A

Foundations of Typestate-Oriented Programming

Ronald Garcia, University of British Columbia
Éric Tanter, University of Chile
Roger Wolff, Carnegie Mellon University
Jonathan Aldrich, Carnegie Mellon University

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, typestate was not treated as a primitive language concept: typestate checkers are an additional verification layer on top of an existing language. In contrast, a *typestate-oriented programming language* directly supports expressing typestates. For example, in the Plaid programming language, the typestate of an object directly corresponds to its class, and that class can change dynamically. Plaid objects have not just typestate-dependent interfaces, but also typestate-dependent behaviors and runtime representations.

This paper lays foundations for typestate-oriented programming by formalizing a nominal object-oriented language with mutable state that integrates typestate change and typestate checking as primitive concepts. We first describe a statically-typed language, called Featherweight Typestate (FT), where the types of object references are augmented with access permissions and state guarantees. We describe a novel flow-sensitive permission-based type system for FT. Because static typestate checking is still too rigid for some applications, we then extend this language into a gradually-typed language, called Gradual Featherweight Typestate (GFT). This 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. The gradual type system of GFT allows programmers to write dynamically safe code even when the static type checker can only partly verify it.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Typestate*; D.2.10 [**Software Engineering**]: Design—*Representation*

General Terms: Languages, Design, Reliability, Theory, Verification

Additional Key Words and Phrases: Access permissions, gradual typing, types, typestates

ACM Reference Format:
Garcia, R., Tanter, É., Wolff, R., Aldrich, J., 2014. Foundations of Typestate-Oriented Programming. ACM Trans. Program. Lang. Syst. Accepted for Publication (May 2014), 46 pages.

An earlier version of this article was presented at the European Conference on Object-Oriented Programming (ECOOP), July 2011 [Wolff et al. 2011].
R. Garcia is supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project and by the Natural Sciences and Engineering Research Council of Canada. É. Tanter is partially funded by FONDECYT Project 1110051, Chile. J. Aldrich and R. Wolff are supported by the National Science Foundation under grants #CCF-0811592 and #CCF-1116907.
Author's addresses: R. Garcia, Software Practices Laboratory, Computer Science Department, University of British Columbia; É. Tanter, PLEIAD Laboratory, Computer Science Department (DCC), University of Chile; J. Aldrich and R. Wolff, Institute for Software Research, Carnegie Mellon University.

1. INTRODUCTION

This paper investigates an approach to increase the expressiveness and flexibility of object-oriented languages, with the goal of improving 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 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 [1986] 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 [Strom and Yemini 1986] 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 in .NET [DeLine and Fähndrich 2004], to verify that Java programs adhere to object protocols [Fink et al. 2008; Bierhoff et al. 2009; Bodden 2010], and to check that groups of objects collaborate with each other according to an interaction specification [Naeem and Lhoták 2008; Jaspan and Aldrich 2009].

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. [Fink et al. 2008]), or with a modular checker based on additional program annotations (e.g. [Bierhoff and Aldrich 2007]). 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 [Aldrich et al. 2009]. For instance, in a class-based language that supports dynamically changing an object's class (such as Smalltalk), typestates can be represented as classes and can 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 catch such errors as early as possible, we want 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 [Fink et al. 2008; Naeem and Lhoták 2008; Bodden 2010]; others retain modularity at the expense of sophisticated type systems, typically based on linear logic [Walker 2005] and requiring many annotations. One kind of annotations is *access permissions*, which specify certain aliasing patterns [Boyland 2003; DeLine and Fähndrich 2004; Bierhoff and Aldrich 2007]. None of these approaches, however, incorporates typestates as a core language concept.

The first contribution of this paper is a core calculus for typestate-oriented programming inspired by Featherweight Java [Igarashi et al. 2001], called Featherweight Typestate (FT). FT is a nominal object-oriented language with mutable state that integrates typestate change and typestate checking as primitive concepts. Much like FJ, which characterizes Java and nominal object-oriented programming, Featherweight Typestate is meant to precisely characterize TSOP and to serve as a platform for exploring extensions to the paradigm and interactions with proven and bleeding-edge language features. A novel flow-sensitive permission-based type system makes it possible to modularly check FT programs.

Unfortunately, FT and all existing static typestate checkers cannot always verify safe code, due to the conservative assumptions they must make. Advanced techniques like fractional permissions [Boyland 2003] 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 about typestates are not supported by any modular approach we know of; one primary objective of this work is to support them.

Once dynamic coercions on typestates are available, they can be used to ease the transition from dynamically- to statically-typed code. For this reason, we extend gradual typing [Siek and Taha 2006, 2007] 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 second contribution of this work is Gradual Featherweight Typestate (GFT), an extension of FT that supports dynamic permission checking and gradual typing. Like FT, 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 supports recovery of precise typing using dynamically-checked assertions, supports the gradual addition of type annotations to a program, and enables permission- and typestate-based reasoning in dynamically typed programs.

Section 2 introduces the key elements of typestate-oriented programming with access permissions and state guarantees. Section 3 describes Featherweight Typestate, including its syntax, static and dynamic semantics, and its metatheory. Section 4 extends FT to Gradual Featherweight Typestate. GFT's dynamic semantics are presented using a type-safe internal language to which GFT translates. The soundness proofs for both languages are available in companion technical reports [Garcia et al. 2013; Wolff et al. 2013]. Section 5 relates the dynamic semantics of FT to that of GFT. In particular, every FT program is also a GFT program, and its translation to GFTIL has the same runtime behavior as running the FT program directly. This connection

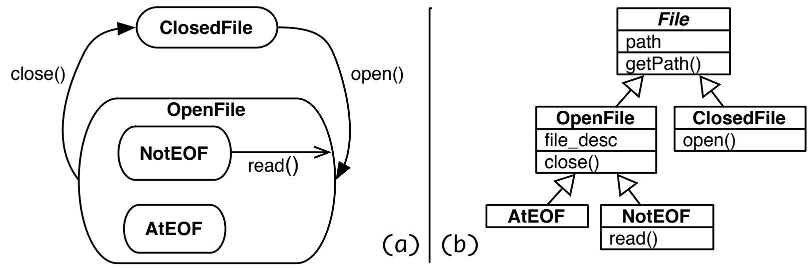

Fig. 1: (a) State diagram of a file. (b) Hierarchy of files states.

is analogous to the relationship between the simply typed, gradually typed, and cast-based languages of [Siek and Taha 2006]. Section 6 concludes. A translator for GFT's source language, type checker for the internal language, and executable runtime semantics are available at:
http://www.cs.ubc.ca/~rxg/gft/gft-toplas.tgz.[1]

2. TYPESTATE-ORIENTED PROGRAMMING

In order to avoid conditionals on flag fields or other indirect mechanisms like the State pattern [Gamma et al. 1994], 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.

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; if we reach the end-of-file, then `read`ing is not available anymore; an open file cannot be opened again, etc. Figure 1a shows a state diagram of a file object, describing the protocol. Figure 1b depicts the corresponding TSOP model of file objects in terms of states, using distinct classes in a subclass hierarchy to represent states. `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 Typestate-oriented programming language supports a state change operation, denoted ←. For instance, the `close` method in `OpenFile` can be defined as:

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

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

[1] An earlier version of this article was presented at the European Conference on Object-Oriented Programming (ECOOP), July 2011 [Wolff et al. 2011]. This paper differs from our previous article in a number of ways. Most importantly, we present the static language Featherweight Typestate (FT) in Section 3. Gradual Typestate's type system is simplified to more clearly reflect its foundations and its relation to FT.

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

Fig. 2: Access permissions.

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 type checker. 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* [Bierhoff and Aldrich 2007; Bierhoff et al. 2009]. 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 2): **`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. Our choice of permissions captures a coherent and self-contained set from the literature that supports common programming idioms. We can easily add more known permissions (e.g., **`immutable`**, **`unique`**, and **`none`**), but they would simply add more complexity to our development without providing any new insights.

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** access 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 [2007] 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.

Consider:

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

While x.read() may change the state of the file by transitioning it to AtEOF, the type system ensures that 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 always obeys the interface C.[2] 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 access permission with an unrestricted update capability (Object state guarantee). Then, a local alias y is created, capturing a **pure** access 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,

[2]In FT, state guarantees are enforced for the rest of program execution. As we will see, however, when we consider gradual typing, a guarantee can be removed if the variable of the guaranteed type goes out of scope, or a run-time assertion on that variable is executed. Extensions such as borrowing can also allow guarantees (e.g. on a borrowed object) to be removed.

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. Consider the example of a socket. A socket (of type `S`) is like a file in that it can be open (`OS`) or closed (`CS`). However, an open socket can also be ready (`RS`) or blocked (`BS`). The `wait` method accepts a blocked socket and waits until it is ready,[3] while the `read` method gets data from the socket. The methods of socket have the following signatures:

```
void wait() [ pure(OS) OS >> pure(OS) RS]
int read() [shared(OS) RS >> shared(OS) OS]
```

Now consider the following program, which waits on a blocked socket and then reads from it:

```
shared(OS) OS x = new OS(...);
x.wait();
x.read();
```

This program is ill-typed due to the downgrading of permissions. In order to invoke `wait`, the permission to `x` is downgraded from **`shared`**`(OS) OS` to **`pure`**`(OS) OS`. Therefore, `read`, which requires a **`shared`**`(OS) RS`, cannot be called, even though the call to `read` is safe: `wait` requires a read-only alias to its argument, and does nothing that would interfere with the caller's **`shared`**`(OS)` permission. This is an unfortunate limitation due to the conservative nature of the type system.

We could attempt to work around this problem by creating a temporary alias to `x` with only a **`pure`** access permission, and use that alias to invoke `wait`. This is however cumbersome and does not allow for permissions to be merged back later. Merging the permission returned by `wait` into the permission held by the client is crucial in this case, because we want `x` to have type **`shared`**`(OS) RS`, taking advantage of the fact that `wait` returns when the socket is ready (`RS`).

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:

```
shared(OS) OS x = new OS(...);
hold[x:shared(OS) OS] { x.wait(); }
x.read();
```

The program is now type correct: **`hold`** retains a **`shared`** access permission to the object referenced by `x`, which is merged back once the body of **`hold`** is evaluated. The call to `wait` is performed with just the necessary access permission, **`pure`**, and the state of the object is merged back into the permission of `x`, enabling the call to `read`. Our **`hold`** construct serves a similar purpose to borrowing [Boyland and Retert 2005; Naden et al. 2012], in that it can be used to ensure that the caller retains the permissions it needs after making a method call. The two differ in that borrowing ensures the callee returns all of the permissions that it was given without storing any in the heap. In contrast hold is for the caller only: the callee can do what it wants with the permissions it receives so long as sufficient permissions are returned to the caller.

[3]Note that `wait` does not actually change the state of the socket itself, but rather asserts the desired `RS` type once the state of the socket has been changed, e.g. by another thread or by a coroutine.

Dynamic permission asserts. As sophisticated as the type system might be, 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 (because `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.

Note that dynamic *class* asserts, which modify the static class of an object but leave permissions alone, need no special support beyond what is needed for a typical object-oriented language. Therefore, a static typestate language that runs on a standard OO backend can support dynamic assertions about the class of an object. Dynamic permission asserts, on the other hand, require special support from the runtime system.

Gradual typing. A statically-typed TSOP program requires more annotations than a comparable object-oriented 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 verifies 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 verifies 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.

While typestate checking has historically been considered only in a fully static setting, supporting gradual typestate checking means that access permissions and state guarantees are *properties that are dynamically enforced*. Just like objects have references to their class, object references have both access permissions and state guarantees. For instance:

```
Dyn x = app.getFile();
pure(OF) OF y = x;
app.process(x);
```

`x` is a dynamic reference to a file, and remains so even after a statically-typed alias `y` is created. However, the static assumptions made by `y` are dynamically enforced: both `x` and `y` refer to (at least) an open file after the execution of `process`. If `process` tries to close `x`, an error is raised.

[4]Note that **`Dyn`** is different from `Object`: if `f` had type `Object` then type checking would fail because `Object` has no `read` method.

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 provides 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[6] (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 23), the `file0` reference no longer has enough permission to close the file, so calls to `logger.log()` are safe. Line 25 passes `logger` to a dynamically-typed method; as a result, `logger` is of type **`Dyn`** after the call. Using **`hold`**, we hold a **`shared`**`(OFL) OFL` permission to the logger while the `dynamicLog` call happens, then restore those permissions before the call to `staticLog`. Had we not held these permissions, the `logger` would have **`Dyn`** type, and the call to `staticLog()` (line 26) would be preceded by an (automatically-inserted) assertion to dynamically ensure that `logger` is of the appropriate type (**`shared`**`(OFL) OFL`). By line 28, `logger` only has **`shared`** access permission, though no other aliases exist. After **`assert`**ing back full access permission, `logger` can close the file log.

3. FEATHERWEIGHT TYPESTATE

In this section we present Featherweight Typestate (FT), a static language for typestate-oriented programming. FT is based on Featherweight Java (FJ) [Igarashi et al. 2001]. FT is the first formalization of a nominal typestate-oriented programming language, with support for representing typestates as classes, modular typestate-checking and state guarantees.

3.1. Syntax

Figure 3 presents FT's syntax. Smallcaps (e.g. FIELDNAMES) indicates syntactic categories, italics (e.g. C) indicates metavariables, and sans serif (e.g. Object) indicates particular elements of a category. An overbar (e.g. $\overline{A}$) indicates possibly empty sequences (e.g. $A_1, ..., A_n$). FT assumes a number of primitive notions, such as identifiers (including this) and method, field, and class names (including Object). An FT program PG is a list of class declarations $\overline{CL}$ paired with an expression e. Class definitions are standard, except that an FT class does not have an explicit constructor: instead, it has an implicit constructor that assigns an initial value to each field. Featherweight Java, for instance, requires an explicit constructor, but its type system forces the same behavior as in FT. Fields F and methods M are mostly 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. Like FJ, we use helper functions like fields, method, etc., whose definitions are deferred to the appendix.

Types in FT extend the Java notion of class names as types. As explained in Section 2, the type of an FT object reference has two components, its permission and its class (or *state*). The permission can be broken down further into its access permission k (described previously in Figure 2) and state guarantee D. We write these *object reference types* in the form $k(D)\ C$. Following the Java tradition, the Void type classifies expressions executed purely for their effects. No source-level values have the Void type.

