Gradual Typestate

Roger Wolff[1], Ronald Garcia[1†], Éric Tanter[2‡], and Jonathan Aldrich[1§]

[1] School of Computer Science – Carnegie Mellon University
first.last@cs.cmu.edu
[2] PLEIAD Laboratory
Computer Science Department (DCC) – University of Chile
etanter@dcc.uchile.cl

Abstract. Typestate reflects how the legal operations on imperative objects can change at runtime as their internal state changes. A typestate checker can statically ensure, for instance, that an object method is only called when the object is in a state for which the operation is well-defined. Prior work has shown how modular typestate checking can be achieved thanks to access permissions and state guarantees. However, static typestate checking is still too rigid for some applications.
This paper formalizes a nominal object-oriented language with mutable state that integrates typestate change and typestate checking as primitive concepts. In addition to augmenting the types of object references with access permissions and state guarantees, the language extends the notion of *gradual typing* to account for typestate: gradual typestate checking seamlessly combines static and dynamic checking by automatically inserting runtime checks into programs. A novel flow-sensitive permission-based type system allows programmers to write safe code even when the static type checker can only partly verify it.

1 Introduction

This paper investigates an approach to increasing the expressiveness and flexibility of object-oriented languages as a means to improve the reliability of software. By introducing *typestate* directly into the language and extending its type system with support for *gradual typing*, useful abstractions can be implemented directly, stronger program properties can be enforced statically, and when necessary dynamic checks can be introduced seamlessly.

An object's type specifies the methods that can be called on it. In most programming languages, this type is constant throughout the object's lifetime, but in practice, the methods that it makes sense to call on an object change as its

[†] Supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project.
[‡] Partially funded by FONDECYT project 1110051.
[§] Support from grants NSF #CCF-0811592, DARPA #HR00110710019, ARO #DAAD19-02-1-0389 to CyLab, and CMU|Portugal Aeminium #CMU-PT/SE/0038/2008

runtime state changes (e.g., an open file cannot be opened again). These constraints typically lie outside the reach of standard type systems, and unintended uses of objects result, at best, in runtime exceptions.

More broadly, types generally denote properties that hold without change, and in mainstream type systems, they fail to account for how changes to mutable state can affect the properties of an object. To address this shortcoming, Strom and Yemini [26] introduced the notion of *typestate* as an extension of the traditional notion of type. Typestate reflects how the legal operations on imperative objects can change at runtime as their internal state changes.

The seminal work on typestate [26] focused primarily on whether variables were properly initialized, and presented a static *typestate checker*. A typestate checker must account for the flow of data and control in a program to ensure that objects are used in accordance with their state at any given point in a computation. Since that original work, typestate has been used to codify and check more sophisticated state-dependent properties of object-oriented programs. It has been used, for instance, to verify object invariants on .NET [10], to verify that Java programs adhere to object protocols [12, 5, 7], and to check that groups of objects collaborate with each other according to an interaction specification [20].

Most imperative languages cannot express typestates directly: rather, typestates are encoded through a disciplined use of member variables. For instance, consider a typical object-oriented file abstraction. A closed file may have a `null` value in its file descriptor field. Accordingly, the `close` method of the file object first checks if the file descriptor is `null`, in which case it throws an exception to signal that the file is already closed. Such typestate encodings hinder program comprehension and correctness. Comprehension is hampered because the protocols underlying the typestate properties, which reflect a programmer's intent, are at best described in the documentation of the code. Also, typestate encodings cannot guarantee by construction that a program does not perform illegal operations. Checking typestate encodings can be done through a whole-program analysis (e.g. [12]), or with a modular checker based on additional program annotations (e.g. [4]). In either case, the lack of integration with the programming language hinders adoption by programmers.

To overcome the shortcomings of typestate encodings, a typestate-oriented programming (TSOP) language directly supports expressing them [2]. For instance, in a class-based language that supports dynamically changing an object's class (like Smalltalk and its `become` statement), typestates can be represented as classes and be dynamically updated: objects can have typestate-dependent interfaces, behaviors, and representations. Protocol violations in a dynamically-typed TSOP language however result in "method not found" errors. To avoid such errors, it is crucial to regain the guarantees provided by static type checking. Static typestate checking is challenging, especially in the presence of aliasing. Some approaches sacrifice modularity and rely on whole program analyses [12, 20, 7]; others retain modularity at the expense of sophisticated type systems, typically based on linear logic [27] and requiring many annotations.

One kind of annotations is *access permissions*, which specify certain aliasing patterns [8, 10, 4]. Unfortunately, these systems cannot always verify safe code, due to the conservative assumptions they must make. Advanced techniques like fractional permissions [8] increase the expressiveness of a type system, within limits, but increase its complexity.

Many practical languages already provide a simple feature for overcoming the limitations of their type systems: dynamic coercions. Although these coercions (a.k.a. casts) may fail at runtime, they are often necessary in specific scenarios where the static machinery is insufficient. Runtime assertions related to typestates are not supported by any modular approach we know of; one primary objective of this work is to support them.

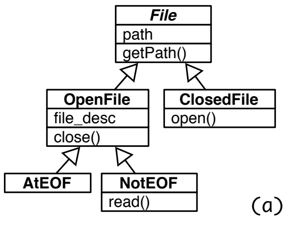
In addition, once dynamic coercions on typestates are available, they can be used to ease the transition from dynamically- to statically-typed code. We therefore extend gradual typing [24, 25] to account for typestates: we make typestate annotations optional, check as much as possible statically, and automatically insert runtime checks into programs where needed. This allows programmers to gradually annotate their code and get progressively more support from the type checker, while still being able to safely run a partially-annotated program.

The primary contribution of this work is Gradual Featherweight Typestate (GFT), a core calculus for typestate-oriented programming inspired by Featherweight Java [17], which supports dynamic permission checking and gradual typing. The proposed language addresses the most important issues of current typestate checkers. GFT directly integrates typestate as a first-class language concept. Its analysis is modular and safe without imposing complex notions like fractional permissions onto programmers. It also supports recovery of precise typing using dynamically-checked assertions and supports the gradual addition of type annotations to a program.

Section 2 introduces the key elements of typestate-oriented programming with access permissions and state guarantees. Section 3 describes the static subset of GFT, and Section 4 presents the extensions for dynamic permission checking and gradual typing. The semantics are presented using a type-safe internal language (Section 5) to which GFT translates (Section 6). The soundness proof is available in a companion technical report [28]. Section 7 concludes. A translator for the source language, type checker for the internal language, and executable runtime semantics are available at:
http://www.cs.cmu.edu/~rxg/gft.html.

2 Typestate-Oriented Programming

In order to avoid conditionals on flag fields or other indirect mechanisms like the State pattern [13], typestate-oriented programming proposes to extend object-oriented programming with an explicit notion of *state* (from here on we use state to mean typestate). In TSOP, objects are modeled not just in terms of classes, but in terms of changing states. Each state may have its own representation and methods, which may transition the object to new states.



