Modular Typestate Verification of Aliased Objects

Kevin Bierhoff[*] **Jonathan Aldrich**[†]

March 2007
CMU-ISRI-07-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Institute for Software Research, Carnegie Mellon University, kevin.bierhoff @ cs.cmu.edu.
[†]Institute for Software Research, Carnegie Mellon University, jonathan.aldrich @ cs.cmu.edu.

This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

Keywords: Protocol enforcement, typestates, aliasing, permissions.

Abstract

A number of type systems have used typestates to specify and statically verify protocol compliance. Aliasing is a major challenge for these systems. This paper proposes a modular type system for a core object-oriented language that leverages linear logic for verifying compliance to more expressive protocol specifications than previously supported. The system improves reasoning about aliased objects by associating references with access permissions that systematically capture what aliases know about and can do to objects. Permissions grant full, shared, or read-only access to a certain part of object state and allow aliasing both on the stack and in the heap. The system supports dynamic state tests, arbitrary callbacks, and open recursion. The system's expressiveness is illustrated with examples from the Java I/O library.

Contents

1 Introduction

In well–written software, different parts of the program interact with each other through abstraction boundaries (interfaces) that hide state and side effects from each other. Although interfaces facilitate understanding and using software, clients of an interface cannot be oblivious to hidden side effects. They typically have to follow a certain *protocol* in using the interface that is intricately tied to the implemented functionality. For example, clients of a stream interface are expected to first read and then close streams and not vice versa. Conformance to such protocols is notoriously hard to verify.

Typestates [30] offer a lightweight way of specifying interesting protocols [4] using abstract state machines. Typestates refine the fixed types of objects with changing abstract states. Operations perform "state transitions" on objects that change their state from one to another. Fugue [11] is the only existing typestate-based object-oriented protocol verification system that we know of.

This paper improves Fugue's reasoning power on two fronts: (1) We propose a type system that can verify compliance to more expressive specifications than previously supported. (2) We improve modular reasoning about protocol compliance of aliased objects. Preliminary case studies suggest that these improvements combined let us go beyond reasoning about objects in isolation and capture object collaborations to some extent.

Expressive protocols. In earlier work we proposed to increase the expressiveness of existing typestate-based protocols to better match object-oriented software [4]. In particular, we found the need to *refine* protocols in subclasses and to relate different objects to one another. The former gives freedom in extending base classes; the latter captures common programming patterns such as dynamic state tests and binary methods. Our proposal was based on a hierarchical notion of state spaces similar to Statecharts [19] that can model orthogonal concerns separately and allows protocol refinement with more fine-grained states. Specifications became logical expressions that could relate objects.

This paper contributes a modular type system that can verify correct usage and implementation of such expressive typestate protocols. Our verification approach is highly inspired by Fugue [11]. We extend state invariants, packing, and frames to work in our context. We improve support for inheritance in comparison to Fugue by decoupling states of frames and reducing overriding requirements. Details about our specification approach will be provided in sect. 2.

Reasoning about aliased objects. Modular verification of protocol compliance in the presence of aliasing is notoriously hard. It basically involves tracking the abstract state of all visible objects and updating these states according to specified state transitions when methods are called. The problem is that method calls could involve invisible manipulation of relevant objects through *aliases* (i.e. other references to those objects). A simple remedy is to enforce that objects have only one reference, ensuring their *linearity* [32]. Since no aliases exist, typestate changes through the one available reference are straightforward to track.

Linearity is extremely restrictive in practice because even method calls (i.e. stack aliasing) become a challenge, let alone storing references in fields (i.e. heap aliasing). Therefore, most approaches to protocol verification allow some amount of aliasing (e.g. [10, 21, 8]). Ultimately, however, permanent state changes require linearity (or at least all aliases to be in scope [10, 11, 15]). Many systems also support *sharing* (i.e. heap aliasing [12, 15, 21, 8]) of objects, but sharing fixes the state. *Focusing* constructs allow temporarily leaving a state, but objects have to return to their fixed state before other aliases access them [12, 15]. Some systems also permit harmless read-only access [21, 8]. In summary, state changes are only permitted if linearity guarantees the absence of unexpected callbacks.

This paper proposes to take the opposite route: as a first approximation, objects always have to be prepared for callbacks. It is the object's protocol that should govern when calls can occur, not the object's linearity. This work shows that such an approach is feasible. Our approach is to use fine-grained access control to objects in order to constrain possible state changes through invisible aliases; object linearity is no longer needed. Specifically, we use *access permissions* to keep track of what a reference can do to and knows about the referenced object. A permission grants *full*, *shared*, or *pure* access to a particular part of the object's state. Other (possibly invisible) permissions to the same object are guaranteed to be consistent: either a distinguished writer (full permission) co-exists with many readers (pure permission) or many writers (shared) co-exist with many readers (pure) of the same state.

Permissions express design intent by capturing very precisely what access a method needs to an object. Since they are resources, we use linear logic [17] to combine permissions into expressive protocol specifications. Correspondingly, linear logic reasoning is used to track permissions as they flow through the program. Newly created objects have one full permission for the entire object state. As aliasing occurs, permissions are *split* according what access each reference needs. Fractions [5] keep track of splits so that they can be *joined* after temporary aliasing. The flexibility achieved with fractions was indispensable when specifying Java iterators [3]. Splitting and joining is handled transparently to the programmer.

In summary, contributions to reasoning about aliased objects include the following. (1) Different references can be constrained to modify orthogonal parts of the referenced object without knowing about each other. This relates state spaces to data groups [24]. (2) An object's state can be partially fixed, giving shared and read-only permissions the ability to *assume* a state. (3) Objects can depend in their invariants on all other objects, even read-only ones. (4) Under certain conditions, a fixed state can be left later even if the object was previously aliased in the heap. We point out that (1) and (2) directly leverage our hierarchical notion of state spaces in access permissions.

Benefits. By allowing object aliasing with permissions and relating objects using linear logic, our approach can capture protocols involving object collaborations such as iterators [3] and stream pipes (sect. 2). Clients and implementations can be checked for compliance to such protocols. To our knowledge, existing typestate-based verification systems lack expressive power needed for these protocols. Furthermore, our approach helped expose a way of breaking an internal invariant of a frequently used class in the Java standard library, `java.io.BufferedInputStream`.

