhibit some of the desired properties of the language. They enforce strict tracking of permissions. The rules check the input context Δ to force their arguments s to have *exactly* the type required. Furthermore, many expressions remove argument references s from the output context, so they cannot be reused later in the program. These restrictions force GFTIL to explicitly encode permission flow. Its dynamic semantics uses this encoding to implement permission tracking.

The (void) rule says that void has Void type. The (ctx-b) rule says that a bare reference has the type dictated by its context and is utterly consumed. The (ctx) rule is an explicit analogue of GFT's (ctx$\Leftarrow$) rule. The (new) rule checks that all its argument types match the class field specifications. The resulting object has full access permissions and the maximally lenient Object state guarantee. The (ref) rule yields the maximal residual type for the field $x.f$, since the object cedes no permissions. For a dynamic field read, (ref$_d$) returns a dynamically typed field reference. The (invoke) rule matches a method's arguments exactly against the method signature. Each argument's output type is dictated by the method's output states. For dynamic method calls, The (invoke$_d$) rule defers all checking to runtime. The (swap) rule checks that its first argument has write permission, and that its second argument's type exactly matches the swapped field. The expression's type matches the field. For dynamic references, the (swap$_d$) rule defers checking to runtime. The (update) and (update$_d$) rules almost mirror GFT's update rules except that its argument types must exactly match the class field specifications. The (rel) rule removes its argument from the type context. The (let) rule is similar to the unannotated GFT rule. However, if x is bound to an object reference type, then for tracking purposes x us required to be consumed by the end of the expression's body. The $\div$ operation indicates removing a possible x : Void binding from Δ. The (assert) and (assert$_d$) rules are explicit analogues of the GFT rules. The former is a safe assert, and is only present to perform explicit permission tracking. The later is a dynamic assert and may fail at runtime. The (hold) rule is the explicit analogue to the GFT typing rule. The (merge) rule expresses how merge annotates the expression e with the information needed to restore the held permissions T_1 back to reference l_2 after e completes. The type T_2' of l_2 after e completes is merged with T_1 to give l_2 type T_3. The type of e is the type of the whole expression.

Well-typed Programs GFTIL programs are typed according to Figure 7. Well-typed method signatures, fields, and classes are the same as for GFT programs, and shown only for completeness sake. Well-typed methods are more strict than their GFT counterparts in that their bodies must exactly match their signatures, and not simply be type consistent with them. Well-typed programs of GFTIL differ only in the syntax of their expression typing judgment.

3.3 Dynamic Semantics

GFTIL's dynamic semantics, presented in Figure 10, require several additional syntactic notions, defined in Figure 8. Ultimately, expressions in the language evaluate to values: void, the result of operations that

$\boxed{\Delta \vdash e : T \dashv \Delta}$ Internal Expression Statics

$$\text{(void)}\ \frac{}{\Delta \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta} \qquad \text{(invoke)}\ \frac{mdecl(m, C_1) = T_r\ m(\overline{T_2 \gg T_2'})[P_1\ C_1 \gg T_1'] \quad}{\Delta, s_1 : P_1\ C_1, \overline{s_2 : T_2} \vdash s_1.m(\overline{s_2}) : T_r \dashv \Delta\downarrow, s_1 : T_1', \overline{s_2 : T_2'}}$$

$$\text{(ctx-b)}\ \frac{}{\Delta, b : T \vdash b : T \dashv \Delta} \qquad \text{(invoke}_d\text{)}\ \frac{}{\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}} \vdash s_1.{}_d m(\overline{s_2}) : \mathsf{Dyn} \dashv \Delta\downarrow, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}}}$$

$$\text{(ctx)}\ \frac{T_1 \Rrightarrow T_2/T_3}{\Delta, s : T_1 \vdash s[T_1 \Rrightarrow T_2/T_3] : T_2 \dashv \Delta, s : T_3} \qquad \text{(swap)}\ \frac{k \in \{\,\mathsf{full}, \mathsf{shared}\,\} \quad (T_2\ f) \in \mathit{fields}(D)}{\Delta, s_1 : k(E)\ D, s_2 : T_2 \vdash s_1.f\ :=:\ s_2 : T_2 \dashv \Delta, s_1 : k(E)\ D}$$

$$\text{(ref)}\ \frac{(T\ f) \in \mathit{fields}(C) \quad T \Downarrow T'}{\Delta, s : P\ C \vdash s.f : T' \dashv \Delta, s : P\ C} \qquad \text{(swap}_d\text{)}\ \frac{}{\Delta, s_1 : \mathsf{Dyn}, s_2 : \mathsf{Dyn} \vdash s_1.f\ :=:_d\ s_2 : \mathsf{Dyn} \dashv \Delta, s_1 : \mathsf{Dyn}}$$

$$\text{(ref}_d\text{)}\ \frac{}{\Delta, s : \mathsf{Dyn} \vdash s.{}_d f : \mathsf{Dyn} \dashv \Delta, s : \mathsf{Dyn}} \qquad \text{(update)}\ \frac{k \in \{\,\mathsf{full}, \mathsf{shared}\,\} \quad C <: E \quad \mathit{fields}(C) = \overline{T\ f}}{\Delta, s_1 : k(E)\ D, \overline{s_2 : T} \vdash s_1 \leftarrow C(\overline{s_2}) : \mathsf{Void} \dashv \Delta\downarrow, s_1 : k(E)\ C}$$

$$\text{(new)}\ \frac{\mathit{fields}(C) = \overline{T\ f}}{\Delta, \overline{s : T} \vdash \mathsf{new}\ C(\overline{s}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta} \qquad \text{(update}_d\text{)}\ \frac{\mathit{fields}(C) = \overline{T\ f}}{\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : T} \vdash s_1 \leftarrow_d C(\overline{s_2}) : \mathsf{Void} \dashv \Delta\downarrow, s_1 : \mathsf{Dyn}}$$

$$\text{(rel)}\ \frac{}{\Delta, s : T \vdash \mathsf{release}[T](s) : \mathsf{Void} \dashv \Delta} \qquad \text{(hold)}\ \frac{T_1 \Rrightarrow T_2/T_3 \quad T_2\downarrow / T_3' \Rrightarrow T_1' \quad \Delta, s : T_3 \vdash e : T \dashv \Delta', s : T_3'}{\Delta, s : T_1 \vdash \mathsf{hold}[s : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) : T \dashv \Delta', s : T_1'}$$

$$\text{(let)}\ \frac{\Delta \vdash e_1 : T_1 \dashv \Delta_1 \quad \Delta_1, x : T_1 \vdash e_2 : T_2 \dashv \Delta_2 \quad x : \mathsf{Void} \in \Delta_2\ \mathit{or}\ x : T_1' \notin \Delta_2}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta_2 \div x} \qquad \text{(merge)}\ \frac{T_1 = T_1\downarrow \qquad T_1/T_2' \Rrightarrow T_3 \quad \Delta, l_2 : T_2 \vdash e : T \dashv \Delta', l_2 : T_2'}{\Delta, l_1 : T_1, l_2 : T_2 \vdash \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](e) : T \dashv \Delta', l_2 : T_3}$$

$$\text{(assert)}\ \frac{T_1 \Rrightarrow T_2}{\Delta, s : T_1 \vdash \mathsf{assert}\langle T_1 \gg T_2\rangle(s) : \mathsf{Void} \dashv \Delta, s : T_2} \qquad \text{(assert}_d\text{)}\ \frac{T_1 \neq \mathsf{Void} \quad T_2 \neq \mathsf{Void}}{\Delta, s : T_1 \vdash \mathsf{assert}_\mathsf{d}\langle T_1 \gg T_2\rangle(s) : \mathsf{Void} \dashv \Delta, s : T_2}$$

Figure 6: Internal Language Typing Rules

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

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F}, \overline{M}\ \} \quad mdecl(D, m) = T_r\ m(\overline{T_x \gg T'_x})[P_t\ E \gg T'_t]}{T_r\ m(\overline{T_x \gg T'_x})[P_t\ C \gg T'_t]\ \textbf{ok in}\ C}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ \overline{F}, \overline{M}\ \} \quad mdecl(D, m)\ \text{undefined}}{T_r\ m(\overline{T_x \gg T'_x})[P_t\ C \gg T'_t]\ \textbf{ok in}\ C}$$

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

$$\frac{T_r\ m(\overline{T_x \gg T'_x\ x})[T_t \gg T'_t]\ \textbf{ok in}\ C_t \quad \overline{x : T_x}, \mathsf{this} : T_t \vdash e : T_r \dashv \mathsf{this} : T'_t, \overline{x : T'_x}}{T_r\ m(\overline{T_x \gg T'_x\ x})\ [T_t \gg T'_t]\ \{\ \mathsf{return}\ e;\ \}\ \textbf{ok in}\ C_t}$$

$\boxed{F \textbf{ ok}}$ Well-typed Field

$$\frac{T{\downarrow} = T}{T\ f\ \textbf{ok}}$$

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

$$\frac{\overline{F\ \textbf{ok}} \quad \overline{M}\ \textbf{ok in}\ C_0}{\mathsf{class}\ C_0\ \mathsf{extends}\ C_1\ \{\ \overline{F};\ \overline{M}\ \}\ \textbf{ok}}$$

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

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

Figure 7: Internal Language Program Typing Rules

$C(\overline{o})$	R	$\in$	OBJECTS	
	R	$::=$	$\overline{P}$	(permission-sets)
	v	$::=$	$\mathsf{void} \mid o$	(values)
	μ	$\in$	OBJECTREFS $\rightharpoonup$ OBJECTS	(stores)
	ρ	$\in$	INDIRECTREFS $\rightharpoonup$ VALUES	(environments)
	$\mathbb{E}$	$::=$	$\square \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e$	(evaluation contexts)
		$\mid$	$\mathsf{merge}[l : T / l : T \Rrightarrow T](\mathbb{E})$	

Figure 8: Dynamic Semantics Support

$\boxed{\mu, \rho, e \to \mu, \rho, e}$ Internal Expression Dynamics

$$\text{(lookup-binder)}\ \frac{}{\mu, \rho, l \to \mu, \rho, \rho(l)} \qquad \text{(new)}\ \frac{o \notin dom(\mu) \quad \mu' = \mu[o \mapsto C(\overline{\rho(l)})\ [\mathsf{full}(\mathsf{Object})]]}{\mu, \rho, \mathsf{new}\ C(\overline{l}) \to \mu', \rho, o}$$

$$\text{(lookup-obj)}\ \frac{\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3}{\mu, \rho, l[T_1 \Rrightarrow T_2/T_3] \to \mu', \rho, \rho(l)} \qquad \text{(rel)}\ \frac{\mu' = \mu - \rho(l) : T}{\mu, \rho, \mathsf{release}[T](l) \to \mu', \rho, \mathsf{void}}$$

$$\text{(swap)}\ \frac{\mu(\rho(l_1)) = C(\overline{o})\ R \quad fields(C) = \overline{T\ f}}{\begin{array}{l}\mu, \rho, l_1.f_i\ :=:\ l_2 \to \\ \quad \mu[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\overline{o})\ R], \rho, o_i\end{array}} \qquad \text{(invoke)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ R \quad method(m, C) = \\ T_r\ m(\overline{T_x \gg T'_x\ x})\ [T_t \gg T'_t]\ \{\ \mathsf{return}\ e;\ \}\end{array}}{\mu, \rho, l_1.m(\overline{l_2}) \to \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e}$$

$$\text{(swap}_d\text{)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ R \quad fields(C) = \overline{T\ f} \\ C_g = \begin{cases} D & \text{if } \mathsf{shared}(D) \in R \\ C & \text{otherwise}\end{cases} \\ P = \mathsf{shared}(C_g) \qquad T_1 = P\ C\end{array}}{\begin{array}{l}\mu, \rho, l_1.f_i\ :=:_d\ l_2 \to \\ \quad \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1\rangle(l_1); \\ \quad \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_i\rangle(l_2); \\ \quad \mathsf{let}\ ret = l_1.f_i\ :=:\ l_2\ \mathsf{in} \\ \qquad \mathsf{assert}\langle T_1 \gg \mathsf{Dyn}\rangle(l_1); \\ \qquad \mathsf{assert}\langle T_i \gg \mathsf{Dyn}\rangle(ret); \\ \qquad ret\end{array}}$$

$$\text{(invoke}_d\text{)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ R \\ mdecl(m, C) = T_r\ m(\overline{T_x \gg T'_x})\ [T_t \gg T'_t] \\ |\ \overline{T_x}\ | = |\ \overline{l_2}\ |\end{array}}{\begin{array}{l}\mu, \rho, l_1.{}_d m(\overline{l_2}) \to \mu, \rho, \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t\rangle(l_1);} \\ \qquad \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_x\rangle(l_2);} \\ \qquad \mathsf{let}\ ret = l_1.m(\overline{l_2})\ \mathsf{in} \\ \qquad\quad \overline{\mathsf{assert}\langle T'_t \gg \mathsf{Dyn}\rangle(l_1);} \\ \qquad\quad \overline{\mathsf{assert}\langle T'_x \gg \mathsf{Dyn}\rangle(l_2);} \\ \qquad\quad \mathsf{assert}\langle T_r \gg \mathsf{Dyn}\rangle(ret); \\ \qquad\quad ret\end{array}}$$

$$\text{(update)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ R \qquad fields(C) = \overline{T\ f} \\ \mu_1 = \mu[\rho(l_1) \mapsto C'(\overline{\rho(l_2)})\ R] \quad \mu' = \mu_1 - \overline{o : T}\end{array}}{\mu, \rho, l_1 \leftarrow C'(\overline{l_2}) \to \mu', \rho, \mathsf{void}} \qquad \text{(hold)}\ \frac{\begin{array}{c}\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3 \\ l' \notin dom(\rho) \qquad \rho' = \rho[l' \mapsto \rho(l)]\end{array}}{\begin{array}{l}\mu, \rho, \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T'_3 \Rrightarrow T'_1](e) \to \\ \mu', \rho', \mathsf{merge}[l' : T_2\!\downarrow / l : T'_3 \Rrightarrow T'_1](e)\end{array}}$$

$$\text{(update}_d\text{)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o_f})\ R \\ C_g = \begin{cases} D & \text{if } \mathsf{shared}(D) \in R \\ C \ddot{\vee} C' & \text{otherwise}\end{cases} \\ C' <: C_g \qquad P = \mathsf{shared}(C_g) \\ T = P\ C \qquad T' = P\ C'\end{array}}{\begin{array}{l}\mu, \rho, l_1 \leftarrow_d C'(\overline{l_2}) \to \\ \quad \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg T\rangle(l_1); \\ \quad l_1 \leftarrow C'(\overline{l_2}); \\ \quad \mathsf{assert}\langle T' \gg \mathsf{Dyn}\rangle(l_1)\end{array}} \qquad \text{(merge)}\ \frac{\mu' = \mu - \rho(l') : T_1 - \rho(l) : T_2 + \rho(l) : T_3}{\mu, \rho, \mathsf{merge}[l' : T_1/l : T_2 \Rrightarrow T_3](v) \to \mu', \rho, v}$$

$$\text{(ref)}\ \frac{\begin{array}{c}\mu(\rho(l)) = C(\overline{o})\ R \quad fields(C) = \overline{T\ f} \\ T_i \Downarrow T' \qquad \mu' = \mu + o_i : T'\end{array}}{\mu, \rho, l.f_i \to \mu', \rho, o_i} \qquad \text{(assert)}\ \frac{\mu' = \mu - \rho(l) : T + \rho(l) : T'}{\mu, \rho, \mathsf{assert}\langle T \gg T'\rangle(l) \to \mu', \rho, \mathsf{void}}$$

$$\text{(ref}_d\text{)}\ \frac{\mu(\rho(l)) = C(\overline{o})\ R \quad fields(C) = \overline{T\ f}}{\mu, \rho, l.{}_d f_i \to \mu, \rho, o_i} \qquad \text{(assert}_d\text{)}\ \frac{\begin{array}{c}\mu' = \mu - \rho(l) : T + \rho(l) : T' \\ \mu'(\rho(l)) = C(\overline{o})\ R \quad \langle R, \leftrightarrow\rangle \text{ is connected} \\ \text{if } T' = P'\ C' \text{ then } C <: C'\end{array}}{\mu, \rho, \mathsf{assert_d}\langle T \gg T'\rangle(l) \to \mu', \rho, \mathsf{void}}$$

$$\text{(let)}\ \frac{l \notin dom(\rho)}{\mu, \rho, \mathsf{let}\ x = v\ \mathsf{in}\ e \to \mu, \rho[l \mapsto v], [l/x]e} \qquad \text{(congr)}\ \frac{\mu, \rho, e \to \mu', \rho', e'}{\mu, \rho, \mathbb{E}[e] \to \mu', \rho', \mathbb{E}[e']}$$

Figure 9: Internal Dynamic Semantics

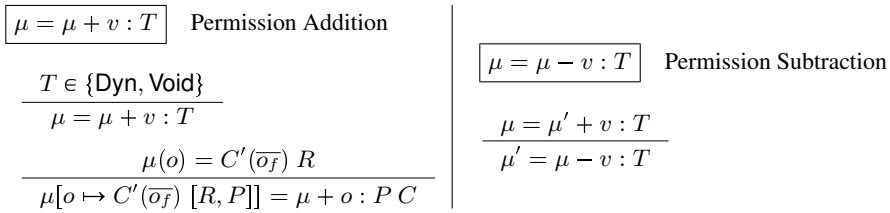

Figure 10: Internal Dynamics Auxiliary Functions

are only interesting for their side-effects, and object references o. Stores μ associate object references to objects. They are represented as partial functions from object references o to objects $C(\overline{o})\ R$, where $C(\overline{o})$ is the runtime object representation and R is the set of outstanding permissions for references to that object.