(b)

	change state?	
	owner	**others**
full	yes	no
shared	yes	yes
pure	no	yes

Fig. 1. (a) Hierarchy of files states. (b) Access permissions.

To illustrate this concept in practice, consider a familiar example. A file object has methods such as `open`, `close` and `read`. However, these methods cannot be called at just any time. A file can only be `read` after it has been `open`ed; even if open, if we have reached the end-of-file, then `read`ing is not available anymore; an open file cannot be opened again, etc. Figure 1a depicts a model of the file example in terms of states, using distinct classes in a subclass hierarchy to represent them. `File` is an abstract state; a file object is either in the `OpenFile` or `ClosedFile` state. Note that the `path` field is present in both states, but that the `file_desc` field, which refers to the low-level operating system resource, is only present in the `OpenFile` state. Any `OpenFile` can be closed; however, it is only possible to read from an open file if the end-of-file has not been reached. Therefore, the `OpenFile` state has two refining substates, `AtEOF` and `NotEOF`.

State change. A TSOP language supports a state change operation, denoted here by `<-`. For instance, the `close` method in `OpenFile` can be defined as:

```
void close() { this <- ClosedFile(this.path); }
```

The expression form `e <- S(...)` transitions the object described by `e` into the state `S`; the arguments are used to initialize the fields of the object. In other words, `<-` behaves like a constructor, but updates the object in-place.

Declaring state changes. A statically-typed TSOP language must track state changes in order to reject programs that invoke methods on objects in inappropriate states. Consider the following:

```
OpenFile f = ...; f.close(); f.close();
```

The type of `f` before the first call to `close` is `OpenFile`. However, the second call to `close` should be rejected by a typechecker. One way to do so is to analyze the body of the `close` method to deduce that it updates the state of its argument to `ClosedFile`. However, this approach sacrifices modularity. Therefore, a method's signature should specify the output state of its arguments as well as that of its receiver. The calculi in this paper specify the state changes of methods by annotating each argument with its input and output state, separated by the `>>` symbol. The input and output states of the receiver object are placed in square brackets after the normal argument list, e.g.:

```
void close() [OpenFile >> ClosedFile] {...}
```

Access permissions. In a language with aliasing, tracking state changes is a subtle process. For instance, consider the following (where `F`, `OF` and `CF` are abbreviations for `File`, `OpenFile` and `ClosedFile`, respectively):

```
void m(OF >> CF f,OF >> OF g) {f.close(); print(g.file_desc.pos);}
```

Because of possible aliasing, `f` and `g` may refer to the same object. In that case, the method body of `m` must not be well-typed, as `g` may refer to a closed file by the time it needs to access its (potentially non-existent) `file_desc` field.

To track state changes in the presence of aliasing, Bierhoff and Aldrich have proposed *access permissions* [4, 5]. An access permission specifies whether a given reference to an object can be used to change its state or not, as well as the access permissions that other aliases to the same object might have. In this work we consider three kinds of access permissions (Figure 1b): full, shared and pure. We say a reference has *write access* if it has the ability to change the state of an object. full and shared have write access, where full implies *exclusive* write access.

One fix for the `m` method is to require that `f` and `g` have exclusive write access to an `OF` in order to ensure that they are not aliases, and therefore that `f.close()` cannot affect `g`'s referent.

```
void m(full OF >> full CF f, full OF >> full OF g){ ... }
```

State guarantees. Requiring `g` to have exclusive write access seems like overkill here. Only a `pure` permission is required to read the field `file_desc`. But we must still ensure that the two parameters are not aliases.

For more flexible reasoning in the presence of aliasing, access permissions are augmented with *state guarantees* (proposed by Bierhoff and Aldrich [4] but formalized and proven sound for the first time here). A state guarantee puts an upper bound on the state change that may be performed by a reference with write access: it can only transition an object to some subclass of the state guarantee. A type specification then has the form `k(D) C` where `k` is the access permission, `D` is the state guarantee, and `C` is the current state of the object. A *permission*, `k(D)`, is the access permission coupled with the state guarantee[3].

Consider:

```
full(Object) NotEOF x = new NotEOF(...);
pure(OF) OF y = x;
x.read();
print(y.file_desc.pos);
```

[3] When it is clear from the context, we sometimes say 'permission' when we really mean 'access permission'.

While `x.read()` may change the state of the file by transitioning it to `AtEOF`, it cannot invalidate the open file assumption held by `y`.

State guarantees improve modular reasoning about typestates substantially. For instance, they recover the ability to express something similar to an ordinary object-oriented type: `shared(C) C` allows an object to be updated but guarantees that it will always obey the interface `C`. Also, it turns out that we can use state guarantees to express an alternative solution to the previous example: restrict `g` to the `pure` access permission it requires, but add a state guarantee of `OF` to ensure that no other reference can transition the object to `ClosedFile`:

```
void m(full(F)  OF >> full(F)  CF f,
       pure(OF) OF >> pure(OF) OF g){ ... }
```

In this case, we can still statically enforce that `f` and `g` are not aliases by carefully choosing exactly how references to objects can be created. In this way, we can allow the programmer more flexibility than always demanding exclusive access to objects.

Permission flows. Permissions are split between all aliases and carefully restricted to ensure safety. This includes aliases in local variables, as well as in object fields. Consider the following snippet:

```
class FileContainer{ shared(OF) OF file; }

full(Object) OF x = new OF(...);
pure(OF) OF y = x;
full(Object) FileContainer z = new FileContainer(x);
```

After construction of the `OF`, the reference `x` has no aliases, so it is safe to give it full permission with an unrestricted update capability (`Object` state guarantee). Then, a local alias `y` is created, capturing a `pure` permission with `OF` guarantee. After this point, any state change done through `x` must respect this guarantee. Therefore, the permission of `x` must be downgraded to `full(OF)`. Finally, a container object is created, passing `x` as argument to the constructor. The field of `z` captures a `shared(OF)` permission. The permission of `x` is downgraded again, this time to `shared(OF)`. At this point, there are three aliases to the same file object: `x` and `z.file` both hold a `shared(OF)` permission, and `y` holds a `pure(OF)`. All aliases must be consistent, in that a state update through one alias must not break the invariants of other references.

Temporarily holding permissions In general, as the program executes, permissions to variables get split and are strictly weakened. There are many ways to refine the static type system in order to increase expressiveness, such as parametric polymorphism, fractional permissions and borrowing [8, 9]. Here we consider one such refinement: a mechanism that can temporarily hold some permissions to a reference while a sub-computation is performed. Consider the following:

```
void printPath(pure(F) F >> pure(F) F);
```

```
full(Object) OF x = new OF(...);
printPath(x);
x.close();
```

This program is ill-typed due to the downgrading of permissions. In order to invoke `printPath`, the permission to `x` is downgraded from `full(Object) OF` to `pure(F) F`. Therefore, `close`, which requires a `full(F) OF`, cannot be called, although the call to `close` is safe: `printPath` requires a read-only alias to its argument, and there are no other writeable aliases to `x`. This is an unfortunate limitation due to the conservative nature of the type system.