Compared to global analyses [1, 20], our approach promises more scalability and less brittleness. Since it operates modularly, it assists programmers like a compiler in using interfaces correctly. At the same time we can handle arbitrary callbacks and open recursion, issues that are difficult to handle in modular approaches. We can also verify correct usage *and* implementation of dynamic state tests. Dynamic tests have received surprisingly little attention [8] considering how common they are (e.g. for testing if a stream is open or a collection is empty).

Outline. The remainder of this paper is organized as follows. In the following section we introduce our typestate specification approach. We give an overview of our formal verification approach in section 3. Section 4 formally defines a object-oriented language with expressive typestate specifications. We discuss protocol verification using this calculus in section 5. Section 6

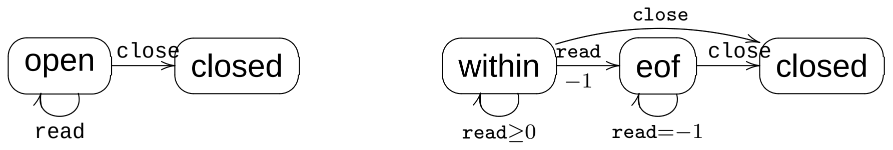

Figure 1: Simple and refined input stream protocols

describes the problem in `BufferedInputStream` that we found. Soundness of our approach for a core fragment of specifications is proven in section 7. Section 8 compares our approach to related work. We conclude in section 9.

2 Typestate Specifications

2.1 Protocols as State Machines

Typestates define protocols as state machines. For example, a simple model of an input stream would have two states open and closed. The stream can be read as long as it is open (fig. 1). Notice that states have an intuitive meaning even though their concrete names are irrelevant.

In an object-oriented language we can associate a class with such a state machine. States express abstractly what conditions an object satisfies at a given time (e.g., it is “open”). They respect object encapsulation because states do not correspond to concrete fields in an implementation. Methods can be specified with the state transitions they can perform (e.g., `close` transitions from open to closed). In this approach, methods are specified independently from each other and state transitions are reminiscent of traditional function types. This allows verifying typestate-based protocol specifications in the manner of a type system.

2.2 State Refinements and Dimensions

Rather than defining the possible states of an object with a “flat” set of mutually exclusive states, we model *state spaces* using dimensions and refinements [4] that loosely correspond to AND- and OR-states in Statecharts [19].

State refinement can be used to distinguish more fine-grained conditions within a state. For example, we could refine open into two mutually exclusive states within and eof to distinguish whether `read` returns a character or the “end of file” (EOF) token (fig. 1). The idea of *state dimensions* is to separate independent aspects of object behavior. For example, Java input streams can be “marked” and later “reset” to the marked position [4]. This is possible independently of the stream’s current position. Thus dimensions obviate the need for “combination” states such as “marked and within” and allow specifications to focus on dimensions of interest. Technically, dimensions are just independent refinements that start from a root state called alive. At runtime, an object will be in exactly one state in each applicable dimension. The state space of streams could e.g. be specified as follows. Note that each dimension has a unique name.

```
stream = open, closed refines alive;
position = within, eof refines open;
mark = unmarked, marked refines open;
```

Subclasses *inherit* the superclass's state space and are free to add their own refinements. This conveniently ensures that states of subclasses always correspond to (possibly more coarse-grained) states of superclasses [4]. Note that dimensions or states do *not* correspond to implementation fields. Instead, *state invariants tie* field values to states (see sect. 4.4).

2.3 Access Permissions

A major complication in verifying protocol specifications is that different variables could *alias* the same object. Care must be taken to keep the "views" of those aliases, i.e. what they assume about the referenced object, consistent.

Our approach is to associate references with *access permissions* that are guaranteed to remain consistent. A permission *perm*(x, n, A) grants different levels of access to a part n of the state space (e.g., a state dimension) to a variable x. Permissions optionally carry additional information A about the exact state inside the part of the state space they cover (omitted otherwise). We use the following access levels.

- full permissions give exclusive right to can change state.
- share permissions give shared modifying access. Many share permissions may be around, but no full permission. This is the de-facto access level in languages without aliasing control such as Java or C#.
- pure permissions give read-only access. There may be other pure permissions and *either* one full permission *or* several share permissions around.

For example, full(*this*, position, within) represents a full permission to change state in the position dimension of an input stream named *this* that is currently in the within state. Permissions to different dimensions of the same object do not interfere and can therefore be used to independently change in "their" dimensions.

As a technical device, we use fractions [5] to keep track of permission splitting. This lets us e.g. "collect" all share permissions to regain a full permission. We usually omit fractions in examples but sect. 4.3 will make fractions precise. Unlike in existing work [5], we use fractions not to fully avoid interference but to keep permissions consistent and to allow temporary aliasing.

2.4 Linear Logic Specifications

As input streams illustrate we need considerable flexibility in specifying methods. In particular, it is crucial to relate states of method receiver, arguments and results to each other and allow non-deterministic behavior [4, 3]. Both are exhibited by the `read` method that can transition to within or eof and indicates its choice by returning different values.

To achieve this expressiveness we specify methods with the decidable multiplicative-additive fragment of linear logic [17] (MALL). Method pre- and post-conditions are separated with a linear implication ($\multimap$) and use conjunction ($\otimes$), internal choice ($\&$), and external choice ($\oplus$).

The following example specifies the `read` method for Java input streams. It requires a share permission for the receiver's open state. The post-condition on the right-hand side of the implication is an external choice between conjunctions indicating that the caller has no influence on whether `read` will return a character or EOF.

$$\begin{aligned}\mathsf{share}(\mathit{this}, \mathsf{position}) \multimap\ &(\mathit{result} \geq 0 \otimes \mathsf{share}(\mathit{this}, \mathsf{open}))\\ &\oplus (\mathit{result} = -1 \otimes \mathsf{share}(\mathit{this}, \mathsf{open}, \mathsf{eof}))\end{aligned}$$

For a more complex example, consider the "pipe" implementation in the Java I/O library [4]. A pipe is created by connecting a `PipedOutputStream` (called "source") to a `PipedInputStream` (called "sink"). Calling `read` on the sink will return the next character from a private buffer. The source's `write` method deposits characters into that buffer using the sink's `receive` method. The source signals that it is closed by invoking `receivedLast` on the sink.

Fig. 2 shows how the sink side of the pipe can be specified using our approach. (For simplicity, we do not consider the sink as a subclass of `InputStream` here.)