In addition to the store, the dynamic semantics uses a second heap, which we call the *environment* ρ, 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, and is not needed in a practical implementation. 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 tracks these differences at runtime. It maps indirect references l to values v. Two indirect references can point to the same object, but the permissions associated with the two indirect references are kept separate.

Figure 10 also defines two helper functions for tracking permissions in the heap. Permission addition augments the permission set for a particular object in the heap. Conversely, permission subtraction removes a permission from the set of tracked permissions for an object. To simplify their use in the dynamic semantics, both operations take an arbitrary value and type and behave like identity when presented with a void value or Dyn typed object reference.

The dynamic semantics of GFTIL is defined as transitions between store/environment/expression triples (as discussed earlier, a practical implementation does not need indirect references, so it could be defined over store/expression pairs). The (let) rule shows that each variable binding is tracked using a distinct indirect reference. Ultimately indirect references are dereferenced to their corresponding values in rules for various expressions, such as the (lookup-*) rules. The (lookup-bare) rule simply dereferences an indirect reference, while the (lookup-obj) rule additionally tracks permissions using its explicit type splitting information. The (assert) rule uses permission addition and subtraction to track permissions, and returns void. The (assert$_d$) does the same, but also confirms dynamically that the assertion is safe. If the new type is incompatible with outstanding permissions (i.e. not all permissions are pairwise compatible $\leftrightarrow$), then the expression is stuck. The (update) rule looks up the object references for the target reference and the arguments to the class constructor, replaces the store object for the target reference with the newly constructed object, and releases the permissions held by the fields of the old object. The (update$_d$) rule, on the other hand, dynamically acquires a shared permission to the object being updated (using assert$_d$), defers to the static update operation, and releases the acquired permission (using assert). This rule mixes run-time code generation with dynamic property lookup so as to succinctly express dynamic update semantics. This rule could be rephrased to subsume the behavior of the assertions and the static update, but it would become more complex. The (swap) rule updates the field of an object, and returns its old value. (swap$_d$) is the dynamic variant of (swap). It first computes the types necessary, T_1 and T_i, of the target and new field value. It then transitions to a sequence of expressions that will assert those permissions, perform the statically-typed sway, before releasing the

permissions. If a shared(D) permission exists to the target referent, than that permission is used, as it is the only permission with write access allowed. If no shared permission exists, then a shared(C) is requested, where C is the current class of the target object. Note that this permission is the least restrictive necessary to perform the swap (but can still get stuck in an assert$_d$. (ref) returns the value of a field, and updates memory by adding the maximal permission. (ref$_d$) does the same but does not update memory, as the expression is typed as Dyn, and references of type Dyn are not tracked. (new) adds a new object in the heap to the store, and transitions to its reference. (rel) simply subtracts any permission in the annotated type from the store, and transitions to void. (invoke) looks up the method body e and transitions to it, after substituting the arguments for the parameter names. It is assumed that the parameters can be renamed as necessary to avoid variable name conflicts. (invoke$_d$) is the dynamic method invocation. As with (update$_d$) and (swap$_d$), this transitions to a sequence of expressions where relevant types are asserted, the static variant of invoke is called, and then acquired permissions are released (via a safe assert). Note that this can be stuck if the method m does not exist, or the arity of m does not match the number of supplied arguments. (hold) updates the store by accounting for the splitting of types as annotated in the expression. A new indirect reference, l' is added to the environment as an alias for l to hold the permission $T_2{\downarrow}$ during execution of e. (merge) completes the hold expression, by merging the held type of l' with the type of its alias l, and updating the store accordingly. Note that after this point, the indirect reference l' is no longer in scope. (congr) is the standard congruence rule for both the hold and let expressions.

3.4 Type Safety

GFTIL's type safety proof must account for the outstanding permissions for each object o and verify that they are mutually compatible. Figure 11 presents the definitions used for this. The $fieldTypes$, $ctxTypes$, and $envTypes$ functions accumulate outstanding type information for objects in the store from the fields of objects, the type context, and the environment respectively. The $objTypes$ function selects just the permission-carrying types for a particular object reference o. These definitions use square brackets to express list comprehensions, and $+\!+$ to express list concatenation.

The $objTypes$ function is used to define *reference consistency*, the judgment that an object in the store and all references to it are sensible. A consistent object reference points to an object that has the proper number of fields, and all references to it must be well-formed, mutually compatible, and tracked in the store.

Reference consistency is used in turn to define *global consistency*, which establishes the mutual compatibility of a store-environment-context triple. Global consistency implies that every object reference in the store satisfies reference consistency, that every reference in the type context is accounted for in the store and environment, and that Void and object-typed indirect references ultimately point to void values and object references respectively. Note that global consistency and permission tracking take into account even objects that are no longer reachable in the program. To recover permissions, a program must explicitly release the fields of an object before it becomes unreachable.

These concepts contribute to stating and proving type-safety.

Theorem 1 (Progress)**.** *If e is a closed expression, $\Delta \vdash e : T \dashv \Delta'$ and μ, Δ, ρ **ok** then one of the following holds*

- *e is a value*
- *$\mu, \rho, e \rightarrow \mu', \rho', e'$ for some μ', ρ', e'*
- *$e = \mathbb{E}[e_d]$*

Helper Functions

$$
\begin{array}{rcl}
objTypes(\mu, \Delta, \rho, o) & = & [T \mid o : T \in types(\mu, \Delta, \rho), T \neq \mathsf{Dyn}] \\
types(\mu, \Delta, \rho) & = & fieldTypes(\mu) +\!\!+ envTypes(\Delta, \rho) +\!\!+ ctxTypes(\Delta) \\
fieldTypes(\mu) & = & \mathop{+\!\!+}_{o' \in dom(\mu)} [o_i : T_i \mid \mu(o') = C(\overline{o})\ R, fields(C) = \overline{T\ f}] \\
envTypes(\Delta, \rho) & = & [o : T \mid \rho(l) = o, l : T \in \Delta] \\
ctxTypes(\Delta) & = & [o : T \mid o : T \in \Delta]
\end{array}
$$

$\boxed{\mu, \Delta, \rho \vdash o\ \mathbf{ok}}$ Reference Consistency

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

$\boxed{\mu, \Delta, \rho\ \mathbf{ok}}$ Type Consistency

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

Figure 11: Permission-Consistency Relations

Proof. See appendix. □

The last case of the progress theorem holds when a program is stuck on a failed dynamically checked expression. All statically checked expressions make progress.

Theorem 2 (Preservation)**.** *If $\Delta \vdash e : T \dashv \Delta'$ and μ, Δ, ρ **ok**, and $\mu, \rho, e \to \mu', \rho', e'$, then there exists Δ'', such that $\Delta'' \vdash e' : T \dashv \Delta'$, and μ', Δ'', ρ' **ok***

Proof. See appendix. □

The preservation theorem states that for any well-typed expression e of type T, that transitions to e', there exists an outgoing context of the transition, Δ'', such that Δ'' can be used to type e' at type T, respecting the output context Δ' of the typing derivation of e).

4 Source to Internal Language Translation

The dynamic semantics of GFT are defined by augmenting its type system to generate GFTIL expressions. The type checking and synthesis judgments become $\Delta \vdash e_1 \Leftarrow T \rightsquigarrow e_2^T \dashv \Delta'$ and $\Delta \vdash e_1 \Rightarrow T \rightsquigarrow e_2^T \dashv \Delta'$ respectively, where e_1 is a GFT expression and e_2^T is its corresponding GFTIL expression. Figure 13 presents them in full. We use the ^{T} superscript to disambiguate GFTIL expressions as needed.

Several rules use the *coerce* partial function, defined in Figure 12, which is only defined for valid coercions and abstracts the generation of static and dynamic assertions:

Expressions Most of these translations are very straightforward, and follow similar patterns. There is a one-to-one correspondence between typing rules of GFT and translation rules. The premises of comparable rules are often identical. Expressions are translated from GFT to GFTIL often simply syntactically (i.e. ctx⇒) by adding the explicit type annotations. Also, implicit splitting is made explicit with the help of additional let expressions coupled with asserts.

$\boxed{coerce(x, T \gg T) = e^T}$ Type Coercion

$$\frac{T \Rrightarrow T'}{coerce(x, T \gg T') = \mathsf{assert}\langle T \gg T' \rangle(x)} \qquad \frac{T = \mathsf{Dyn} \quad T' \neq \mathsf{Void}}{coerce(x, T \gg T') = \mathsf{assert}_{\mathsf{d}}\langle T \gg T' \rangle(x)}$$

Figure 12: Translation Auxiliary Functions

We prove a translation preservation theorem,

Theorem 3 (Translation Preservation)**.** *If* $\Delta \vdash e \Leftrightarrow T \dashv \Delta'$*, then* $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta'$ *and* $\Delta \vdash e^T : T \dashv \Delta'$*, for some* e^T*.*

Proof. See appendix. □

which states that any well-typed GFT expression can be translated to a well-typed GFTIL expression of a corresponding type (and corresponding input and output contexts).

Programs Figure 14 defines program translation. This is a straightforward definition based on expression translation. Note the definition of method translation which coerces the parameters to those defined in the method signature before returning the result from the translated body of the expression.

We can now prove a program translation theorem,

Theorem 4 (Program Translation Preservation)**.** *If* PG ***ok****, then there exists* PG^T *such that* $PG \rightsquigarrow PG^T$*, and* PG^T ***ok****.*

Proof. See appendix. □

which states that a well-typed program in GFT can be translated to a well-typed GFTIL program. By the Expression Translation Preservation theorem, the type of the main expression of the program is preserved as well.

$\boxed{\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta}$ Source to Internal Language Translation

$$\text{(ctx}\Rightarrow\text{)}\ \frac{T_1 \Downarrow T_2}{\Delta, x : T_1 \vdash x \Rightarrow T_1 \rightsquigarrow x[T_1 \Rightarrow T_1/T_2] \dashv \Delta, x : T_2}$$

$$\text{(new}\Rightarrow\text{)}\ \frac{fields(C) = \overline{T'\ f} \qquad \overline{x : T \vdash x \Leftarrow T' \rightsquigarrow e^T \dashv x : T''}}{\Delta, \overline{x : T} \vdash \mathsf{new}\ C(\overline{x}) \Rightarrow \mathsf{full}(\mathsf{Object})\ C \rightsquigarrow \overline{\mathsf{let}\ x' = e^T\ \mathsf{in}}\ \mathsf{new}\ C(\overline{x'}) \dashv \Delta, x : T''}$$

$$\text{(ctx}\Leftarrow\text{)}\ \frac{T_1 \Rightarrow T_2/T_3}{\Delta, x : T_1 \vdash x \Leftarrow T_2 \rightsquigarrow x[T_1 \Rightarrow T_2/T_3] \dashv \Delta, x : T_3}$$

$$\text{(invoke}\Rightarrow\text{)}\ \frac{\begin{array}{c} mdecl(m, C_1) = T\ m(\overline{T_x \gg T'_x})[T_t \gg T'_t] \\ coerce(x_1, P_1\ C_1 \gg T_t) = e_1^T \\ \overline{coerce(x_2, T_2 \gg T_x) = e_2^T} \end{array}}{\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow T \rightsquigarrow e_1^T;\ \overline{e_2^T};\ x_1.m(\overline{x_2}) \dashv \Delta\downarrow, x_1 : T'_t, \overline{x_2 : T'_x}}$$

$$\text{(ctx}_d \Leftarrow\text{)}\ \frac{}{\begin{array}{l} \Delta, x : \mathsf{Dyn} \vdash x \Leftarrow P\ C \rightsquigarrow \\ \quad \mathsf{let}\ ret = x[\mathsf{Dyn} \Rightarrow \mathsf{Dyn}/\mathsf{Dyn}]\ \mathsf{in} \\ \quad\quad \mathsf{assert_d}\langle \mathsf{Dyn} \gg P\ C \rangle(ret); \\ \quad\quad ret \dashv \Delta, x : \mathsf{Dyn} \end{array}}$$

$$\text{(invoke}_d \Rightarrow\text{)}\ \frac{\overline{coerce(x_2, T_2 \gg \mathsf{Dyn}) = e_2^T}}{\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow \mathsf{Dyn} \rightsquigarrow \overline{e_2^T};\ x_1.{}_d m(\overline{x_2}) \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}}$$

$$\text{(let}\Leftrightarrow\text{)}\ \frac{\Delta \vdash e_1 \Rightarrow T_1 \rightsquigarrow e_1^T \dashv \Delta_1 \qquad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \rightsquigarrow e_2^T \dashv \Delta', x : T'_1}{\begin{array}{l} \Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \rightsquigarrow \mathsf{let}\ x = e_1^T\ \mathsf{in} \\ \quad \mathsf{let}\ ret = e_2^T\ \mathsf{in} \\ \quad\quad \mathsf{release}[T'_1](x); ret \dashv \Delta' \end{array}}$$

$$\text{(swap}\Rightarrow\text{)}\ \frac{\begin{array}{c} k_1 \in \{\mathsf{full}, \mathsf{shared}\} \qquad T'_2\ f \in fields(C_1) \\ x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \rightsquigarrow e_2^T \dashv x_2 : T''_2 \end{array}}{\begin{array}{l} \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash \\ \quad x_1.f\ :=:\ x_2 \Rightarrow T''_2 \rightsquigarrow \\ \quad\quad \mathsf{let}\ x'_2 = e_2^T\ \mathsf{in} \\ \quad\quad\quad x_1.f\ :=:\ x'_2 \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T''_2 \end{array}}$$

$$\text{(letT}\Leftrightarrow\text{)}\ \frac{\Delta \vdash e_1 \Leftarrow T_1 \rightsquigarrow e_1^T \dashv \Delta_1 \qquad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \rightsquigarrow e_2^T \dashv \Delta', x : T'_1}{\begin{array}{l} \Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \rightsquigarrow \\ \quad \mathsf{let}\ x = e_1^T\ \mathsf{in} \\ \quad\quad \mathsf{let}\ ret = e_2^T\ \mathsf{in} \\ \quad\quad\quad \mathsf{release}[T'_1](x); ret \dashv \Delta' \end{array}}$$

$$\text{(swap}_d \Rightarrow\text{)}\ \frac{x_2 : T_2 \vdash x_2 \Leftarrow \mathsf{Dyn} \rightsquigarrow e_2^T \dashv x_2 : T''_2}{\begin{array}{l} \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash x_1.f\ :=:\ x_2 \Rightarrow \mathsf{Dyn} \rightsquigarrow \\ \quad \mathsf{let}\ x'_2 = e_2^T\ \mathsf{in} \\ \quad\quad x_1.f\ :=:_d\ x'_2 \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T''_2 \end{array}}$$

$$\text{(}\hat{e} \Leftarrow\text{)}\ \frac{\Delta \vdash \hat{e} \Rightarrow T_1 \rightsquigarrow e_1^T \dashv \Delta' \qquad coerce(ret, T_1 \gg T_2) = e_2^T}{\Delta \vdash \hat{e} \Leftarrow T_2 \rightsquigarrow \mathsf{let}\ ret = e_1^T\ \mathsf{in}\ e_2^T; ret \dashv \Delta'}$$

$$\text{(update}\Rightarrow\text{)}\ \frac{\begin{array}{c} k \in \{\mathsf{full}, \mathsf{shared}\} \qquad C <: E \qquad fields(C) = \overline{T'_2\ f} \\ \overline{x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \rightsquigarrow e_2^T \dashv x_2 : T''_2} \end{array}}{\begin{array}{l} \Delta, x_1 : k(E)\ D, \overline{x_2 : T_2} \vdash x_1 \leftarrow C(\overline{x_2}) \Rightarrow \mathsf{Void} \rightsquigarrow \\ \quad \overline{\mathsf{let}\ x'_2 = e_2^T\ \mathsf{in}} \\ \quad\quad x_1 \leftarrow C(\overline{x'_2}) \dashv \Delta\downarrow, x_1 : k(E)\ C, \overline{x_2 : T''_2\downarrow} \end{array}}$$

$$\text{(hold}\Rightarrow\text{)}\ \frac{T_1 \Rightarrow T_2/T_3 \qquad T_2\downarrow / T'_3 \Rightarrow T'_1 \qquad \Delta, x : T_3 \vdash e \Rightarrow T \rightsquigarrow e^T \dashv \Delta', x : T'_3}{\begin{array}{l} \Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) \Rightarrow T \rightsquigarrow \\ \quad \mathsf{hold}[x : T_1 \Rightarrow T_2/T_3 \gg T'_3 \Rightarrow T'_1](e^T) \\ \quad \dashv \Delta', x : T'_1 \end{array}}$$

$$\text{(update}_d \Rightarrow\text{)}\ \frac{fields(C) = \overline{T'_2\ f} \qquad \overline{x_2 : T_2 \vdash x_2 \Leftarrow T'_2 \rightsquigarrow e_2^T \dashv x_2 : T''_2}}{\begin{array}{l} \Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash x_1 \leftarrow C(\overline{x_2}) \Rightarrow \mathsf{Void} \rightsquigarrow \\ \quad \overline{\mathsf{let}\ x'_2 = e_2^T\ \mathsf{in}} \\ \quad\quad x_1 \leftarrow_d C(\overline{x'_2}) \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : T''_2\downarrow} \end{array}}$$

$$\text{(ref}\Rightarrow\text{)}\ \frac{T_2\ f \in fields(C_1) \qquad T_2 \Downarrow T'_2}{\Delta, x : P_1\ C_1 \vdash x.f \Rightarrow T'_2 \rightsquigarrow x.f \dashv \Delta, x : P_1\ C_1}$$

$$\text{(assert}\Rightarrow\text{)}\ \frac{T \Rightarrow T'}{\Delta, x : T \vdash \mathsf{assert}\langle T' \rangle(x) \Rightarrow \mathsf{Void} \rightsquigarrow \mathsf{assert}\langle T \gg T' \rangle(x) \dashv \Delta, x : T'}$$

$$\text{(ref}_d \Rightarrow\text{)}\ \frac{}{\Delta, x : \mathsf{Dyn} \vdash x.f \Rightarrow \mathsf{Dyn} \rightsquigarrow x.{}_d f \dashv \Delta, x : \mathsf{Dyn}}$$

$$\text{(assert}_d \Rightarrow\text{)}\ \frac{T \neq \mathsf{Void} \qquad T' \neq \mathsf{Void}}{\Delta, x : T \vdash \mathsf{assert}\langle T' \rangle(x) \Rightarrow \mathsf{Void} \rightsquigarrow \mathsf{assert_d}\langle T \gg T' \rangle(x) \dashv \Delta, x : T'}$$