A workaround is to introduce a temporary alias to `x` with only a `pure` permission, and use that alias to invoke `printPath`. This is however cumbersome and does not allow for permissions to be merged back later on. In order to properly support this pattern, we introduce a novel expression, `hold`, which reserves a permission to a variable for use within a lexical scope, and then *merges* that permission with that of the variable at the end of the scope. For instance:

```
full(Object) OF x = new OF(...);
hold[x:full(F) OF] { printPath(x); }
x.close();
```

Dynamic asserts. As sophisticated as the type system might be (supporting hold, borrowing, etc.), it is still necessarily conservative and therefore loses precision. Dynamic checks, like runtime casts, are often useful to recover such precision. For instance, consider the following extension of the `FileContainer` snippet seen previously in which both `y` and `z` are updated to release their aliases to `x`.

```
...
y = new OF(...);
z <- Object();
assert<full(F) OF> x;
x.close();
```

Assuming `close` requires a `full(F)` permission to its receiver, the type system is unable to determine that `x` can be closed, even though it is safe to do so (`x` is once again the sole reference to the object). A *dynamic assert* allows this permission to be recovered. Like casts, dynamic asserts may fail at runtime.

Gradual typing. A statically typed TSOP program requires more annotations than a comparable OO program. This may be prohibitively burdensome for a programmer, especially during the initial stages of development. For this reason, we develop a gradually typed calculus that supports a dynamic type `Dyn`. Precise type annotations can then be omitted from an early draft of a program as in the following code:

```
Dyn f = ...; f.read();
```

A runtime check will verify that `f` refers to an object that has a `read` method[4]. Assume that `read` is annotated with a receiver type `full(OF) NotEOF`. In this case, we must ensure that we have an adequate permission to the receiver. Thus, a further runtime check will verify that `f` refers to an object that is currently in the `NotEOF` state, that no aliases have write access, and that all aliases have a state guarantee that is a superstate of `OF`. The last two conditions ensure that invariants of aliases to `f` cannot be broken. Gradual typing thus enables dynamically and statically-typed parts of a program to coexist without compromising safety.

Putting it all together. Listing 1 exhibits the above capabilities in a small logging example that generalizes to other shared resources[5]. The `OpenFileLogger` (`OFL`) state holds a reference to a file object (`OF`) and presents a `log` method for logging messages to it. When logging is complete, the `close` method acquires all permissions to the file by swapping in a sentinel value (with :=:, explained in the next section), closes the file, and transitions the logger to the `FileLogger` (`FL`) state, which has no file handle. The client code declares and uses two logging interfaces, `staticLog` and `dynamicLog`. They are somewhat contrived, but are meant to represent APIs that utilize a file logger but do not store it. After creating `logger` (line 17), the `file` reference no longer has enough permission to close the file, so calls to `logger.log` are safe. Line 19 `hold`s a shared permission to `logger` during a dynamically-typed method call. By line 22, `logger` only has shared permission, though no other aliases exist. After asserting back full permission, `logger` can close the file log.

```
1   class FileLogger { /* Logging−related Data and Methods */ }
2   class OpenFileLogger : FileLogger {
3       full(OF) OF file;
4
5       Void log(string s)[shared(OFL) OFL >> shared(OFL) OFL] {...}
6       Void close()[full(FL) OFL >> full(FL) FL] {
7           full(OF) fileT = (this.file :=: new File(”/dev/null”));
8           fileT.close();
9           this <− FileLogger();
10      }
11  }
12  // Client Code
13  Void staticLog(shared(OFL) logger >> shared(OFL) logger) { logger.log(”in staticLog”); }
14  Dyn dynamicLog(Dyn logger) { logger.log(”in dynamicLog”); }
15
16  full(OF) OF file = new OF(...);
17  full(OFL) OFL logger = new OFL(file);
18
19  hold[logger:shared(OFL) OFL]{ dynamicLog(logger); }
20  staticLog(logger);
21
22  assert<full(FL) OFL>(logger);
23  logger.close();
```

Listing 1. Sample Typestate-Oriented code.

[4] Note that `Dyn` is different from `Object`; if `f` had type `Object` the typechecker would check for a method read in `Object`, and would raise an error.

[5] A practical language would provide means to abbreviate our type annotations.

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

Fig. 2. Gradual Featherweight Typestate: Syntax (static subset)

3 Static Featherweight Typestate

We present a formal model for a language with integrated support for gradual typestate. The language is inspired by Featherweight Java (FJ) [17], so we call it Gradual Featherweight Typestate (GFT). Gradually-typed languages are typically presented as two distinct languages: a fully static language and its gradual extension. We only describe the gradual language; this section focuses on its static aspects. Sections 4 and beyond present the extensions for gradual typing. Garcia et al. [14] formalizes a fully static variant of GFT, called Featherweight Typestate. The static subset of the language is novel in its own right: it is the first formalization of a nominal TSOP language, with support for representing typestates as classes, modular typestate-checking and state guarantees, and an algorithmic flow-sensitive type system specification.

3.1 Syntax

Figure 2 presents GFT's syntax. Smallcaps (e.g. $\textsc{FieldNames}$) 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 (including this) and method, field, and class names (including Object). A GFT program PG is a list of class declarations $\overline{CL}$ paired with an expression e. Class definitions are standard, except that a GFT class does not have an explicit constructor: instead, it has an implicit constructor that assigns an initial value to each field. Fields F and methods M are standard. Each method parameter is annotated with its input and output types, and the method itself carries an annotation (in square brackets) for the receiver object. We use helper functions like *fields*, *method*, etc., whose definitions are omitted for brevity.

Types in GFT extend the Java notion of class names as types. As explained in Section 2, the type of a GFT object reference has two components, its permission

$$\boxed{k \Rrightarrow k/k} \quad \text{Access Permission Splitting}$$

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

$$\boxed{P \leftrightarrow P} \quad \text{Compatible Permissions}$$

$$\frac{E <: D}{k(E) \leftrightarrow \mathsf{pure}(D)} \qquad \frac{P_1 \leftrightarrow P_2}{P_2 \leftrightarrow P_1} \qquad \frac{}{\mathsf{shared}(D) \leftrightarrow \mathsf{shared}(D)} \qquad \frac{\forall i, j : i \neq j . P_i \leftrightarrow P_j}{\overline{P} \text{ compatible}}$$

$$\boxed{P \prec: P} \quad \text{Subpermission}$$

$$\frac{k_1 \Rrightarrow k_2}{k_1(D) \prec: k_2(D)} \qquad \frac{E <: D}{\mathsf{pure}(E) \prec: \mathsf{pure}(D)} \qquad \frac{D <: E}{\mathsf{full}(E) \prec: \mathsf{full}(D)} \qquad \frac{P_1 \prec: P_2 \quad P_2 \prec: P_3}{P_1 \prec: P_3}$$