Connecting the pipe (using the sink's constructor) creates two shared permissions to the sink. One is used by the source for calls to `receive` and is "consumed" by the sink when calling `receiveLast`. Only then can the sink reach eof and join the two shared permissions (from the source side and the sink's client) to close the stream. We use explicit fractions (see sect. 4.3) to ensure the presence of exactly two shared permissions. They allow us to permanently change the sink's state to closed even though it was previously aliased in the heap. To improve readability, the explicit fractions needed for this example are expressed with half permissions that represent share permissions with "half" ($1/2$) fractions as detailed in section 4.

Notice how different kinds of permissions with different assumptions, e.g. in `read`, `close`, and `isClosed`, express design intent *and* allow aliasing: permissions capture what part of the object's state a method can affect. Aliasing is permitted because other permissions can exist while a method executes.

The specification prevents several error conditions that cause runtime exceptions in the Java implementation: (1) closing the sink before the source, (2) calling `receive` after `receiveLast`, and (3) reading from a closed sink.

3 Verification Approach

In the remainder of the paper we formalize sound modular verification of typestate specifications based on access permissions for a core object-oriented language based on Featherweight Java (FJ, [22]). This section summarizes our verification approach before the following sections go into more detail.

A set of expression checking rules track state and permission changes. The rules are syntax-directed up to reasoning about permission requirements for e.g. method calls. The intuition is that a method body "guides" the search for a proof that the method's post-condition can be satisfied from

```
class PipedInputStream {
  stream = open, closed refines alive;
  position = within, eof refines open;
  buffer = empty, nonEmpty refines within;
  filling = partial, filled refines nonEmpty;
  source = sourceOpen, sourceClosed refines nonEmpty;

  public PipedInputStream(PipedOutputStream src) :
    full(src, alive, raw) ⊸ half(this, open) ⊗ full(src, alive, open)

  void close() : half(this, open, eof) ⊸ unique(this, alive, closed)
  boolean isClosed() : pure(this, alive) ⊸ (result = true ⊗ pure(this, alive, closed))
                                        ⊕ (result = false ⊗ pure(this, alive, open))

  int read() : share(this, open) ⊸ (result ≥ 0 ⊗ share(this, open))
                                        ⊕ (result = −1 ⊗ share(this, open, eof))

  void receive(int b) : half(this, open) ⊗ b ≥ 0 ⊸ half(this, open, nonEmpty)
  void receivedLast() : half(this, open) ⊸ 1
}
```

Figure 2: Java `PipedInputStream` protocol (simplified)

its pre-condition by telling the checker which implications (i.e. methods) to apply in which order. Resource management, i.e. decisions about which permission to use when, is handled transparently with linear logic reasoning.

Behavioral subtyping. A state space is associated with each class. Subclasses inherit state spaces and can define additional state refinements. Overriding methods are free to define their own specifications, e.g. by using more fine-grained states. We devise a simple check that ensures behavioral subtyping [26] between overridden and overriding methods.

Primitive Booleans. Booleans are primitive and a conditional construct allows us to statically distinguish outcomes of Boolean tests. This lets us encode dynamic state tests and statically reason about their correct implementation. Typestates do *not* need a runtime representation in our approach. Protocol compliance is fully guaranteed at compile time.

Let-normal form. We syntactically distinguish pure terms (in particular of Boolean type) from expressions. Terms cannot affect permissions while expressions can. We require arguments of atomic expressions (such as method calls) to be terms in order to simplify permission reasoning. Results of expressions can be bound to fresh variables using a `let` construct. It can be used to simulate recursive expressions [27, 8].

Data groups. We map each field into a part of the class's state space. Thus nodes in the state space serve as data groups for object fields [24]. A modifying permission to a data group only permits assignments to fields contained in it.

State invariants. State invariants define abstract typestates in terms of permissions to fields. For example, a file-based stream, while open, could require a field to hold a valid file descriptor. Only states within a field's data group (and class) can depend on the field. State invariants serve as an explicit abstraction function between class interface and implementation. They let us reason about clients of a class fully separately from the class itself. They also let us verify that an implementation is conforms to its interface.

Unpacking. Permissions are explicitly *unpacked* in order to gain access to fields [11]. Access is granted according to state invariants implied by the permission's state assumption. Full and shared permissions grant modifying access while pure permissions grant read-only access to fields. Because of inheritance, only permissions to the method receiver can be unpacked. After manipulating fields, objects can be re-packed into a potentially different state.

Intermediate packing. The receiver must be fully packed before method calls. This guarantees that objects are consistent in case of a callback. This is not a strong restriction because the object can always be packed to an intermediate state. But it lets us enforce that only one permission is unpacked at any time, allowing us to apply *focus* [12] when unpacking shared permissions. Note that we could avoid intermediate packing in the absence of callbacks.

Frame permissions. Objects are compartmentalized into frames [11]. Each frame corresponds to a class in the object's subclass hierarchy. The frame corresponding to the object's runtime type is called *virtual frame*. We associate each frame with a separate *frame permission*. A frame's abstract typestate can only depend on fields defined in the same frame and the abstract state of the inherited frame.

Open recursion. A frame permission to an object's virtual frame is called *object permission*. Frame permissions are needed for unpacking and statically dispatched calls while object permissions are needed for dynamic dispatch. Dynamic dispatch can treat object permissions (to the receiver) as frame permissions (to the virtual frame). This only requires methods to be overridden as in Fugue [11] if they need frame permissions but overriding is not required for methods that only need object permissions.

Soundness. Intuitively, for every permission we maintain the invariant that the statically tracked state assumption is a sound approximation of the referenced object's runtime state (unless it is unpacked). While an object is unpacked we ensure that field assignments do not affect other permissions. Packing brings the object into a new state that is soundly approximated by other permissions. Since at most one object is unpacked at any time we are guaranteed that objects are consistent with the permission that is used for unpacking them. Using these intuitions, we prove soundness of a core language fragment in section 7.