Figure 13: Translation Rules

$\boxed{M \rightsquigarrow M^T}$ Method Translation

$$\frac{\mathsf{this} : T_t, \overline{x : T} \vdash e \Leftarrow T_r \rightsquigarrow e^T \dashv \mathsf{this} : T_t'', \overline{x : T_x''} \quad e_1^T = \mathsf{let}\ ret = e^T\ \mathsf{in}\ coerce(\mathsf{this}, T_t'' \gg T_t'); \overline{coerce(x, T'' \gg T'')}; ret}{T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \} \rightsquigarrow T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e_1^T;\ \}}$$

$\boxed{F \rightsquigarrow F^T}$ Field Translation

$$\frac{}{F \rightsquigarrow F}$$

$\boxed{CL \rightsquigarrow CL}$ Class Translation

$$\frac{\overline{F \rightsquigarrow F^T} \qquad \overline{M \rightsquigarrow M^T}}{\mathsf{class}\ C_0\ \mathsf{extends}\ C_1\ \{\ \overline{F};\ \overline{M}\ \} \rightsquigarrow \mathsf{class}\ C_0\ \mathsf{extends}\ C_1\ \{\ \overline{F^T};\ \overline{M^T}\ \}}$$

$\boxed{PG \rightsquigarrow PG^T}$ Program Translation

$$\frac{\cdot \vdash e \Rightarrow T \rightsquigarrow e^T \dashv \cdot \qquad \overline{CL \rightsquigarrow CL^T}}{\langle \overline{CL}, e \rangle \rightsquigarrow \langle \overline{CL^T}, e^T \rangle}$$

Figure 14: Program Translation Rules

Appendix: Proofs of Type Safety

Lemma 5 (Coercion). *If* $coerce(x, T_1 \gg T_2) = e$, *then* $\Delta, x : T_1 \vdash e : \mathsf{Void} \dashv \Delta, x : T_2$

Proof. By case analysis of derivation of $coerce(x, T_1 \gg T_2)$
Case (Coerce).

1. By assumption
   (a) $T_1 \Rrightarrow T_2$
   (b) $coerce(x, T_1 \gg T_2) = \mathsf{assert}\langle T_1 \gg T_2\rangle(x)$
2. $\Delta, x : T_1 \vdash \mathsf{assert}\langle T_1 \gg T_2\rangle(x) : \mathsf{Void} \dashv \Delta, x : T_2$ – by 1a, (assert)

Case (Coerce$_d$).

1. By assumption
   (a) $T_1 = \mathsf{Dyn}$
   (b) $T_2 \neq \mathsf{Void}$
   (c) $coerce(x, T_1 \gg T_2) = \mathsf{assert_d}\langle T_1 \gg T_2\rangle(x)$
2. $\Delta, x : T_1 \vdash \mathsf{assert}\langle T_1 \gg T_2\rangle(x) : \mathsf{Void} \dashv \Delta, x : T_2$ – by 1a-b,(assert$_d$)

□

Lemma 6 (Translation Weakening). *If* $\Delta \vdash b \Leftarrow T \rightsquigarrow e' \dashv \Delta'$, *then* $\Delta, y : T_y \vdash b \Leftarrow T \rightsquigarrow e' \dashv \Delta', y : T_y$

Proof. Case analysis of $(ctx \Leftarrow)$, and $(ctx_d \Leftarrow)$ □

Theorem 7 (Translation Preservation). *If* $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta'$ *then* $\Delta \vdash e^T : T \dashv \Delta'$.

Proof. Note that the premise is implicitly indexed by the class table of the source program, and that the conclusion is indexed by the class table of the internal program. However, as we have defined program translation, only difference between the two are the method bodies. In particular, the subtyping relation <:, and the auxiliary functions $mddecl, fields$ are identical for the source and target programs. We proceed by induction on derivations of $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta'$.
Case (ctx⇒).

1. By assumption
   (a) $\Delta, x : T_1 \vdash x \Rightarrow T_1 \rightsquigarrow x[T_1 \Rrightarrow T_1/T_2] \dashv \Delta, x : T_2$
   (b) $T_1 \Downarrow T_2$
2. $T_1 \Rrightarrow T_1/T_2$ – by 1a,definition of $\Downarrow$
3. $\Delta, x : T_1 \vdash x[T_1 \Rrightarrow T_1/T_2] : T_1 \dashv \Delta, x : T_2$ – by 2,(ctx)

Case (ctx⇐).

1. By assumption
   (a) $\Delta, x : T_1 \vdash x \Rightarrow T_2 \rightsquigarrow x[T_1 \Rrightarrow T_2/T_3] \dashv \Delta, x : T_3$
   (b) $T_1 \Rrightarrow T_2/T_3$
2. $\Delta, x : T_1 \vdash x[T_1 \Rrightarrow T_2/T_3] : T_2 \dashv \Delta, x : T_3$ – by 1b,(ctx)

Case (ctx$_d$ ⇐).

1. By assumption
   (a) $\Delta, x : \mathsf{Dyn} \vdash x \Leftarrow P\ C \rightsquigarrow \mathsf{let}\ ret = x[\mathsf{Dyn} \Rrightarrow \mathsf{Dyn}/\mathsf{Dyn}]\ \mathsf{in}\ \mathsf{assert}_\mathsf{d}\langle \mathsf{Dyn} \gg P\ C \rangle(ret); ret \dashv \Delta, x : \mathsf{Dyn}$
2. $ret \notin dom(\Delta)$ – by α-renaming
3. $\Delta, x : \mathsf{Dyn} \vdash x[\mathsf{Dyn} \Rrightarrow \mathsf{Dyn}/\mathsf{Dyn}] : \mathsf{Dyn} \dashv \Delta, x : \mathsf{Dyn}$ – by (Split-Dyn) and (ctx)
4. $\Delta, x : \mathsf{Dyn}, ret : \mathsf{Dyn} \vdash \mathsf{assert}_\mathsf{d}\langle \mathsf{Dyn} \gg P\ C \rangle(ret) : \mathsf{Void} \dashv \Delta, x : \mathsf{Dyn}, ret : P\ C$ – by (assert$_d$)
5. $\Delta, x : \mathsf{Dyn}, ret : P\ C \vdash ret : P\ C \dashv \Delta, x : \mathsf{Dyn}$ – by (ctx-binder)
6. $\Delta, x : \mathsf{Dyn} \vdash \mathsf{let}\ ret = x[\mathsf{Dyn} \Rrightarrow \mathsf{Dyn}/\mathsf{Dyn}]\ \mathsf{in}\ \mathsf{assert}_\mathsf{d}\langle \mathsf{Dyn} \gg P\ C \rangle(ret); ret : P\ C \dashv \Delta, x : \mathsf{Dyn}$ – by 2-5, (let)

Case (letT⇔).

1. By assumption
   (a) $\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Rrightarrow T_2 \rightsquigarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ \mathsf{let}\ ret = e_2'\ \mathsf{in}\ \mathsf{release}[T_1'](x); ret \dashv \Delta'$
   (b) $\Delta \vdash e_1 \Leftarrow T_1 \rightsquigarrow e_1' \dashv \Delta_1$
   (c) $\Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \rightsquigarrow e_2' \dashv \Delta', x : T_1'$
2. $x, ret \notin dom(\Delta)$ – by α-renaming
3. $\Delta \vdash e_1' : T_1 \dashv \Delta_1$ – by induction on 1b
4. $\Delta_1, x : T_1 \vdash e_2' : T_2 \dashv \Delta', x : T_1'$ – by induction on 1c
5. $\Delta', x : T_1', ret : T_2 \vdash \mathsf{release}[T_1'](x) : \mathsf{Void} \dashv \Delta', ret : T_2$ – by (rel)
6. $\Delta', ret : T_2 \vdash ret : T_2 \dashv \Delta'$ – by (ctx-binder)
7. $\Delta \vdash \mathsf{let}\ x = e_1'\ \mathsf{in}\ \mathsf{let}\ ret = e_2'\ \mathsf{in}\ \mathsf{release}[T_1'](x); ret : T_2 \dashv \Delta'$ – by 4-9, (let)

Case (let⇔).

1. Same as case (letT⇔), with assumption $\Delta \vdash e_1 \Rrightarrow T_1 \rightsquigarrow e_1' \dashv \Delta_1$

Case (sub⇐).

1. By assumption
   (a) $\Delta \vdash \widehat{e} \Leftarrow T_2 \rightsquigarrow \mathsf{let}\ ret = e_1\ \mathsf{in}\ e_2; ret \dashv \Delta'$
   (b) $\Delta \vdash \widehat{e} \Rrightarrow T_1 \rightsquigarrow e_1 \dashv \Delta'$
   (c) $coerce(ret, T_1 \gg T_2) = e_2$
2. $ret \notin dom(\Delta)$ – by α-renaming
3. $\Delta \vdash e_1 : T_1 \dashv \Delta'$ – by induction on 1b
4. $\Delta', ret : T_1 \vdash e_2 : T_2 \dashv \Delta', ret : T_2$ – by 3, Coercion Lemma
5. $\Delta', ret : T_2 \vdash ret : T_2 \dashv \Delta'$ – by (ctx-binder)
6. $\Delta \vdash \mathsf{let}\ ret = e_1\ \mathsf{in}\ e_2; ret : T_2 \dashv \Delta'$ – by 3-5, (let)

Case (new⇒).

1. By assumption
   (a) $\Delta, \overline{x : T} \vdash \mathsf{new}\ C(\overline{x}) \Rrightarrow \mathsf{full}(\mathsf{Object})\ C \rightsquigarrow \overline{\mathsf{let}\ x' = e\ \mathsf{in}}\ \mathsf{new}\ C(\overline{x'}) \dashv \Delta, x : T''$
   (b) $fields(C) = \overline{T'\ f}$

(c) $\overline{x : T \vdash x \Leftarrow T' \rightsquigarrow e \dashv x : T''}$

2. Let $n = |\mathit{fields}(C)|$
3. $\overline{x}, \overline{x'} \notin dom(\Delta)$ – by α-renaming
4. $\Delta, \Delta_x, x_i : T_i \vdash x_i \Leftarrow T_i' \rightsquigarrow e_i \dashv \Delta, \Delta_x, x_i : T_i''$ – by 3 and Translation Weakening, for $i \in [1..n]$, where $\Delta_x = x_1 : T_1'', ..., x_{i-1} : T_{i-1}'', x_{i+1} : T_{i+1}, ..., x_n : T_n, x_1' : T_1', ..., x_{i-1}' : T_{i-1}'$
5. $\Delta, \overline{x : T''}, \overline{x' : T'} \vdash \mathsf{new}\ C(\overline{x'}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta, \overline{x : T''}$ – by 1b,(new)
6. $\Delta, \overline{x : T} \vdash \overline{\mathsf{let}\ x' = e\ \mathsf{in}}\ \mathsf{new}\ C(\overline{x'}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta, x : T''$ – by 3-5, (let)

Case (assert⇒).

1. By assumption
   (a) $\Delta, x : T \vdash \mathsf{assert}\langle T' \rangle(x) \Rightarrow \mathsf{Void} \rightsquigarrow \mathsf{assert}\langle T \gg T' \rangle(x) \dashv \Delta, x : T'$
   (b) $T \Rrightarrow T'$
2. $\Delta, x : T \vdash \mathsf{assert}\langle T \gg T' \rangle(x) : \mathsf{Void} \dashv \Delta, x : T'$, by 1b,(assert)

Case (assert$_d$ ⇒).

1. By assumption
   (a) $\Delta, x : T \vdash \mathsf{assert}\langle T' \rangle(x) \Rightarrow \mathsf{Void} \rightsquigarrow \mathsf{assert}_\mathsf{d}\langle T \gg T' \rangle(x) \dashv \Delta, x : T'$
   (b) $T \neq \mathsf{Void}$
   (c) $T' \neq \mathsf{Void}$
2. $\Delta, x : T \vdash \mathsf{assert}_\mathsf{d}\langle T \gg T' \rangle(x) : \mathsf{Void} \dashv \Delta, x : T'$ – by 1b-c,(assert$_d$)

Case (ref⇒).

1. By assumption
   (a) $\Delta, x : P_1\ C_1 \vdash x.f \Rightarrow T_2' \rightsquigarrow x.f \dashv \Delta, x : P_1\ C_1$
   (b) $T_2\ f \in \mathit{fields}(C_1)$
   (c) $T_2 \Downarrow T_2'$
2. $\Delta, x : P_1\ C_1 \vdash x.f : T_2' \dashv \Delta, x : P_1\ C_1$ – by 1b-c,(ref)

Case (ref$_d$ ⇒).

1. By assumption
   (a) $\Delta, x : \mathsf{Dyn} \vdash x.f \Rightarrow \mathsf{Dyn} \rightsquigarrow x.{}_d f \dashv \Delta, x : \mathsf{Dyn}$
2. $\Delta, x : \mathsf{Dyn} \vdash x.{}_d f : \mathsf{Dyn} \dashv \Delta, x : \mathsf{Dyn}$ – by (ref$_d$)

Case (update⇒).

1. By assumption
   (a) $\Delta, x_1 : k(E)\ D, \overline{x_2 : T_2} \vdash x_1 \leftarrow C(\overline{x_2}) \Rightarrow \mathsf{Void}$ – by assumption
   $\overset{\rightsquigarrow}{\overline{\mathsf{let}\ x_2' = e_2\ \mathsf{in}}\ x_1 \leftarrow C(\overline{x_2'}) \dashv \Delta\downarrow, x_1 : k(E)\ C, \overline{x_2 : T_2''\downarrow}}$
   (b) $k \in \{\mathsf{full}, \mathsf{shared}\}$
   (c) $C <: E$
   (d) $fields(C) = \overline{T_2'\ f}$

(e) $\overline{x_2 : T_2 \vdash x_2 \Leftarrow T_2' \rightsquigarrow e_2 \dashv x_2 : T_2''}$

2. $\overline{x_2'} \notin dom(\Delta)$ – by α-renaming
3. Let $n = |\mathit{fields}(C)|$
4. $\Delta, x_1 : k(E)\ D, \Delta_x, x_{2_i} : T_{2_i} \vdash x_{2_i} \Leftarrow T'_{2_i} \rightsquigarrow e_{2_i} \dashv \Delta, x_1 : k(E)\ D, \Delta_x, x_{2_i} : T''_{2_i}$ – by 1e,Translation Weakening, for $i \in [1..n]$, where $\Delta_x = x_{2_1} : T''_{2_1}, ..., x_{2_{i-1}} : T''_{2_{i-1}}, x_{2_{i+1}} : T_{2_{i+1}}, ..., x_{2_n} : T_{2_n}, x'_{2_1} : T'_{2_1}, ..., x'_{2_{i-1}} : T'_{2_{i-1}}$
5. $\Delta, x_1 : k(E)\ D, \overline{x_2 : T_2''}, \overline{x_2' : T_2'} \vdash x_1 \leftarrow C(\overline{x_2'}) : \mathsf{Void} \dashv \Delta\downarrow, k(E)\ C, \overline{x_2 : T_2''\downarrow}$ – by 1b-d,(update)
6. $\Delta, x_1 : k(E)\ D, \overline{x_2 : T_2} \vdash \overline{\mathsf{let}\ x_2' = e_2\ \mathsf{in}}\ x_1 \leftarrow C(\overline{x_2'}) : \mathsf{Void} \dashv \Delta\downarrow, x_1 : k(E)\ C, \overline{x_2 : T_2''\downarrow}$ – by 2-5,(let)

Case (update$_d$ $\Rightarrow$).

1. Almost identical to (update$\Rightarrow$)

Case (swap$\Rightarrow$).

1. By assumption
   (a) $\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash x_1.f \ :=: \ x_2 \Rightarrow T_2''$
   $\overset{\rightsquigarrow}{\mathsf{let}\ x_2' = e_2\ \mathsf{in}}\ x_1.f \ :=: \ x_2' \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2''$
   (b) $k_1 \in \{\mathsf{full}, \mathsf{shared}\}$
   (c) $T_2'\ f \in \mathit{fields}(C_1)$
   (d) $x_2 : T_2 \vdash x_2 \Leftarrow T_2' \rightsquigarrow e_2 \dashv x_2 : T_2''$
2. $x_2' \notin dom(\Delta)$ – by α-renaming
3. $\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash x_2 \Leftarrow T_2' \rightsquigarrow e_2 \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2''$ – by 1d,Translation Weakening
4. $\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash e_2 : T_2' \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2''$ – by induction on 3
5. $\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2'', x_2' : T_2' \vdash x_1.f \ :=: \ x_2' : T_2'' \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2''$ – by 1b-c,(swap)
6. $\Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2 \vdash \mathsf{let}\ x_2' = e_2\ \mathsf{in}\ x_1.f \ :=: \ x_2' : T_2'' \dashv \Delta, x_1 : k_1(D_1)\ C_1, x_2 : T_2''$ – by 2,4-5,(let)

Case (swap$_d$ $\Rightarrow$).