[5]When the output type is the same as the input type, we omit it for brevity; a practical language would provide means to further abbreviate our type annotations.
[6]A practical language would support nullable references, but for simplicity we omit this.

```
class FileLogger { /* Logging−related data and methods */ }

class OpenFileLogger : FileLogger {
  full(OF) OF file;

  void log(string s)[shared(OFL) OFL] {...}

  void close()[full(FL) OFL ≫ full(FL) FL] {
        full(OF) OF fileT = (this.file :=: new OF("/dev/null"));
        assert<full(F) OF>(fileT);
        fileT.close();
        this ← FileLogger();
  }
}

// Client code
void staticLog(shared(OFL) OFL logger) {
 logger.log("in staticLog");
}
Dyn dynamicLog(Dyn logger) { logger.log("in dynamicLog"); }

full(OF) OF file0 = new OF(...);
full(OFL) OFL logger = new OFL(file0);

hold[logger:shared(OFL) OFL]{ dynamicLog(logger); }
staticLog(logger);

assert<full(FL) OFL>(logger);
logger.close();
```

Listing 1: Sample Typestate-Oriented Code.

To simplify the description of the type system, expressions in FT are restricted to A-normal form [Sabry and Felleisen 1993], so let expressions explicitly sequence all complex operations (we write $e_1; e_2$ as shorthand for the standard encoding).

Apart from method invocation, field reference and object creation (all standard), FT includes the update operation $x_0 \leftarrow C(\overline{x_1})$, which lets programs directly express typestate change. It replaces the object referred to by x_0 with a 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 of object x_0 and returns the old value as its result. Section 3.3 explains why this is needed.

The assert operation changes the static type of an object reference. Asserts are similar to casts in that an assert up the subclass hierarchy succeeds immediately, while an assert down the class hierarchy requires a runtime check. Assertions are strictly more powerful than casts: they change the type of an existing reference, whereas casts produce a new reference with a different type. In fact, a type cast $(T)\ x$ can be encoded using class assertions:

$$(\mathsf{T})\ \mathsf{x} \triangleq \begin{array}{l} \mathsf{let}\ \mathsf{y} : \mathsf{T}_0 = \mathsf{x} \\ \ \mathsf{in}\ \mathsf{let}\ \mathsf{z} : \mathsf{Void} = \mathsf{assert}\langle \mathsf{T} \rangle(\mathsf{y}) \\ \ \mathsf{in}\ \mathsf{y} \end{array}$$

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

Fig. 3: Featherweight Typestate: Syntax

where y and z are fresh, and the type T_0 depends on how much permissions are to be taken from x. The assert operation of FT cannot change the permission of a variable, but the run-time permission tracking introduced in GFT will allow us to add support for permission-changing assertions there.

The hold 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.

3.2. Managing Permissions

Before we present FT's typing judgments, 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 4 presents several auxiliary judgments that specify how permissions may be safely split, and how they relate to typing.

First, *access permission splitting* $k_1 \Rrightarrow k_2/k_3$ describes how given a k_1 access permission, k_2 can be acquired, leaving behind k_3 as the residual. When we are only concerned that k_2 can be split from k_1 (i.e. the residual access permission is irrelevant), we write $k_1 \Rrightarrow k_2$. For instance, given any access 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, analogous to subtyping. The notation $P_1 <: P_2$ says that P_1 is a *subpermission* of P_2, which means that a reference with P_1 permission may be used wherever an object reference with P_2 permission is needed. As expected, the subpermission relation is reflexive, transitive, and anti-symmetric. The first subpermission rule says that splitting an access permission produces a lesser (or identical) permission. The subpermission rules for pure and full access permissions respectively capture how state guarantees affect the strength of permissions. Pure access permissions covary with their state guarantee because a pure reference with a superclass state guarantee assumes less reading capability. Full access permissions contravary with their state guarantee because a full reference with a subclass state guarantee assumes less writing capability (i.e. it can update to fewer possible states). Although full access permissions also allow reads, those reads can only see writes through the full reference itself; therefore contravariance is enough and we do not have to enforce invariance. The last rule ensures that subpermissions are transitive.

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

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

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

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

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

$$\frac{}{\mathsf{shared}(D) \leftrightarrow \mathsf{shared}(D)}$$

$$\frac{\forall i, j : i \neq j. P_i \leftrightarrow P_j}{\overline{P} \text{ compatible}}$$

$\boxed{P <: P}$ Subpermission

$$\frac{k_1 \Rrightarrow k_2}{k_1(D) <: k_2(D)} \qquad \frac{E <: D}{\mathsf{pure}(E) <: \mathsf{pure}(D)}$$

$$\frac{D <: E}{\mathsf{full}(E) <: \mathsf{full}(D)} \qquad \frac{P_1 <: P_2 \quad P_2 <: P_3}{P_1 <: P_3}$$

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

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

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

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

$$\frac{}{\mathsf{Void} \Rrightarrow \mathsf{Void}/\mathsf{Void}}$$

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

$$\frac{P = P_1 \mathbin{\underset{\cdot\cdot}{\wedge}} P_2 \quad C = C_1 \mathbin{\underset{\cdot\cdot}{\wedge}} C_2}{P_1\ C_1/P_2\ C_2 \Rrightarrow P\ C} \qquad \frac{}{T/\mathsf{Dyn} \Rrightarrow T} \qquad \frac{}{\mathsf{Dyn}/T \Rrightarrow T}$$

$\boxed{T <: T}$ Subtyping

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

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

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

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

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

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

Fig. 4: Permission and Type Management Relations

Permission splitting extends access permission splitting to account for state guarantees. First, if $k_1(D_1) <: k_2(D_2)$, then the latter can safely be split from the former. The remaining task then is to determine the proper residual permission $k_3(D_3)$. The residual access permission k_3 comes directly from access permission splitting. For the residual state guarantee, observe that $k_1(D_1) <: k_2(D_2)$ implies that D_1 and D_2 are related by subclassing. By considering the possible cases of permission splitting, we find that the state guarantee should be whichever of D_1 and D_2 is subclass of the other: this is necessary if k_3 is a write access, and ideal if k_3 is pure. We denote this as the greatest lower bound of D_1 and D_2 in the subclass hierarchy $D_1 \mathbin{\underset{\cdot\cdot}{\wedge}} D_2$, an operation that we use (along with greatest lower-bound permission $P_1 \mathbin{\underset{\cdot\cdot}{\wedge}} P_2$) several times in the language formalization.

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. Type splitting has a special case called the *maximum residual*, the most permissions that can be split without changing the original type. Type splitting determines the notion of subtyping $T <: T$ used in FT. As with access 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.

Converse to type splitting is *type merging*, denoted $T/T \Rrightarrow T$. The type merging relation describes how two separate permissions to the same underlying object may be combined.

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 $\mathsf{full}(C) \leftrightarrow \mathsf{pure}(\mathsf{Object})$. On the other hand, $\mathsf{full}(C) \not\leftrightarrow \mathsf{full}(C)$ because if one of these references updated its state guarantee (further down the subclass hierarchy), then the other reference could violate it during a state change operation. The compatible permission relation is used to define the relation $\overline{P}$ *compatible*: that the outstanding permissions $\overline{P}$ of references to a particular object can all coexist. These concepts are critical for showing that well-typed programs remain in a consistent state as they run.

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

3.3. Static Semantics

Armed with the permission management relations, we now discuss the most salient feature of FT's static semantics: flow-sensitive typing.

As with FJ, the FT type system relies upon type contexts. 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. In our notation, $\Delta, x : T$ specifies a context Δ' that includes all of the bindings in Δ plus the binding $x : T$, which *requires that* Δ *contains no entry for* x. In FT'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 and assert operations may change the class of an object during execution.

The FT typing judgment is a quaternary relation of the form $\Delta_1 \vdash e : T \dashv \Delta_2$, which means "given the typing assumptions Δ_1, the expression e can be assigned the type T and doing so 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.

Typing rules. Figure 5 presents the typing rules for FT expressions (all prefixed with "ST": S for "static typing" and T for "typing").

The (STvar) typing rule, for variable references, demonstrates flow-sensitive typing immediately. If the type context binds a variable x to a type T_1, and that variable is referenced at type T_2, then the output type context resets the type assumption for x according to the type splitting relation. Observe that the (STvar) rule implies that in general a variable reference can be given many possible types:

LEMMA 3.1. *If* $\Delta \vdash x : T_1 \dashv \Delta'$ *and* $T_1 <: T_2$ *Then* $\Delta \vdash x : T_2 \dashv \Delta''$ *for some* Δ''.

This is similar to the standard subsumption rule for object-oriented languages, but changing the type from T_1 to T_2 also changes the output context.

The (STlet) rule reflects the standard value-binding behavior for let, but it also sequences permission-consuming operations. After typing the expression bound to x, the new typing context is updated with a type assumption for x, which is used to type the body of the let. To preserve lexical scoping, x (and its associated permission) is removed from the output context.

$\boxed{\Delta \vdash e : T \dashv \Delta}$ Well-typed Expression

$$\text{(STvar)}\,\frac{T_1 \Rrightarrow T_2/T_3}{\Delta, x : T_1 \vdash x : T_2 \dashv \Delta, x : T_3} \qquad \text{(STlet)}\,\frac{\begin{array}{c}\Delta \vdash e_1 : T_1 \dashv \Delta_1 \\ \Delta_1, x : T_1 \vdash e_2 : T_2 \dashv \Delta', x : T_1'\end{array}}{\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta'}$$

$$\text{(STnew)}\,\frac{\begin{array}{c}\mathit{fields}(C) = \overline{T\ f} \\ \Delta \vdash \overline{x : T} \dashv \Delta'\end{array}}{\Delta \vdash \mathsf{new}\ C(\overline{x}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta'}$$

$$\text{(STupdate)}\,\frac{\begin{array}{c}k \in \{\mathsf{full}, \mathsf{shared}\} \quad \mathit{fields}(C) = \overline{T\ f} \quad C <: D \\ \Delta \vdash \overline{x_2 : T} \dashv \Delta', x_1 : k(D)\ E\end{array}}{\Delta \vdash x_1 \leftarrow C(\overline{x_2}) : \mathsf{Void} \dashv \Delta'\!\downarrow, x_1 : k(D)\ C}$$

$$\text{(STfield)}\,\frac{T_2\ f \in \mathit{fields}(C_1) \quad T_2 \Downarrow T_2'}{\Delta, x : P_1\ C_1 \vdash x.f : T_2' \dashv \Delta, x : P_1\ C_1} \qquad \text{(STswap)}\,\frac{\begin{array}{c}T_2\ f \in \mathit{fields}(C_1) \\ \Delta, x_1 : P_1\ C_1 \vdash x_2 : T_2 \dashv \Delta'\end{array}}{\Delta, x_1 : P_1\ C_1 \vdash x_1.f :=: x_2 : T_2 \dashv \Delta'}$$

$$\text{(STinvoke)}\,\frac{P_1\ C_1 <: T_t \quad \mathit{mdecl}(m, C_1) = T\ m(\overline{T_i \gg T_i'})\ [T_t \gg T_t'] \quad \overline{T_2 <: T_i}}{\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) : T \dashv \Delta\!\downarrow, x_1 : T_t', \overline{x_2 : T_i'}}$$

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

$$\text{(SThold)}\,\frac{\begin{array}{c}T_1 \Rrightarrow T_2/T_3 \quad T_2\!\downarrow / T_3' \Rrightarrow T_1' \\ \Delta, x : T_3 \vdash e : T \dashv \Delta', x : T_3'\end{array}}{\Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) : T \dashv \Delta', x : T_1'}$$

where

$$\Delta \vdash \overline{x : T} \dashv \Delta' \triangleq \Delta = \Delta_0 \vdash x_0 : T_0 \dashv \Delta_1; \quad \Delta_1 \vdash x_1 : T_1 \dashv \Delta_2; \cdots \Delta_n \vdash x_n : T_n \dashv \Delta_{n+1} = \Delta'$$

Fig. 5: Featherweight Typestate: Expression Typing Rules

The (STnew) rule, for creating a new object, is analogous to the equivalent Java rule. The prominent difference is that in FT a new object also has permissions associated with it. The reference to a new object is given $\mathsf{full}(\mathsf{Object})$ permissions because it is unique, so it can update the object arbitrarily without concern about aliases. This rule relies on an auxiliary judgment that captures the idea of a well-typed constructor call: the arguments to the constructor are iteratively checked against the class fields and the typing context is iteratively updated accordingly. This means that a variable may be given for more than one argument to the constructor, but because of flow-sensitive typing, it may be typed differently each time.

The (STassert) rule reflects how the assert operation $\mathsf{assert}\langle T\rangle(x)$ changes class type information of the reference x. Though FT types consist of more than the object's class, this operation can only affect the class part of an object's type: the permission must stay the same. The assert operation could safely decrease permissions to an object, but it would add no expressiveness to the language.

The (SThold) rule reflects how the hold expression acquires permissions to an object for the dynamic extent of its subordinate expression. Once that expression completes, the held permissions are returned to the reference from which they were acquired. It types the subexpression e after splitting T_2 from variable x. The resulting type of x is the merge of the demotion of T_2 (the type being being held) and T_3', the resulting output type of x after evaluation of e.

Class update. The (STupdate) rule type checks FT's novel update operation, $x_1 \leftarrow C_2(\overline{x_2})$, which replaces the receiving object referenced by x_1 with $C_2(\overline{x_2})$. This operation is only possible if the reference to the receiving object has shared or full access permissions to the underlying object. The possible target states of an object are implicitly constrained by the state guarantee that the object has *after* the arguments to the constructor have been typed, since $k(D)\ C$ must be a well-formed type. This ensures that the outstanding references to the updated object (including possibly its own fields) all have a consistent view of the object. The type of the update operation is Void since it is performed solely for its effect on the heap. The type of the updated object in the output context 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 4) 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 (STupdate) rule necessarily demotes types: 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.

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. $T{\downarrow} = T$). This means that a field with shared or pure access permission has the same class type as its state guarantee. 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. As a result only local variable types need ever be demoted.

The classes of variables in Δ_2 are demoted to their state guarantees since state change may have invalidated those stronger assumptions. Only one object is updated by this operation, but it may affect any number of outstanding references.

Field Operations. As was mentioned in Section 3.1, two operations operate directly on an object field: field reference and swapping assignment. Field reference (STfield) does not relinquish any of the permissions held by the 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.

Swap operations (STswap) 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 access permissions to an object, then field reference can yield the same amount of permission; however, if a field has full access permission to an object, only swapping can yield that full access permission.