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

$$\frac{k_1(D_1) \prec: k_2(D_2) \quad k_1 \Rrightarrow k_2/k_3 \qquad D_3 = D_1 \mathbin{\underset{..}{\wedge}} D_2}{k_1(D_1) \Rrightarrow k_2(D_2)/k_3(D_3)}$$

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

$$\frac{P_1 \Rrightarrow P_2/P_3 \quad C_1 <: C_2}{P_1\ C_1 \Rrightarrow P_2\ C_2/P_3\ C_1} \qquad \frac{}{\mathsf{Void} \Rrightarrow \mathsf{Void}/\mathsf{Void}}$$

$$\boxed{T <: T} \quad \text{Subtyping}$$

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

$$\boxed{T \Downarrow T} \quad \text{Max. Residual}$$

$$\frac{T_1 \Rrightarrow T_1/T_2}{T_1 \Downarrow T_2}$$

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

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

$$\boxed{\text{A type } T \equiv k(D)\ C \text{ is well-formed iff } C <: D}$$

Fig. 3. Permission and Type Management Relations

and its class (or *state*). The permission can be broken down further into its access permission k and state guarantee D. We write these *object reference types* in the form $k(D)\ C$. The Void type classifies expressions executed purely for their effects. No source-level values have the Void type.

To simplify the description of the type system, expressions in GFT are restricted to A-normal form [22], so let expressions explicitly sequence all complex operations (we write $e_1; e_2$ as shorthand). An optional type ascription provides fine-grained control over how permissions are distributed to the bound variable (Section 3.3).

Apart from method invocation, field reference and object creation (all standard), GFT includes the update operation $x_0 \leftarrow C(\overline{x_1})$ 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. Also non-standard is the swapping assignment $x_0.f :=: x_1$. It assigns the value of x_1 to the field f and returns the old value as its result. The need for this expression is detailed in Section 3.3.

3.2 Managing Permissions

Before we present typing judgments for GFT, we must explain how permissions are treated. Permissions to an object are a resource that is split among the variables and fields that reference it. Figure 3 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 $P_1 \prec: P_2$ says that P_1 is a *subpermission* of P_2, which means that a reference with P_1 permission may be used wherever an object reference with P_2 permission 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 $k_1(D_1) \prec: k_2(D_2)$, splitting is safe. The question then is to determine the proper residual permission $k_3(D_3)$. The k_3 residual 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 $D_1 \mathbin{\underset{..}{\wedge}} D_2$[6].

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. 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 $\mathsf{shared}(D)\ C \Downarrow \mathsf{shared}(D)\ C$.

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. For instance, $\mathsf{shared}(C) \leftrightarrow \mathsf{shared}(C)$, and $\mathsf{full}(C) \leftrightarrow \mathsf{pure}(\mathsf{Object})$. This notion is used to define the relation $\overline{P}$ *compatible*: that the outstanding permissions $\overline{P}$ of references to a particular object can all coexist.

Finally, we defer the discussion of *type demotion* to the end of Section 3.3.

[6] $k_1(D_1) \prec: k_2(D_2)$ implies that D_1 and D_2 are related by subclassing

3.3 Static Semantics

Armed with the permission management relations, we now discuss the salient features of the static semantics of GFT: flow-sensitive, deterministic typing through bidirectional type checking.

Flow-sensitive typing As with FJ, the GFT type system relies upon type contexts Δ. Whereas Γ is the standard metavariable for type contexts, we use a different metavariable Δ to emphasize that the typing contexts are not merely lexical: they are *linear* [16]. In GFT's type system, the types of identifiers are flow-sensitive in the sense that they 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.

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 type assumptions.

Deterministic type checking The type system specification must be elaborated 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 derivations, with each derivation representing a different split of permissions between this particular variable reference and later references to x. Nondeterminism is incompatible with dynamic permission assertions[7]: a system could succeed sometimes and fail other times if permissions could flow more than one way for the same code.

Rather than requiring type annotations for all variable references, we use bidirectional typing [21] to define a deterministic type system in which annotations are used only to tune how permissions are split.

The type system is 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 an input to the judgment, and the only output is the context Δ_2.

[7] Determinism is not important for the fully static case: Featherweight Typestate uses non-deterministic typing rules [14].

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

$$(\hat{e}{\Leftarrow})\ \frac{\Delta \vdash \hat{e} \Rightarrow T_2 \dashv \Delta_1 \qquad T_2 <: T_1}{\Delta \vdash \hat{e} \Leftarrow T_1 \dashv \Delta_1}$$

$$(\text{let}{\Leftrightarrow})\ \frac{\Delta \vdash e_1 \Rightarrow T_1 \dashv \Delta_1 \qquad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \dashv \Delta_2, x : T}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \dashv \Delta_2}$$

$$(\text{letT}{\Leftrightarrow})\ \frac{\Delta \vdash e_1 \Leftarrow T_1 \dashv \Delta_1 \qquad \Delta_1, x : T_1 \vdash e_2 \Leftrightarrow T_2 \dashv \Delta_2, x : T}{\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 \Leftrightarrow T_2 \dashv \Delta_2}$$

$$(\text{update}{\Rightarrow})\ \frac{\begin{array}{cc} \mathit{fields}(C_2) = \overline{T_2\ f} & \Delta \vdash \overline{x_2 \Leftarrow T_2} \dashv \Delta', x_1 : k(D)\ C \\ k \in \{\,\mathsf{full}, \mathsf{shared}\,\} & C_2 <: D \end{array}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \mathsf{Void} \dashv \Delta'{\downarrow}, x_1 : k(D)\ C_2}$$

$$(\text{ref}{\Rightarrow})\ \frac{T_2\ f \in \mathit{fields}(C_1) \qquad T_2 \Downarrow T_2'}{\Delta, x : P_1\ C_1 \vdash x.f \Rightarrow T_2' \dashv \Delta, x : P_1\ C_1}$$

$$(\text{swap}{\Rightarrow})\ \frac{\begin{array}{c} T_2\ f \in \mathit{fields}(C_1) \\ \Delta, x_1 : P_1\ C_1 \vdash x_2 \Leftarrow T_2 \dashv \Delta' \end{array}}{\Delta, x_1 : P_1\ C_1 \vdash x_1.f :=: x_2 \Rightarrow T_2 \dashv \Delta'}$$

Fig. 4. Select Expression Typing Rules

Typing rules Figure 4 presents some of the typing rules for GFT expressions[8]. A variable reference is typed differently depending on whether its type is synthesized or checked. The synthesis rule (ctx⇒) yields maximal permissions to the referenced object. Its output context associates the maximum residual permissions to the variable. In contrast, the checking rule (ctx⇐) just ensures that the desired type can be split from the starting type, and leaves the corresponding residual in the output context.