4 Formal Language

4.1 Syntax

Fig. 3 shows the syntax of a simple class-based object-oriented language. The language is inspired by Featherweight Java (FJ, [22]); we will extend it to include typestate protocols in the following

programs	PR	$::=$	$\langle\overline{CL}, e\rangle$	
class decl.	CL	$::=$	$\texttt{class C extends C}' \{ \overline{\texttt{F}}\ \overline{\texttt{R}}\ \texttt{I}\ \overline{\texttt{N}}\ \overline{\texttt{M}} \}$	*I, N in fig. 5*
field decl.	F	$::=$	$f : T\ \texttt{in}\ n$	
method decl.	M	$::=$	$T\ m(\overline{T\ x}) : MS = e$	*MS in fig. 5*
refinements	R	$::=$	$d = \overline{s}\ \texttt{refines}\ s_0$	
terms	t	$::=$	$x \mid l \mid \texttt{true} \mid \texttt{false}$	*atoms*
		$\mid$	$t_1\ \texttt{and}\ t_2 \mid t_1\ \texttt{or}\ t_2 \mid \texttt{not}\ t$	*connectives*
expressions	e	$::=$	$t \mid f \mid \texttt{assign}\ f := t$	*terms, fields*
		$\mid$	$\texttt{new}\ C(\bar{t}) \mid t_0.m(\bar{t}) \mid \texttt{super}.m(\bar{t})$	*construction, calls*
		$\mid$	$\texttt{if}(t, e_1, e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	*condition, binding*
values	v	$::=$	$l \mid \texttt{true} \mid \texttt{false}$	
references	r	$::=$	$x \mid f \mid l$	
types	T	$::=$	$C \mid \texttt{bool}$	
nodes	n	$::=$	$s \mid d$	

classes	C	*fields*	f	*variables*	x	*methods*	m
locations	l	*states*	s	*dimensions*	d		

Figure 3: Base language syntax

subsections. We identify classes (C), methods (m), and fields (f) with their names. We use an overbar notation to abbreviate a list of elements. For example, $\overline{x : T} = x_1{:}T_1, \ \ldots, \ x_n{:}T_n$. Types ($T$) in our system include Booleans (`bool`) and classes.

Programs are defined with a list of class declarations and a main expression. A class declaration CL gives the class a unique name C and defines its fields, methods, typestates, and state invariants. A constructor is implicitly defined with the class's own and inherited fields. Fields (F) are declared with their name and type. Each field is mapped into a part of the state space n that can depend on the field (details in sect. 5.2). A method (M) declares its result type, formal parameters, specification and a body expression. State refinements R will be explained in the next section; method specifications MS and state invariants N are deferred to sect. 4.4.

We syntactically distinguish pure terms t and possibly effectful expressions e. Arguments to method calls and object construction are restricted to terms. This simplifies reasoning about effects [27, 8]. A translation from a more conventional syntax with recursive expressions into our let-normal form is straightforward. Notice that neither field access nor assignment are prefixed with a term. This syntactically restricts field access and assignment to fields of the receiver class. Explicit "getter" and "setter" methods can be defined to give other objects access to fields. We define the result of an assignment to be the *previous* field value.

$$\frac{}{\mathsf{refinements}(\texttt{Object}) = \cdot} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{F}\ \overline{R}\ \ldots\}\quad \mathsf{refinements}(C') = \overline{R'}}{\mathsf{refinements}(C) = \overline{R'}, \overline{R}}$$

$$\frac{n\ \textit{in}\ \mathsf{refinements}(C)}{C \vdash n\ \mathsf{wf}} \qquad \frac{C \vdash A_1\ \mathsf{wf} \quad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2\ \mathsf{wf}} \qquad \frac{C \vdash A_1\ \mathsf{wf} \quad A_1 \# A_2 \quad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2\ \mathsf{wf}}$$

$$\frac{d = \overline{s}\ \texttt{refines}\ s \in \mathsf{refinements}(C)}{C \vdash s_i \leq d \qquad C \vdash d \leq s} \qquad \frac{C \vdash n\ \mathsf{wf}}{C \vdash n \leq n} \qquad \frac{C \vdash n \leq n'' \quad C \vdash n'' \leq n'}{C \vdash n \leq n'}$$

$$\frac{d = \overline{s}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C) \quad d' = \overline{s'}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C) \quad d \neq d'}{C \vdash d \# d'}$$

$$\frac{C \vdash n_1 \leq n_1' \quad C \vdash n_1' \# n_2' \quad C \vdash n_2 \leq n_2'}{C \vdash n_1 \# n_2} \qquad \frac{C \vdash A' \# A}{C \vdash A \# A'} \qquad \frac{C \vdash A_{1,2} \# A}{C \vdash A_1 \otimes A_2 \# A}$$

$$\frac{C \vdash A_{1,2} \# A}{C \vdash A_1 \oplus A_2 \# A} \qquad \frac{C \vdash n' \leq n}{C \vdash n' \prec n} \qquad \frac{C \vdash A_{1,2} \prec n \quad C \vdash A_1 \otimes A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2 \prec n}$$

$$\frac{C \vdash A_{1,2} \prec n \quad C \vdash A_1 \oplus A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2 \prec n} \qquad \frac{C \vdash A \prec n \quad \forall n' : C \vdash A \prec n'\ \textit{implies}\ n \leq n'}{C \vdash A \ll n}$$

Figure 4: State space judgments

4.2 State Spaces

State spaces are formally defined as a list of state refinements (see fig. 3). A state refinement (R) refines an existing state in a new dimension with a set of mutually exclusive sub-states. We use s and d to range over state and dimension names, respectively. A node n in a state space can be a state or dimension. State refinements are inherited by subclasses. We assume a root state alive that is defined in the root class `Object`.

We define a variety of helper judgments for state spaces in fig. 4. $\mathsf{refinements}(C)$ determines the list of state refinements available in class C. $C \vdash A$ wf defines well–formed state assumptions. Conjunctive assumptions have to cover orthogonal parts of the state space. $C \vdash n \leq n'$ defines the substate relation for a class. $C \vdash A \# A'$ defines orthogonality of state assumptions. A and A' are orthogonal if they refer to different (orthogonal) state dimensions. $C \vdash A \prec n$ defines that a state assumption A only refers to states underneath a root node n. $C \vdash A \ll n$ finds the tightest such n.

4.3 Access Permissions

Access permissions p give references permission to access an object. Permissions to objects are uniformly represented with $\mathsf{access}(r, n, g, k, A)$ (fig. 5). (For simplicity, we omitted g and k in section 2.) The components have the following meaning.

- Permissions are granted to references r. References can in general be variables, locations, and fields (of the current receiver object) that are defined in the current scope.

- Permissions apply to a particular *subtree* in the space space of r. The subtree is identified by its root node n. The root node is a state or dimension defined in C. Other parts of the state space are unaffected by the permission. The type system can always assume that the referenced object is in state n.

- The *fraction function* g tracks for each node on the path from n to alive a symbolic fraction [5]. The fraction function keeps track of how often permissions were split at different nodes in the state space so they can be coalesced later (see sect. 5.5).

- The *subtree fraction* k encodes the level of access granted by the permission. $k > 0$ grants modifying access. $k < 1$ implies that other potentially modifying permissions exist. Fraction variables range over fractions strictly greater than 0. The different access levels are summarized in the following table.