Method Invocation. The (STinvoke) rule describes how method invocations are type checked. When invoking a method, first the method declaration is looked up based on the type of the receiver. Next, both the receiver and the arguments are checked for compatibility. The resulting type of the expression is, as usual, specified by the method declaration. The outgoing context demotes the references in Δ. This is necessary to keep type checking modular, since the method call may perform typestate update operations. The outgoing types for the receiver and the arguments are, however, listed in

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

$$\frac{\begin{array}{c}\text{class } C \text{ extends } D\ \{\cdots\}\\ mdecl(D,m) \text{ undefined}\end{array}}{T_r\ m(\overline{T_i \gg T_i'})[P_t\ C \gg T_t']\ \textbf{ok in}\ C} \qquad \frac{\begin{array}{c}\text{class } C \text{ extends } D\ \{\cdots\}\\ mdecl(D,m) = T_r\ m(\overline{T_i \gg T_i'})[P_t\ E \gg T_t']\end{array}}{T_r\ m(\overline{T_i \gg T_i'})[P_t\ C \gg T_t']\ \textbf{ok in}\ C}$$

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

$$\frac{\begin{array}{c}T_r\ m(\overline{T_i \gg T_i''})[P_t\ C_t \gg T_t'']\ \textbf{ok in}\ C_t\\ \text{this} : P_t\ C_t, \overline{x : T_i} \vdash e : T_r \dashv \text{this} : T_t', \overline{x : T_i'}\\ T_t' <: T_t'' \qquad\qquad \overline{T_i' <: T_i''}\end{array}}{T_r\ m(\overline{T_i \gg T_i''\ x})\ [P_t\ C_t \gg T_t'']\ \{\ \text{return } e;\ \}\ \textbf{ok in}\ C_t}$$

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

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

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

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

Fig. 6: FT Program Typing Rules

the method's declaration, and as such are available to the program when the method returns.

Note that in several other expressions (new, for example), permissions to certain variables (arguments to the constructor) are implicitly split, and residual permissions are left over for typing the remainder of the program. Method invocation is different. To keep FT's design simple, the method invocation rule checks that method arguments (including the receiver) have enough permission to type the method call, and discards any residual permissions. Also, the structure of (STinvoke) requires all method arguments to be unique, e.g., $x.m(y, y)$ is untypeable (See Section 3.7).

Typing Programs. Recall that an FT program is a pair of a class table and an expression. To formalize the notion of a well-typed program, we introduce a few more judgments (Figure 6).

First we consider the interface or *declaration* of a method:

$$Md ::= T\ m(\overline{T \gg T})\ [T \gg T]$$

The method declaration judgment Md **ok in** C checks that the interface specification for a method is compatible with a particular class, which holds if the method is altogether new, or a proper override of a superclass method. This is used by the method typing judgment M **ok in** C, which checks that a method M is well-typed if it is defined as part of class C. To type the body of the method, the rule assumes the input types from the method declaration. On completion of typing the method, the arguments and this are checked against the method's output specification. This typing rule allows this and the arguments x to be subtypes of the output types specified by the declaration. The method is well-typed so long as enough permissions remain for these variables to match the declared output specification. If the type system required the output context Δ_0 to exactly match the output specification, then the language would need more mechanisms (such as an explicit subsumption rule).

For a class definition to be well-typed, all of its fields must have object reference types, all of its methods must be well-typed, and its superclass hierarchy must lead to Object. This implies that all intermediate superclasses are defined and that every chain of superclasses ends at Object, i.e. there are no inheritance cycles. Also, as explained above, we require that the permissions associated with field types be invariant under demotion.

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

3.4. Dynamic Semantics

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

$$
\begin{array}{rcll}
o & \in & \textsc{ObjectRefs} & \\
l & \in & \textsc{IndirectRefs} & \\
v & \in & \textsc{Values} & \\
C(\overline{o}) & \in & \textsc{Objects} & \\
e & ::= & s \mid v \mid \mathsf{let}\ x : T = e\ \mathsf{in}\ e \mid \mathsf{new}\ C(\overline{s}) \mid s.f & \text{(expressions)} \\
& \mid & s.m(\overline{s}) \mid s.f :=: s \mid s \leftarrow C(\overline{s}) \mid \mathsf{assert}\langle T\rangle(s) & \\
& \mid & \mathsf{merge}[l : T/l](e) & \\
s & ::= & x \mid l & \text{(simple expressions)} \\
v & ::= & \mathsf{void} \mid o & \text{(values)} \\
\mu & \in & \textsc{ObjectRefs} \rightharpoonup \textsc{Objects} & \text{(stores)} \\
\rho & \in & \textsc{IndirectRefs} \rightharpoonup \textsc{Values} & \text{(environments)} \\
\mathbb{E} & ::= & \square \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \mid \mathsf{merge}[l : T/l](\mathbb{E}) & \text{(evaluation contexts)}
\end{array}
$$

Ultimately, expressions in the language evaluate to values, i.e. void or an object reference o. Since the language is imperative, the value void is used as the result of operations that are only interesting for their side-effects. In other object-oriented languages, a void object is unnecessary: imperative operations can return some arbitrary object reference. However, FT must explicitly consider how permissions to an object are distributed, so providing a void object lets us clearly indicate when no permissions to any object are returned.

The merge expression is a technical device that models how held permissions are treated dynamically. It tracks held permissions at runtime and ultimately merges those held permissions back into their associated indirect reference. This expression is purely a tool for proving type safety.

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

In addition to the traditional heap, the dynamic semantics uses a second heap, which we call the *environment*, that mediates between variable references and the object store. The environment serves a purely formal purpose: it supports the proof of type safety by keeping precise track of the outstanding permissions associated with different references to objects at runtime. In the source language, two variables could refer to the same object in the store, but each can have different permissions to that object. The environment 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. The runtime language therefore adds a notion of simple expressions s, which include true variables x and indirect references l, and may be used in the runtime language everywhere that variables can be used in the programmer-visible language (except, of course, variable definition). The environment is not needed in a practical implementation of the language. As we show later (Section 3.6), well-typed programs can be safely run on a traditional single-heap machine where object references are simple expressions.

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

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

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

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

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

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

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

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

$$\text{(SEcongr)}\frac{\mu, \rho, e_1 \rightarrow \mu', \rho', e_1'}{\mu, \rho, \mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2 \rightarrow \mu', \rho', \mathsf{let}\ x : T = e_1'\ \mathsf{in}\ e_2}$$

$$\text{(SEhold)}\frac{l' \notin \mathit{dom}(\rho)}{\mu, \rho, \mathsf{hold}[l : T](e) \rightarrow \mu, \rho[l' \mapsto \rho(l)], \mathsf{merge}[l : (T{\downarrow})/l'](e)}$$

$$\text{(SEmcongr)}\frac{\mu, \rho, e \rightarrow \mu', \rho', e'}{\mu, \rho, \mathsf{merge}[l_1 : T/l_2](e) \rightarrow \mu', \rho', \mathsf{merge}[l_1 : T/l_2](e')}$$

$$\text{(SEmerge)}\frac{}{\mu, \rho, \mathsf{merge}[l_1 : T/l_2](v) \rightarrow \mu, \rho, v}$$

Fig. 7: FT Dynamic Semantics

To state and prove our notion of type safety, we use a notion of evaluation contexts $\mathbb{E}$. Evaluation contexts are expressions with *holes*, notation $\square$, in them. An expression can be plugged into the hole to produce a program. Following the presentation of Featherweight Java by Pierce [2002], we use evaluation contexts to capture the possibility of a program getting stuck at a bad assertion.

The dynamic semantics of FT is formalized as a structural operational semantics defined over store/environment/expression triples. Figure 7 presents the rules (prefixed by "SE": "S" for static typing, "E" for evaluation).

The (SElookup) rule dereferences an indirect reference to get the underlying value. The (SEnew) rule creates a new object based on the constructor expression given. The arguments to the constructor are dereferenced so that the objects in the heap contain object references. The (SElet) rule handles a variable binding by allocating a new indirect reference, associating the object reference in question to it in the environment and substituting the fresh reference into the body of the let expression. The (SEupdate) rule replaces a binding in the store with a newly-constructed object. The (SEfield) rule looks up the field of an object in the heap and returns the corresponding object reference. The (SEswap) rule swaps the field of an object with a new object reference and returns the old one. The (SEassert) rule checks that a reference points to an object with a type compatible with the assertion. If the assertion succeeds, the program returns a void value; if not, the program gets stuck. The (SEinvoke) rule substitutes the arguments to the method invocation into the method body and continues executing. The (SEcongr) rule ensures that the bound expression in a let is computed before the body of the let. The (SEhold) rule initiates the bookkeeping process of holding on to permissions while a subexpression executes. It uses a new indirect reference l' to hold its permissions. The (SEmcongr) rule allows the expression inside of the merge expression to execute. The (SEmerge) expression removes the bookkeeping information once the relevant subexpression has evaluated to a final value.

3.5. Type Safety

In order to establish type safety, the type system must be extended to account for runtime phenomena. First, we must type the void value.

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

To type runtime programs, type contexts must be extended to account for runtime references.

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

Since runtime expressions have no free variables but may now contain indirect references l and object references o, a typing context may only have entries of the form $l : T$ and $o : T$. As such, the type rules must account for references in a runtime program, e.g.:

$$\text{(STvar)}\frac{T_1 \Rrightarrow T_2/T_3}{\Delta, \boxed{s} : T_1 \vdash \boxed{s} : T_2 \dashv \Delta, \boxed{s} : T_3} \qquad \text{(STobj)}\frac{}{\Delta, o : T \vdash o : T \dashv \Delta}$$

Since variables are replaced by indirect references at runtime, they should be typed similarly. On the other hand, an object reference may only appear once in a program, as the result of a variable reference, which will either be bound to a variable immediately, or returned as the final program result. As such, it is safe to consume it entirely.

Other references to variables in FT's type system should now consider simple expressions (variables or indirect references) rather than just variables, for example:

$$\text{(STnew)}\frac{\mathit{fields}(C) = \overline{T\ f} \qquad \Delta \vdash \boxed{\overline{s : T}} \dashv \Delta'}{\Delta \vdash \mathsf{new}\ C(\boxed{\overline{s}}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta'}$$

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

In addition, we require a typing rule for the runtime merge expression:

$$\text{(STmerge)}\frac{T_1 = T\!\downarrow \qquad \Delta, l_2 : T_2 \vdash e : T \vdash \Delta_1, l_2 : T_2' \qquad T_1/T_2' \Rrightarrow T_3}{\Delta, l_1 : T_1, l_2 : T_2 \vdash \mathsf{merge}[l_1 : T_1/l_2](e) : T \dashv \Delta_1, l_2 : T_3}$$

The merge expression owns the indirect reference l_1, which it uses to store the permissions that it is holding to later merge back into l_2. Thus the outgoing permissions of l_2 combine the output of the computation e with the held permissions.

Helper Functions

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

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

$$\frac{\mu(o) = C(\overline{o'}) \qquad |\overline{o'}| = |\mathit{fields}(C)| \qquad \mathit{refTypes}(\mu,\Delta,\rho,o) = \overline{k(E)\ D} \qquad C <: \overline{D} \qquad \overline{k(E)} \text{ compatible}}{\mu,\Delta,\rho \vdash o\ \mathbf{ok}}$$

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

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

$\boxed{\rho \vdash e\ \mathbf{mc}}$ Merge Consistency

$$\frac{\forall \mathbb{E}.\ e = \mathbb{E}[\mathsf{merge}[l_1 : T_1/l_2](e')] \Rightarrow \rho(l_1) = o = \rho(l_2)}{\rho \vdash e\ \mathbf{mc}}$$

Fig. 8: FT Permission Consistency Relations

To prove type safety, we must account for the outstanding permissions associated with references to each object o and make sure that they are mutually consistent. To achieve this, we appeal to some helpers, presented in Figure 8. The $\mathit{fieldTypes}$ function takes a heap and an object reference in the domain of the heap and produces a list of the type declarations for every field reference to that object. This function disregards object references that are not bound to some field of some object. The $\mathit{envTypes}$ function performs the analogous operation for the indirect references in an environment that have bindings in the context. This function disregards indirect references in the environment that have no typing in the context. The $\mathit{ctxTypes}$ function does the same for object references that occur in a type context. The $\mathit{refTypes}$ function takes a heap, context, environment, and object and yields the list of type declarations for outstanding heap, environment, and context references. These definitions use square brackets to express list comprehensions, and $+\!\!+$ to express list concatenation.

Using the $\mathit{refTypes}$ function and permission compatibility, we can define a notion of *reference consistency* that verifies the mutual compatibility of the types of all outstanding references to some object in the heap. A consistent object reference points to an object that has the proper number of fields, and all references to it are well-formed, assume a plausible class, are mutually compatible, and are 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 prove that preservation holds, we require an additional notion of consistency, called *merge consistency*, to ensure that only indirect references to the same underlying object are ever merged. This judgment helps us guarantee that permissions produced at runtime by hold expressions are only combined in sound ways.

These concepts contribute to the statement (and proof) of type safety.

THEOREM 3.2 (PROGRESS). *If e is a closed expression and $\Delta \vdash e : T \dashv \Delta'$, then either e is a value or for any store μ and environment ρ such that μ, Δ, ρ **ok**, either $\mu, \rho, e \to \mu', \rho', e'$ for some store μ', environment ρ', and expression e', or e is stuck at a bad assert, i.e., $e = \mathbb{E}[\mathsf{assert}\langle D\rangle(l)]$ where $\mu(\rho(l)) = C(\cdots)$, and $C \not<: D$.*

PROOF. By induction on the derivation of $\Delta \vdash e : T \dashv \Delta'$. $\square$

To facilitate our proof of type preservation, we define and establish an invariant of program evaluation. Our semantics has many rules that evaluate to an object reference, but the reference is either returned as the final result of the program, or immediately bound to an identifier (as in let $x : T = o$ in e). Furthermore, at most one object reference appears in a program at any point of execution. We capture these invariants as follows:

Definition 3.3. An expression e is in *head reference form*, notation $hdref(e)$ iff either

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

When a program is in head reference form and takes a step that produces or consumes an object reference, we can easily characterize a type context that establishes preservation.

Finally, we establish a relationship between type contexts that helps us show that evaluating a subterm retains enough output permissions to continue executing the rest of the program. This is needed in particular to support method invocations, since the permissions resulting from evaluating a method body may be be stronger than the method interface declares.

Definition 3.4. A context Δ is *stronger* than a context Δ', notation $\Delta < \Delta'$ if and only if for all $l : T' \in \Delta'$, there is some $T <: T'$ such that $l : T \in \Delta$.

Using these formal helpers, we can state and prove a preservation theorem.