Each of the typing rules for let represents both a checking and synthesis rule. Replacing the ⇔ with ⇐ or ⇒ gives the checking or synthesis rule, respectively. The (let⇔) and (letT⇔) rules differ only in how the bound expression is typed. 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, e_1's type 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.

[8] As with FJ, well-typing is extended to apply to whole programs. The details are covered in the technical report [28].

Type checking is 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 (see Section 6).

A variable reference can only perform an update if it has write access. The arguments to the constructor—which can include the reference being updated—are type checked at the target class's field types. We use the shorthand notation $\Delta \vdash \overline{x \Leftarrow T} \dashv \Delta'$ to stand for iteratively checking the arguments:

$$\Delta \;=\; \Delta_0 \vdash x_0 \Leftarrow T_0 \dashv \Delta_1; \quad \ldots \quad ; \Delta_n \vdash x_n \Leftarrow T_n \dashv \Delta_{n+1} \;=\; \Delta'.$$

The target class of the update must respect the updated reference's state guarantee, taking into account any uses of that reference in the construction of the new object. The update operation is performed solely for its effect on the heap, so the type of the overall expression is Void. The output type of the updated object reflects its new class.

Type demotion. 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 $T{\downarrow}$ (Figure 3) 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.

The synthesis rule for the update operation in Figure 4 makes use of type demotion: type assumptions from the input context are demoted in the output context to ensure that any aliases to the updated object retain a conservative approximation of the object's current class[9].

Note that type demotion does not imply any runtime overhead: it is a purely static process. Furthermore, types of class fields have the restriction that 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. Also, as a result only local variable types need ever be demoted.

Field Operations As was mentioned in Section 3.1, two operations operate directly on the an object field: field reference and swapping assignment. Their type synthesis rules appear in Figure 4. Field reference operations do not relinquish any of the permissions held by a field, so the result type is determined by taking the maximal residual T_2' of the field type T_2. This operation does not affect the permissions of the object reference used to access the field.

[9] The language could retains more precise types by demoting only objects with types related to the updated object. For simplicity of presentation, we demote uniformly.

Swap operations cause an object to relinquish all permissions to a field and replace it with a new reference. The swap expression has two purposes. The first is to reassign a field value in the heap. The second is to return the old field value as the result of the expression. If a field has shared or pure permissions to an object, then field reference can yield the same amount of permission; however, if a field has full permission to an object, only swapping can yield a full permission.

3.4 Holding Permissions

The static fragment of GFT described above captures the essence of a TSOP language, but the design can be usefully extended. For instance, Section 2 introduced the hold expression $\mathsf{hold}[x : T](e)$, which 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.

$$e ::= \ldots \mid \mathsf{hold}[x : T](e)$$

The hold expression is a useful but orthogonal addition to the type system. A practical TSOP language would build hold's capabilities directly into the method invocation mechanism so as to preserve permissions wherever possible. For simplicity and exposition, we express holding as a distinct feature.

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

$$\frac{\begin{array}{c} P = P_1 \underset{\cdot\cdot}{\wedge} P_2 \\ C = C_1 \underset{\cdot\cdot}{\wedge} C_2 \end{array}}{P_1\ C_1/P_2\ C_2 \Rrightarrow P\ C} \qquad (\text{hold}{\Rightarrow})\ \frac{\begin{array}{c} T_1 \Rrightarrow T_2/T_3 \qquad T_2{\downarrow}\ /T_3' \Rrightarrow T_1' \\ \Delta, x : T_3 \vdash e \Rightarrow T \dashv \Delta', x : T_3' \end{array}}{\Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) \Rightarrow T \dashv \Delta', x : T_1'}$$

The typing rule for hold depends on a notion of *type merging* $T/T \Rrightarrow T$, which captures how two separate permissions to an object may be combined. Type merging is defined in terms of the $\underset{\cdot\cdot}{\wedge}$ and $\underset{\cdot\cdot}{\wedge}$ relations, where $\underset{\cdot\cdot}{\wedge}$ is the analogue of $\underset{\cdot\cdot}{\wedge}$ for subpermissions. For example, if we know that variable x has both type $\mathsf{full}(C)\ C$ and $\mathsf{pure}(\mathsf{Object})\ C$, then we can merge those types, and safely conclude that x can be typed at $\mathsf{full}(C)\ C$.

For space reasons, the rest of the technical development of hold (e.g. translation and dynamic semantics) is omitted, but the details can be found in the technical report [28].

4 Gradual Featherweight Typestate

The previous section presents the essence of GFT: an FJ-like calculus where classes model states, with an update operation to dynamically change the state of objects; types encode states as well as permissions and state guarantees, and the type system is both flow-sensitive and deterministic. Even if features like hold increase the expressiveness of the type system, there are still cases where it is necessary to resort to dynamic assertions about typestates. This section presents the support for such assertions, as well as gradual typing.

$\boxed{T \Rrightarrow T/T}$ Type Splitting $\boxed{T \lesssim: T}$ Consistent Subtyping

$$\frac{}{T \Rrightarrow \mathsf{Dyn}/T} \qquad \frac{T_1 <: T_2}{T_1 \lesssim: T_2} \quad \frac{T \neq \mathsf{Dyn}}{\mathsf{Dyn} \lesssim: T}$$

Fig. 5. Hybrid Permission Management Relations

$$(\mathrm{ctx}_d \Leftarrow)\frac{T \neq \mathsf{Dyn}}{\Delta, x : \mathsf{Dyn} \vdash x \Leftarrow T \dashv \Delta, x : \mathsf{Dyn}}$$

$$(\mathrm{update}_d \Rightarrow)\frac{\mathit{fields}(C_2) = \overline{T_2\ f} \qquad \Delta \vdash \overline{x_2 \Leftarrow T_2} \dashv \Delta', x_1 : \mathsf{Dyn}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \mathsf{Void} \dashv \Delta'\!\downarrow, x_1 : \mathsf{Dyn}}$$

Fig. 6. Select Dynamic Typing Rules

Type assertions GFT supports an assert expression for explicitly changing the type assumptions of a variable.

$$e ::= \cdots \mid \mathsf{assert}\langle T\rangle(x)$$

Its type synthesis rule is as follows.

$$(\mathrm{assert}{\Rightarrow})\frac{}{\Delta, x : T_1 \vdash \mathsf{assert}\langle T_2\rangle(x) \Rightarrow \mathsf{Void} \dashv \Delta, x : T_2}$$

Assert is like a cast, but instead of returning a value of the given type it changes the type of the target variable[10]. When $T_1 <: T_2$, the assert is statically safe; otherwise, a runtime check is required (see Section 6).

Gradual typing To support gradual typing, GFT provides a type for dynamically checked values.

$$T ::= \cdots \mid \mathsf{Dyn}$$

The type system treats the Dyn type with greater leniency: type checks on Dyn objects are deferred to runtime.

To account for these dynamic features, Figure 5 presents several adjustments to the type system from the last section. First, type splitting is extended to account for Dyn. In particular, any reference can split off a Dyn without affecting its original type or permissions. Type consistency is extended analogously.