1. By assumption
   (a) $\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash x_1.f \ :=: \ x_2 \Rightarrow \mathsf{Dyn}$
   $\overset{\rightsquigarrow}{\mathsf{let}\ x_2' = e_2\ \mathsf{in}}\ x_1.f \ :=:_d \ x_2' \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2''$
   (b) $x_2 : T_2 \vdash x_2 \Leftarrow \mathsf{Dyn} \rightsquigarrow e_2 \dashv x_2 : T_2''$
2. $x_2' \notin dom(\Delta)$ – by α-renaming
3. $\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash x_2 \Leftarrow \mathsf{Dyn} \rightsquigarrow e_2 \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2''$ – by 1b,Transition Weakening
4. $\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash e_2 : \mathsf{Dyn} \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2''$ – by induction on 3
5. $\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2'', x_2' : \mathsf{Dyn} \vdash x_1.f \ :=:_d \ x_2' : \mathsf{Dyn} \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2''$ – by (swap$_d$)
6. $\Delta, x_1 : \mathsf{Dyn}, x_2 : T_2 \vdash \mathsf{let}\ x_2' = e_2\ \mathsf{in}\ x_1.f \ :=:_d \ x_2' : \mathsf{Dyn} \dashv \Delta, x_1 : \mathsf{Dyn}, x_2 : T_2''$ – by 2,4-5,(let)

Case (invoke$\Rightarrow$).

1. By assumption

(a) $\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow T$
$\rightsquigarrow$
$e_1;\ \overline{e_2;}\ x_1.m(\overline{x_2}) \dashv \Delta\downarrow, x_1 : T_t', \overline{x_2 : T_x'}$

(b) $mdecl(m, C_1) = T\ m(\overline{T_x \gg T_x'})[T_t \gg T_t']$

(c) $coerce(x_1, P_1\ C_1 \gg T_t) = e_1$

(d) $\overline{coerce(x_2, T_2 \gg T_x) = e_2}$

2. Let $n = |\ x_2\ |$
3. $\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash e_1 : \mathsf{Void} \dashv \Delta, x_1 : T_t, \overline{x_2 : T_2}$ – by 1c,Coercion Lemma
4. $\Delta, x_1 : T_t, \Delta_x, x_2 : T_2 \vdash e_{2_i} : \mathsf{Void} \dashv \Delta, x_1 : T_t, \Delta_x, x_2 : T_{x_i}$ – by 1d,Coercion Lemma, for $i \in [1..n]$, where $\Delta_x = x_{2_1} : T_{x_1}, ..., x_{2_{i-1}} : T_{x_{i-i}}, x_{2_{i+1}} : T_{2_{i+1}}, ...x_{2_n} : T_{2_n}$
5. $\Delta, x_1 : T_t, \overline{x_2 : T_2} \vdash \overline{e_2} : \mathsf{Void} \dashv \Delta, x_1 : T_t, \overline{x_2 : T_x}$ – by 4, where $i = [1..n]$,(let)
6. $\Delta, x_1 : T_t, \overline{x_2 : T_x} \vdash x_1.m(\overline{x_2}) : T \dashv \Delta\downarrow, x_1 : T_t', \overline{x_2 : T_x'}$ – by 1b,(invoke)
7. $\Delta, x_1 : T_t, \overline{x_2 : T_2} \vdash e_1;\ \overline{e_2;}\ x_1.m(\overline{x_2}) : T \dashv \Delta\downarrow, x_1 : T_t', \overline{x_2 : T_x'}$ – by 2-3,5-6,(let)

Case (invoke$_d$ $\Rightarrow$).

1. By assumption
   (a) $\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow \mathsf{Dyn}$
   $\rightsquigarrow$
   $\overline{e_2;}\ x_1.{}_d m(\overline{x_2}) \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}$
   (b) $\overline{coerce(x_2, T_2 \gg \mathsf{Dyn}) = e_2}$
2. Let $n = |\overline{x_2}|$
3. $\Delta, x_1 : \mathsf{Dyn}, \Delta_x, x_{2_i} : T_{2_i} \vdash e_2 : \mathsf{Void} \dashv \Delta, x_1 : \mathsf{Dyn}, \Delta_x, x_2 : \mathsf{Dyn}$ – by 1b and Coercion Lemma, for $i \in [1..n]$, where $\Delta_x = x_{2_1} : \mathsf{Dyn}, ..., x_{2_{i-1}} : \mathsf{Dyn}, x_{2_{i+1}} : T_{2_{i+1}}, ...x_{2_n} : T_{2_n}$
4. $\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash \overline{e_2;} : \mathsf{Void} \dashv \Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}$ – by 3 where $i = [1..n]$,(let)
5. $\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}} \vdash x_1.{}_d m(\overline{x_2}) : \mathsf{Dyn} \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}$ – by (invoke$_d$)
6. $\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash \overline{e_2;}\ x_1.{}_d m(\overline{x_2}) : \mathsf{Dyn} \dashv \Delta\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}$ – by 4-5,(let)

Case (hold$\Rightarrow$).

1. By assumption
   (a) $\Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) \Rightarrow T \rightsquigarrow \mathsf{hold}[x : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e') \dashv \Delta', x : T_1'$
   (b) $T_1 \Rrightarrow T_2/T_3$
   (c) $\Delta, x : T_3 \vdash e \Rightarrow T \rightsquigarrow e' \dashv \Delta', x : T_3'$
   (d) $T_2\downarrow / T_3' \Rrightarrow T_1'$
2. $\Delta, x : T_3 \vdash e' : T \dashv \Delta', x : T_3'$ – by induction on 1c
3. $\Delta, x : T_1 \vdash \mathsf{hold}[x : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e') : T \dashv \Delta', x : T_1'$ – by 1b,1d,2,(hold)

□

Lemma 8 (Merge Consistency)**.** *If $\langle [R, P_1, P_2], \leftrightarrow \rangle$ is connected , $P_1\ C_1 / P_2\ C_2 \Rrightarrow P_3\ C_3$, and $P_3 = k(E)$ then $\langle [R, P_3], \leftrightarrow \rangle$ is connected*

Proof. □

Lemma 9 (Memory Consistency)**.**

1. *If* $\mu, (\Delta, l : T), \rho$ ***ok*** *and* $\rho(l) \neq \mathsf{void}$ *then* $\mu, (\Delta, \rho(l) : T), \rho$ ***ok***
2. *If* $\mu, (\Delta, l : T), \rho$ ***ok*** *and* $l' \notin dom(\rho)$ *then* $\mu, (\Delta, l' : T), \rho[l' \mapsto \rho(l)]$ ***ok***
3. *If* μ, Δ, ρ ***ok*** *and* $l \notin dom(\rho)$ *then* $\mu, (\Delta, l : \mathsf{Void}), \rho[l \mapsto \mathsf{void}]$ ***ok***
4. *If* $\mu, (\Delta, l : T), \rho$ ***ok*** *then* $(\mu - \rho(l) : T), \Delta, \rho$ ***ok***
5. *If* μ, Δ, ρ ***ok*** *and* $o \in dom(\mu)$ *then* $\mu, (\Delta, o : \mathsf{Dyn}), \rho$ ***ok***
6. *If* $\mu, (\Delta, l : T_1), \rho$ ***ok*** *and* $T_1 \Rrightarrow T_2$ *then* $(\mu - \rho(l) : T_1 + \rho(l) : T_2), (\Delta, l : T_2), \rho$ ***ok***
7. *If* $\mu, (\Delta, l : T_1), \rho$ ***ok*** *and* $T_1 \Rrightarrow T_2/T_3$ *then* $(\mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3), (\Delta, l : T_2, l : T_3), \rho$ ***ok***
8. *If* $\mu, (\Delta, l : T_1, l : T_2), \rho$ ***ok*** *and* $T_1/T_2 \Rrightarrow T_3$ *then* $\mu - \rho(l) : T_1 - \rho(l) : T_2 + \rho(l) : T_3, (\Delta, l : T_3), \rho$ ***ok***
9. *If* $\mu, (\Delta, \overline{l : T}), \rho$ ***ok*** *and* $fields(C) = \overline{T\ f}$ *and* $o \notin dom(\mu)$ *then* $(\mu[o \mapsto C(\overline{\rho(l)})\ \cdot]), \Delta, \rho$ ***ok***
10. *If* μ, Δ, ρ ***ok****, and* $\mu(o) = C(\overline{o_f})\ R$ *and* $\langle [R, P], \leftrightarrow \rangle$ *is connected and* $C <: C'$ *then* $(\mu + o : P\ C'), (\Delta, o : P\ C'), \rho$ ***ok***
11. *If* μ, Δ, ρ ***ok*** *and* $\mu(o) = C(\overline{o_f})\ R$ *and* $fields(C) = \overline{T\ f}$ *and* $T_i \Downarrow T_i'$ *then* $\mu + o_i : T_i', (\Delta, o_i : T_i'), \rho$ ***ok***
12. *If* $\mu, (\Delta, l : T), \rho$ ***ok*** *then* $\mu, (\Delta, l : T\downarrow), \rho$ ***ok***
13. *If* $\mu, (\Delta, l : P\ C), \rho$ ***ok*** *and* $\mu(\rho(l)) = C'(\overline{o_f})\ R$ *then* $\mu, (\Delta, l : P\ C'), \rho$ ***ok***
14. *If* $\mu, (\Delta, l_1 : P\ C, l_2 : T_i), \rho$ ***ok*** *and* $fields(C) = \overline{T\ f}$ *and* $\mu(\rho(l_1)) = C'(\overline{o})\ R$ *and* $n = \mid \overline{o} \mid$ *then* $\mu[\rho(l_1) \mapsto C'(o_1, ..., o_{i-1}, \rho(l_2), o_{i+1}, ..., o_n)\ R], (\Delta, l_1 : P\ C, o_i : T_i), \rho$ ***ok***
15. *If* $\mu, (\Delta, l_1 : k_1(E_1)\ C_1, \overline{l_2 : T_d}), \rho$ ***ok*** *and* $k_1 \in \{\mathsf{full}, \mathsf{shared}\}$ *and* $D <: E_1$ *and* $fields(D) = \overline{T_d\ f_d}$ *and* $\mu(\rho(l_1)) = C(\overline{o})\ R$ *and* $fields(C) = \overline{T\ f}$ *then* $\mu[\rho(l_1) \mapsto D(\overline{\rho(l_2)})\ R], (\Delta\downarrow, \overline{o : T}, l_1 : k(E)\ D), \rho$ ***ok***

Proof.

1. Environment map – Assuming $\mu, (\Delta, l : T), \rho$ **ok** and $\rho(l) \neq \mathsf{void}$ we need to show that $\mu, (\Delta, \rho(l) : T), \rho$ **ok**. Memory does not change. The only object potentially affected is $\rho(l)$, which since we assume is not void, is equal to o, say. Since $types(\mu, (\Delta, l : T), \rho, o)) = types(\mu, (\Delta, o : T), \rho, o)$, we can conclude that $\mu, (\Delta, o : T), \rho \vdash o$ **ok**, and therefore $\mu, (\Delta, o : T), \rho$ **ok**
2. Environment rename
   Assuming $\mu, (\Delta, l : T), \rho$ **ok** and $l' \notin dom(\rho)$, we need to show that $\mu, (\Delta, l' : T), \rho[l' \mapsto \rho(l)]$ **ok**. If $T \neq \mathsf{Void}$, the only object affected can be $\rho(l)$. By the same argument above, we can conclude that $\mu, (\Delta, l' : T), \rho[l' \mapsto \rho(l)] \vdash \rho(l)$ **ok**. If $T = \mathsf{Void}$, then no objects are affected. Either way $\mu, (\Delta, l' : T), \rho[l' \mapsto \rho(l)] \vdash dom(\mu)$ **ok**. The rest of the premises for $\mu, (\Delta, l' : T), \rho[l' \mapsto \rho(l)]$ **ok**$dom(\mu)$ are trivial to show.
3. Adding Void
   Assuming μ, Δ, ρ **ok** and $l \notin dom(\rho)$ we need to show that $\mu, (\Delta, l : \mathsf{Void}), \rho[l \mapsto \mathsf{void}]$ **ok**. No objects are affected. The rest of the premises for $\mu, (\Delta, l : \mathsf{Void}), \rho[l \mapsto \mathsf{void}]$ **ok** are trivial to show.
4. Context subtraction
   Assuming $\mu, (\Delta, l : T), \rho$ **ok** we need to show that μ', Δ, ρ **ok**, where $\mu' = (\mu - \rho(l) : T)$. If $T \in \{\mathsf{Void}, \mathsf{Dyn}\}$, then $\mu = \mu'$, and therefore $\mu', (\Delta, l : T), \rho$ **ok**, and since $l : T$ does not affect any premises of memory consistency, we can also conclude μ', Δ, ρ **ok** Let us assume that $T \notin \{\mathsf{Void}, \mathsf{Dyn}\}$, so $T = k_l(E_l)\ C_l$, say. Our assumption also dictates that $\rho(l) = o$ for some o. Since $\mu, (\Delta, l : k_l(E_l)\ C_l), \rho \vdash o$ **ok**, we know that $\mu(o) = C(\overline{o'})\ R$, $types(\mu, (\Delta, l : k_l(E_l)\ C_l), \rho, o) = \overline{k(E)\ D}, k_l(E_l)\ C_l, \langle [\overline{k(E)\ D}, k_l(E_l)\ C_l], \leftrightarrow \rangle$ is connected and $\overline{k(E)}, k_l(E_l) = R$. Therefore, $\mu'(o) = C(\overline{o'})\ [\overline{k(E)}]$, $types(\mu', \Delta, \rho, o) = \overline{k(E)\ D}$, and $\langle [\overline{k(E)}], \leftrightarrow \rangle$ is connected , so we can conclude $\mu', \Delta, \rho \vdash o$ **ok**. The rest of the premises of μ', Δ, ρ **ok** are not affected, and are true by assumption.

5. Adding Dyn
Assuming μ, Δ, ρ **ok** and $o \in dom(\mu)$, we need to show that $\mu, (\Delta, o : \mathsf{Dyn}), \rho$ **ok**. The only object affected is o, and since $types(\mu, \Delta, \rho, o) = types(\mu, (\Delta, o : \mathsf{Dyn}), \rho, o)$, we can show that $\mu, \Delta, \rho \vdash o$ **ok**.

6. Type downgrade
Assuming $\mu, (\Delta, l : T_1), \rho$ **ok** and $T_1 \Rrightarrow T_2$ we need to show that $\mu', (\Delta, l : T_2), \rho$ **ok**, where $\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2$. If $T_1 \in \{\mathsf{Void}, \mathsf{Dyn}\}$, then $T_2 = T_1$, $\mu' = \mu$, and $\mu', (\Delta, l : T_2), \rho$ **ok** is true trivially. Therefore, we can assume that $T_1 = k_1(E_1)\ C_1$. If $T_2 = \mathsf{Dyn}$, then μ', Δ, ρ **ok** by case memory subtraction, and $\mu', (\Delta, l : T_2), \rho$ **ok**, by case Dyn addition. So we can assume that $T_2 = k_2(E_2)\ C_2$. Say $\rho(l) = o$ and $\mu(o) = C(\overline{o_f})\ R$ for some $o, C, \overline{o_f}, R$. The only object affected is o, and it suffices to show that $\mu', (\Delta, l : T_2), \rho \vdash o$ **ok**. By assumption, we know that $types(\mu, (\Delta, l : k_1(E_1)\ C_1), \rho, o) = \overline{k(E)\ C}, k_1(C_1)\ E_1, \langle[\overline{k(E)}, k_1(E_1)], \leftrightarrow\rangle$ is connected , and $C <: C_1$, and $\overline{k(E)}, k_1(C_1) = R$. By split consistency, $\langle[\overline{k(E)}, k_2(E_2)], \leftrightarrow\rangle$ is connected . By inversion on the derivation of type splitting (SplitP-P), $C_1 <: C_2$. Since $\mu'(o) = C(\overline{o_f})\ [\overline{k(E)}, k_2(C_2)]$, and $C <: C_2$, we can conclude $\mu', (\Delta, l : T_2), \rho \vdash o$ **ok**.

7. Type splitting
This follows the same argument as the case above, appealing to the split consistency lemma that if $\langle[\overline{k(E)}, k_1(E_1)], \leftrightarrow\rangle$ is connect
and $k_1(E_1)\ C_1 \Rrightarrow k_2(E_2)\ C_2/k_3(E_3)\ C_3$, then $\langle[\overline{k(E)}, k_2(E_2), k_3(C_3)], \leftrightarrow\rangle$ is connected

8. Type merging
Assuming $\mu, (\Delta, l : T_1, l : T_2), \rho$ **ok** and $T_1/T_2 \Rrightarrow T_3$, we need to show that $\mu - \rho(l) : T_1 - \rho(l) : T_2 + \rho(l) : T_3, (\Delta, l : T_3), \rho$ **ok**. This follows the same argument above, but appeals to the merge consistency lemma.