Access level	This permission	Other permissions	Subtree Fraction
full	exclusive modifying access	only read-only	$k = 1$
share	shared modifying access	modifying and read-only	$0 < k < 1$
pure	read-only access	modifying and read-only	$k = 0$

- An optional *state assumption* A expresses additional state knowledge within the permission's subtree. Modifying permissions can be used to change the current state within the permission's subtree. If other modifying permissions exist then the state assumption is *temporary*, i.e. lost on any effectful expression (because the object's state may change without the knowledge of r). Thus only full permissions can permanently make state assumptions until they modify the object's state themselves. If no state assumption is given then the object is still guaranteed to be in state n.

As mentioned above, the subtree fraction k lets us recover our original three permission kinds that we write as full, share, and pure. They can be encoded as follows. Note that this equates $\mathsf{full}(r, n, g, A) \equiv \mathsf{share}(r, n, g, 1, A)$ which conforms with our intuition.

$$\begin{array}{rcl} \mathsf{access}(r, n, g, 1, A) & \equiv & \mathsf{full}(r, n, g, A) \\ \mathsf{access}(r, n, g, k, A) & \equiv & \mathsf{share}(r, n, g, k, A) \quad (k \neq 0) \\ \mathsf{access}(r, n, g, 0, A) & \equiv & \mathsf{pure}(r, n, g, A) \end{array}$$

4.4 Permission-Based Specifications

Objects often dependent on each other. For example, we want to be able to express that an object is in a particular state only if a Boolean value is true. Since permissions act as linear resources we use a decidable subset of linear logic connectives to relate multiple objects (fig. 5).

The atoms of our predicate language are the permissions p and facts q about Boolean values. Facts about values have the same role as state information about objects although state information

permissions	p	$::=$	$\mathsf{access}(r, n, g, k, A)$	*access perm.*
facts	q	$::=$	$t = \texttt{true} \mid t = \texttt{false}$	*boolean values*
assumptions	A	$::=$	$n \mid A_1 \otimes A_2 \mid A_1 \oplus A_2$	*node, conj., disj.*
fraction fct.	g	$::=$	$z \mid \overline{n \mapsto v}$	*variable, mapping*
		$\mid$	$g/2 \mid g_1, g_2$	*split, extension*
fractions	k	$::=$	$1 \mid 0 \mid z \mid k/2$	*full, zero, variable, split*
predicates	P	$::=$	$p \mid q$	*permissions, facts*
		$\mid$	$P_1 \otimes P_2 \mid \mathbf{1}$	*conjunction*
		$\mid$	$P_1 \mathbin{\&} P_2 \mid \top$	*internal choice*
		$\mid$	$P_1 \oplus P_2 \mid \mathbf{0}$	*external choice*
		$\mid$	$\exists z : H.P \mid \forall z : H.P$	*fraction quantification*
method specs	MS	$::=$	$P \multimap E$	
expr. types	E	$::=$	$\exists x : T.P$	
state inv.	N	$::=$	$n = P$	
initial state	I	$::=$	$\texttt{initially}\ \langle \exists \overline{f : T}.P, s_1 \otimes \ldots \otimes s_n \rangle$	
fraction terms	h	$::=$	$g \mid k$	
fraction types	H	$::=$	$\mathsf{Fract} \mid \overline{n} \rightarrow \mathsf{Fract}$	*value, function*
fraction vars.	z			

Figure 5: Permissions, predicates, and specifications

A changes over time while facts q remain true. These atoms can be combined with the linear operators multiplicative conjunction ($\otimes$), additive conjunction ($\&$), and additive disjunction ($\oplus$). We also include existential ($\exists z : H.P$) and universal quantification of fractions ($\forall z : H.P$) into our permissions. Quantification of fractions alleviates the programmer from writing concrete fraction functions in most cases. We will use an existential quantification over types to type all expressions (E).

Method specifications. Methods are specified with a linear implication ($\multimap$) of predicates (MS). This captures the intuition that a method "takes" a number of permissions and returns potentially different permissions. The left-hand side of the implication (essentially the method pre-condition) may refer to method receiver and formal parameters. The right-hand side (post-condition) existentially quantifies the result (a similar technique is used in Vault [10]). We always refer to the receiver with *this* and usually call the return value *result*.

State invariants. We also use predicates to define state invariants. State invariants were proposed in Fugue [11] as a generalization of class invariants. We decided to use linear logic predicates for state invariants as well (N). In general, several of the defined state invariants will have to be satisfied at the same time. This is due to the hierarchical nature of the state space and the existence of orthogonal state dimensions. Usually, state invariants will use existential quantification to abstract from concrete fraction functions. Each class declares an initial state as a conjunction of states (I). It must be established during object construction.

4.5 Handling Inheritance

Specifications of object behavior are usually not oblivious to inheritance, and our approach is no exception. One of the problems is that each class in a class hierarchy defines its own fields and manipulates them. Fugue proposed to compartmentalize objects into *frames* [11]. Each frame corresponds to a class in the object's class hierarchy.

Unlike state refinements, state invariants are *not* inherited by subclasses. Separate state invariants for each class let us associate separate typestates with each frame. The state of the "virtual" frame (that corresponds to the runtime type of the object) represents the overall state of the object. Fugue essentially forced all frames of an object to be in the same typestate. Moreover, all methods had to be overridden by all subclasses. Calls to *super* were possible and essentially required in order to keep frame typestates consistent.

Following previous work we allow subclasses to explicitly express their expectations of the super-frame's state, thereby decoupling typestates of different frames. This is for example necessary for defining a buffered stream as a subclass of a "filter" that forwards calls to an "underlying" stream, as implemented in the Java I/O library [4]. The filter's state is always the same as the underlying stream's. But the buffered input stream caches characters internally and can therefore still be in state within while the inherited filter is already eof.

In order to realize this idea we allow the specification of permissions for `super` in state invariants. State invariants can refer to fields defined in the current class and typestates of the immediately extended class. Thus all fields are "private" to a class frame.

$$\textit{references} \quad r \quad ::= \quad \ldots \mid \texttt{super} \mid \texttt{this}_{\mathsf{fr}} \qquad \textit{super frame, this frame}$$

Thus permissions actually give access to a particular frame. The *object permissions* we defined in sect. 4.3 are permissions to the "virtual frame". They can be used for "entering" an object through a dynamically dispatched call. In method specifications we distinguish permissions for the receiver's "current" frame with $\texttt{this}_{\mathsf{fr}}$ from normal permissions.