THEOREM 3.5 (PRESERVATION). *If e is a closed expression, $\Delta \vdash e : T \dashv \Delta''$, μ, Δ, ρ **ok**, $hdref(e)$, $\rho \vdash e$ **mc**, and $\mu, \rho, e \to \mu', \rho', e'$ then for some Δ', $\Delta' \vdash e' : T \dashv \Delta'''$, μ', Δ', ρ' **ok**, $\rho' \vdash e'$ **mc**, and $\Delta''' <^{l} \Delta''$.*

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

3.6. Single-Heap Implementation Model

As we have previously mentioned, the environment in the FT dynamic semantics is specifically a tool for proving type safety. In particular, we need indirect references so that we can independently track the permissions to a particular object held by individual aliases. Similarly, hold and merge expressions only play a role in statically allocating permissions, and need not be considered after type checking FT programs. Here we formally show that a practical implementation of the language can use a traditional heap and can do without hold and merge. Figure 9 presents the rules (prefixed by "SI": "S" for static, "I" for implementation). The implementation semantics almost exactly matches the dynamic semantics, but leaves out the extra layer of indirection imposed by indirect references l and environments ρ. Note as well that there are no rules for hold or merge. This is because Featherweight Typestate does not need to track runtime permissions in practice. The hold expression is a purely static means of

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

$$\text{(SInew)}\ \frac{o \notin dom(\mu)}{\mu, \mathsf{new}\ C(\overline{o'}) \rightarrow \mu[o \mapsto C(\overline{o'})], o} \qquad \text{(SIlet)}\ \frac{}{\mu, \mathsf{let}\ x : T = v\ \mathsf{in}\ e \rightarrow \mu, [v/x]e}$$

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

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

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

$$\text{(SIcongr)}\ \frac{\mu, e_1 \rightarrow \mu', e'_1}{\mu, \mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2 \rightarrow \mu', \mathsf{let}\ x : T = e'_1\ \mathsf{in}\ e_2}$$

Fig. 9: FT Implementation Semantics

controlling permission flows, and merge is merely a technical device for proving type safety, so the implementation semantics for FT can discard them.

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

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

Where the erasure function $\mathcal{E}(e)$ is the natural extension of the following equations:

$$\mathcal{E}(\mathsf{hold}[l : T](e)) = \mathcal{E}(e)$$
$$\mathcal{E}(\mathsf{merge}[l_1 : T/l_2](e) = \mathcal{E}(e)$$

and where $\rho(e)$ is the natural extension of $\rho(l)$ to arbitrary expressions. Note that this relation is defined up to choice of object references.

PROPOSITION 3.6.

(*1*) *If* e *is a source program, then* $\emptyset, \emptyset, e \sim \emptyset, \mathcal{E}(e)$.
(*2*) *If* $\mu_1, \rho_1, e_1 \sim \mu'_1, e'_1$ *and* $\mu_1, \rho, e_1 \rightarrow \mu_2, \rho_2, e_2$ *then* $\mu'_1, e'_1 \rightarrow^* \mu'_2, e'_2$ *and* $\mu_2, \rho_2, e_2 \sim \mu'_2, e'_2$, *for some store* μ'_2, *and some expression* e'_2.

PROOF.

(1) Immediate.
(2) by induction on $\mu_1, \rho_1, e_1 \rightarrow \mu_2, \rho_2, e_2$.

□

3.7. Discussion

In the design of Featherweight Typestate we made a number of decisions based on the desire to simplify the resulting calculus. We now discuss these decisions and their alternatives.

Method calls. In FT, two particular restrictions on method calls are made to simplify the type system design for clarity of presentation. First, the (STinvoke) rule enforces that a variable may be passed only once as an argument to a method call. For example, $x_1.\mathsf{m}(x_2, x_2)$ would never type check because x_2 is passed as the argument for two method parameters. Type checking duplicated arguments like these adds substantial complexity to the type system and the type safety proof specifically because method parameters change state. For instance, suppose that m were declared as $\mathsf{Void}\ \mathsf{m}(T_1 \gg T_2, T_1 \gg T_3)[T \gg T]$, where $T_2 \neq T_3$. Then the question arises: what is the type of x_2 after the method call? One could define a sensible merging of T_2 and T_3 as its output type. However, this is not sufficient because when proving type preservation, a single indirect reference would be substituted for two different method parameters into a method body that was type checked using two independent variables. One solution in the formalism is to use a generalized form of merge to temporarily split a reference into two references for the dynamic extent of the method call.

The second simplification we make to method calls is that when a method call takes a variable argument, it drops permissions on the floor. For example, consider the method call $x_1.\mathsf{m}(x_2)$ where x_2 has type $\mathsf{full}(\mathsf{Object})\mathsf{C}$ but the method is declared as $\mathsf{Void}\ \mathsf{m}(\mathsf{pure}(\mathsf{Object})\mathsf{C} \gg \mathsf{pure}(\mathsf{Object})\mathsf{C})[T \gg T]$. Then the extra full permission is lost during the call and is not recovered after the method returns. A practical version of this language would allow methods calls to preserve extra permissions so that, for instance, x_2 could recover its full permission. We can use hold to implement this explicitly in our model language: a practical language would integrate hold semantics directly into method calls.

Method overriding. For clarity and simplicity, the language definition provides conservative constraints on what counts as a legal method override. The overriding rule from Figure 6 says that the overriding method's signature must match the superclass method exactly except that the incoming class of the receiver object must be the class in which the override is being declared. One side-effect of this restriction is that calling an overridden method on an object of statically known subclass type can lose type information. For example, consider two classes:

```
class C { Void m()[full(Object) C >> full(Object) C] { ... } }
class D { Void m()[full(Object) D >> full(Object) C] { ... } }
```

Because of the method type restriction, the following code:

```
x = new D();
x.m()
```

results in the type of x being C rather than D. This particular drawback can be rectified by loosening the restriction on the output type of the receiver, but the language benefits much more from a generally broader notion of legal method overrides. In particular, a method can be a legal override of an existing method if:

(1) The input *permission* of the receiver is a superpermission of the overridden method's receiver input permission;
(2) The input *class* of the receiver must match the current class definition, which is therefore a subclass of the overridden method's receiver input class. Note that covariance in the receiver class is standard in object-oriented type systems, and is sound because we dispatch on the receiver.
(3) The output type of the receiver is a subtype of the overridden method's receiver output type;
(4) The input types of the arguments are supertypes of the overridden method's input types;
(5) The output types of the arguments are subtypes of the overridden method's output types; and

(6) The return type is a subtype of the overridden method's return type.

Demotion. The process of demoting environment references at update operations is quite coarse in the current design. Many objects that need not be demoted in particular cases are currently. For example, if an object is updated, then any existing environment variable whose type is unrelated is surely not an alias to the object at hand. As such, it need not be demoted. In addition, methods could also be annotated to indicate that they do not perform state change. Such safe methods need not cause any variables to be demoted when they are called. For simplicity of presentation, we demote uniformly.

Kinds of permissions. The literature on modular typestate checking with permissions (*e.g.* [Bierhoff and Aldrich 2007; Naden et al. 2012]) introduces other kinds of access permissions, such as **none**, which provides no guarantees about the behavior of other aliases, **unique**, which guarantees that there are no other usable aliases, and **immutable**, which guarantees that no one can change the underlying object. Note that the semantics of **none** and **unique** make their state guarantees essentially irrelevant, so each could be limited to **none**(Object) and **unique**(Object) respectively, or alternatively **none** and **unique** could be treated as permissions P rather than access permissions k.

We integrate **full**, **pure**, and **shared** into Featherweight Typestate because they constitute a self-contained and representative set of access permissions, especially in a language that supports state change for aliased objects. The **full** permission embodies the concept of granting a single alias the ability to change state (much like **unique**); the **pure** permission embodies the inability to change state (much like **immutable**); and the **shared** permission characterizes support for multiple sources of state change. The other permissions described above can all be integrated into Featherweight Typestate without any additional machinery.

In general, as a 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 [Boyland 2003; Boyland and Retert 2005; Naden et al. 2012]. We believe that hold is a simple but expressive means of recovering permissions, and is complementary to these more sophisticated but complex mechanisms.

Syntactic sugar. In addition to increasing expressiveness, a practical language could also implement some convenient shorthands that would make programs more concise while retaining their expressiveness and precision. For example, many method arguments are likely to have the same incoming and outgoing type. A language can abbreviate this idiom by allowing a single type parameter specification T x to be equivalent to an identical type transition specification T $\gg$ T x.

A practical typestate-oriented language could easily simplify the presentation of class field types. Since field types must be invariant under demotion, any field with pure or shared access permission has the same class assumption and state guarantee, e.g. shared(C) C. In these cases, a field type can be abbreviated to include the access permission and a single class, e.g. shared C. In the case of full, the state guarantee must be specified to be precise, though the same abbreviation could have the same meaning as a common case.

Unicity of typing. As with many object-oriented languages, the FT expressions are not uniquely typed. In particular, because of subtyping, most values could be assigned many possible types. This absence of type unicity can be traced specifically to the variable reference rule (STvar), which can assign to a variable reference any subtype of that variable's current type. In most cases, however, the type of a variable reference is restricted by the surrounding context.

To see these phenomena in practice, consider the following program:

$$\mathsf{let}\ \mathsf{x} : \mathsf{full}(\mathsf{Object})\ \mathsf{C} = \mathsf{y}\ \mathsf{in}\ \mathsf{x}$$

The type annotation on x's declaration restricts how (STvar) applies to y: the type of this reference must match the annotation. On the other hand, no such annotation constrains the reference to x in the body of the let. Treated as an entire program, this whole expression could be assigned any subtype of x. In fact, because programs are in A-normal form, this flexibility of typing manifests only for the top-level program type.

Even with programs in A-normal form, it is possible to extend Featherweight Typestate to have even more flexible typing. Such changes do not increase the expressive power of the language, but they do make some programs more convenient to write. First, type annotations on let-bindings could be elided from the language, thereby requiring the type system to guess a type for each variable. This kind of design would increase the nondeterminism of typing. Consider its effect on the program above:

$$\mathsf{let}\ \mathsf{x} = \mathsf{y}\ \mathsf{in}\ \mathsf{x}$$

As before, the reference to x can be typed many ways. However, the type of x is no longer fixed when it is declared, so the reference to y can be typed many ways, and y's output type varies accordingly.

Second, a full subsumption rule could be added to the language:

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

Its effect would be to allow any expression, not just variable references, to be typed many ways.

These two proposed changes to the type system, and their increase in nondeterminism of typing, add no significant expressive power to the type system. Once a method body is type checked, any extra permissions left over are either discarded (in the case of local variables) or adjusted to match the method interface specification (in the case of method arguments). This means that adding subsumption has no effect on the set of typeable FT programs. Since permissions are simply a type-checking device, and have no effect on runtime behavior in FT, there is no particular need for subsumption.

We find in the next section that such nondeterminism in typing is incompatible with a runtime treatment of permissions, which is needed to support gradual typing. In that context we depend on the determinism of typing that comes from the design presented in this section.

4. GRADUAL FEATHERWEIGHT TYPESTATE

Despite its sophistication, Featherweight Typestate cannot statically typecheck all typestate-oriented programs that one might want to write. In this section, we present Gradual Featherweight Typestate (GFT), a gradually typed [Siek and Taha 2007] extension of Featherweight Typestate. GFT seamlessly enhances FT's static type system with support for dynamic typestate checking. To support GFT, we extend concepts of gradual typing to encapsulate the sophistication of permissions, typestate change, and modular flow-sensitive typing.

4.1. Considerations

The design of Gradual Featherweight Typestate is driven by several interacting forces. Here, we outline three primary observations that inform how we extend Featherweight Typestate.

4.1.1. Dynamic typing. The most visible feature of a gradually typed programming language is the presence of dynamically typed values. To support this, GFT adds a dynamic type:

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

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

A Dyn typed value is quite different from an object with Object type. This can be seen by looking at programs (in the sugared syntax of Section 2) that are legal with a Dyn value and not legal with an Object value:

```
full(Object) Object y = ...;
Dyn ydyn = y; // y's type does not change
shared(Object) Object ystc = y; // y's type changes

full(Object) Object xs1 = ystc; // Type error
full(Object) Object xs2 = ydyn; // Okay

ystc.f(y); // Type error: no method f
ydyn.f(y); // Okay

// Void f([full(Object) Object >> pure(Object) Object]) [T >> T]
m.f(ystc); // Type error: incompatible permissions
m.f(ydyn); // ydyn now has type pure(Object) Object
```

Each of the scenarios above captures a difference between static types and dynamic types. Assigning a statically typed variable `y` to a dynamically typed variable `ydyn` does not change `y`'s permissions. As seen in Section 2, this is not generally true for static types. Furthermore, assigning `y` to `ystc` may fail if, for example `y` were **`pure`**. Conversely, a dynamic variable can be assigned to any other variable, regardless of its type: safety is checked at runtime. However, assigning a static variable to another static variable is always checked. Next, method calls on dynamic objects are always safe, and any arguments are treated as dynamic. This is not the case for static method calls. Finally, static method calls on static objects are checked for conformance. On the other hand, a dynamic object can always be passed as an argument to a method call. Note, however, that the type of the dynamic object after the method call matches the method declaration. Newly discovered static information is not automatically discarded, but as we show below, a program can choose to discard this type information.

On the surface, adding **`Dyn`**, a single syntactic difference, is the only necessary addition for gradual typing, but this small interface change implies substantial underlying formal and implementation machinery, which we outline in this section. The fact that it is almost trivial syntactically is one of the great strengths of gradual typing.

4.1.2. Type assertions. Runtime type tests are at the heart of gradual typing, though they need not appear in the surface syntax of a gradual language. However, type tests in the form of casts are a standard feature of object-oriented programming. As discussed earlier, Featherweight Typestate's assert operation is analogous to traditional object-oriented language support for type casting, but FT does not track runtime information about permissions. For this reason, FT assertions cannot manipulate variable permissions. Since GFT requires runtime permission information to support gradual typing, we can expose them at the source language by extending the semantics of assert to manipulate the full type of an object reference, not just its class. For instance, using assert the method call example from Section 4.1.1 can be extended to revert the `ydyn` variable back to **`Dyn`**.

```
m.f(ydyn); // ydyn now has type pure(Object) Object
assert<Dyn>(ydyn); // ydyn now has Dyn type.
```

4.1.3. Dynamic Permissions Need Deterministic Typing. In Section 3.7, we observed that Featherweight Typestate's type system could be made more nondeterministic by removing type annotations on let-bound variables and by adding full subsumption. This kind of nondeterminism would be problematic for the semantics of a gradual language that depends on dynamic permission tracking. To understand this phenomenon, consider the following hypothetical example in a gradual language with the above extensions. Suppose x has type $\mathsf{full}(D)\ D$, and that class C has one field of type $\mathsf{pure}(D)\ D$, and consider the following expression:

$$\begin{array}{l}\mathsf{let}\ \mathsf{y} = \mathsf{x}\ \mathsf{in} \\ \mathsf{let}\ \mathsf{z} = \mathsf{new}\ \mathsf{C}(\mathsf{y})\ \mathsf{in}\ \mathsf{z}\end{array}$$

What are the types of x and y at the end? The answer depends on what type was given to the x reference when it was bound to y. If x was given type $\mathsf{full}(D)\ D$, then x would have type $\mathsf{pure}(D)\ D$ and y would have type $\mathsf{full}(D)\ D$; but if the reference to x was given type $\mathsf{pure}(D)\ D$, then the reverse would be true: x would have type $\mathsf{full}(D) D$ and y would have type $\mathsf{pure}(D)\ D$; finally if the reference to x were given type $\mathsf{shared}(D)\ D$, then both x and y would end up with that type.