9. New object
Assuming $\mu, (\Delta, \overline{l : T}), \rho$ **ok** and $fields(C) = \overline{T\ f}$ and $o \notin dom(\mu)$, we need to show that μ', Δ, ρ **ok**, where $\mu' = \mu[o \mapsto C(\overline{\rho(l)})\ \cdot]$. By the restriction of field types, we know that $\overline{\rho(l) = o'}$ for some objects $\overline{o'}$. The only objects affected are $o, \overline{o'}$. Since $\overline{\mu(o') = \mu'(o')}$, and $\overline{types(\mu, (\Delta, l : T), \rho, o') = types(\mu', \Delta, \rho, o')}$, we can conclude that $\overline{\mu', \Delta, \rho \vdash o'}$ **ok**. Since $types(\mu', \Delta, \rho, o) = \cdot$, we can also conclude that $\mu', \Delta, \rho \vdash o$ **ok**.

10. Checked type
Assuming μ, Δ, ρ **ok** and $\mu(o) = C(\overline{o_f})\ R$ and $\langle[R, P'], \leftrightarrow\rangle$ is connected and $C <: C'$, we need to show that $\mu', (\Delta, o : P'\ C'), \rho$ **ok**, for $\mu' = \mu + o : P'\ C'$. The only object affected is o. From $\mu, \Delta, \rho \vdash o$ **ok**, $C <: C'$, and $\langle[R, P'], \leftrightarrow\rangle$ is connected , we can conclude that $\mu', (\Delta, o : P'\ C'), \rho \vdash o$ **ok**.

11. Field read
Assuming μ, Δ, ρ **ok** and $\mu(o) = C(\overline{o_f})\ R$ and $fields(C) = \overline{T\ f}$ and $T_i \Downarrow T_i'$, we need to show that $\mu', (\Delta, o_i : T_i'), \rho$ **ok**, where $\mu' = \mu + o_i : T_i'$. The only object affected is o_i. If $T = \mathsf{Dyn}$, then $\mu', ((\Delta, o_i : \mathsf{Dyn})), \rho \vdash$ **ok**, by adding Dyn case. Otherwise, we can assume that $T_i = k_i(E_i)\ D_i$, and $T_i{\downarrow} = k_i'(E_i)\ D_i$ for some k_i'. By assumption $T_i \in fieldTypes(\mu, o)$. Let $types(\mu, \Delta, \rho, o_i) = \overline{k(E)\ D}, T_i$. We know that $\langle[\overline{k(E)}, k_i(E_i)], \leftrightarrow\rangle$ is connected . By type splitting of $T_i \Rrightarrow T_i/T_i'$, we also know that $\langle[\overline{k(E)}, k_i(E_i), k_i'(E_i')], \leftrightarrow\rangle$ is connected
This is enough to show that $\mu', (\Delta, o_i : T_i'), \rho \vdash o_i$ **ok**.

12. Type demotion
Assuming $\mu, (\Delta, l : T), \rho$ **ok**, we need to show that $\mu, (\Delta, l : T{\downarrow}), \rho$ **ok**. If $T = T{\downarrow}$, then this is trivial. Otherwise, $T = k(E)\ D$, for some k, E, D, where $k \in \{\mathsf{pure}, \mathsf{shared}\}$, and $T' = k(E)\ E$. Let $\rho(l) = o$. By the well-formedness of T, it is necessarily the case that $D <: E$. Therefore, if $\mu, (\Delta, l : T), \rho \vdash o$ **ok**, then so is $\mu, (\Delta, l : T{\downarrow}), \rho \vdash o$ **ok**.

13. Type strengthening
Assuming $\mu, (\Delta, l : P\ C), \rho$ **ok** and $\mu(\rho(l)) = C'(\overline{o_f})\ R$, we need to show that $\mu, (\Delta, l : P\ C'), \rho$ **ok**. The only object affected is $\rho(l)$, which is o, say. We know that $\mu, (\Delta, l : P\ C), \rho \vdash o$ **ok**. The only premise to $\mu, (\Delta, l : P\ C'), \rho \vdash o$ **ok** which changes as a result is that we need to ensure that $C' <: C'$, which is true by definition.

14. Field swap
Assuming $\mu, (\Delta, l_1 : P\ C, l_2 : T_i), \rho$ **ok** and $fields(C) = \overline{T\ f}$ and $\mu(\rho(l_1)) = C'(\overline{o})\ R$ and $n = |\ \overline{o}\ |$, we need

to show that $\mu', (\Delta, l_1 : P\ C, o_i : T_i), \rho$ **ok**, where $\mu' = \mu[\rho(l_1) \mapsto C'(o_1, ..., o_{i-1}, \rho(l_2), o_{i+1}, ..., o_n)\ R]$. Only one object is affected, namely o_i. But since $types(\mu, (\Delta, l_1 : P\ C, l_2 : T_i), \rho, o_i) = types(\mu', (\Delta, l_1 : P\ C, o_i : T_i), \rho, o_i)$, knowing that $\mu, (\Delta, l_1 : P\ C, l_2 : T_i), \rho \vdash o_i$ **ok** lets us conclude that $\mu', (\Delta, l_1 : P\ C, o_i : T_i), \rho \vdash o_i$ **ok**.

15. Object update
    Assuming μ, Δ, ρ **ok**, $\Delta = (\Delta_x, l_1 : k_1(E_1)\ C_1, \overline{l_2 : T_d})$ and $k_1 \in \{\mathsf{full}, \mathsf{shared}\}$ and $D <: E_1$ and $fields(D) = \overline{T_d\ f_d}$ and $\mu(\rho(l_1)) = C(\overline{o_f})\ R$ and $fields(C) = \overline{T\ f}$, we need to show that μ', Δ', ρ **ok**, where $\mu' = \mu[\rho(l_1) \mapsto D(\overline{\rho(l_2)})\ R]$, and $\Delta' = (\Delta_x\downarrow, \overline{o_f : T}, l_1 : k(E)\ D)$. Several objects are affected here, $\rho(l_1)$, which is o_1, say, all objects that its fields point to, and all objects pointed to by $\overline{\rho(l_2)}$. Let's consider object o_1. By well-formedness of the class table, we know that $\overline{T_d\downarrow = T_d}$ and $\overline{T\downarrow = T}$.

    Say that $types(\mu, \Delta, \rho, o_1) = [\overline{T_x}, \overline{T_2}, \overline{T_c}, k_1(E_1)\ C_1]$, where the permissions come from $\Delta_x, \overline{l_2 : T_d}, \overline{o_f : T}, l_1 : k_1(E_1)\ C_1$, respectively. It is clear that $types(\mu', \Delta', \rho, o_1) = [\overline{T_x\downarrow}, \overline{T_2}, \overline{T_c}, k_1(E_1)\ C_1]$. Since permissions do not change, in order to show $\mu', \Delta', \rho \vdash o_1$ **ok**, it is enough to show that the current state respects the subtyping relation. Let $\overline{T_x = k_x(E_x)\ D_x}$. There are two cases to consider, if $k_1 = \mathsf{full}$, or if $k_1 = \mathsf{shared}$.

    Assuming that $k_1 = \mathsf{full}$, memory consistency of the assumption dictates that $\overline{k_x} = \mathsf{pure}$, and $E_1 <: \overline{E_x}$, and that $C <: \overline{D_x}$. By restriction on valid types, we know that $\overline{D_x <: E_x}$. Together with the initial assumption $D <: E_1$, and the transitivity of the subtyping relation, $D <: \overline{E_x}$. Since $\overline{T_x\downarrow} = k_x(E_x)\ E_x$, we can conclude that $\overline{T_x\downarrow}$ respects the subtyping relation for memory consistency.

    A similar argument can be made if $k_1 = \mathsf{shared}$. And that argument must be repeated with $\overline{T_2}$ and $\overline{T_c}$ – with the class table restriction that $\overline{T_c\downarrow = T_c}$ and $\overline{T_d\downarrow = T_d}$ – before you can conclude that $\mu', \Delta', \rho \vdash o_1$ **ok**.

    The other objects who are affected are not as interesting. $types(\mu', \Delta', \rho, o) = types(\mu, \Delta, \rho, o)\downarrow$, for all objects $o \neq o_1$. This, and $\mu, \Delta, \rho \vdash o$ **ok** is enough to show that $\mu', \Delta', \rho \vdash o$ **ok**.

□

Lemma 10 (Memory Addition)**.** *If $v \in dom(\mu) \cup \mathsf{void}$, and $v = \mathsf{void}$ iff $T = \mathsf{Void}$ then $\mu + v : T$ is defined*

Proof. If $T \in \{\mathsf{Void}, \mathsf{Dyn}\}$ then $\mu + v : T = \mu$ by (memadd-nop). If $T = P\ C$ then $v \neq \mathsf{Void}$ and $v \in dom(\mu)$, so $\mu(v) = C(\overline{o})\ R$, for some $C, \overline{o}, R$, and $\mu + v : T = \mu[v \mapsto C(\overline{o})\ [R, P]]$ by (memadd-perm). □

Theorem 11 (Progress)**.** *If e is a closed expression, $\Delta \vdash e : T \dashv \Delta'$ and μ, Δ, ρ **ok** then one of the following holds*

- *e is a value*
- *$\mu, \rho, e \rightarrow \mu', \rho', e'$ for some μ', ρ', e'*
- *$e = \mathbb{E}[e_d]$*

Proof.

Case (ctx-binder).

1. By assumption
   (a) $\Delta, b : T \vdash b : T \dashv \Delta$
   (b) $\mu, (\Delta, b : T), \rho$ **ok**
2. One of the following cases hold:
   (a) $b = x$
      i. Since b is closed, contradiction
   (b) $b = o$
      i. b is a value
   (c) $b = l$

i. $\rho(l)$ is defined – by memory consistency
ii. $\mu, \rho, l \to \mu, \rho, \rho(l)$ – by (lookup-binder)

Case (ctx).

1. By assumption
   (a) $\Delta, s : T_1 \vdash s[T_1 \Rrightarrow T_2/T_3] : T_2 \dashv \Delta, s : T_3$
   (b) $T_1 \Rrightarrow T_2/T_3$
   (c) $\mu, (\Delta, s : T_1), \rho$ **ok**
2. $s = l$ for some l – since the expression is closed
3. $(\mu - \rho(l)), \Delta, \rho$ **ok** – by 1c,2,memory consistency lemma
4. Let $\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3$
5. μ' is defined – by 3,memory addition
6. $\rho(l)$ is defined – by memory consistency
7. $\mu, \rho, l[T_1 \Rrightarrow T_2/T_3] \to \mu', \rho, \rho(l)$ – by (lookup-obj)

Case (void).

1. By assumption
   (a) $\Delta \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta$
   (b) μ, Δ, ρ **ok**
2. void is a value

Case (new).

1. By assumption
   (a) $\Delta, \overline{s : T} \vdash \mathsf{new}\ C(\overline{s}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta$
   (b) $fields(C) = \overline{T\ F}$
   (c) $\mu, (\Delta, \overline{s : T}), \rho$ **ok**
2. $\overline{s = l}$, for some $\overline{l}$ – the expression is closed
3. $\overline{\rho(l)}$ is defined – by memory consistency
4. Let $o \notin dom(\mu)$
5. $\mu, \rho, \mathsf{new}\ C(\overline{l}) \to \mu[o \mapsto C(\overline{\rho(l)})\ [\mathsf{full}(\mathsf{Object})]], \rho, o$ – by (new)

Case (ref).

1. By assumption
   (a) $\Delta, s : P\ C \vdash s.f : T' \dashv \Delta, s : P\ C$
   (b) $(T\ f) \in fields(C)$
   (c) $T \Downarrow T'$
   (d) $\mu, (\Delta, s : P\ C), \rho$ **ok**
2. $s = l$ for some l – the expression is closed
3. $\mu(\rho(l)) = C'(\overline{o})\ R$ for some $C', \overline{o}, R$ – by memory consistency

4. $C' <: C$ – by memory consistency
5. $(T\ f) \in \mathit{fields}(C')$ for some index i – by type consistency
6. $\mu' = \mu + o_i : T'$ is defined – by memory addition lemma
7. $\mu, \rho, l.f \rightarrow \mu', \rho, o_i$ – by 3,5,7-8,(ref)

Case (ref$_d$).

1. By assumption
   (a) $\Delta, s : \mathsf{Dyn} \vdash s.{}_d f : \mathsf{Dyn} \dashv \Delta, s : \mathsf{Dyn}$
   (b) $\mu, (\Delta, s : \mathsf{Dyn}), \rho$ **ok**
2. $s.{}_d f$ – is a runtime-checked expression

Case (invoke).

1. By assumption
   (a) $\Delta, s_1 : P_1\ C_1, \overline{s_2 : T_2} \vdash s_1.m(\overline{s_2}) : T_r \dashv \Delta\downarrow, s_1 : T_1', \overline{s_2 : T_2'}$
   (b) $\mathit{mdecl}(m, C_1) = T_r\ m(\overline{T_2 \gg T_2'})[P_1\ C_1 \gg T_1']$
   (c) $\mu, (\Delta, s_1 : P_1\ C_1, \overline{s_2 : T_2}), \rho$ **ok**
2. By 1a, and that this is a closed expression
   (a) $s_1 = l_1$ for some l_1
   (b) $\overline{s_2 = l_2}$ for some $\overline{l_2}$
3. $\mu(\rho(l_1)) = C(\overline{o})\ R$ for some $C, \overline{o}, R$ – by memory consistency
4. By 1c,3,type consistency:
   (a) $C <: C_1$
   (b) $\mathit{method}(m, C) = T_r\ m(\overline{T_x \gg T_x'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}$
   (c) $|\ T_x\ | = |\ T_2\ |$
5. $\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e$, by 3,4b,(invoke)

Case (invoke$_d$).

1. By assumption
   (a) $\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}} \vdash s_1.{}_d m(\overline{s_2}) : \mathsf{Dyn} \dashv \Delta\downarrow, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}}$
   (b) $\mu, (\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}}), \rho$ **ok**
2. $s_1.{}_d m(\overline{s_2})$ – is a runtime-checked expression

Case (swap).

1. By assumption
   (a) $\Delta, s_1 : k(E)\ D, s_2 : T_2 \vdash s_1.f\ :=:\ s_2 : T_2 \dashv \Delta, s_1 : k(E)\ D$
   (b) $k \in \{\ \mathsf{full}, \mathsf{shared}\ \}$
   (c) $(T_2\ f) \in \mathit{fields}(D)$
   (d) $\mu, (\Delta, s_1 : k(E)\ D, s_2 : T_2), \rho$ **ok**
2. By 1a, closed expression

(a) $s_1 = l_1$ for some l_1

(b) $s_2 = l_2$ for some l_2

3. By 1d,2,memory consistency

   (a) $\mu(\rho(l_1)) = C(\overline{o})\ R$ for some $C', \overline{o}, R$

   (b) $\rho(l_2)$ is defined

4. By 1c-d,2,3a,type consistency

   (a) $C <: D$

   (b) $(T_2\ f) \in \mathit{fields}(C)$ at some index i

5. $\mu, \rho, l_1.f_i\ :=:\ l_2 \to \mu[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\overline{o})\ R], \rho, o_i$ – by (swap)

Case (swap$_d$).

1. By assumption

   (a) $\Delta, s_1 : \mathsf{Dyn}, s_2 : \mathsf{Dyn} \vdash s_1.f\ :=:_d\ s_2 : \mathsf{Dyn} \dashv \Delta, s_1 : \mathsf{Dyn}$

   (b) $\mu, (\Delta, s_1 : \mathsf{Dyn}, s_2 : \mathsf{Dyn}), \rho$ **ok**

2. $s_1.f\ :=:_d\ s_2$ – is a runtime-checked expression

Case (update).

1. By assumption

   (a) $\Delta, s_1 : k(E)\ D, \overline{s_2 : T} \vdash s_1 \leftarrow C(\overline{s_2}) : \mathsf{Void} \dashv \Delta\downarrow, s_1 : k(E)\ C$

   (b) $k \in \{\mathsf{full}, \mathsf{shared}\}$

   (c) $C <: E$

   (d) $\mathit{fields}(C) = \overline{T\ f}$

   (e) $\mu, (\Delta, s_1 : k(E)\ D, \overline{s_2 : T}), \rho$ **ok**

2. By 1a, closed expression

   (a) $s_1 = l_1$ for some l_1

   (b) $\overline{s_2 = l_2}$ for some $\overline{l_2}$

3. $\mu(\rho(l_1)) = C'(\overline{o})\ R$ for some $C', \overline{o}, R$ – by 1b,memory consistency

4. Let $\mathit{fields}(C') = \overline{T'\ f'}$

5. Let $\mu_1 = \mu[\rho(l_1) \mapsto C(\overline{\rho(l_2)})\ R]$

6. $\mu_1, (\Delta\downarrow, l_1 : k(E)\ D, o : T'), \rho$ **ok** – by 1b-e,3-4,memory consistency lemma

7. Let $\mu' = \mu - \overline{o : T'}$

8. $\mu', (\Delta\downarrow, l_1 : k(E)\ D), \rho$ **ok** – by 6-7,memory consistency lemma

9. $\mu'(\rho(l_1)) = C'(\overline{o})\ R'$ – by definition of memory subtraction

10. $\mu, \rho, l_1 \leftarrow C(\overline{l_2}) \to \mu', \rho, \mathsf{void}$ – by 3-5,7,(update)

Case (update$_d$).

1. By assumption

   (a) $\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : T} \vdash s_1 \leftarrow_d C(\overline{s_2}) : \mathsf{Void} \dashv \Delta\downarrow, s_1 : \mathsf{Dyn}$

(b) $\mu, (\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : T}), \rho$ **ok**

2. $s_1 \leftarrow_d C(\overline{s_2})$ – is a runtime-checked expression

Case (let).