Only methods that require frame permissions have to be overridden; this lets us treat object permissions as frame permissions in dynamically dispatched calls. Permissions for the receiver's current frame are needed for methods that access fields. If a method merely forwards calls then it only needs object permissions and need not be overridden.[1] We believe that this distinction significantly reduces overriding burden.

4.6 Behavioral Subtyping

Subclasses should be allowed to define their own specifications, e.g. to add precision or support additional behavior [4]. However, subclasses need to be *behavioral subtypes* [26] of the extended class. Our system enforces behavioral subtyping in two steps. Firstly, state space inheritance conveniently guarantees that states of subclasses *always* correspond to states defined in superclasses [4]. Secondly, we make sure that every overriding method's specification implies the overridden

[1]A call can be forwarded to an argument or to the receiver itself. The latter occurs when base class methods implement functionality in terms of other methods.

$$\frac{(z : H) \in \Gamma}{\Gamma \vdash z : H} \qquad \frac{}{\Gamma \vdash 1 : \textsf{Fract}} \qquad \frac{}{\Gamma \vdash 0 : \textsf{Fract}} \qquad \frac{\Gamma \vdash k : \textsf{Fract}}{\Gamma \vdash k/2 : \textsf{Fract}}$$

$$\frac{\Gamma \vdash \overline{k} : \textsf{Fract} \quad (k \neq 0)}{\Gamma \vdash \overline{n \mapsto k} : \overline{n} \to \textsf{Fract}} \quad \frac{\Gamma \vdash g : \overline{n} \to \textsf{Fract}}{\Gamma \vdash g/2 : \overline{n} \to \textsf{Fract}} \quad \frac{\Gamma \vdash g : \overline{n} \to \textsf{Fract} \quad \Gamma \vdash g' : \overline{n'} \to \textsf{Fract}}{\Gamma \vdash g, g' : \overline{n}, \overline{n'} \to \textsf{Fract}}$$

Figure 6: Fraction typing

$$\frac{\begin{array}{c}\Gamma \vdash r : C \quad C \vdash A \prec n \\ \Gamma \vdash g : \textsf{up}_C(n) \to \textsf{Fract} \quad \Gamma \vdash k : \textsf{Fract}\end{array}}{\Gamma \vdash \textsf{access}(r, n, g, k, A) \textsf{ Permission}}$$

Figure 7: Well–formed permissions

method's specification [4] using the override judgment (fig. 10) that is used in checking method declarations (fig. 9). This check leads to method specifications that are contra-variant in the domain and co-variant in the range as required by behavioral subtyping.

5 Modular Typestate Verification

5.1 Permission Tracking

This section shows how we check method implementations against the permission-based specifications introduced in the last section. What we describe here is a modular static typestate checking technique that allows us to guarantee at compile-time that the behavioral specifications of a program will never be violated at runtime. We emphasize that our approach does not require tracking typestates at run time.

We permission-check an expression e with the judgment $\Gamma; \Delta \vdash^i_C e : \exists x : T.P \setminus \mathcal{E}$. This is read as, "in valid context Γ and linear context Δ, an expression e executed within receiver class C produces an object of type T and permissions P and affects fields in $\mathcal{E}$". The permissions in Δ are consumed in the process. We omit the receiver C where it is not required for checking a particular syntactic form. The set $\mathcal{E}$ keeps track of fields that were assigned to, which is important for the correct handling of permissions to fields. It is omitted when empty. The marker i in the judgment can be 0 or 1 where $i = 1$ indicates that states of objects in the context may change during evaluation of the expression. This will help us reason about temporary state assumptions. A combination of markers with $i \vee j$ is 1 if at least one of the markers is 1.

$$\begin{array}{rrcl} \textit{valid contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, z : H \mid \Gamma, l : C \mid \Gamma, q \\ \textit{linear contexts} & \Delta & ::= & \cdot \mid \Delta, P \\ \textit{effects} & \mathcal{E} & ::= & \cdot \mid \mathcal{E}, f \end{array}$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{(l : C) \in \Gamma}{\Gamma \vdash l : C}\ \textsc{T-Loc}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textsc{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textsc{T-False}$$

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \quad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1\ \texttt{and}\ t_2 : \texttt{bool}}\ \textsc{T-And} \qquad \frac{\Gamma \vdash t_1 : \texttt{bool} \quad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1\ \texttt{or}\ t_2 : \texttt{bool}}\ \textsc{T-Or}$$

$$\frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash \texttt{not}\ t : \texttt{bool}}\ \textsc{T-Not} \qquad \frac{\Gamma \vdash t : C' \quad C'\ \textsf{extends}\ C}{\Gamma \vdash t : C}\ \textsc{T-Sub}$$

Figure 8: Term typechecking

Valid and linear contexts distinguish valid (permanent) facts (Γ) from resources (Δ). Resources are tracked linearly, forbidding their duplication, while facts can be used arbitrarily often. (In logical terms, contraction is defined for facts only). The valid context types object variables, fraction variables, and location types and keeps track of facts about terms q. Fraction variables are tracked in order to handle fraction quantification correctly. The linear context holds currently available resource predicates.

Fractions and fraction functions are formally typed in figure 6. Note that fraction function types keep track of exactly which nodes are mapped. We use this to check that the fraction function of a permission covers exactly the nodes between (and including) the permission's root node and the state space root alive. Fraction typing lets us define permission validity (figure 7

The judgment $\Gamma \vdash t : T$ types terms (figure 8). It includes the usual rule for subsumption using nominal subtyping induced by the extends relation. Term typing is used in expression checking.

Our expression checking rules are syntax-directed up to reasoning about permissions. Permission reasoning is deferred to a separate judgment $\Gamma; \Delta \vdash P$ that uses the rules of linear logic to prove the availability of permissions P in a given context. This judgment will be discussed in sect. 5.5. Permission checking rules for most expressions appear in fig. 9 and are described in turn. Packing, method calls, and field assignment are discussed in following subsections. Helper judgments are summarized in fig. 10. The notation $[t/r]e$ substitutes t for occurrences of r in e.

- P-TERM embeds terms. It formalizes the standard logical judgment for existential introduction and has no effect on existing objects.

- P-FIELD checks field accesses in a similar way to P-TERM.