This flexibility allows many more programs to be typed without the programmer having to annotate every variable binding, or change those annotations as the program changes, but such nondeterminism is incompatible with dynamic permission assertions. Suppose we extend the example with a dynamic assertion:

$$\begin{array}{l}\mathsf{let}\ \mathsf{y} = \mathsf{x}\ \mathsf{in} \\ \mathsf{let}\ \mathsf{z} = \mathsf{new}\ C(\mathsf{y})\ \mathsf{in} \\ \mathsf{let}\ \mathsf{w} = \mathsf{assert}\langle \mathsf{shared}(D)\ D\rangle(\mathsf{y})\ \mathsf{in}\ \mathsf{z}\end{array}$$

Then the behavior of this example depends on how the types are resolved. If y has shared or full access permission, then the assertion is a safe "upcast" that always succeeds; if y ends up with pure permission, then the assertion is a "downcast" that must be checked dynamically (and in this case fails because x's full permission is not compatible with a shared alias).

These issues do not arise in FT because it cannot check permissions dynamically. As such it only needs to find *some* valid typing, after which the permission information is discarded for runtime. Gradual typing, on the other hand, can detect how permissions flow in a program at runtime, so permissions must have some deterministic specification if gradually typed programs are to behave deterministically. In the following development, we leverage the fact that FT typing is more deterministic than strictly necessary to support dynamic permissions and thereby support gradual typing as a pure extension.[7]

4.2. Making Featherweight Typestate Gradual

Now that we have brought to light the primary challenges of developing a gradually typed typestate-oriented language like Gradual Featherweight Typestate, we can provide an overview of the language and describe how its design addresses these considerations.

[7] As shown in [Wolff et al. 2011], a typestate-oriented language can simultaneously enjoy deterministic typing and low annotation overhead.

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

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

Fig. 10: Hybrid Permission Management Relations

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

$$\text{(GTvar}_d\text{)}\ \frac{}{\Delta, x : \mathsf{Dyn} \vdash x : T \dashv \Delta, x : \mathsf{Dyn}} \qquad \text{(GTupdate}_d\text{)}\ \frac{\begin{array}{c} \mathit{fields}(C) = \overline{T\ f} \\ \Delta \vdash \overline{x_2 : T} \dashv \Delta', x_1 : \mathsf{Dyn} \end{array}}{\Delta \vdash x_1 \leftarrow C(\overline{x_2}) : \mathsf{Void} \dashv \Delta'{\downarrow}, x_1 : \mathsf{Dyn}}$$

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

$$\text{(GTswap}_d\text{)}\ \frac{\Delta, x_1 : \mathsf{Dyn} = \Delta', x_2 : T}{\Delta, x_1 : \mathsf{Dyn} \vdash x_1.f \ :=: \ x_2 : \mathsf{Dyn} \dashv \Delta', x_2 : \mathsf{Dyn}}$$

$$\text{(GTinvoke}_d\text{)}\ \frac{}{\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) : \mathsf{Dyn} \dashv \Delta{\downarrow}, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}}$$

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

Fig. 11: Gradual Featherweight Typestate: Expression Typing Extensions

Aside from the introduction of a dynamic type Dyn, the syntax of GFT is the same as that of FT. The key extensions to the language can be found in its typing rules and its runtime semantics.

4.2.1. Managing Permissions. Now that the Dyn type has been introduced to the language, we must consider how it interacts with the family of type operations that supports typestate-oriented programming

Figure 10 presents the necessary adjustments. 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. This captures the intuition that dynamically typed objects do not intrinsically carry any permissions.

Following Siek and Taha [2007], we replace subtyping in our rules with a notion of *consistent subtyping* $T \lesssim T$. Consistent subtyping is the union of the notion of *type consistency* $T \sim T$ from gradual typing—which codifies *possibly* safe substitution—with the notion of subtyping $T <: T$ for Featherweight Typestate—which codifies *definitely* safe substitutability. According to consistent subtyping, Dyn $\lesssim T$, and also $T \lesssim$ Dyn because modified type splitting now forces $T <:$ Dyn (See Section 4.5). We restrict the rules to ensure determinism, which facilitates our translation semantics.

4.2.2. Static Semantics. The fundamental differences between Featherweight Typestate and Gradual Featherweight Typestate are found in its type system. All of FT's typing rules are valid for GFT, so Figure 11 presents only the extensions that GFT adds to FT's type system (all prefixed with "GT": G for "gradual typing" and T for "typing").

The (GTassert) rule for assert subsumes the analogous rule in Featherweight Typestate, though now it considers and affects the entire type of its argument, including in particular the permissions associated with an object. When $T_1 <: T_2$, the assert is statically safe; otherwise, a runtime check is required (see Section 4.4).

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

Fig. 12: Internal Language Syntax

The full language adds new typing rules for each operation in the case when the primary object being operated on is dynamically typed. The rest of the new typing rules account for how Dyn-typed references to objects can be used, as well as their effect on permissions and type information. The (GTvar$_d$) rule says that a Dyn-typed variable can be referenced at any type. Note that because of our extensions to type splitting, x : Dyn can already be typed at Dyn using FT's (STvar) rule. The (GTupdate$_d$) 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 4.4). The (GTfield$_d$) rule says that accessing a field of a dynamic object yields another dynamic object (if it succeeds). The (GTswap$_d$) rule allows an object to be swapped into the field of a dynamic object. Permissions are checked at runtime for safety. Finally, the (GTinvoke$_d$) rule calls a method with objects of any type. However, the output type of the method's arguments are all dynamic, since the effect on their permissions cannot be known until runtime.

4.3. Internal Language

Gradually typed languages are characterized in terms of three languages: a fully statically typed language, the gradually typed language itself, and the internal implementation language. For instance, the original work on gradual typing presented the simply-typed lambda calculus, the gradual lambda calculus, and the cast calculus as the necessary three components [Siek and Taha 2006]. Here we have already presented the first two components: Featherweight Typestate and Gradual Featherweight Typestate. We must now introduce our analogue to the cast calculus.

The semantics of GFT are defined by type-directed translation to GFTIL, an internal language that makes the details of dynamic permission management explicit. This section presents the syntax, type system, and dynamic semantics of the internal language. Section 4.4 discusses how the source language is mapped to it.

4.3.1. Syntax. GFTIL is structured much like GFT but elaborates several concepts (Figure 12). 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 the same runtime constructs as were added to Featherweight Typestate: object references, indirect references, and the void object.

In GFTIL, 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.

The notion of a well-typed GFTIL program (see Appendix C) is almost identical in form to that notion in FT. One notable difference is the typing of method bodies: since GFTIL explicitly tracks resources, it requires a method's returned value as well as the output states of all its parameters (and this) to exactly match the method signature, for which $\mathsf{release}[T](s)$ is introduced. In contrast, both FT and GFT allow subtyping to implicitly fill the gap.

4.3.2. Static Semantics. The rules for GFTIL's typing judgment $\Delta \vdash e : T \dashv \Delta$ are defined using 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 Δ to force their arguments s to have *exactly* the type required. GFTIL's dynamic semantics uses this encoding to track permissions.

Figure 13 presents some of GFTIL's typing rules (rules are prefixed with TI for "Typing" the "Internal language"). For brevity, we only present the rules for invoke, update and assert, together with their dynamically-typed variants here: the full set can be found in Appendix C. The (TIinvoke) 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 (TIupdate) rule almost mirrors GFT's update rule except that its argument types must exactly match the class field specifications. The (TIassert) 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 (TIupdate$_d$) rule only checks that the arguments match the constructor, and the (TIassert$_d$) rule applies when the destination type cannot be split from the source type. The (TIhold) rule is the explicit analogue to the GFT typing rule. The (TImerge) 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.

4.3.3. Dynamic Semantics. The dynamic semantics of GFTIL, presented in Figure 14, depend on the same runtime structures as Featherweight Typestate: environments ρ and stores μ. One significant difference, though, is that GFTIL heaps map object references to *tracked* objects:

$$\begin{aligned} C(\overline{o})\ \overline{P} &\in \textsc{TrackedObjects} \\ \mu &\in \textsc{ObjectRefs} \rightharpoonup \textsc{TrackedObjects} \quad \text{(stores)} \end{aligned}$$

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 collection of outstanding permissions for references to that object, $\overline{P}$. The dynamic semantics of GFTIL is defined as transitions between store/environment/expression triples.

Figure 14 presents some select dynamic semantics rules of GFTIL (prefixed with GE, for "Gradual typing" and "Evaluation"). Certain rules use two helper functions for tracking permissions in the heap, whose definitions are given in Figure 15. 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. Both operations take an arbitrary value and type, but be-

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

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

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

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

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

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

$$(\text{TIhold})\ \frac{T_1 \Rrightarrow T_2/T_3 \quad T_2{\downarrow}\,/T_3' \Rrightarrow T_1' \quad \Delta, s : T_3 \vdash e : T \dashv \Delta_1, s : T_3'}{\Delta, s : T_1 \vdash \mathsf{hold}[s : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) : T \dashv \Delta_1, s : T_1'}$$

$$(\text{TImerge})\ \frac{T_1 = T_1{\downarrow} \quad T_1/T_2' \Rrightarrow T_3 \quad \Delta, l_2 : T_2 \vdash e : T \dashv \Delta_1, l_2 : T_2'}{\Delta, l_1 : T_1, l_2 : T_2 \vdash \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](e) : T \dashv \Delta_1, l_2 : T_3}$$

Fig. 13: Select Internal Language Typing Rules

have like identity when presented with a type that does not represent a permission, like Void or Dyn. The (GEinvoke) rule is straightforward. The (GEupdate) 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 (GEassert) rule uses permission addition and subtraction to track permissions, and returns void. Rules for dynamic operators, like (GEinvoke$_d$) and (GEupdate$_d$), dynamically assert the necessary permissions (using assert$_\mathsf{d}$), defer to the corresponding static operation, and then statically release the acquired permission (using assert). The (GEassert$_d$) rule confirms dynamically that its type assertion is safe. The (GEhold) rule performs the splitting of permissions (one permission to be used through the execution of the subexpression, and one to be held around it), and evaluates to a merge. 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. Finally, the (GEmerge) rule applies when the subexpression is fully evaluated, and roughly reverses the (GEhold) rule. It merges the held type of l' with the type of its alias l, and updates the store accordingly. Note that after this point, the indirect reference l' is no longer in scope.

4.3.4. Type safety. As for FT, the type safety proof of GFTIL must account for the outstanding permissions for each object o and verify that they are mutually compatible. Figure 16 presents representative updates to FT's permission accounting operations needed for GFTIL. The basic reference type operations must be updated to filter out the Dyn type and to expect tracked objects rather than just objects. The most important difference, though, is that when checking reference consistency, that an object is **ok** with respect to a context-environment-heap triple, it's now necessary to check that the heap is properly tracking permissions.

The definition of global consistency does not change from that of FT. Recall that under global consistency, every reference in the type context is accounted for in the store and environment, and that Void and object-typed indirect references ultimately point

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

$$\text{(GEinvoke)}\frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \\ \mathit{method}(m, C) = T_r\ m(\overline{T_i \gg T_i'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}\end{array}}{\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e}$$

$$\text{(GEinvoke}_d\text{)}\frac{\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \quad \mathit{mdecl}(m, C) = T_r\ m(\overline{T_i \gg T_i'})\ [T_t \gg T_t'] \quad |\ \overline{T_i}\ | = |\ \overline{l_2}\ |}{\begin{array}{l}\mu, \rho, l_1 ._d m(\overline{l_2}) \rightarrow \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t \rangle(l_1);\ \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_i \rangle(l_2);} \\ \qquad \mathsf{let}\ \mathit{ret} = l_1.m(\overline{l_2})\ \mathsf{in}\ \mathsf{assert}\langle T_t' \gg \mathsf{Dyn}\rangle(l_1); \\ \qquad \overline{\mathsf{assert}\langle T_i' \gg \mathsf{Dyn}\rangle(l_2);}\ \mathsf{assert}\langle T_r \gg \mathsf{Dyn}\rangle(\mathit{ret}); \\ \qquad \mathit{ret}\end{array}}$$

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

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

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

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

$$\text{(GEhold)}\frac{\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3 \quad l' \notin \mathit{dom}(\rho) \quad \rho' = \rho[l' \mapsto \rho(l)]}{\mu, \rho, \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) \rightarrow \mu', \rho', \mathsf{merge}[l' : T_2{\downarrow}\ /l : T_3' \Rrightarrow T_1'](e)}$$

$$\text{(GEmerge)}\frac{\mu' = \mu - \rho(l') : T_1 - \rho(l) : T_2 + \rho(l) : T_3}{\mu, \rho, \mathsf{merge}[l' : T_1/l : T_2 \Rrightarrow T_3](v) \rightarrow \mu', \rho, v}$$

$$\text{(GEmcongr)}\frac{\mu, \rho, e \rightarrow \mu', \rho', e'}{\mu, \rho, \mathsf{merge}[l_1 : T/l_2](e) \rightarrow \mu', \rho', \mathsf{merge}[l_1 : T/l_2](e')}$$

Fig. 14: Select Internal Language Dynamic Semantics Rules

$\boxed{\mu = \mu + v : T}$ Permission Addition

$$\frac{T \in \{\mathsf{Dyn}, \mathsf{Void}\}}{\mu = \mu + v : T}$$

$$\frac{\mu(o) = C(\overline{o_f})\ \overline{P}}{\mu[o \mapsto C(\overline{o_f})\ \overline{P}, P'] = \mu + o : P'\ C'}$$

$\boxed{\mu = \mu - v : T}$ Permission Subtraction

$$\frac{\mu = \mu' + v : T}{\mu' = \mu - v : T}$$

Fig. 15: Internal Dynamics Auxiliary Functions

Helper Functions

$$\mathit{fieldTypes}(\mu, o) \;=\; \underset{o' \in dom(\mu)}{+\!\!+} \left[T_i \;\boxed{\neq \mathsf{Dyn}}\; \mid \mu(o') = C(\overline{o''}) \;\boxed{\overline{P_2}}\,,\; \mathit{fields}(C) = \overline{T\;f},\; o''_i = o,\; \text{and } T_i \neq \mathsf{Dyn} \right]$$