1. By assumption
   (a) $\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta \div x$
   (b) $\Delta \vdash e_1 : T_1 \dashv \Delta_1$
   (c) $\Delta, x : T_1 \vdash e_2 : T_2 \dashv \Delta_2$
   (d) $x : \mathsf{Void} \in \Delta_2$ *or* $x : T_1' \notin \Delta_2$
   (e) μ, Δ, ρ **ok**
2. One of the following three cases hold – by induction on 1b,1e
   (a) e_1 is a value
      i. Let $l \notin dom(\mu)$
      ii. $e_1 = v$ for some v
      iii. $\mu, \rho, \mathsf{let}\ x = v\ \mathsf{in}\ e_2 \rightarrow \mu, \rho[l \mapsto v], [l/x]e_2$ – by (let)
   (b) $\mu, \rho, e_1 \rightarrow \mu', \rho', e_1'$
      i. $\mu, \rho, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightarrow \mu', \rho', \mathsf{let}\ x = e_1\ \mathsf{in}\ e_1'$ – by 2b,(let-congr)
   (c) $e_1 = \mathbb{E}[e_{1_d}]$ for some runtime-checked expression e_{1_d}
      i. $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 = \mathbb{E}'[e_{1_d}]$, where $\mathbb{E}' = \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e_2$

Case (rel).

1. By assumption
   (a) $\Delta, s : T \vdash \mathsf{release}[T](s) : \mathsf{Void} \dashv \Delta$
   (b) $\mu, (\Delta, s : T), \rho$ **ok**
2. $s = l$ for some l – closed expression
3. $\mu, (\Delta, \rho(s) : T), \rho$ **ok** – by 1b,memory consistency lemma
4. $(\mu - \rho(s) : T), \Delta, \rho$ **ok** – by 3,memory consistency lemma
5. Let $\mu' = \mu - \rho(s) : T$
6. $\mu, \rho, \mathsf{release}[T](l) \rightarrow \mu', \rho, \mathsf{void}$ – by 3,(rel)

Case (assert).

1. By assumption
   (a) $\Delta, s : T_1 \vdash \mathsf{assert}\langle T_1 \gg T_2 \rangle(s) : \mathsf{Void} \dashv \Delta, s : T_2$
   (b) $T_1 \Rrightarrow T_2$
   (c) $\mu, (\Delta, s : T_1), \rho$ **ok**
2. $s = l$ for some l – closed expression
3. $\mu, (\Delta, \rho(s) : T_1), \rho$ **ok** – by 1c,memory consistency lemma
4. $(\mu - \rho(s) : T_1 + \rho(s) : T_2), (\Delta, \rho(s) : T_2), \rho$ **ok** – by 3,memory consistency lemma
5. Let $\mu' = \mu - \rho(s) : T_1 + \rho(s) : T_2$

6. $\mu, \rho, \mathsf{assert}\langle T_1 \gg T_2 \rangle(l) \to \mu', \rho, \mathsf{void}$ – by 5,(assert)

Case (assert$_d$).

1. By assumption
   (a) $\Delta, s : T_1 \vdash \mathsf{assert_d}\langle T_1 \gg T_2 \rangle(s) : \mathsf{Void} \dashv \Delta, s : T_2$
   (b) $\mu, (\Delta, s : T_1), \rho$ **ok**
2. $\mathsf{assert_d}\langle T_1 \gg T_2 \rangle(s)$ – is a runtime-checked expression

Case (hold).

1. By assumption
   (a) $\Delta, s : T_1 \vdash \mathsf{hold}[s : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) : T \dashv \Delta', s : T_1'$
   (b) $T_1 \Rrightarrow T_2/T_3$
   (c) $\mu, (\Delta, s : T_1), \rho$ **ok**
2. $s = l$ for some l – closed expression
3. $\mu, (\Delta, \rho(l) : T_1), \rho$ **ok** – by 1c,2,memory consistency lemma
4. $(\mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3), (\Delta, \rho(l) : T_2, \rho(l) : T_3), \rho$ **ok** – by 1b,3,memory consistency lemma
5. Let $\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3$
6. Choose $l' \notin dom(\rho)$
7. Let $\rho' = \rho[l' \mapsto \rho(l)]$
8. $\mu, \rho, \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) \to \mu', \rho', \mathsf{merge}[l' : T_2\!\downarrow / l : T_3' \Rrightarrow T_1'](e)$ – by (hold)

Case (merge).

1. By assumption
   (a) $\Delta, l_1 : T_1, l_2 : T_2 \vdash \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](e) : T \dashv \Delta', l_2 : T_3$
   (b) $\Delta, l_2 : T_2 \vdash e : T \dashv \Delta', l_2 : T_2'$
   (c) $T_1/T_2' \Rrightarrow T_3$
   (d) $T_1 = T_1\!\downarrow$
   (e) $\mu, (\Delta, l_1 : T_1, l_2 : T_2), \rho$ **ok**
2. One of three cases hold – by induction on 1b,1e
   (a) $e = v$ for some value v
      i. $T_2 = T_2'$ – by inversion of typing ((void) or (ctx-b)) on 1b,2a
      ii. $\mu, (\Delta_x, l_1 : T_1, l_2 : T_2), \rho$ **ok** – by 1e,3ai
      iii. $\mu, (\Delta_x, \rho(l_1) : T_1, \rho(l_2) : T_2), \rho$ **ok** – 3aii,memory consistency lemma
      iv. $(\mu - \rho(l) : T_1 - \rho(l) : T_2 + \rho(l) : T_3), (\Delta_x, \rho(l) : T_3), \rho$ **ok** – by 3aiii,2ai,memory consistency lemma
      v. Let $\mu' = \mu - \rho(l) : T_1 - \rho(l) : T_2 + \rho(l) : T_3$
      vi. $\mu, \rho, \mathsf{merge}[l_1 : T_1/l_2 : T_2 \Rrightarrow T_3](v) \to \mu', \rho, v$ – by (merge)
   (b) $\mu, \rho, e \to \mu', \rho', e'$ for some μ', ρ', e'
      i. $\mu, \rho, \mathsf{merge}[l_1 : T_1/l_2 : T_2 \Rrightarrow T_3](e) \to \mu', \rho', \mathsf{merge}[l_1 : T_1/l_2 : T_2 \Rrightarrow T_3](e')$ – by (congr)
   (c) $e = \mathbb{E}[e_d]$ for some runtime-checked expression e_d

i. $\mathsf{merge}[l_1 : T_1/l_2 : T_2 \Rrightarrow T_3](e) = \mathbb{E}'[e_d]$, where $\mathbb{E}' = \mathsf{merge}[l_1 : T_1/l_2 : T_2 \Rrightarrow T_3](\mathbb{E})$

□

Lemma 12 (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 $k_0(C_0) \leftrightarrow \mathsf{pure}(C_1)$ and $C_0 <: C_1$.

1. if $k_0 \Rrightarrow k_0'$ then $k_0'(C_0) \leftrightarrow \mathsf{pure}(C_1)$ by (pure).
2. then $\mathsf{pure} \Rrightarrow \mathsf{pure}$, and (pure) applies.

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

1. Suppose $\mathsf{shared} \Rrightarrow k$. Then proceed by cases.
   (a) If $\mathsf{shared} \Rrightarrow \mathsf{shared}$ 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 13. *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 14 (Context Binder Consistency?)**.** *If $\Delta \vdash e : T \dashv \Delta'$ then*

1. *If $s : T_1' \in \Delta'$ then $s : T_1 \in \Delta$ for some T_1*
2. *If $s : T_1{\downarrow} \in \Delta$ and s does not appear in e then $s : T_1{\downarrow} \in \Delta'$*

Proof. □

Lemma 15 (Double Demotion)**.** $T{\downarrow} = (T{\downarrow}){\downarrow}$

Proof. If $T = k(E)\ C$, where $k \in \{\mathsf{pure}, \mathsf{shared}\}$, then $T{\downarrow} = (T{\downarrow}) = k(E)\ E$. Otherwise, $T{\downarrow} = (T{\downarrow}){\downarrow} = T$. □

Lemma 16 (Weakening)**.** *If $\Delta \vdash e : T \dashv \Delta'$ then $\Delta, s : T_s{\downarrow} \vdash e : T \dashv \Delta, s : T_s{\downarrow}$*

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

Lemma 17 (Strengthening)**.** *If $\Delta, l : T_l \vdash e : T \dashv \Delta', l : T_l'$ and l does not occur in e, then $\Delta \vdash e : T \dashv \Delta'$.*

Proof. By induction on derivation of $\Delta, l : T_l \vdash e : T \dashv \Delta', l : T_l'$. □

Lemma 18 (Substitution)**.** *If $\Delta \vdash e : T \dashv \Delta'$ then $[s'/s]\Delta \vdash [s'/s]e : T \dashv [s'/s]\Delta$*

Proof. Substitute s' for s throughout the derivation of $\Delta \vdash e : T \dashv \Delta'$. □

Lemma 19 (Indirect Reference Weakening)**.** *If $\mu, \rho, e \to \mu', \rho', e'$, and $l' \notin dom(\rho) \cup dom(\rho')$ and l' does not occur in e, then for any value v, $\mu, (\rho, l' \mapsto v), e \to \mu', (\rho', l' \mapsto v), e'$.*

Proof. Induction on the derivation of $\mu, \rho, e \to \mu', \rho', e'$. □

Lemma 20 (Context Variable Conservation)**.** *If* $\Delta \vdash e : T \dashv \Delta'$ *then*

1. *If* $\Delta' = \Delta'_x, l : T'_l$ *then* $\Delta = \Delta_x, l : T_l$ *for some* T_l*. Furthermore, if* l *does not occur in* e*, and* $T'_l{\downarrow} = T_l$ *then* $T_l = T'_l$*.*
2. *If* $l \notin dom(\Delta')$ *then* $l \notin dom(\Delta)$*.*

Proof. Induction on the derivation of $\Delta \vdash e : T \dashv \Delta', l : T'_l$. □

Theorem 21 (Preservation-internal)**.** *If* $\Delta \vdash e : T \dashv \Delta'$*,* μ, Δ, ρ ***ok****, and* $\mu, \rho, e \to \mu', \rho', e'$*, then there exists* Δ''*, such that* $\Delta'' \vdash e' : T \dashv \Delta'$*, and* μ', Δ'', ρ' ***ok***

Proof.

Case (lookup-binder).

1. By assumption
   (a) $\Delta \vdash l : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l \to \mu, \rho, \rho(l)$
2. By inversion on 1a
   (a) $\Delta = \Delta', l : T$
3. $\mu, (\Delta', l : T), \rho$ **ok** – by 1b,2a
4. Case analyze on the type T
   (a) $T \neq \mathsf{Void}$
      i. $\rho(l) = o$ for some o – by memory consistency
      ii. Let $\Delta'' = \Delta', \rho(l) : T$
      iii. $\Delta', o : T \vdash o : T \dashv \Delta'$ – by (ctx-binder)
      iv. $\Delta'' \vdash \rho(l) : T \dashv \Delta'$ – by 4aii-iii
      v. $\mu, (\Delta', \rho(l) : T), \rho$ **ok** – by 3,4ai,memory consistency lemma
      vi. μ, Δ'', ρ **ok** – by 4aii,4av
   (b) $T = \mathsf{Void}$
      i. $\rho(l) = \mathsf{void}$ – by memory consistency
      ii. Let $\Delta'' = \Delta'$
      iii. $\Delta' \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta'$ – by (void)
      iv. $\Delta'' \vdash \rho(l) : T \dashv \Delta'$ – by 4b,4bi-ii
      v. $(\mu - \rho(l) : T), \Delta', \rho$ **ok** – by 3,memory consistency lemma
      vi. $\mu = \mu - \mathsf{void} : \mathsf{Void}$ – by (memadd-nop),(memsub)
      vii. μ, Δ'', ρ **ok** – by 4bii,4bv-vi
5. q.e.d. – by 4aiv,4avi,4biv,4bvii

Case (lookup-obj).

1. By assumption
   (a) $\Delta \vdash l[T_1 \Rrightarrow T_2/T_3] : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l[T_1 \Rrightarrow T_2/T_3] \to \mu', \rho, \rho(l)$

(d) $\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3$

2. By inversion on 1a
   (a) $T = T_2$
   (b) $\Delta = \Delta_x, l : T_1$
   (c) $\Delta' = \Delta_x, l : T_3$
   (d) $T_1 \Rrightarrow T_2/T_3$
3. Let $\Delta'' = \Delta_x, \rho(l) : T_2, l : T_3$
4. $\mu, (\Delta_x, l : T_1), \rho$ **ok** – by 1b,2b
5. $(\mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3), (\Delta_x, l : T_2, l : T_3), \rho$ **ok** – by 2d,4,memory consistency lemma
6. $(\mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3), (\Delta_x, \rho(l) : T_2, l : T_3), \rho$ **ok** – by 5,memory consistency lemma
7. μ', Δ'', ρ **ok** – by 1d,3,6
8. $\Delta_x, \rho(l) : T_2, l : T_3 \vdash \rho(l) : T_2 \dashv \Delta_x, l : T_3$ – by (ctx-binder)
9. $\Delta'' \vdash \rho(l) : T \dashv \Delta'$ – by 2a,2c,3,8
10. q.e.d. – by 7,9

Case (new).

1. By assumption
   (a) $\Delta \vdash \mathsf{new}\ C(\bar{l}) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, \mathsf{new}\ C(\bar{l}) \rightarrow \mu', \rho, o$
   (d) $o \notin dom(\mu)$
   (e) $\mu' = \mu[o \mapsto C(\overline{\rho(l)})\ [\mathsf{full}(\mathsf{Object})]]$
2. By inversion on 1a
   (a) $T = \mathsf{full}(\mathsf{Object})\ C$
   (b) $fields(C) = \overline{T_f\ f}$
   (c) $\Delta = \Delta', \overline{l : T_f}$
3. Let $\Delta'' = \Delta', o : \mathsf{full}(\mathsf{Object})\ C$
4. $\Delta', o : \mathsf{full}(\mathsf{Object})\ C \vdash o : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta'$ – by (ctx-binder)
5. $\Delta'' \vdash o : T \dashv \Delta'$ – by 2a,3-4
6. $\mu, (\Delta', \overline{l : T_f}), \rho$ **ok** – by 1b,2c
7. $(\mu[o \mapsto C(\overline{\rho(l)})\ \cdot]), \Delta', \rho$ **ok** – by 1d,2b,6,memory consistency lemma
8. $\mu' = \mu[o \mapsto C(\overline{\rho(l)})\ \cdot] + o : \mathsf{full}(\mathsf{Object})\ C$ – by 1e,(memadd-perm)
9. $\mu', (\Delta', o : \mathsf{full}(\mathsf{Object})\ C), \rho$ **ok** – by 7-8,memory consistency lemma
10. μ', Δ'', ρ **ok** – by 3,9
11. q.e.d. – by 5,10

Case (ref).

1. By assumption
   (a) $\Delta \vdash l.f_i : T_e \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l.f_i \rightarrow \mu', \rho, o_i$
   (d) $\mu(\rho(l)) = C(\overline{o})\ R$
   (e) $fields(C) = \overline{T\ f}$
   (f) $T_i \Downarrow T_i'$
   (g) $\mu' = \mu + o_i : T_i'$
2. By inversion on 1a
   (a) $\Delta = \Delta'$
   (b) $\Delta = \Delta_x, l : P\ C'$
   (c) $(T_f\ f_i) \in \mathit{fields}(C')$
   (d) $T_f \Downarrow T_e$
3. $C <: C'$ – by 1b,1d,2b,memory consistency
4. By 1e,2c,3,type consistency
   (a) $(T_f\ f_i) \in \mathit{fields}(C)$
   (b) $T_f = T_i$
5. $T_e = T_i'$ – by 1f,2d,4b
6. Let $\Delta'' = \Delta', o_i : T_i'$
7. $\Delta', o_i : T_i' \vdash o_i : T_i' \dashv \Delta'$ – by (ctx-binder)
8. $\Delta'' \vdash o_i : T_e \dashv \Delta'$ – by 5-7
9. $(\mu + o_i : T_i'), (\Delta, o_i : T_i'), \rho$ **ok** – by 1b,1d-f,memory consistency lemma
10. μ', Δ'', ρ **ok** – 1g,6,9
11. q.e.d. – by 8,10

Case (ref$_d$).

1. By assumption
   (a) $\Delta \vdash l._d f_i : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l._d f_i \rightarrow \mu, \rho, o_i$
   (d) $\mu(\rho(l)) = C(\overline{o})\ R$
   (e) $fields(C) = \overline{T_f\ f}$
2. By inversion on 1a
   (a) $\Delta = \Delta' = \Delta_x, l : \mathsf{Dyn}$
   (b) $T = \mathsf{Dyn}$
3. Let $\Delta'' = \Delta', o_i : \mathsf{Dyn}$
4. $\Delta', o_i : \mathsf{Dyn} \vdash o_i : \mathsf{Dyn} \dashv \Delta'$ – by (ctx-binder)

5. $\Delta'' \vdash o_i : T \dashv \Delta'$ – by 2b,3-4
6. $\mu, (\Delta, o_i : \mathsf{Dyn}), \rho$ **ok** – by 1b,memory consistency lemma
7. μ, Δ'', ρ **ok** – by 3,6
8. $q.e.d$ – by 5,7

Case (invoke).