- P-NEW checks object construction. The parameters passed to the constructor have to satisfy initialization predicate P and become the object's initial field values. The new existentially quantified object is associated with a full permission to the root state (with full fraction) that makes state assumptions according to the declared start state A. Object construction has no effect on existing objects.

$$\frac{\Gamma \vdash t : T \quad \Gamma; \Delta \vdash [t/x]P}{\Gamma; \Delta \vdash^0 t : \exists x : T.P} \text{ P-TERM} \qquad \frac{\mathsf{localFields}(C) = \overline{f : T} \quad \Gamma; \Delta \vdash [f_i/x]P}{\Gamma; \Delta \vdash^0_C f_i : \exists x : T_i.P} \text{ P-FIELD}$$

$$\frac{\Gamma \vdash \overline{t : T} \quad \mathsf{init}(C) = \langle \exists \overline{f : T}.P, A \rangle \quad \Gamma; \Delta \vdash [\overline{t/f}]P}{\Gamma; \Delta \vdash^0 \texttt{new}\ C(\overline{t}) : \exists x : C.\mathsf{full}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, A)} \text{ P-NEW}$$

$$\frac{\begin{array}{c}(\Gamma, t = \texttt{true}); \Delta \vdash^i e_1 : \exists x : T.P_1 \setminus \mathcal{E}_1 \\ \Gamma \vdash t : \texttt{bool} \quad (\Gamma, t = \texttt{false}); \Delta \vdash^j e_2 : \exists x : T.P_2 \setminus \mathcal{E}_2\end{array}}{\Gamma; \Delta \vdash^{i \vee j} \texttt{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-IF}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash^i e_1 : \exists x : T.P \setminus \mathcal{E}_1 \quad (\Gamma, x : T); (\Delta', P) \vdash^j e_2 : E_2 \setminus \mathcal{E}_2 \\ i = 1 \textit{ implies no temporary assumptions in } \Delta' \quad \textit{Fields } \mathcal{E}_1 \textit{ do not occur in } \Delta'\end{array}}{\Gamma; (\Delta, \Delta') \vdash^{i \vee j} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : E_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-LET}$$

$$\frac{\begin{array}{c}(\overline{x : T}, \texttt{this} : C); P \vdash^i_C e : \exists \textit{result} : T_r.P_r \otimes \top \setminus \mathcal{E} \quad E = \exists \textit{result} : T_r.P_r \\ \mathsf{override}(m, C, \forall \overline{x : T}.P \multimap E)\end{array}}{T_r\ m(\overline{T\ x}) : P \multimap E = e\ \mathsf{ok\ in}\ C} \text{ P-METH}$$

$$\frac{\ldots \quad \overline{M}\ \mathsf{ok\ in}\ C \quad \overline{M} \textit{ overrides all methods with frame permissions in } C'}{\texttt{class C extends C}'\ \{\ \overline{\texttt{F}}\ \overline{\texttt{R}}\ \texttt{I}\ \overline{\texttt{N}}\ \overline{\texttt{M}}\ \}\ \mathsf{ok}} \text{ P-CLASS}$$

$$\frac{\overline{CL}\ \mathsf{ok} \quad \cdot; \cdot \vdash^i_{\_} e : E \setminus \mathcal{E}}{\langle \overline{CL}, e \rangle : E} \text{ P-PROG}$$

Figure 9: Permission checking for expressions (part 1)

The judgment init (fig. 10) yields initialization predicate and initial state for a class. The start state is a conjunction of states (fig. 5). The initialization predicate is the invariant needed for the start state.

- P-IF introduces non-determinism into the system, reflected by the disjunction in its type. We make sure that the predicate is of Boolean type and then assume its truth in checking the *then* branch (e_1). Similarly, we assume the falsehood of the predicate in checking the *else* branch (e_2). This approach lets branches make use of the conditional.

- P-LET checks a `let` binding. Since variables are terms, `let` can be used to bind new objects, fields, or method results in subsequent expressions. The linear context used in checking the second subexpression must not contain permissions for fields affected by the first expression. This makes sure that old permissions to fields do not "survive" assignments and packing. Moreover, temporary state information are dropped if the first subexpression has side effects.

A program consists of a list of classes and a main expression (P-PROG, fig. 9). As usual, the class table $\overline{CL}$ is globally available. The main expression is checked with initially empty contexts. The judgment CL ok (P-CLASS) checks a class declaration. It checks fields, states, and invariants for syntactic correctness (omitted here) and verifies consistency between method specifications and implementations using the judgment M $\mathsf{ok\ in}$ C. P-METH assumes the specified pre-condition of a method (i.e. the left-hand side of the linear implication) and verifies that the method's body expression produces the declared post-condition (i.e. the right-hand side of the implication). Conjunction with $\top$ drops excess permissions, e.g. for garbage-collected objects. Notice that a method itself is not a linear resource since all resources it uses (including the receiver) are passed in upon invocation.

5.2 Packing and Unpacking

We use a refined notion of *unpacking* [11]: we unpack and pack a specific permission. Unpacking a permission gives access to the part of the object covered by that permission. The access we gain to fields reflects the kind of permission we unpacked. Full and shared permissions give modifying access, while a pure permission gives read-only access to underlying fields.

To avoid inconsistencies, objects are always fully packed when methods are called. Thus at any given time, only one method can unpack an object. To further simplify the situation, only one permission can be unpacked at the same time. Intuitively, this approach "focuses" [12] on the permission being unpacked. This lets us improve usability of share permissions by unpacking them like full permissions, gaining full rather than shared access to underlying fields (if available). The syntax for packing and unpacking is as follows.

$$\begin{array}{llllll} \textit{expressions} & e & ::= & \dots & \mid \ \texttt{unpack}(n,k,A)\ \texttt{in}\ e & \textit{unpacking} \\ & & & & \mid \ \texttt{pack}\ n\ \texttt{to}\ A\ \texttt{in}\ e & \textit{packing} \end{array}$$

Packing and unpacking always affects the receiver of the currently executed method. The parameters to packing and unpacking express the programmer's expectations about the permission she is unpacking. In particular, n denotes the subtree in the state space the permission should cover. A are the assumptions about states within that subtree that need to be satisfied. For simplicity, an explicit subtree fraction k is part of packing expressions. It could be inferred from a programmer-provided permission kind like "share".

In order for pack to work properly we have to "remember" the permission we unpacked. Therefore we introduce unpacked as an additional linear predicate.

$$\textit{permissions} \quad p \quad ::= \quad \dots \quad \mid \ \mathsf{unpacked}(n,g,k,A)$$

The checking rules for packing and unpacking are given in fig. 11. Notice that packing and unpacking always affects permissions to $\texttt{this}_{\mathsf{fr}}$, the frame of the receiver in which the surrounding method is defined. (We ignore substitution of this with a location at runtime here.) P-UNPACK first derives the permission to be unpacked. The helper judgment inv determines a predicate that describes the receiver's fields based on the permission being unpacked. It is used for checking the body expression e. An unpacked predicate is added into the linear context that lets field assignments and packing work correctly. We can prevent multiple permissions from being unpacked at the same time using a straightforward dataflow analysis [9] (omitted here).