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

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

Fig. 16: Changes to Permission-Consistency Relations

to void values and object references respectively. In extending to GFT, Dyn-typed references can be ignored because they may point to anything. 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 4.1 (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 4.2 (PRESERVATION). *If $\Delta \vdash e : T \dashv \Delta'$, and μ, Δ, ρ **ok**, and $\rho \vdash e$ **mc**, and $\mu, \rho, e \to \mu', \rho', e'$, then $\Delta'' \vdash e' : T \dashv \Delta'$ and μ', Δ'', ρ' **ok**, and $\rho' \vdash e'$ **mc** for some Δ''.*

4.4. Source to Target Translation

The dynamic semantics of GFT are defined by augmenting its type system to generate GFTIL expressions. The typing judgment becomes $\Delta \vdash e_1 : T \rightsquigarrow e_2^{\mathcal{I}} \dashv \Delta'$, where e_1 is a GFT expression and $e_2^{\mathcal{I}}$ is its corresponding GFTIL expression. Figure 17 presents these rules. We use the $\mathcal{I}$ superscript to disambiguate GFTIL expressions as needed. Several rules use the *coerce* partial function, which translates consistent subtyping judgments $T \lesssim T$ into variable assertions:

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

Most of the translations are straightforward, and follow similar patterns. For instance, the (TRupdate) 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 (TRupdate$_d$) 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 (TRassert) simply

$\boxed{\Delta \vdash e : T \leadsto e^{\mathcal{I}} \dashv \Delta}$ Source to Internal Language Translation

$$\text{(TRvar)}\ \dfrac{T_1 \Rrightarrow T_2/T_3}{\begin{array}{l}\Delta, x : T_1 \vdash x : T_2 \leadsto \\ \quad x[T_1 \Rrightarrow T_2/T_3] \dashv \Delta, x : T_3\end{array}} \qquad \text{(TRvar}_d\text{)}\ \dfrac{T \neq \mathsf{Dyn}}{\begin{array}{l}\Delta, x : \mathsf{Dyn} \vdash x : T \leadsto \mathsf{let}\ ret = x[\mathsf{Dyn} \Rrightarrow \mathsf{Dyn}/\mathsf{Dyn}]\ \mathsf{in} \\ \qquad \mathsf{assert_d}\langle \mathsf{Dyn} \gg T\rangle(ret); \\ \qquad ret \dashv \Delta, x : \mathsf{Dyn}\end{array}}$$

$$\text{(TRlet)}\ \dfrac{\begin{array}{c}\Delta \vdash e_1 : T_1 \leadsto e_1^{\mathcal{I}} \dashv \Delta_1 \\ \Delta_1, x : T_1 \vdash e_2 : T_2 \leadsto e_2^{\mathcal{I}} \dashv \Delta', x : T_1'\end{array}}{\begin{array}{l}\Delta \vdash \mathsf{let}\ x : T_1 = e_1\ \mathsf{in}\ e_2 : T_2 \leadsto \\ \quad \mathsf{let}\ x = e_1^{\mathcal{I}}\ \mathsf{in} \\ \qquad \mathsf{let}\ ret = e_2^{\mathcal{I}}\ \mathsf{in} \\ \qquad\quad \mathsf{release}[T_1'](x); ret \dashv \Delta'\end{array}} \qquad \text{(TRnew)}\ \dfrac{\mathit{fields}(C) = \overline{T\ f} \quad \Delta \vdash \overline{x : T \leadsto e^{\mathcal{I}}} \dashv \Delta'}{\begin{array}{l}\Delta \vdash \mathsf{new}\ C(\overline{x}) : \mathsf{full}(\mathsf{Object})\ C \leadsto \\ \qquad \overline{\mathsf{let}\ x' = e^{\mathcal{I}}\ \mathsf{in}}\ \mathsf{new}\ C(\overline{x'}) \dashv \Delta'\end{array}}$$

$$\text{(TRinvk)}\ \dfrac{\begin{array}{c} mdecl(m, C_1) = T\ m(\overline{T_i \gg T_i'})[T_t \gg T_t'] \\ coerce(x_1, P_1\ C_1, T_t) = e_1^{\mathcal{I}} \\ \overline{coerce(x_2, T_2, T_i) = e_2^{\mathcal{I}}}\end{array}}{\begin{array}{l}\Delta, x_1 : P_1\ C_1, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) : T \leadsto \\ \quad e_1^{\mathcal{I}};\ \overline{e_2^{\mathcal{I}}};\ x_1.m(\overline{x_2}) \dashv \Delta\!\downarrow, x_1 : T_t', \overline{x_2 : T_i'}\end{array}} \qquad \text{(TRinvk}_d\text{)}\ \dfrac{\overline{coerce(x_2, T_2, \mathsf{Dyn}) = e_2^{\mathcal{I}}}}{\begin{array}{l}\Delta, x_1 : \mathsf{Dyn}, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) : \mathsf{Dyn} \leadsto \\ \quad \overline{e_2^{\mathcal{I}}};\ x_1 ._d m(\overline{x_2}) \dashv \Delta\!\downarrow, x_1 : \mathsf{Dyn}, \overline{x_2 : \mathsf{Dyn}}\end{array}}$$

$$\text{(TRswap)}\ \dfrac{\begin{array}{c} T_2\ f \in \mathit{fields}(C_1) \\ \Delta, x_1 : P_1\ C_1 \vdash x_2 : T_2 \leadsto e_2^{\mathcal{I}} \dashv \Delta'\end{array}}{\begin{array}{l}\Delta, x_1 : P_1\ C_1 \vdash x_1.f\ :=:\ x_2 : T_2 \leadsto \\ \quad \mathsf{let}\ x_2' = e_2^{\mathcal{I}}\ \mathsf{in}\ x_1.f\ :=:\ x_2' \dashv \Delta'\end{array}} \qquad \text{(TRswap}_d\text{)}\ \dfrac{\Delta, x_1 : \mathsf{Dyn} = \Delta', x_2 : T}{\begin{array}{l}\Delta, x_1 : \mathsf{Dyn} \vdash x_1.f\ :=:\ x_2 : \mathsf{Dyn} \leadsto \\ \quad \mathsf{assert}\langle T \gg \mathsf{Dyn}\rangle(x_2); \\ \quad \mathsf{let}\ x_2' = x_2[\mathsf{Dyn} \Rrightarrow \mathsf{Dyn}/\mathsf{Dyn}]\ \mathsf{in} \\ \quad\ x_1.f\ :=:_d\ x_2' \dashv \Delta', x_2 : \mathsf{Dyn}\end{array}}$$

$$\text{(TRupdate)}\ \dfrac{\begin{array}{c} \mathit{fields}(C) = \overline{T\ f} \\ \Delta \vdash \overline{x_2 : T \leadsto e_2^{\mathcal{I}}} \dashv \Delta', x_1 : k(D)\ E \\ k \in \{\mathsf{full}, \mathsf{shared}\} \qquad C <: D\end{array}}{\begin{array}{l}\Delta \vdash x_1 \leftarrow C(\overline{x_2}) : \mathsf{Void} \leadsto \\ \overline{\mathsf{let}\ x_2' = e_2^{\mathcal{I}}\ \mathsf{in}} \\ \quad x_1 \leftarrow C(\overline{x_2'}) \dashv \Delta'\!\downarrow, x_1 : k(D)\ C\end{array}} \qquad \text{(TRupdate}_d\text{)}\ \dfrac{\begin{array}{c} \mathit{fields}(C) = \overline{T_2\ f} \\ \Delta \vdash \overline{x_2 : T_2 \leadsto e_2^{\mathcal{I}}} \dashv \Delta', x_1 : \mathsf{Dyn}\end{array}}{\begin{array}{l}\Delta \vdash x_1 \leftarrow C(\overline{x_2}) : \mathsf{Void} \leadsto \\ \overline{\mathsf{let}\ x_2' = e_2^{\mathcal{I}}\ \mathsf{in}} \\ \quad x_1 \leftarrow_d C(\overline{x_2'}) \dashv \Delta'\!\downarrow, x_1 : \mathsf{Dyn}\end{array}}$$

$$\text{(TRfield)}\ \dfrac{T_2\ f \in \mathit{fields}(C_1) \qquad T_2 \Downarrow T_2'}{\begin{array}{l}\Delta, x : P_1\ C_1 \vdash x.f : T_2' \leadsto \\ \quad x.f \dashv \Delta, x : P_1\ C_1\end{array}} \qquad \text{(TRfield}_d\text{)}\ \dfrac{}{\begin{array}{l}\Delta, x : \mathsf{Dyn} \vdash x.f : \mathsf{Dyn} \leadsto \\ \quad x._d f \dashv \Delta, x : \mathsf{Dyn}\end{array}}$$

$$\text{(TRassert)}\ \dfrac{T \Rrightarrow T'}{\begin{array}{l}\Delta, x : T \vdash \mathsf{assert}\langle T'\rangle(x) : \mathsf{Void} \leadsto \\ \quad \mathsf{assert}\langle T \gg T'\rangle(x) \dashv \Delta, x : T'\end{array}} \qquad \text{(TRassert}_d\text{)}\ \dfrac{T \not\Rrightarrow T'}{\begin{array}{l}\Delta, x : T \vdash \mathsf{assert}\langle T'\rangle(x) : \mathsf{Void} \leadsto \\ \quad \mathsf{assert_d}\langle T \gg T'\rangle(x) \dashv \Delta, x : T'\end{array}}$$

$$\text{(TRhold)}\ \dfrac{\begin{array}{c} T_1 \Rrightarrow T_2/T_3 \qquad T_2\!\downarrow / T_3' \Rrightarrow T_1' \\ \Delta, x : T_3 \vdash e : T \leadsto e^{\mathcal{I}} \dashv \Delta', x : T_3'\end{array}}{\begin{array}{l}\Delta, x : T_1 \vdash \mathsf{hold}[x : T_2](e) : T \leadsto \\ \quad \mathsf{hold}[x : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e^{\mathcal{I}}) \\ \qquad \dashv \Delta', x : T_1'\end{array}}$$

Fig. 17: Type-directed Translation from GFT to GFTIL

use the typing rule to expose the needed extra type annotations for the corresponding GFTIL expression. The (TRhold) rule specifies how the source-level hold is translated to the internal expression, which is fully annotated with the intermediate types used in the derivation.

As intended, the translation rules preserve well-typing:

THEOREM 4.3 (TRANSLATION SOUNDNESS). *If* $\Delta \vdash e : T \leadsto e^{\mathcal{I}} \dashv \Delta'$ *then* $\Delta \vdash e^{\mathcal{I}} : T \dashv \Delta'$.

This theorem extends straightforwardly to whole programs.

4.5. Discussion

In Featherweight Typestate, permissions are a compile-time phenomenon and need not be represented at runtime. However, permissions are an integral component of FT types, so being able to reason about them at runtime is critical to support the dynamic type checking that is at the heart of gradual typing. For this reason, Gradual Featherweight Typestate is designed to support runtime tracking and querying of permissions.

In order to achieve this combination of static and dynamic typestate checking, several challenges needed to be overcome. First, given that the language includes objects whose type changes over time, it is necessary to determine what might be a reasonable behavior for dynamically typed objects. Since dynamically typed objects include object references that would otherwise have permissions associated with them, it was necessary to introduce a notion of runtime-checked permissions, a feature that could also be applied to purely dynamically typed typestate-oriented languages. Nonetheless, this change alone necessitated removing non-determinism from the type system of FT, while still providing a convenient programming model.

Once runtime permission tracking and dynamic assertions are added, the introduction of the Dyn type of gradual typing can be viewed as a pure language extension, since any program with no Dyn types falls in the non-gradual subset of the language. To keep the development simple, our presentation introduces gradual typing by making some modifications to the existing permission management operations and typing rules. However, the Dyn type could have been introduced to GFT as a pure extension atop the language with dynamic type assertions. First, we could have preserved a full separation between dynamic typing and type-splitting/subtyping by only specifying that $\mathsf{Dyn} \Rightarrow \mathsf{Dyn}/\mathsf{Dyn}$, which is standard for any type that does not track permissions (like Void). We could then have introduced a distinct notion of *dynamic type splitting* $T \sim T/T$ solely for handling the special properties of the Dyn type. Its two rules would be $T \sim \mathsf{Dyn}/T$ and $\mathsf{Dyn} \sim T/\mathsf{Dyn}$. The type system could then be extended with special rules for checking variables and complex expressions at Dyn, as well as checking Dyn-typed variables at non-dyn types. Furthermore, we could define *consistent type splitting* as the union of standard type splitting and dynamic type splitting. This would lead to the definition of consistent subtyping that we ultimately used, though by a more circuitous route. We found it simpler to allow Dyn to be the head of the subclass hierarchy and then extend subtyping to consistent subtyping directly.

In Featherweight Typestate, hold is a purely static notion, and supports the permission-based type discipline, but is not needed at runtime. In a gradually-typed setting, however, we must account for temporarily-held permissions at runtime, so both hold and merge have GFTIL counterparts that implement the necessary permission bookkeeping. Compared to prior work on borrowing, the semantics of **hold** is novel in two ways that can be ascribed to its straightforward and effective integration with gradual typing. First, in order to provide a static guarantee that the held permissions remain valid, **hold** must do runtime bookkeeping to ensure that the code inside the nested block does not assert an incompatible permission. Second, hold does not always restore the exact original permission; rather, it agnostically the held permission with the available pending permissions. Because of dynamic assertions that can occur within the nested block, the merged permission may be stronger or weaker than the original permissions. Borrowing has to date been conceived only in a static context, and it recovers exactly the permissions that were loaned to a function call. It remains to be explored how borrowing interacts with dynamic permission assertion.

5. GFT SIMPLY EXTENDS FT

The prior sections present two source languages, Featherweight Typestate and Gradual Featherweight Typestate, as well as type systems and operational semantics for both. However, despite the presence of two separate operational semantics, we claim that GFT is simply an extension of the FT language, with support for gradual typing and dynamic permission management. This section clarifies the sense in which this is so.

We start with the syntax and static semantics of these languages. As discussed in Section 4, FT is syntactically a subset of GFT, with the only extension being the addition of the Dyn type. Furthermore, GFT's type system accepts all FT programs. So the syntax and static semantics of the two languages are in sync.

From here, however, things appear to diverge. We give FT a direct operational semantics. On the other hand, GFT is defined by type-directed translation to GFTIL, an intermediate language that is given its own operational semantics, independent of that of FT.

To complete the connection between FT and GFT, we bridge the difference between these operational semantics. In particular, since every FT program is also a GFT program, we show that translating an FT program to GFTIL and then running it produces the same behavior as running the FT program directly.