1. By assumption
   (a) $\Delta \vdash l_1.m(\overline{l_2}) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e$
   (d) $\mu(\rho(l_1)) = C(\overline{o})\ R$
   (e) $method(m, C) = T_r\ m(\overline{T_x \gg T'_x\ x})\ [T_t \gg T'_t]\ \{\ \mathsf{return}\ e;\ \}$
2. By inversion of 1a
   (a) $\Delta = \Delta_x, l_1 : P_1\ C_1, \overline{l_2 : T_2}$
   (b) $mdecl(m, C_1) = T_{res}\ m(\overline{T_2 \gg T'_2})\ [P_1\ C_1 \gg T'_1]$
   (c) $\Delta' = \Delta_x\downarrow, l_1 : T'_1, \overline{l_2 : T'_2}$
   (d) $T = T_{res}$
3. By 1-2,type consistency
   (a) $C <: C_1$
   (b) $T_{res} = T_r$
   (c) $\overline{T_2 = T_x}$
   (d) $\overline{T'_2 = T'_x}$
   (e) $\overline{T'_1 = T'_t}$
   (f) $T_t = P\ C$
   (g) $P_1 = P$
4. $\mathsf{this}, \overline{x} \notin dom(\Delta)$ – by $\alpha-$renaming
5. $\mathsf{this} : T_t, \overline{x : T_x} \vdash e : T_r \dashv \mathsf{this} : T'_t, \overline{x : T'_x}$ – by method typing
6. $\Delta_x\downarrow, \mathsf{this} : T_t, \overline{x : T_x} \vdash e : T_r \dashv \Delta_x\downarrow, \mathsf{this} : T'_t, \overline{x : T'_x}$ – by 5,weakening
7. Let $\Delta'' = \Delta_x\downarrow, l_1 : T_t, \overline{l_2 : T_x}$
8. $\Delta_x\downarrow, l_1 : T_t, \overline{l_2 : T_x} \vdash [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e : T_r \dashv \Delta_x\downarrow, l_1 : T'_t, \overline{l_2 : T'_x}$ – by substitution
9. $\Delta'' \vdash [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e : T \dashv \Delta'$ – by 7-8,3d-e,2d,3b
10. $\mu, (\Delta_x\downarrow, l_1 : P_1\ C_1, \overline{l_2 : T_2}), \rho$ **ok** – by 1b,2a,memory consistency lemma
11. $\mu, (\Delta_x\downarrow, l_1 : P_1\ C, \overline{l_2 : T_2}), \rho$ **ok** – by 1d,3a,10,memory consistency lemma
12. μ, Δ'', ρ **ok** – by 3f-g,7,11
13. q.e.d – by 9,12

Case (invoke$_d$).

1. By assumption

(a) $\Delta \vdash l_1.{}_d m(\overline{l_2}) : T \dashv \Delta'$

(b) μ, Δ, ρ **ok**

(c) $\mu, \rho, l_1.{}_d m(\overline{l_2}) \rightarrow \mu, \rho,$
$\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t \rangle(l_1);$
$\overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_x \rangle(l_2);}$
$\mathsf{let}\ ret = l_1.m(\overline{l_2})\ \mathsf{in}$
$\quad \mathsf{assert}\langle T_t' \gg \mathsf{Dyn} \rangle(l_1);$
$\quad \overline{\mathsf{assert}\langle T_x' \gg \mathsf{Dyn} \rangle(l_2);}$
$\quad \mathsf{assert}\langle T_r \gg \mathsf{Dyn} \rangle(ret);$
$\quad ret$

(d) $\mu(\rho(l_1)) = C(\overline{o})\ R$

(e) $mdecl(m, C) = T_r\ m(\overline{T_x \gg T_x'})\ [T_t \gg T_t']$

2. By inversion on 1a

(a) $\Delta = \Delta' = \Delta_x, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}$

(b) $T = \mathsf{Dyn}$

3. $ret \notin dom(\Delta)$ – by α-renaming

4. $\Delta_x, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}} \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t \rangle(l_1) : \mathsf{Void} \dashv \Delta_x, l_1 : T_t, \overline{l_2 : \mathsf{Dyn}}$ – by (assert$_d$)

5. $\Delta_x, l_1 : T_t, \overline{l_2 : \mathsf{Dyn}} \vdash \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_x \rangle(l_2);} : \mathsf{Void} \dashv \Delta_x, l_1 : T_t, \overline{l_2 : T_x}$ – by (assert$_d$)

6. $\Delta_x, l_1 : T_t, \overline{l_2 : T_x} \vdash l_1.m(\overline{l_2}) : T_r \dashv \Delta_x\downarrow, l_1 : T_t', \overline{l_2 : T_x'}$ – by (invoke)

7. $\Delta_x\downarrow, l_1 : T_t', \overline{l_2 : T_x'}, ret : T_r \vdash \mathsf{assert}\langle T_t' \gg \mathsf{Dyn} \rangle(l_1) : \mathsf{Void} \dashv \Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : T_x'}, ret : T_r$ – by (assert)

8. $\Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : T_x'}, ret : T_r \vdash \overline{\mathsf{assert}\langle T_x' \gg \mathsf{Dyn} \rangle(l_2);} : \mathsf{Void} \dashv \Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}, ret : T_r$ – by (assert)

9. $\Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}, ret : T_r \vdash \mathsf{assert}\langle T_r \gg \mathsf{Dyn} \rangle(ret) : \mathsf{Void} \dashv \Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}, ret : \mathsf{Dyn}$ – by (assert)

10. $\Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}, ret : \mathsf{Dyn} \vdash ret : \mathsf{Dyn} \dashv \Delta_x\downarrow, l_1 : \mathsf{Dyn}, \overline{l_2 : \mathsf{Dyn}}$ – by ctx-binder

11. $\Delta \vdash \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t \rangle(l_1);} \quad : T \dashv \Delta'$ – by 3-10,(let)
$\overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_x \rangle(l_2);}$
$\mathsf{let}\ ret = l_1.m(\overline{l_2})\ \mathsf{in}$
$\quad \mathsf{assert}\langle T_t' \gg \mathsf{Dyn} \rangle(l_1);$
$\quad \overline{\mathsf{assert}\langle T_x' \gg \mathsf{Dyn} \rangle(l_2);}$
$\quad \mathsf{assert}\langle T_r \gg \mathsf{Dyn} \rangle(ret);$
$\quad ret$

12. q.e.d. – by 1b,11

Case (swap).

1. By assumption

(a) $\Delta \vdash l_1.f_i\ :=:\ l_2 : T \dashv \Delta'$

(b) μ, Δ, ρ **ok**

(c) $\mu(\rho(l_1)) = C(\overline{o})\ R$

(d) $\mu' = \mu[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\overline{o})\ R]$

(e) $fields(C) = \overline{T_f\ f}$

2. By inversion on 1a

(a) $\Delta = \Delta_x, l_1 : k_1(E_1)\ C_1, l_2 : T_2$

(b) $\Delta' = \Delta_x, l_1 : k_1(E_1)\ C_1$

(c) $k_1 \in \{\mathsf{full}, \mathsf{shared}\}$

(d) $(T_2\ f_i) \in \mathit{fields}(C_1)$

(e) $T_2 = T$

3. Let $\Delta'' = \Delta', o_i : T$
4. $\Delta', o_i : T \vdash o_i : T \dashv \Delta'$ – by (ctx-binder)
5. $\Delta'' \vdash o_i : T \dashv \Delta'$ – by 3-4
6. $\mu[\rho(l_1) \mapsto C(o_1, ..., o_{i-1}, \rho(l_2), o_{i+1}, ..., o_n)\ R], (\Delta_x, l_1 : k_1(E_1)\ C_1, o_i : T_2), \rho$ **ok** – by 1b-c,2a,memory consistency lemma
7. μ', Δ'', ρ **ok** – by 1d,6,2b,3
8. q.e.d – by 5,7

Case (swap$_d$).

1. By assumption

(a) $\Delta \vdash l_1.f_i\ :=:_d\ l_2 : T \dashv \Delta'$

(b) μ, Δ, ρ **ok**

(c) $\mu, \rho, l_1.f_i\ :=:_d\ l_2 \rightarrow \mu, \rho,$
$\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1);$
$\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_{f_i} \rangle(l_2);$
$\mathsf{let}\ ret = l_1.f_i\ :=:\ l_2\ \mathsf{in}$
$\quad \mathsf{assert}\langle T_1 \gg \mathsf{Dyn} \rangle(l_1);$
$\quad \mathsf{assert}\langle T_{f_i} \gg \mathsf{Dyn} \rangle(ret);$
$\quad ret$

(d) $\mu(\rho(l)) = C(\overline{o})\ R$

(e) $\mathit{fields}(C) = \overline{T_f\ f}$

(f) $C_g = \{D :\ if\ \mathsf{shared}(D) \in R,\ C : otherwise\}$

(g) $P = \mathsf{shared}(C_g)$

(h) $T_1 = P\ C$

2. By inversion on 1a

(a) $T = \mathsf{Dyn}$

(b) $\Delta = \Delta_x, l_1 : \mathsf{Dyn}, l_2 : \mathsf{Dyn}$

(c) $\Delta' = \Delta_x, l_1 : \mathsf{Dyn}$

3. $\Delta_x, l_1 : \mathsf{Dyn}, l_2 : \mathsf{Dyn} \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1) : \mathsf{Void} \dashv \Delta_x, l_1 : T_1, l_2 : \mathsf{Dyn}$ – by (assert$_d$)
4. $\Delta_x, l_1 : T_1, l_2 : \mathsf{Dyn} \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_{f_i} \rangle(l_2) : \mathsf{Void} \dashv \Delta_x, l_1 : T_1, l_2 : T_{f_i}$ – by (assert$_d$)
5. $\Delta_x, l_1 : T_1, l_2 : T_{f_i} \vdash l_1.f_i\ :=:\ l_2 : T_{f_i} \dashv \Delta_x, l_1 : T_1$ – by 1e,1g-h,(swap)
6. $ret \notin dom(\Delta_x)$ – by α-renaming
7. $\Delta_x, l_1 : T_1, ret : T_{f_i} \vdash \mathsf{assert}\langle T_1 \gg \mathsf{Dyn} \rangle(l_1) : \mathsf{Void} \dashv \Delta_x, l_1 : \mathsf{Dyn}, ret : T_{f_i}$ – by (assert)
8. $\Delta_x, l_1 : \mathsf{Dyn}, ret : T_{f_i} \vdash \mathsf{assert}\langle T_{f_i} \gg \mathsf{Dyn} \rangle(ret) : \mathsf{Void} \dashv \Delta_x, l_1 : \mathsf{Dyn}, ret : \mathsf{Dyn}$ – by (assert)
9. $\Delta_x, l_1 : \mathsf{Dyn}, ret : \mathsf{Dyn} \vdash ret : \mathsf{Dyn} \dashv \Delta_x, l_1 : \mathsf{Dyn}$ – by (ctx-binder)

10. $\Delta \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1);$ $\quad : T \dashv \Delta'$ – by 2b-c,3-9
    $\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_{f_i} \rangle(l_2);$
    $\mathsf{let}\ ret = l_1.f_i\ :=:\ l_2\ \mathsf{in}$
    $\quad \mathsf{assert}\langle T_1 \gg \mathsf{Dyn} \rangle(l_1);$
    $\quad \mathsf{assert}\langle T_{f_i} \gg \mathsf{Dyn} \rangle(ret);$
    $\quad ret$
11. q.e.d – by 1b,10

Case (update).

1. By assumption
   (a) $\Delta \vdash l_1 \leftarrow C'(\overline{l_2}) : T_e \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu(\rho(l_1)) = C(\overline{o})\ R$
   (d) $fields(C) = \overline{T\ f}$
   (e) $\mu_1 = \mu[\rho(l_1) \mapsto C'(\overline{\rho(l_2)})\ R]$
   (f) $\mu' = \mu_1 - \overline{o : T}$
   (g) $\mu, \rho, l_1 \leftarrow C'(\overline{l_2}) \rightarrow \mu', \rho, \mathsf{void}$
2. By inversion on 1a
   (a) $\Delta = \Delta_x, l_1 : k_1(E_1)\ D_1, \overline{l_2 : T_2}$
   (b) $\Delta' = \Delta_x\downarrow, l_1 : k_1(E_1)\ C'$
   (c) $T_e = \mathsf{Void}$
   (d) $k_1 \in \{\mathsf{full}, \mathsf{shared}\}$
   (e) $C' <: E_1$
   (f) $fields(C') = \overline{T_2\ f_2}$
3. $\Delta' \vdash \mathsf{void} : T_e \dashv \Delta'$ – by 2c,(void)
4. $\mu, (\Delta_x\downarrow, l_1 : k_1(E_1)\ D_1, \overline{l_2 : T_2}), \rho$ **ok** – by 1b,2a,memory consistency lemma
5. $\mu_1, (\Delta_x\downarrow, l_1 : k_1(E_1)\ C', \overline{o : T}), \rho$ **ok** – by 4,1c-d,1e,2d-f,memory consistency lemma
6. $\mu', (\Delta_x\downarrow, l_1 : k_1(E_1)\ C'), \rho$ **ok** – by 1f,5,memory consistency lemma
7. μ', Δ', ρ **ok**
8. q.e.d. – by 3,7

Case (update$_d$).

1. By assumption
   (a) $\Delta \vdash l_1 \leftarrow_d C'(\overline{l_2}) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, l_1 \leftarrow_d C'(\overline{l_2}) \rightarrow \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1);$
   $\quad l_1 \leftarrow C'(\overline{l_2});$
   $\quad \mathsf{assert}\langle T_1' \gg \mathsf{Dyn} \rangle(l_1)$
   (d) $\mu(\rho(l_1)) = C(\overline{o_f})\ R$
   (e) $C_g = \{D :\ if\ \mathsf{shared}(D) \in R, C \ddot{\vee} C' : otherwise\}$
   (f) $C' <: C_g$

(g) $P = \mathsf{shared}(C_g)$

(h) $T_1 = P\ C$

(i) $T_1' = P\ C'$

2. By inversion on 1a

   (a) $\Delta = \Delta_x, l_1 : \mathsf{Dyn}, \overline{l_2 : T_f}$

   (b) $\Delta' = \Delta_x\downarrow, l_1 : \mathsf{Dyn}$

   (c) $T = \mathsf{Void}$

   (d) $fields(C') = \overline{T_f\ f}$

3. $\Delta_x, l_1 : \mathsf{Dyn}, \overline{l_2 : T_f} \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1) : \mathsf{Void} \dashv \Delta_x, l_1 : T_1, \overline{l_2 : T_f}$ – by (assert$_d$)

4. $\Delta_x, l_1 : T_1, \overline{l_2 : T_f} \vdash l_1 \leftarrow C'(\overline{l_2}) : \mathsf{Void} \dashv \Delta_x\downarrow, l_1 : T_1'$ – by 1e-h,2d,(update)

5. $\Delta_x\downarrow, l_1 : T_1' \vdash \mathsf{assert}\langle T_1' \gg \mathsf{Dyn} \rangle(l_1) : \mathsf{Void} \dashv \Delta_x\downarrow, l_1 : \mathsf{Dyn}$ – by (Split-Dyn),(assert)

6. $\Delta \vdash \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_1 \rangle(l_1);\ : T \dashv \Delta'$ – by 2a-c,3-5, (let)
   $l_1 \leftarrow C'(\overline{l_2})$;
   $\mathsf{assert}\langle T_1' \gg \mathsf{Dyn} \rangle(l_1)$

7. q.e.d – by 1b,6

Case (let).

1. By assumption

   (a) $\Delta \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : T \dashv \Delta'$

   (b) μ, Δ, ρ **ok**

   (c) $\mu, \rho, \mathsf{let}\ x = v\ \mathsf{in}\ e \rightarrow \mu, \rho[l \mapsto v], [l/x]e$

   (d) $l \notin dom(\rho)$

2. By inversion on 1a

   (a) $\Delta \vdash v : T_1 \dashv \Delta_1$

   (b) $\Delta_1, x : T_1 \vdash e : T \dashv \Delta_2$

   (c) $x : \mathsf{Void} \in \Delta_2$ *or* $x : T_1 \notin \Delta_2$

   (d) $\Delta' = \Delta_2 \div x$

3. Let $\Delta'' = \Delta_1, l : T_1$

4. $\Delta_1, l : T_1 \vdash [l/x]e : T \dashv [l/x]\Delta_2$ – by 1d,2b,substitution

5. $\Delta'' \vdash [l/x]e : T \dashv \Delta'$ – by 2d,5

6. By case analysis on v

   (a) $v = \mathsf{void}$

      i. $\Delta = \Delta_1$ – by inversion of 2a

      ii. $T = \mathsf{Void}$ – by context consistency

      iii. $\mu, (\Delta_1, l : T), \rho[l \mapsto v]$ **ok** – by 1b,6ai-ii,memory consistency lemma

   (b) $v = o$ for some o

      i. $\Delta = \Delta_1, o : T$ – by inversion of 2a

      ii. $\mu, (\Delta_1, v : T), \rho$ **ok**

iii. $\mu, (\Delta_1, l : T), \rho[l \mapsto v]$ **ok** – by 6bii,memory consistency lemma

7. $\mu, \Delta'', \rho[l \mapsto v]$ **ok** – 3,6aiii,6biii
8. q.e.d. – by 5,7

Case (let-congr).

1. By assumption
   (a) $\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \to \mu', \rho', \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2$
   (d) $\mu, \rho, e_1 \to \mu', \rho', e_1'$
2. By inversion on 1a
   (a) $\Delta \vdash e_1 : T_1 \dashv \Delta_1$
   (b) $\Delta_1, x : T_1 \vdash e_2 : T_2 \dashv \Delta_2$
   (c) $x : \mathsf{Void} \in \Delta_2$ *or* $x : T_1' \notin \Delta_2$
   (d) $\Delta' = \Delta_2 \div x$
3. By induction on 1b,1d,2a
   (a) $\Delta'' \vdash e_1' : T_1 \dashv \Delta_1$, for some Δ''
   (b) μ', Δ'', ρ' **ok**
4. $\Delta'' \vdash \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 : T \dashv \Delta'$ – by 3a,2b-d,(let)
5. $q.e.d$ – by 3b,4

Case (rel).