Following Siek and Taha [25], consistent subtyping $T \lesssim: T$ extends subtyping to support using Dyn-typed values wherever another type is expected and vice-versa. Consistent subtyping explicitly states that $\mathsf{Dyn} \lesssim: T$, but also $T \lesssim: \mathsf{Dyn}$ because type splitting now forces $T <: \mathsf{Dyn}$. In accordance, the $(\hat{e} \Leftarrow)$ type checking rule from the last section now uses $\lesssim:$ in place of $<:$.

Figure 6 presents some extra typing rules that are needed to account for uses of Dyn-typed references. The $(\mathrm{ctx}_d \Leftarrow)$ rule says that a Dyn-typed variable can

[10] In fact, assertions are strictly more powerful than casts: casts can be implemented using assertions.

$$
\begin{array}{rcll}
o & \in & \textsc{ObjectRefs} & \\
l & \in & \textsc{IndirectRefs} & \\
s & ::= & x \mid l & \text{(simple exprs)} \\
b & ::= & x \mid l \mid o & \text{(bare expr)} \\
e & ::= & e_s \mid e_d & \text{(expressions)} \\
e_s & ::= & b \mid \mathsf{void} \mid s[T \Rrightarrow T/T] \mid \mathsf{new}\ C(\overline{s}) & \text{(static exprs)} \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{release}[T](s) \mid s.f \mid s.m(\overline{s}) & \\
 & \mid & s.f\ :=:\ s \mid s \leftarrow C(\overline{s}) \mid \mathsf{assert}\langle T \gg T\rangle(s) & \\
e_d & ::= & s._d f \mid s._d m(\overline{s}) \mid s.f\ :=:_d\ s & \text{(dynamic exprs)} \\
 & \mid & s \leftarrow_d C(\overline{s}) \mid \mathsf{assert_d}\langle T \gg T\rangle(s) & \\
\Delta & ::= & \overline{b : T} & \text{(type context)}
\end{array}
$$

Fig. 7. Internal Language Syntax

be checked at any type. Note that $x : \mathsf{Dyn}$ can be synthesized and checked at Dyn by the (ctx$\Rightarrow$) and (ctx$\Leftarrow$) rules respectively. The (update$_d$ $\Rightarrow$) rule accounts for updating a dynamically typed variable. The type system checks that the arguments to the constructor are suitable, but the checks on the target of the update are deferred to runtime (see Section 6).

5 Internal Language

GFT's semantics are defined by type-directed translation to GFTIL, an internal language that makes the details of dynamic permission management explicit.

5.1 Syntax

GFTIL is structured much like GFT but elaborates several concepts (Figure 7). First, the internal language introduces explicitly dynamic variants e_d of some operations from the source language. Static variants are ensured to be safe by the type system; dynamic variants require runtime checks. 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 runs. These type annotations play a role in both the type system and the dynamic semantics.

Finally, GFTIL adds several constructs that only occur at runtime. Object references and indirect references point to runtime objects[11]. GFTIL is also in A-normal form, though at runtime the arguments to expressions are generalized to simple expressions s: variable names or indirect references. 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 starting type, result type, and the residual type of the reference. The $\mathsf{release}[T](s)$ expression explicitly releases a reference and its permissions, after which it can no longer be used.

[11] Object references correspond to heap pointers; indirect references are an artifact that facilitates the type safety proof (see Section 5.4).

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

$$\text{(invoke}_d\text{)}\ \frac{}{\varDelta, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}} \vdash s_1 ._d m(\overline{s_2}) : \mathsf{Dyn} \dashv \varDelta{\downarrow}, s_1 : \mathsf{Dyn}, \overline{s_2 : \mathsf{Dyn}}}$$

$$\text{(update)}\ \frac{k \in \{\,\mathsf{full}, \mathsf{shared}\,\} \quad C_2 <: D \quad \mathit{fields}(C_2) = \overline{T\ f}}{\varDelta, s_1 : k(D)\ C_1, \overline{s_2 : T} \vdash s_1 \leftarrow C_2(\overline{s_2}) : \mathsf{Void} \dashv \varDelta{\downarrow}, s_1 : k(D)\ C_2}$$

$$\text{(update}_d\text{)}\ \frac{\mathit{fields}(C_2) = \overline{T\ f}}{\varDelta, s_1 : \mathsf{Dyn}, \overline{s_2 : T} \vdash s_1 \leftarrow_d C_2(\overline{s_2}) : \mathsf{Void} \dashv \varDelta{\downarrow}, s_1 : \mathsf{Dyn}}$$

$$\text{(assert)}\ \frac{T_1 \Rrightarrow T_2}{\varDelta, s : T_1 \vdash \mathsf{assert}\langle T_1 \gg T_2\rangle(s) : \mathsf{Void} \dashv \varDelta, s : T_2}$$

$$\text{(assert}_d\text{)}\ \frac{T_1 \not\Rrightarrow T_2}{\varDelta, s : T_1 \vdash \mathsf{assert_d}\langle T_1 \gg T_2\rangle(s) : \mathsf{Void} \dashv \varDelta, s : T_2}$$

Fig. 8. Select Internal Language Typing Rules

5.2 Static Semantics

Because of GFTIL's explicit form, its typing judgment $\varDelta \vdash e : T \dashv \varDelta$ does not need to be bidirectional. Furthermore, its typing rules use the same permission and type management relations as the source language. GFTIL's typing rules explicitly and strictly encode permission flow by checking the input context $\varDelta$ to force their arguments s to have *exactly* the type required. GFTIL's dynamic semantics uses this encoding to track permissions.

Figure 8 presents some of GFTIL's typing rules. For space reasons, we only present the rules for invoke, update and assert, together with their dynamically-typed variants. 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. The (update) rule almost mirrors GFT's update rule except that its argument types must exactly match the class field specifications. The (assert) rule is the safe subset of GFT's rule, though GFTIL's assert is explicitly annotated with its argument's source type. The dynamic variants of these expressions enforce very little statically: the (update$_d$) rule only checks that the arguments match the constructor, and the (assert$_d$) rule applies when the destination type cannot be split from the source type.