The key observation underlying this connection is that many GFTIL expressions are designed to maintain proper permission accounting so that information may be queried whenever runtime permission checks are needed. FT, being a static language, never needs to query runtime permissions (though assert may check class identity in the case of a downcast). Furthermore, as shown in Section 3.6, indirect references and their environment are irrelevant to the behavior of programs: it's the structure of the heap that matters. Thus, we want to show that FT programs produce the same heap structures when run on the FT semantics and the GFTIL semantics.

The relationship between FT, GFT, and GFTIL programs is reminiscent of the connection between Siek and Taha [2006]'s simply typed, gradually typed, and cast calculus programs. Every simply-typed program is also a gradually-typed program and thus translates to a cast calculus program which has the same semantics. The correspondence between semantics in their system is immediately evident and needs no proof. In our present case, we must account for GFTIL's strict permission tracking and show that it does not affect the behavior of FT programs.

First, we establish what it means for an FT state and a GFTIL state to be in correspondence. We must appeal to the GFT translation for this.

Definition 5.1. Let $\Delta \vdash \mu, \rho, e \sim \mu^{\mathcal{I}}, \rho^{\mathcal{I}}, e^{\mathcal{I}}$ if and only if

(1) μ, Δ, ρ **ok**;
(2) $\mu^{\mathcal{I}}, \Delta, \rho^{\mathcal{I}}$ **ok**;
(3) $\mu = |\mu^{\mathcal{I}}|$;
(4) $\rho \subset \rho^{\mathcal{I}}$;
(5) $\Delta \vdash e : T \rightsquigarrow e_0^{\mathcal{I}} \dashv \Delta_1$;
(6) $e_0^{\mathcal{I}}$ **expands to** $e^{\mathcal{I}}$; and
(7) $\Delta \vdash e^{\mathcal{I}} \dashv \Delta_2$;

The above definition relies on several auxiliary concepts. The $|\mu^{\mathcal{I}}|$ operation converts a GFTIL heap $\mu^{\mathcal{I}}$ to an FT heap by discarding permission information. Also, the relation $e_1^{\mathcal{I}}$ **expands to** $e_2^{\mathcal{I}}$ is defined by the following rules:

$$\text{(release)}\ \frac{e_1^{\mathcal{I}}\ \textbf{expands to}\ e_2^{\mathcal{I}}}{e_1^{\mathcal{I}}\ \textbf{expands to}\ \mathsf{let}\ ret = e_2^{\mathcal{I}}\ \mathsf{in}\ \mathsf{release}[T](l); ret}$$

$$\text{(assert)}\ \frac{e_1^{\mathcal{I}}\ \textbf{expands to}\ e_2^{\mathcal{I}} \qquad \overline{C <: D}}{e_1^{\mathcal{I}}\ \textbf{expands to}\ \mathsf{let}\ ret = e_2^{\mathcal{I}}\ \mathsf{in}\ \overline{\mathsf{assert}\langle P\ C \gg P\ D\rangle(l)}; ret.}$$

$$\text{(refl)}\ \frac{}{e^{\mathcal{I}}\ \textbf{expands to}\ e^{\mathcal{I}}} \qquad \text{(let)}\ \frac{e_1^{\mathcal{I}}\ \textbf{expands to}\ e_2^{\mathcal{I}} \qquad e_3^{\mathcal{I}}\ \textbf{expands to}\ e_4^{\mathcal{I}}}{\mathsf{let}\ x = e_1^{\mathcal{I}}\ \mathsf{in}\ e_3^{\mathcal{I}}\ \textbf{expands to}\ \mathsf{let}\ x = e_2^{\mathcal{I}}\ \mathsf{in}\ e_4^{\mathcal{I}}}$$

This relation accounts for the extra code added by the translation of let expressions and method bodies.

The resulting correspondence $\Delta \vdash \mu, \rho, e \sim \mu^{\mathcal{I}}, \rho^{\mathcal{I}}, e^{\mathcal{I}}$ captures the idea that we can consider an FT and GFTIL state to be in sync if they are the same apart from indirect references and permission tracking steps.

Armed with these definitions, we can establish correspondence.

PROPOSITION 5.2.

(*1*) *If* $\bullet \vdash e : T \rightsquigarrow e^{\mathcal{I}} \dashv \bullet$ *then* $\bullet \vdash \varnothing, \varnothing, e \sim \varnothing, \varnothing, e^{\mathcal{I}}$.
(*2*) *Let* $e_1^{\mathcal{I}}$ *be one of:*
  (*a*) *a value* v*;*
  (*b*) *a reference* $l[T_1 \Rrightarrow T_2/T_3]$*; or*
  (*c*) *an assertion* $\mathsf{assert}\langle T \gg T\rangle(l)$.
  If $e_1^{\mathcal{I}}$ ***expands to*** $e_2^{\mathcal{I}}$, μ, Δ, ρ ***ok****,* $\Delta \vdash e_2^{\mathcal{I}} : T \dashv \Delta'$*, and* $\mu^{\mathcal{I}}, \rho_1^{\mathcal{I}}, e_1^{\mathcal{I}} \longrightarrow^* \mu, \rho_2^{\mathcal{I}}, v$ *then* $\mu^{\mathcal{I}}, \rho_1^{\mathcal{I}}, e_2^{\mathcal{I}} \longrightarrow^* \mu^{\mathcal{I}}, \rho_3^{\mathcal{I}}, v$ *where* $\rho_2 \subset \rho_3$.
(*3*) *If* $\Delta_1 \vdash \mu_1, \rho_1, e_1 \sim \mu_1^{\mathcal{I}}, \rho_1^{\mathcal{I}}, e_1^{\mathcal{I}}$ *and* $\mu_1, \rho_1, e_1 \to \mu_2, \rho_2, e_2$ *then* $\mu_1^{\mathcal{I}}, \rho_1^{\mathcal{I}}, e_1^{\mathcal{I}} \to^* \mu_2^{\mathcal{I}}, \rho_2^{\mathcal{I}}, e_2^{\mathcal{I}}$ *and* $\Delta_2 \vdash \mu_2, \rho_2, e_2 \sim \mu_2^{\mathcal{I}}, \rho_2^{\mathcal{I}}, e_2^{\mathcal{I}}$ *for some* Δ_2.

PROOF SKETCH.

(1) Straightforward
(2) By induction on $e_1^{\mathcal{I}}$ **expands to** $e_2^{\mathcal{I}}$.
(3) By simultaneous induction on $\mu_1, \rho_1, e_1 \to \mu_2, \rho_2, e_2$ and $e_1^{\mathcal{I}}$ **expands to** $e_2^{\mathcal{I}}$.
  Cases (SEassert) and (SEinvoke) make explicit use of well-typed translation. In particular, Some assert expressions in FT translate to assert_d in GFTIL, but they never modify the permissions, only the class.
  To account for the let-bound arguments introduced by translation, cases (SEnew), (SEupdate), and (SEinvoke) appeal to part (2) and use a nested simultaneous induction on the $e_1^{\mathcal{I}}$ **expands to** $e_2^{\mathcal{I}}$ relation and the number of let bindings in $e_1^{\mathcal{I}}$.
  Finally, to properly translate running programs, we extend (TRref) to include indirect references and add the following rules:

$$\text{(TRvoid)}\ \frac{}{\Delta \vdash \mathsf{void} : \mathsf{Void} \rightsquigarrow \mathsf{void} \dashv \Delta} \qquad \text{(TRobj)}\ \frac{}{\Delta, o : T \vdash o : T \rightsquigarrow o \dashv \Delta}$$

$$\text{(STmerge)}\ \frac{T_1 = T{\downarrow} \qquad \Delta, l_2 : T_2 \vdash e : T \vdash \Delta_1, l_2 : T_2' \qquad T_1/T_2' \Rrightarrow T_3}{\Delta, l_1 : T_1, l_2 : T_2 \vdash \begin{array}{c}\mathsf{merge}[l_1 : T_1/l_2](e) : T \rightsquigarrow \\ \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](e)\end{array} \dashv \Delta_1, l_2 : T_3}$$

□

6. CONCLUSION

Related Work. A lot of research has been done on typestates since they were first introduced by Strom and Yemini [1986]. Most typestate analyses are whole-program analyses, which makes them very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [Fink et al. 2008]) rely on a global alias analysis and generally assume that the protocol implementation is correct and only verify client conformance. Naeem and Lhoták [2008] 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 [DeLine and Fähndrich 2004] 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 [2007] extended this approach by supporting more expressive method specifications based on linear logic [Girard 1987]. 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 [2003], to support patterns like borrowing and adoption [Boyland and Retert 2005]. The Plural tool supports modular typestate checking with access permissions for Java. It has been used in a number of practical studies [Bierhoff et al. 2009]. Although Plural introduced state guarantees, this paper provides their first formalization. Nanda et al. [2005] present a system for deriving typestate information from Java programs. In general, type and typestate inference techniques are complementary and orthogonal to gradual typing [Siek and Vachharajani 2008].

Work on distributed session types [Gay et al. 2010] 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 proposed by Aldrich et al. [2009]; its defining characteristic is supporting run-time changes to the representation of objects in the dynamic semantics and type system. The programming language Plaid[8] is the first language to integrate typestates in the core programming model. Saini et al. [2010] 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 access permissions and state guarantees for reasoning about typestate in the presence of aliasing. Earlier work related to TSOP includes the Fickle system [Drossopoulou et al. 2001], 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 [Knowles and Flanagan 2010] and gradual typing [Siek and Taha 2006, 2007; Bierman et al. 2010]. Gradual Featherweight Typestate is a considerable advance in this sense, by showing how to gradually check flow-sensitive resources in a modular fashion. Bodden [2010] 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. Turning a conventional alias analysis into a modular analysis would require heavy low-level annotations (such as abstract locations) that are not directly meaningful to programmers. In contrast, permissions are designed to match human abstractions.

Ahmed et al. [2007] 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

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

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 access 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. Extended L3 is a parametrized framework: one must add one's own type system to associate a proof with the capability request.

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 [2007], and looking at how a gradual type system could support Plaid's Statechart-like multidimensional, compositional state model [Sunshine et al. 2011]. Another interesting direction is examining how gradual permissions could be leveraged in Plaid's support for concurrency [Stork 2013]. 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 [Boyland and Retert 2005; Naden et al. 2012], 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 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, not for memory management, but for enabling runtime permission checks. Standard optimization techniques like deferred increments [Baker 1994] and update coalescing [Levanoni and Petrank 2006] will be applied. We believe these techniques will reduce reference count overhead to a small percentage of runtime, 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. Featherweight Typestate (FT) and Gradual Featherweight Typestate (GFT) are nominal core calculi for typestate-oriented programming. By introducing typestate directly into the languages and extending their type systems 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. Both languages support a rich set of access permissions together with state guarantees for substantial reasoning about typestate in the presence of aliasing. Furthermore, this work paves the way for further gradual approaches by showing how to modularly and gradually check flow-sensitive resources.

REFERENCES

AHMED, A., FLUET, M., AND MORRISETT, G. 2007. L3: A linear language with locations. *Fundamenta Informaticae 77,* 4, 397–449.

ALDRICH, J., SUNSHINE, J., SAINI, D., AND SPARKS, Z. 2009. Typestate-oriented programming. In *Proc. Onward! 2009*. ACM, New York, NY, USA, 1015–1022.

BAKER, H. G. 1994. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not. 29*, 38–43.

BIERHOFF, K. AND ALDRICH, J. 2007. Modular typestate checking of aliased objects. In *Proc. Conference on Object-oriented Programming Systems and Applications*. ACM, New York, NY, USA, 301–320.

BIERHOFF, K., BECKMAN, N. E., AND ALDRICH, J. 2009. Practical API protocol checking with access permissions. In *Proc. European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 195–219.

BIERMAN, G., MEIJER, E., AND TORGERSEN, M. 2010. Adding dynamic types to C#. In *Proc. European Conference on Object-oriented Programming*. ECOOP'10. Springer-Verlag, Berlin, Heidelberg, 76–100.

BODDEN, E. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. International Conference on Software Engineering*. ACM, New York, NY, USA, 5–14.

BOYLAND, J. 2003. Checking interference with fractional permissions. In *Proc. Static Analysis (SAS)*. Springer-Verlag, Berlin, Heidelberg, 55–72.

BOYLAND, J. AND RETERT, W. 2005. Connecting effects and uniqueness with adoption. In *Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 283—-295.

DELINE, R. AND FÄHNDRICH, M. 2004. Typestates for objects. In *Proc. European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 465–490.

DROSSOPOULOU, S., DAMIANI, F., DEZANI-CIANCAGLINI, M., AND GIANNINI, P. 2001. Fickle: Dynamic object re-classification. In *Proc. European Conference on Object-Oriented Programming*. Springer-Verlag, Berlin, Heidelberg.

FINK, S. J., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol. 17,* 2, 1–34.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Boston, MA, USA.

GARCIA, R., WOLFF, R., TANTER, É., AND ALDRICH, J. 2013. Featherweight Typestate. Tech. Rep. CMU-ISR-13-112, Carnegie Mellon University. Sept.

GAY, S., VASCONCELOS, V., RAVARA, A., GESBERT, N., AND CALDEIRA, A. 2010. Modular session types for distributed object-oriented programming. In *Symposium on Principles of programming languages*. ACM, New York, NY, USA, 299–312.

GIRARD, J.-Y. 1987. Linear logic. *Theor. Comput. Sci. 50,* 1, 1–102.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23,* 3, 396–450.

JASPAN, C. AND ALDRICH, J. 2009. Checking framework interactions with relationships. In *Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP 2009)*, M. Mezini, Ed. Lecture Notes in Computer Science Series, vol. 5653. Springer, Genova, Italy, 27–51.

KNOWLES, K. AND FLANAGAN, C. 2010. Hybrid type checking. *ACM Trans. Program. Lang. Syst. 32,* 2, 6:1–6:34.

LEVANONI, Y. AND PETRANK, E. 2006. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst. 28*, 1–69.

NADEN, K., BOCCHINO, R., ALDRICH, J., AND BIERHOFF, K. 2012. A type system for borrowing permissions. In *Symposium on Principles of Programming Languages*. POPL '12. ACM, New York, NY, USA, 557–570.

NAEEM, N. A. AND LHOTÁK, O. 2008. Typestate-like analysis of multiple interacting objects. In *Proc. Conference on Object-oriented programming systems languages and applications*. ACM, New York, NY, USA, 347–366.

NANDA, M. G., GROTHOFF, C., AND CHANDRA, S. 2005. Deriving object typestates in the presence of inter-object references. In *Proc. Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 77–96.

PIERCE, B. C. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA.

SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation-passing style. *Lisp Symb. Comput. 6,* 3-4, 289–360.

SAINI, D., SUNSHINE, J., AND ALDRICH, J. 2010. A theory of typestate-oriented programming. In *Formal Techniques for Java-like Programs*. ACM, New York, NY, USA.

SIEK, J. AND TAHA, W. 2006. Gradual typing for functional languages. In *Proc. Scheme and Functional Programming Workshop*. ACM, New York, NY.

SIEK, J. AND TAHA, W. 2007. Gradual typing for objects. In *Proc. European Conference on Object-oriented Programming*. Springer-Verlag, Berlin, Heidelberg, 2–27.

SIEK, J. G. AND VACHHARAJANI, M. 2008. Gradual typing with unification-based inference. In *Proc. Symposium on Dynamic languages*. ACM, New York, NY, USA, 7:1–7:12.

STORK, S. 2013. Æminium: Freeing Programmers from the Shackles of Sequentiality. Ph.D. thesis, Carnegie Mellon University.

STROM, R. E. AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12,* 1, 157–171.

SUNSHINE, J., NADEN, K., STORK, S., ALDRICH, J., AND TANTER, É. 2011. First-class state change in plaid. In *Proc. Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA.

WALKER, D. 2005. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. MIT Press, Cambridge, MA, Chapter 1, 3–43.

WOLFF, R., GARCIA, R., TANTER, É., AND ALDRICH, J. 2011. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, M. Mezini, Ed. Lecture Notes in Computer Science Series, vol. 6813. Springer, Lancaster, UK, 459–483.

WOLFF, R., GARCIA, R., TANTER, É., AND ALDRICH, J. 2013. Gradual Featherweight Typestate. Tech. Rep. CMU-ISR-13-113, Carnegie Mellon University. Sept.

A. HELPERS

$\boxed{C <: C}$ Subclass

$$\frac{\textsf{class } C \textsf{ extends } D \{ \overline{F},\ \overline{M} \}}{C <: D}$$

$$\frac{}{C <: C}$$

$$\frac{C <: D \qquad D <: E}{C <: E}$$

$\boxed{\mathit{fields}(C)}$ Class Field Declarations

$$\text{(fields-object)}\ \frac{}{\mathit{fields}(\textsf{Object}) = \cdot}$$

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

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

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

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

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

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

B. GFT PROGRAM TYPING RULES

$\boxed{Md\ \mathbf{ok\ in}\ C}$ Well-typed Method Declaration

$$\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \{ \overline{F}, \overline{M} \} \\ \mathit{mdecl}(D, m) = T_r\ m(\overline{T_i \gg T'_i})[P_t\ E \gg T'_t]\end{array}}{T_r\ m(\overline{T_i \gg T'_i})[P_t\ C \gg T'_t]\ \mathbf{ok\ in}\ C}$$

$$\frac{\begin{array}{c}\textsf{class } C \textsf{ extends } D \{ \overline{F}, \overline{M} \} \\ \mathit{mdecl}(D, m) \text{ undefined}\end{array}}{T_r\ m(\overline{T_i \gg T'_i})[P_t\ C \gg T'_t]\ \mathbf{ok\ in}\ C}$$

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

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

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

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

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

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

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

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

C. GFT INTERNAL LANGUAGE (GFTIL)

$\boxed{\Delta \vdash e : T \dashv \Delta}$ Well-typed Expression

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

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

$$(\text{TIvar-b})\ \frac{}{\Delta, b : T \vdash b : T \dashv \Delta}$$

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

$$(\text{TIvar})\ \frac{T_1 \Rrightarrow T_2/T_3}{\Delta, s : T_1 \vdash s[T_1 \Rrightarrow T_2/T_3] : T_2 \dashv \Delta, s : T_3}$$

$$(\text{TIswap})\ \frac{(T_2\ f) \in \mathit{fields}(C_1)}{\begin{array}{l}\Delta, s_1 : P_1\ C_1, s_2 : T_2 \vdash \\ \quad s_1.f\ :=:\ s_2 : T_2 \dashv \Delta, s_1 : P_1\ C_1\end{array}}$$

$$(\text{TIfield})\ \frac{(T\ f) \in \mathit{fields}(C) \qquad T \Downarrow T'}{\Delta, s : P\ C \vdash s.f : T' \dashv \Delta, s : P\ C}$$

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

$$(\text{TIfield}_d)\ \frac{}{\Delta, s : \mathsf{Dyn} \vdash s._d f : \mathsf{Dyn} \dashv \Delta, s : \mathsf{Dyn}}$$

$$(\text{TIupdate})\ \frac{k_1 \in \{\mathsf{full}, \mathsf{shared}\} \qquad C_1' <: D_1 \qquad \mathit{fields}(C_1') = \overline{T_2\ f}}{\Delta, s_1 : k_1(D_1)\ C_1, \overline{s_2 : T_2} \vdash s_1 \leftarrow C_1'(\overline{s_2}) : \mathsf{Void} \dashv \Delta\!\downarrow, s_1 : k_1(D_1)\ C_1'}$$

$$(\text{TInew})\ \frac{\mathit{fields}(C) = \overline{T\ f}}{\Delta, \overline{s : T} \vdash \mathsf{new}\ C(\overline{s}) : \mathsf{full}(\mathsf{Object})\ C \dashv \Delta}$$

$$(\text{TIupdate}_d)\ \frac{\mathit{fields}(C) = \overline{T_2\ f}}{\Delta, s_1 : \mathsf{Dyn}, \overline{s_2 : T_2} \vdash s_1 \leftarrow_d C(\overline{s_2}) : \mathsf{Void} \dashv \Delta\!\downarrow, s_1 : \mathsf{Dyn}}$$

$$(\text{TIrel})\ \frac{}{\Delta, s : T \vdash \mathsf{release}[T](s) : \mathsf{Void} \dashv \Delta}$$

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

$$(\text{TIlet})\ \frac{\begin{array}{c}\Delta \vdash e_1 : T_1 \dashv \Delta_1 \\ \Delta_1, x : T_1 \vdash e_2 : T_2 \dashv \Delta_2 \\ x : \mathsf{Void} \in \Delta_2\ \mathit{or}\ x : T_1' \notin \Delta_2\end{array}}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2 \dashv \Delta_2 \div x}$$

$$(\text{TImerge})\ \frac{\begin{array}{c}T_1 = T_1\!\downarrow \qquad T_1/T_2' \Rrightarrow T_3 \\ \Delta, l_2 : T_2 \vdash e : T \dashv \Delta', l_2 : T_2'\end{array}}{\Delta, l_1 : T_1, l_2 : T_2 \vdash \mathsf{merge}[l_1 : T_1/l_2 : T_2' \Rrightarrow T_3](e) : T \dashv \Delta', l_2 : T_3}$$

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

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

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

$$\frac{\text{class } C \text{ extends } D\ \{\ \overline{F}, \overline{M}\ \} \qquad mdecl(D,m) = T_r\ m(\overline{T_i \gg T'_i})[P_t\ E \gg T'_t]}{T_r\ m(\overline{T_i \gg T'_i})[P_t\ C \gg T'_t]\ \textbf{ok in}\ C}$$

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

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

$$\frac{T_r\ m(\overline{T_i \gg T'_i\ x})[T_t \gg T'_t]\ \textbf{ok in}\ C_t \qquad \text{this} : T_t, \overline{x : T_i} \vdash e : T_r \dashv \text{this} : T'_t, \overline{x : T'_i}}{T_r\ m(\overline{T_i \gg T'_i\ x})\ [T_t \gg T'_t]\ \{\ \text{return } e;\ \}\ \textbf{ok in}\ C_t}$$

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

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

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

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

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

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

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

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

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

$$\text{(GEswap)}\ \frac{\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \qquad \mathit{fields}(C) = \overline{T\ f}}{\begin{array}{l}\mu, \rho, l_1.f_i\ :=:\ l_2 \rightarrow \\ \quad \mu[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\overline{o})\ \overline{P}], \rho, o_i\end{array}}$$

$$\text{(GEinvoke)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \qquad \mathit{method}(m, C) = \\ T_r\ m(\overline{T_i \gg T_i'\ x})\ [T_t \gg T_t']\ \{\ \mathsf{return}\ e;\ \}\end{array}}{\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1, \overline{l_2}/\mathsf{this}, \overline{x}]e}$$

$$\text{(GEswap}_d\text{)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \qquad \mathit{fields}(C) = \overline{T\ f} \\ D_g = \curlywedge \{D \mid k(D) \in \overline{P}\}\end{array}}{\begin{array}{l}\mu, \rho, l_1.f_i\ :=:_d\ l_2 \rightarrow \\ \quad \mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg \mathsf{shared}(D_g)\ C\rangle(l_1); \\ \quad \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_i\rangle(l_2); \\ \quad \mathsf{let}\ ret = l_1.f_i\ :=:\ l_2\ \mathsf{in} \\ \qquad \mathsf{assert}\langle \mathsf{shared}(D_g)\ C \gg \mathsf{Dyn}\rangle(l_1); \\ \qquad \mathsf{assert}\langle T_i \gg \mathsf{Dyn}\rangle(ret); \\ \qquad ret\end{array}}$$

$$\text{(GEinvoke}_d\text{)}\ \frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} \\ \mathit{mdecl}(m, C) = T_r\ m(\overline{T_i \gg T_i'})\ [T_t \gg T_t'] \\ \mid \overline{T_i} \mid = \mid \overline{l_2} \mid\end{array}}{\begin{array}{ll}\mu, \rho, l_1 ._d m(\overline{l_2}) \rightarrow \mu, \rho, & \mathsf{assert_d}\langle \mathsf{Dyn} \gg T_t\rangle(l_1); \\ & \overline{\mathsf{assert_d}\langle \mathsf{Dyn} \gg T_i\rangle(l_2);} \\ & \mathsf{let}\ ret = l_1.m(\overline{l_2})\ \mathsf{in} \\ & \quad \mathsf{assert}\langle T_t' \gg \mathsf{Dyn}\rangle(l_1); \\ & \quad \overline{\mathsf{assert}\langle T_i' \gg \mathsf{Dyn}\rangle(l_2);} \\ & \quad \mathsf{assert}\langle T_r \gg \mathsf{Dyn}\rangle(ret); \\ & \quad ret\end{array}}$$

$$\text{(GEupdate)}\ \frac{\begin{array}{cc}\mu(\rho(l_1)) = C(\overline{o})\ \overline{P} & \mathit{fields}(C) = \overline{T\ f} \\ \mu_1 = \mu[\rho(l_1) \mapsto C'(\overline{\rho(l_2)})\ \overline{P}] & \mu' = \mu_1 - \overline{o : T}\end{array}}{\mu, \rho, l_1 \leftarrow C'(\overline{l_2}) \rightarrow \mu', \rho, \mathsf{void}}$$

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

$$(\text{GEfield})\ \frac{\begin{array}{cc}\mu(\rho(l)) = C(\overline{o})\ \overline{P} & \mathit{fields}(C) = \overline{T\ f} \\ T_i \Downarrow T' & \mu' = \mu + o_i : T'\end{array}}{\mu, \rho, l.f_i \rightarrow \mu', \rho, o_i} \qquad (\text{GEassert})\ \frac{\mu' = \mu - \rho(l) : T + \rho(l) : T'}{\mu, \rho, \mathsf{assert}\langle T \gg T'\rangle(l) \rightarrow \mu', \rho, \mathsf{void}}$$

$$(\text{GEfield}_d)\ \frac{\mu(\rho(l)) = C(\overline{o})\ \overline{P} \quad \mathit{fields}(C) = \overline{T\ f}}{\mu, \rho, l._d f_i \rightarrow \mu, \rho, o_i}$$

$$(\text{GEassert}_d\text{v})\ \frac{\rho(l) = \mathsf{void}}{\mu, \rho, \mathsf{assert_d}\langle \mathsf{Dyn} \gg \mathsf{Void}\rangle(l) \rightarrow \mu, \rho, \mathsf{void}}$$

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

$$(\text{GEhold})\ \frac{\mu' = \mu - \rho(l) : T_1 + \rho(l) : T_2 + \rho(l) : T_3 \quad l' \notin \mathit{dom}(\rho) \quad \rho' = \rho[l' \mapsto \rho(l)]}{\mu, \rho, \mathsf{hold}[l : T_1 \Rrightarrow T_2/T_3 \gg T_3' \Rrightarrow T_1'](e) \rightarrow \mu', \rho', \mathsf{merge}[l' : T_2{\downarrow}\ /l : T_3' \Rrightarrow T_1'](e)}$$

$$(\text{GEmerge})\ \frac{\mu' = \mu - \rho(l') : T_1 - \rho(l) : T_2 + \rho(l) : T_3}{\mu, \rho, \mathsf{merge}[l' : T_1/l : T_2 \Rrightarrow T_3](v) \rightarrow \mu', \rho, v}$$

$$(\text{GEmcongr})\ \frac{\mu, \rho, e \rightarrow \mu', \rho', e'}{\mu, \rho, \mathsf{merge}[l_1 : T/l_2](e) \rightarrow \mu', \rho', \mathsf{merge}[l_1 : T/l_2](e')}$$

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

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

D. TYPE-DIRECTED TRANSLATION FROM GFT TO GFTIL

$\boxed{M \rightsquigarrow M^{\mathcal{I}}}$ Method Translation

$$\frac{\begin{array}{c}\textsf{this} : T_t, \overline{x : T} \vdash e : T_r \rightsquigarrow e^{\mathcal{I}} \dashv \textsf{this} : T_t'', \overline{x : T''} \\ e_1^{\mathcal{I}} = \textsf{let } ret = e^{\mathcal{I}} \textsf{ in } coerce(\textsf{this}, T_t'', T_t'); \overline{coerce(x, T'', T')}; ret\end{array}}{T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \textsf{return } e;\ \} \rightsquigarrow T_r\ m(\overline{T \gg T'\ x})\ [T_t \gg T_t']\ \{\ \textsf{return } e_1^{\mathcal{I}};\ \}}$$

$\boxed{F \rightsquigarrow F^{\mathcal{I}}}$ Field Translation

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

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

$$\frac{\overline{F \rightsquigarrow F^{\mathcal{I}}} \qquad \overline{M \rightsquigarrow M^{\mathcal{I}}}}{\begin{array}{l}\textsf{class } C_0 \textsf{ extends } C_1\ \{\ \overline{F};\ \overline{M}\ \} \\ \quad \rightsquigarrow \textsf{class } C_0 \textsf{ extends } C_1\ \{\ \overline{F^{\mathcal{I}}};\ \overline{M^{\mathcal{I}}}\ \}\end{array}}$$

$\boxed{PG \rightsquigarrow PG^{\mathcal{I}}}$ Program Translation

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