1. By assumption
   (a) $\Delta \vdash \mathsf{release}[T_l](l) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, \mathsf{release}[T_l](l) \to \mu', \rho, \mathsf{void}$
   (d) $\mu' = \mu - \rho(l) : T_l$
2. By inversion on 1a
   (a) $T = \mathsf{Void}$
   (b) $\Delta = \Delta', l : T_l$
3. $\Delta' \vdash \mathsf{void} : T \dashv \Delta'$ – by 7, (void)
4. μ', Δ', ρ **ok** – by 1b,2b,4,memory consistency lemma
5. q.e.d – by 3-4

Case (assert).

1. By assumption
   (a) $\Delta \vdash \mathsf{assert}\langle T \gg T' \rangle(l) : T_e \dashv \Delta'$
   (b) μ, Δ, ρ **ok**

(c) $\mu, \rho, \mathsf{assert}\langle T \gg T' \rangle(l) \rightarrow \mu', \rho, \mathsf{void}$

(d) $\mu' = \mu - \rho(l) : T + \rho(l) : T'$

2. By inversion on 1a

(a) $T_e = \mathsf{Void}$

(b) $\Delta = \Delta_x, l : T$

(c) $\Delta' = \Delta_x, l : T'$

(d) $T \Rrightarrow T'$

3. $\Delta' \vdash \mathsf{void} : T_e \dashv \Delta'$ – by 5,(void)

4. μ', Δ', ρ **ok** by 1b,1d,2b-c,memory consistency

5. q.e.d. – by 3-4

Case (assert$_d$).

1. By assumption

(a) $\Delta \vdash \mathsf{assert_d}\langle T \gg T' \rangle(l) : T_e \dashv \Delta'$

(b) μ, Δ, ρ **ok**

(c) $\mu, \rho, \mathsf{assert_d}\langle T \gg T' \rangle(l) \rightarrow \mu', \rho, \mathsf{void}$

(d) $\mu' = \mu - \rho(l) : T + \rho(l) : T'$

(e) $\mu'(\rho(l)) = C\,(\overline{o})\;R$

(f) $\langle R, \leftrightarrow \rangle$ is connected

(g) if $T' = P'\;C'$ then $C <: C'$

2. By inversion on 1a

(a) $T_e = \mathsf{Void}$

(b) $\Delta = \Delta_x, l : T$

(c) $\Delta' = \Delta_x, l : T'$

(d) $T \neq \mathsf{Void}$

(e) $T' \neq \mathsf{Void}$

3. $\Delta' \vdash \mathsf{void} : T_e \dashv \Delta'$ – by 2a,(void)

4. $(\mu - \rho(l) : T), \Delta_x, \rho$ **ok** – by 1b,2b,memory consistency

5. By 2e, $T' = \mathsf{Dyn}$ or $T' = P'\;C'$

(a) Assume $T' = \mathsf{Dyn}$

i. $(\mu - \rho(l) : T), (\Delta_x, l : \mathsf{Dyn}), \rho$ **ok** – by 4,memory consistency

ii. $(\mu - \rho(l) : T + \rho(l) : \mathsf{Dyn}), (\Delta_x, l : \mathsf{Dyn}), \rho$ **ok** – by (memadd-nop)

(b) Assume $T' = P'\;C'$

i. $C <: C'$ – by 5b,1g

ii. $(\mu - \rho(l) : T + \rho(l) : T'), (\Delta_x, l : T'), \rho$ **ok** – by 4,5bi,memory consistency

6. μ', Δ', ρ **ok** – by 5aii,5bii

7. q.e.d. – by 3,6

Case (hold).

1. By assumption
   (a) $\Delta \vdash \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3$
   (d) $l' \notin dom(\rho)$
   (e) $\rho' = \rho[l' \mapsto \rho(l)]$
   (f) $\mu, \rho, \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) \rightarrow \mu', \rho', \mathsf{merge}[l' : T_2{\downarrow} \,/ l : T_3' \Rrightarrow T_1'](e)$
2. By inversion on 1a
   (a) $\Delta = \Delta_x, l : T_1$
   (b) $T_1 \Rrightarrow T_2/T_3$
   (c) $\Delta_x, l : T_3 \vdash e : T \dashv \Delta_x', l : T_3'$
   (d) $T_2{\downarrow} \,/ T_3' \Rrightarrow T_1'$
   (e) $\Delta' = \Delta_x', l : T_1'$
3. $T_2{\downarrow} = (T_2{\downarrow}){\downarrow}$ – double demotion
4. Let $\Delta'' = \Delta_x, l : T_3, l' : T_2{\downarrow}$
5. $\Delta_x, l : T_2{\downarrow}, l : T_3 \vdash \mathsf{merge}[l' : T_2{\downarrow} \,/ l : T_3' \Rrightarrow T_1'](e) : T \dashv \Delta_x', l : T_1'$ – by 2c-d,3,(merge)
6. $\Delta'' \vdash \mathsf{merge}[l' : T_2{\downarrow} \,/ l : T_3' \Rrightarrow T_1'](e) : T \dashv \Delta'$ – by 2e,4-5
7. $\mu, (\Delta_x, l : T_1), \rho$ **ok** – by 1b,2a
8. $\mu', (\Delta_x, l : T_2, l : T_3), \rho$ **ok** – by 7,2b,memory consistency lemma
9. $\mu', (\Delta_x, l : T_2{\downarrow}, l : T_3), \rho$ **ok** – by 8,memory consistency lemma
10. μ', Δ'', ρ **ok** – by 9,4
11. q.e.d. – by 6,10

Case (merge).

1. By assumption
   (a) $\Delta \vdash \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](v) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu' = \mu - \rho(l_1) : T_1 - \rho(l_2) : T_2' + \rho(l_2) : T_3$
   (d) $\mu, \rho, \mathsf{merge}[l : T_1/l_2 : T_2' \Rrightarrow T_3](v) \rightarrow \mu', \rho, v$
2. By inversion on 1a
   (a) $\Delta = \Delta_x, l_1 : T_1, l_2 : T_2$ for some T_2
   (b) $T_1 = T_1{\downarrow}$
   (c) $T_1/T_2' \Rrightarrow T_3$
   (d) $\Delta_x, l_2 : T_2 \vdash v : T \dashv \Delta_x', l_2 : T_2'$
   (e) $\Delta' = \Delta_x', l_2 : T_3$
3. Either $v = o$ or $v = \mathsf{void}$
   (a) Assume $v = o$

i. By inversion of 2d,3a
   A. $\Delta_x = \Delta'_x, v : T$
   B. $T_2 = T'_2$
ii. Let $\Delta'' = \Delta', v : T$
iii. $\Delta'' \vdash v : T \dashv \Delta'$ – by 3aii,(ctx-b)
iv. $\mu, (\Delta_x, l_1 : T_1, l_2 : T_2), \rho$ **ok** – by 1b,2a
v. $\mu, (\Delta_x, l_2 : T_1, l_2 : T_2), \rho$ **ok** – by 3aiv, mem consistency lemma, and $\rho(l_1) = \rho(l_2)$ by construction
vi. $\mu', (\Delta_x, l_2 : T_3), \rho$ **ok** – 3av, 2bc,3aiB, mem consistency lemma
vii. $\mu', (\Delta'_x, v : T, l_2 : T_3), \rho$ **ok** – by 3avi,2aiA
viii. $\mu', (\Delta', v : T), \rho$ **ok** – by 3avii, 2e
ix. μ', Δ'', ρ **ok** – by 3aviii, 3aii

(b) Assume $v = \mathsf{void}$

i. By inversion of 2d,3a
   A. $\Delta_x = \Delta'_x$
   B. $T_2 = T'_2$
   C. $T = \mathsf{Void}$
ii. Let $\Delta'' = \Delta'$
iii. $\Delta'' \vdash \mathsf{void} : \mathsf{Void} \dashv \Delta'$ – by 3aii,(void)
iv. $\mu, (\Delta_x, l_1 : T_1, l_2 : T_2), \rho$ **ok** – by 1b,2a
v. $\mu, (\Delta_x, l_2 : T_1, l_2 : T_2), \rho$ **ok** – by 3biv, mem consistency lemma, and $\rho(l_1) = \rho(l_2)$ by construction
vi. $\mu', (\Delta_x, l_2 : T_3), \rho$ **ok** – 3bv, 2bc,3biB, mem consistency lemma
vii. $\mu', (\Delta'_x, l_2 : T_3), \rho$ **ok** – by 3bvi,2aiA
viii. $\mu', (\Delta'), \rho$ **ok** – by 3bvii, 2e
ix. μ', Δ'', ρ **ok** – by 3bviii, 3bii

4. q.e.d. – by 3aiii,3aix,3biii,3bix

Case (merge-congr).

1. By assumption
   (a) $\Delta \vdash \mathsf{merge}[l_1 : T_1 / l_2 : T'_2 \Rrightarrow T_3](e) : T \dashv \Delta'$
   (b) μ, Δ, ρ **ok**
   (c) $\mu, \rho, e \to \mu', \rho', e'$
   (d) $\mu, \rho, \mathsf{merge}[l_1 : T_1 / l_2 : T'_2 \Rrightarrow T_3](e) \to \mu', \rho', \mathsf{merge}[l_1 : T_1 / l_2 : T'_2 \Rrightarrow T_3](e')$
2. By inversion on 1a
   (a) $\Delta = \Delta_x, l_1 : T_1, l_2 : T_2$ for some T_2
   (b) $T_1 = T_1{\downarrow}$
   (c) $T_1 / T'_2 \Rrightarrow T_3$
   (d) $\Delta_x, l_2 : T_2 \vdash e : T \dashv \Delta'_x, l_2 : T'_2$
   (e) $\Delta' = \Delta'_x, l_2 : T_3$
3. $l_1 \notin FV(e)$ – by construction
4. $\Delta_x, l_2 : T_2, l_1 : T_1{\downarrow} \vdash e : T \dashv \Delta'_x, l_2 : T'_2, l_1 : T_1{\downarrow}$ – by 2d,3,weakening

5. $\Delta \vdash e : T \dashv \Delta'_x, l_2 : T'_2, l_1 : T_1\downarrow$ – by 2a-b,4
6. By induction on 1b,5, there exists Δ'' such that:
   (a) μ', Δ'', ρ' **ok**
   (b) $\Delta'' \vdash e' : T \dashv \Delta'_x, l_2 : T'_2, l_1 : T_1\downarrow$
7. $\Delta'' = \Delta_{2x}, l_1 : T_1\downarrow, l_2 : T''_2$ – by 6b, Context Variable Conservation
8. $\Delta_{2x}, l_2 : T''_2 \vdash e' : T \dashv \Delta'_x, l_2 : T'_2$ – by strengthening
9. $\Delta_{2x}, l_2 : T''_2, l_1 : T_1\downarrow \vdash \mathsf{merge}[l_1 : T_1\downarrow / l_2 : T'_2 \Rrightarrow T_3](e) : T \dashv \Delta'_x, l_2 : T_3$ – by 8,2b-c,(merge)
10. $\Delta'' \vdash \mathsf{merge}[l_1 : T_1\downarrow / l_2 : T'_2 \Rrightarrow T_3](e) : T \dashv \Delta'$ – by 7,9
11. q.e.d – by 11-12

□

Lemma 22 (CompatCoerce)**.**
If $T_1 \lesssim T_2$ then $coerce(x, T_1 \gg T_2)$ is defined.

Proof. By induction on the derivation $T_1 \lesssim T_2$. Either $T_1 <: T_2$, or $T_2 = \mathsf{Dyn}$, in which case $T_1 \Rrightarrow T_2$, and $coerce(x, T_1 \gg T_2)$ is defined by (Coerce). Otherwise, $T_1 = \mathsf{Dyn}$ and $T_2 = P\ C$ and $coerce(x, T_1 \gg T_2)$ is defined by (Coerce$_d$). □

Lemma 23 (CompatDyn)**.**
If $T \lesssim \mathsf{Dyn}$ then, $x : T \vdash x \Leftarrow \mathsf{Dyn} \rightsquigarrow e \dashv x : T$, for some e.

Proof. Assume $T \lesssim \mathsf{Dyn}$. Either $T = \mathsf{Dyn}$ or $T = P\ C$. In either case, $T \Rrightarrow \mathsf{Dyn}$. Therefore, $x : \mathsf{Dyn} \vdash x \Leftarrow \mathsf{Dyn} \rightsquigarrow e \dashv x : \mathsf{Dyn}$, for some e, by (ctx⇐). □

Lemma 24 (Preservation-translation)**.** *If $\Delta \vdash e \Leftrightarrow T \dashv \Delta'$, then $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e' \dashv \Delta'$ for some e'.*

Proof. With CompatCoerce and CompatDyn lemmas, this is trivially proved by induction on derivation of $\Delta \vdash e \Leftrightarrow T \dashv \Delta'$. For example, look at the case (src-letT⇒). We assume $\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Rightarrow T_2 \dashv \Delta'$, $\Delta \vdash e_1 \Leftarrow T_1 \dashv \Delta_1$, and $\Delta_1, x : T_1 \vdash e_2 \Rightarrow T_2 \dashv \Delta', x : T'_1$. By induction on the last two terms, we get $\Delta \vdash e_1 \Leftarrow T_1 \rightsquigarrow e'_1 \dashv \Delta_1$, and $\Delta_1, x : T_1 \vdash e_2 \Rightarrow T_2 \rightsquigarrow e'_2 \dashv \Delta', x : T'_1$, for some e'_1, e'_2. By (letT⇒), we get $\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Rightarrow T_2 \rightsquigarrow e' \dashv \Delta'$, for some e'. All cases proceed similarly.

□

Theorem 25 (Preservation-source)**.** *If $\Delta \vdash e \Leftrightarrow T \dashv \Delta'$, then $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e' \dashv \Delta'$ and $\Delta \vdash e' : T \dashv \Delta'$, for some e'.*

Proof. Assume $\Delta \vdash e \Leftrightarrow T \dashv \Delta'$. By preservation-translation, there exists e' such that $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e' \dashv \Delta'$. By translation lemma, $\Delta \vdash e' : T \dashv \Delta'$. □

Lemma 26 (Method translation)**.** *If M **ok in** C in the source language, then there exists M' such that $M \rightsquigarrow M'$, and M' **ok in** C in the internal language.*

Proof.

1. By assumption
   (a) $T_r\ m(\overline{T_x \gg T'_x\ x})\ [T_t \gg T'_t]\ \{\ \mathsf{return}\ e;\ \}$ **ok in** C_t
   (b) $T_r\ m(\overline{T_x \gg T'_x\ x})[T_t \gg T'_t]$ **ok in** C_t
   (c) $\overline{x : T_x}, \mathsf{this} : T_t \vdash e \Leftarrow T_r \dashv \mathsf{this} : T''_t, \overline{x : T''_x}$
   (d) $T''_t \lesssim T'_t$

(e) $\overline{T''_x \lesssim: T'_x}$

2. There exists e' such that – by 1c,source preservation
   (a) $\mathsf{this} : T_t, \overline{x : T} \vdash e \Leftarrow T_r \rightsquigarrow e' \dashv \mathsf{this} : T''_t, \overline{x : T''_x}$
   (b) $\mathsf{this} : T_t, \overline{x : T} \vdash e' : T_r \dashv \mathsf{this} : T''_t, \overline{x : T''_x}$
3. By (CompatCoerce) – 1d-e. Note that we are using the already defined $\lesssim:$ relation on classes of the source language to reason about internal language expressions. This is reasonable since we are not translating class names, nor the class hierarchy at all
   (a) $coerce(\mathsf{this}, T''_t \gg T'_t)$ is defined
   (b) $\overline{coerce(x, T'' \gg T')}$ is defined
4. Let $e'' = \mathsf{let}\ ret = e'\ \mathsf{in}\ coerce(\mathsf{this}, T''_t \gg T'_t); \overline{coerce(x, T'' \gg T')}; ret$ – by 2-3
5. $\mathsf{this} : T''_t, \overline{x : T''_x}, ret : T_r \vdash coerce(\mathsf{this}, T''_t \gg T'_t) : \mathsf{Void} \dashv \mathsf{this} : T'_t, \overline{x : T''_x}, ret : T_r$ – by 2a, Coercion lemma
6. $\mathsf{this} : T'_t, \overline{x : T''_x}, ret : T_r \vdash \overline{coerce(x, T'' \gg T'')}; : \mathsf{Void} \dashv \mathsf{this} : T'_t, \overline{x : T'_x}, ret : T_r$ – by 2b, Coercion lemma
7. $\mathsf{this} : T'_t, \overline{x : T'_x}, ret : T_r \vdash ret : T_r \dashv \mathsf{this} : T'_t, \overline{x : T'_x}$ – by (ctx-binder)
8. $\mathsf{this} : T_t, \overline{x : T} \vdash e'' : T_r \dashv \mathsf{this} : T'_t, \overline{x : T'_x}$ – by 4-7, (let)
9. M' **ok in** C in internal language – by 1a, 8

□

Theorem 27. *If $PG : T$ **ok** in the source language, then there exists PG' such that $PG : T \rightsquigarrow PG' : T$, and $PG' : T$ **ok** in the internal language.*

Proof. Follows directly from method translation and preservation-source lemmas. □

References

Ronald Garcia, Roger Wolff, Éric Tanter, and Jonathan Aldrich. Featherweight typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University, July 2010.

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 and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

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.