5.3 Dynamic Semantics

GFTIL's dynamic semantics, presented in Figure 9, requires several additional syntactic notions, defined below:

$$\begin{array}{rll}
C(\overline{o})\ \overline{P} \in & \textsc{Objects} & \\
v ::= & \mathsf{void} \mid o & \text{(values)} \\
\mu \in & \textsc{ObjectRefs} \rightharpoonup \textsc{Objects} & \text{(stores)} \\
\rho \in & \textsc{IndirectRefs} \rightharpoonup \textsc{Values} & \text{(environments)} \\
\mathbb{E} ::= & \square \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e & \text{(evaluation contexts)}
\end{array}$$

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

$$\text{(invoke)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \\ \mathit{method}(m, C) = T_r\ m(\overline{T_x \gg T'_x\ x})\ [T_t \gg T'_t]\ \{\ \mathsf{return}\ e;\ \}\end{array}}{\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e}$$

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

$$\text{(update)}\ \frac{\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \qquad \mathit{fields}(C) = \overline{T\ f} \qquad \mu' = \Big(\mu[\rho(l_1) \mapsto C'(\overline{\rho(l_2)})\ \overline{P}]\Big) - \overline{o : T}}{\mu, \rho, l_1 \leftarrow C'(\overline{l_2}) \rightarrow \mu', \rho, \mathsf{void}}$$

$$\text{(update}_d\text{)}\ \frac{\mu(\rho(l_1)) = C(\overline{o_f})\ \overline{P} \qquad D_g = \underset{\cdot\cdot}{\wedge}\ \{\ D \mid k(D) \in \overline{P}\ \} \qquad C' <: D_g}{\begin{array}{l}\mu, \rho, l_1 \leftarrow_d C'(\overline{l_2}) \rightarrow \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg \mathsf{shared}(D_g)\ C\rangle(l_1); \\ \qquad\qquad l_1 \leftarrow C'(\overline{l_2}); \\ \qquad\qquad \mathsf{assert}\langle \mathsf{shared}(D_g)\ C' \gg \mathsf{Dyn}\rangle(l_1)\end{array}}$$

$$\text{(assert)}\ \frac{\mu' = \mu - \rho(l) : T + \rho(l) : T'}{\mu, \rho, \mathsf{assert}\langle T \gg T'\rangle(l) \rightarrow \mu', \rho, \mathsf{void}} \qquad \text{(assert}_d\text{v)}\ \frac{\rho(l) = \mathsf{void}}{\mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg \mathsf{Void}\rangle(l) \rightarrow \mu, \rho, \mathsf{void}}$$

$$\text{(assert}_d\text{o)}\ \frac{\rho(l) = o \qquad \mu' = \mu - o : T + o : P'\ C' \qquad \mu'(o) = C(\overline{o_f})\ \overline{P} \qquad C <: C' \qquad \overline{P}\ \text{compatible}}{\mu, \rho, \mathsf{assert_d}\langle T \gg P'\ C'\rangle(l) \rightarrow \mu', \rho, \mathsf{void}}$$

Fig. 9. Select Internal Language Dynamic Semantics Rules

Expressions in the language evaluate to values, including void and object references o. Stores μ associate object references to objects. The novelty of GFTIL is that an object in the store $C(\overline{o})$ is annotated with the set of outstanding permissions for references to that object, $\overline{P}$.

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. 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. The dynamic semantics of GFTIL is defined as transitions between store/environment/expression triples[12].

Figure 9 presents some select dynamic semantics rules of GFTIL. Certain rules use two helper functions for tracking permissions in the heap, whose definitions are straightforward and as such omitted for brevity. Permission addition $+$ augments the permission set for a particular object in the heap. Conversely, permission subtraction $-$ removes a permission from the set of tracked permis-

[12] 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.

Helper Functions

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

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

$$\frac{\begin{array}{c} \mu(o) = C(\overline{o'})\ \overline{P} \qquad \left|\overline{o'}\right| = |\mathit{fields}(C)| \\ \mathit{objTypes}(\mu, \Delta, \rho, o) = \overline{k(E)\ D} \\ C <: \overline{D} \qquad \overline{k(E)} \text{ compatible} \\ \overline{k(E)} = \overline{P} \end{array}}{\mu, \Delta, \rho \vdash o\ \mathbf{ok}}$$

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

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

Fig. 10. Permission-Consistency Relations

sions for an object. The (invoke) rule is standard. 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 (assert) rule uses permission addition and subtraction to track permissions, and returns void. Rules for dynamic operators, like (invoke$_d$) and (update$_d$), dynamically assert the necessary permissions (using $\mathsf{assert_d}$), defer to the corresponding static operation, and then release the acquired permission (using assert). Finally, the (assert$_d$) rule confirms dynamically that its type assertion is safe.

5.4 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 10 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 are 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[13]. 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 the statement (and proof) of type safety.

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

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

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 $\Delta'' \vdash e' : T \dashv \Delta'$ and μ', Δ'', ρ' **ok** for some Δ''.*

6 Source to Target 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 11 presents some of these rules. We use the ^{T} superscript to disambiguate GFTIL expressions as needed. Several rules use the *coerce* partial function, which translates consistent subtyping assertions $T \lesssim T$ into variable assertions:

$$\begin{aligned} &coerce(x, T_1 \lesssim T_2) &&= \mathsf{assert}\langle T_1 \gg T_2\rangle(x) \quad \text{if} \quad T_1 <: T_2 \\ &coerce(x, \mathsf{Dyn} \lesssim T) &&= \mathsf{assert_d}\langle \mathsf{Dyn} \gg T\rangle(x) \quad \text{if} \quad T \neq \mathsf{Dyn} \end{aligned}$$

Most of the translations are straightforward, and follow similar patterns. For instance, the (update⇒) rule, which applies when the target of the update is statically typed, let-binds all of the arguments to the object constructor so as to extract the exact permissions that it needs before calling GFTIL's static update. The (update$_d$ ⇒) rule, in contrast, applies when the target of the update is dynamically typed. It translates to a dynamic update operation $\leftarrow_d$, but is otherwise the same. Operations on dynamically typed objects translate to dynamic operations. Other rules like (assert⇒) simply use the typing rule to ascertain the intended type annotations for the corresponding GFTIL expression.

As intended, the translation rules preserve well-typing:

Theorem 3 (Translation Soundness).
If $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta'$ then $\Delta \vdash e^T : T \dashv \Delta'$.

This theorem extends straightforwardly to whole programs.

[13] Dyn references may point to either Void or object references.

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

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

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

$$(\text{update}{\Rightarrow})\ \frac{\begin{array}{cc} \mathit{fields}(C_2) = \overline{T_2\ f} & \Delta \vdash \overline{x_2 \Leftarrow T_2 \leadsto e_2^T} \dashv \Delta', x_1 : k(D)\ C \\ k \in \{\mathsf{full}, \mathsf{shared}\} & C_2 <: D \end{array}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \mathsf{Void} \leadsto \overline{\mathsf{let}\ x'_2 = e_2^T}\ \mathsf{in}\ x_1 \leftarrow C_2(\overline{x'_2}) \dashv \Delta'{\downarrow}, x_1 : k(D)\ C_2}$$

$$(\text{update}_d \Rightarrow)\ \frac{\mathit{fields}(C_2) = \overline{T_2\ f} \qquad \Delta \vdash \overline{x_2 \Leftarrow T_2 \leadsto e_2^T} \dashv \Delta', x_1 : \mathsf{Dyn}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \mathsf{Void} \leadsto \overline{\mathsf{let}\ x'_2 = e_2^T}\ \mathsf{in}\ x_1 \leftarrow_d C_2(\overline{x'_2}) \dashv \Delta'{\downarrow}, x_1 : \mathsf{Dyn}}$$

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

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

Fig. 11. Select Translation Rules from GFT to GFTIL

7 Discussion

Related Work A lot of research has been done on typestates since they were first introduced by Strom and Yemini [26]. Most typestate analyses are whole-program analyses, which makes them very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [12]) rely on a global alias analysis and generally assume that the protocol implementation is correct and only verify client conformance. Naeem and Lhoták [20] developed an analysis for checking typestate properties over multiple interacting objects. These global analyses typically run on the complete code base, only once a system is fully implemented, and are time consuming.

Fugue [10] was the first modular typestate verification system for object-oriented software. It tracks objects as “not aliased” or “maybe aliased”; only “not aliased” objects can change state. Bierhoff and Aldrich [4] extended this approach by supporting more expressive method specifications based on linear logic [16]. They introduce the notion of access permissions in order to allow state changes even in the presence of aliasing. They also use fractions, first proposed by Boyland [8], to support patterns like borrowing and adoption [9]. The Plural tool supports modular typestate checking with access permissions for Java. It has been used in a number of practical studies [5]. Although Plural introduced state guarantees, this paper provides their first formalization.

Recent work on distributed session types [15] provides essentially the same expressiveness as Plural, but with protocols expressed in the structural setting of a process algebra instead of the setting of nominal typestates. It considers communication over distributed channels as well as object protocols, but does not allow aliasing for objects with protocols.

The above approaches do not address typestate-oriented programming, as they are not integrating typestates within the programming model, but rather overlay static typestate analysis on top of an existing language. TSOP has been recently proposed by Aldrich et al. [2]; its defining characteristic is supporting run-time changes to the representation of objects in the dynamic semantics and type system. The programming language Plaid[14] is the first language to integrate typestates in the core programming model. Saini et al. [23] recently developed the first core calculus for a TSOP language; their language is object-based and relies on structural types. Gradual Featherweight Typestate builds on this work but adapts it to a class-based, nominal approach with shared permissions and state guarantees for reasoning about typestate in the presence of aliasing. Earlier work related to TSOP includes the Fickle system [11], which can change the class of an object at runtime, but has limited ability to reason about the states of an object's fields.

This work also builds upon existing techniques for partial typing, like hybrid typing [18] and gradual typing [24, 25, 6]. Gradual Featherweight Typestate is a considerable advance in this sense, by showing how to gradually check flow-sensitive resources in a modular fashion. Recently, Bodden [7] presented a hybrid approach to typestate checking. A static typestate analysis is performed to avoid unnecessary instrumentation of programs for monitoring typestates at runtime. While the hybrid perspective is shared with this work, the proposed analysis is global.

Ahmed et al. [1] define a core functional programming language that supports *strong updates*, i.e. changing the type of an object in a reference cell. Similarly to our approach, it uses linear typing. They present two languages, L3, and extended L3. L3 allows aliasing, but only has exclusive access, through a capability: only one reference can read/write to an object. In contrast, full, shared and pure permissions allow for more varied aliasing patterns. Extended L3 allows recovering a capability, but the programmer must provide a proof that no other capabilities exist to the reference cell.

Future Work Gradual Featherweight Typestate is at the core of the Plaid language design project at CMU. We are integrating other access permissions from Bierhoff and Aldrich [4]. Most importantly, we are exploring ways to extend the power of the static type system in order to avoid resorting to dynamic asserts. An example of such an extension is permission borrowing [9], which, if specified in method signatures, avoids having to dynamically reassert permissions after "lending" them to a sub-computation. The language we present here already includes one such refinement, namely hold, used to hold some permissions to

[14] Under development at CMU: `http://plaid-lang.org`

a reference while a sub-computation is performed. Importantly, it remains an outstanding research question if the cost of dynamic permission checking can be amortized over the number of permission checks. As it now stands, enabling dynamic permission checking mandates a fully-instrumented runtime semantics to keep track of permissions. In Plaid, we intend to address this with reference counting. Standard optimization techniques like deferred increments [3] and update coalescing [19] will be applied. We believe these techniques will reduce reference count overhead to a small percentage of runtime, as it does for garbage collection, and will study this empirically in future. The formalism presented here establishes a baseline from which to explore this capability and develop new models for permission tracking.

Conclusion Gradual Featherweight Typestate (GFT) is a nominal core calculus for typestate-oriented programming. By introducing typestate directly into the language and extending its type system with support for gradual typing, state abstractions can be implemented directly, stronger program properties can be enforced statically, and when necessary dynamic checks can be introduced seamlessly. In particular GFT supports a rich set of access permissions together with state guarantees for substantial reasoning about typestate in the presence of aliasing. This work paves the way for further gradual approaches by showing how to modularly and gradually check flow-sensitive resources.

Acknowledgments We thank the members of the Plaid research group at Carnegie Mellon University, especially Darpan Saini and Nels Beckman, as well as the anonymous reviewers, for feedback on this work.

References

1. Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.
2. Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. Onward! 2009*, pages 1015–1022. ACM, 2009.
3. Henry G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.*, 29:38–43, September 1994.
4. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proc. Conference on Object-oriented Programming Systems and Applications*, pages 301–320. ACM, 2007.
5. Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proc. European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009.
6. Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *Proc. European Conference on Object-oriented Programming*, ECOOP'10, pages 76–100, Berlin, Heidelberg, 2010. Springer-Verlag.
7. Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. International Conference on Software Engineering*, pages 5–14. ACM, 2010.

8. John Boyland. Checking interference with fractional permissions. In *Proc. Static Analysis (SAS)*, pages 55–72. Springer, 2003.
9. John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Symposium on Principles of Programming Languages*, pages 283–295. ACM, January 2005.
10. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.
11. Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *Proc. European Conference on Object-Oriented Programming*, 2001.
12. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.
14. Ronald Garcia, Roger Wolff, Éric Tanter, and Jonathan Aldrich. Featherweight Typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University, July 2010.
15. Simon Gay, Vasco Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Caldeira. Modular session types for distributed object-oriented programming. In *Symposium on Principles of programming languages*, pages 299–312. ACM, 2010.
16. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.
17. 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.
18. Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, January 2010.
19. Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28:1–69, January 2006.
20. Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proc. Conference on Object-oriented programming systems languages and applications*, pages 347–366. ACM, 2008.
21. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
22. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp Symb. Comput.*, 6(3-4):289–360, 1993.
23. Darpan Saini, Joshua Sunshine, and Jonathan Aldrich. A theory of typestate-oriented programming. In *Formal Techniques for Java-like Programs*, 2010.
24. Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proc. Scheme and Functional Programming Workshop*, September 2006.
25. Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proc. European Conference on Object-oriented Programming*, pages 2–27. Springer, 2007.
26. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
27. David Walker. Substructural type systems. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, Cambridge, MA, 2005.
28. Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual Featherweight Typestate. Technical Report CMU-ISR-10-116R, Carnegie Mellon University, July 2010.