P-PACK does the opposite of P-UNPACK. It derives the field predicate necessary for packing the given permission and then assumes that permission in checking the body expression. Notice how P-PACK verifies that the receiver was unpacked before. The state assumption A can differ from before only if a modifying permission was unpacked. Finally, the rule ensures that field permissions do not "survive" packing.

Invariant transformation. The judgment $\mathsf{inv}_C(n, g, k, A)$ essentially determines what it means to possess an atomic permission $\mathsf{access}(\mathit{this}_{\mathsf{fr}}, n, g, k, A)$ for an object of (runtime) class C. It is defined in fig. 12. It uses the purify function (fig. 13) that converts all atomic permissions into pure permissions. Unpacking a full or shared permission with root node n yields purified permissions for nodes "above" n and includes invariants following from state assumptions as–is. Conversely, unpacking a pure permission yields completely purified permissions.

Example: Dynamic State Tests. A dynamic state test for a state *s* is a method with a type like $\forall g : \mathsf{alive} \to \mathsf{Fract}.\mathsf{pure}(\mathtt{this}, \mathsf{alive}, g) \multimap \exists b : \mathtt{bool}.(b = \mathtt{true} \otimes \mathsf{pure}(\mathtt{this}, \mathsf{alive}, g, s)) \oplus (b = \mathtt{false} \otimes \mathsf{pure}(\mathtt{this}, \mathsf{alive}, g, s'))$ that can be implemented as follows. This example makes the simplifying assumption that the object contains a Boolean field *flag* that is true iff the object is in state *s*.

$$\begin{array}{l}\mathtt{unpack}(\mathsf{alive}, 0, \mathsf{alive})\ \mathtt{in}\ \mathtt{let}\ x = \mathit{flag}\ \mathtt{in} \\ \quad \mathtt{if}(x, \mathtt{pack}\ \mathsf{alive}\ \mathtt{to}\ s\ \mathtt{in}\ \mathtt{true}, \mathtt{pack}\ \mathsf{alive}\ \mathtt{to}\ s'\ \mathtt{in}\ \mathtt{false})\end{array}$$

5.3 Calling Methods

We distinguish virtual calls and calls to inherited methods. Checking any method call expression involves proving the method's pre-condition. The expression is typed with the corresponding post-condition. Unfortunately, calling a method can result into callbacks. In order to ensure that objects are always consistent when called we require them to be fully packed before method calls. This can be ensured with a simple dataflow analysis [9].

While this rule may seem unnatural at first, it reflects that aliased objects have to be prepared for callbacks. Note that the packing requirement is not a strong limitation. We can always pack to some intermediate state. Moreover, intermediate packing removes the need for adoption as in existing work [12]. Instead, the intermediate state represents the situation where an adopted object was taken out of the adopting object. Inferring intermediate states as well as identifying where callbacks are impossible are areas for future research.

Virtual calls. Virtual calls are dynamically dispatched (rule P-CALL). In virtual calls, frame and object permissions are identical because object permissions simply refer to the object's virtual frame. This is achieved by substituting the receiver for both *this* and $\mathit{this}_{\mathsf{fr}}$.

Super calls. Super calls are statically dispatched (rule P-SUPER). We substitute `super` only for $\mathit{this}_{\mathsf{fr}}$. Recall that `super` is used to identify permissions to the super-frame. We omit a substitution of *this* for the receiver (*this* again) for clarity.

5.4 Field Assignments

Assignments to fields change the state of the receiver's current frame. We point out that assignments to a field do *not* change states of objects referenced by the field. Therefore reasoning about assignments mostly has to be concerned with preserving invariants of the receiver. Again, unpacked predicates help us with this task.

Our intuition is that assignment to a field requires unpacking the surrounding object to the point where all states that refer to the assigned field in their invariants are revealed. Notice that the object does not have to be unpacked completely in this scheme. For simplicity, each field is annotated with the subtree that can depend on it. Thus we interpret subtrees as data groups [24], and every field is mapped into one of them.

The rule P-ASSIGN (fig. 11) assigns a given object t to a field f_i and returns the old field value as an existential x'. This preserves information about that value. It verifies that the new object is of the correct type and that a suitable full or share permission is currently unpacked. By recording an effect on f_i we ensure that information about the old field value cannot flow around the assignment (which would be unsound).

5.5 Permission Splitting and Joining

Our permission checking rules rely on the ability to prove a permission with the current resources, written $\Gamma; \Delta \vdash P$ (figure 14). We use standard rules for the multiplicative-additive fragment of linear logic (MALL) with quantifiers that only range over fractions. This fragment has been proven decidable [25]. Following Boyland [6] we add a rule SUBST that introduces a notion of substitution into the logic. It allows to substitute a set of linear resources with an equivalent one.

$$\frac{\Gamma; \Delta \vdash P' \quad P' \Rrightarrow P}{\Gamma; \Delta \vdash P} \text{ SUBST}$$

The judgment $P \Rrightarrow P'$ defines legal transformations similar to a subtyping judgment in conventional type systems. We use substitutions for splitting and joining permissions with the rules shown in fig. 15. The symbol $\Lleftarrow\Rrightarrow$ indicates that transformations are allowed in both directions. We explain each rule in turn.

SYM symmetrically splits a permission into two equivalent permissions. Notice how fractions are split. ASYM asymmetrically splits a pure permission off a given permission. Here, the subtree fraction k is untouched, reflecting the asymmetric split. Both transformations can be inverted.

F-SPLIT-$\otimes$ splits a full permission with a conjunctive state assumption into a conjunction of full permissions. F-JOIN-$\otimes$ inverts F-SPLIT-$\otimes$ but requires the fraction on the new root node to be 1. This guarantees that no additional full or shared permissions exist in the new permission's subtree. F-$\oplus$ splits and conjoins full permissions with a disjunction of state assumptions. Since only one of the two state assumptions can be true at a given time we do not need to split fractions.

F-DOWN limits a full permission to a smaller subtree by moving the root node down in the state space. The fraction function is appended with additional 1 fractions for nodes that are above the moved root. Notice that this operation is only allowed if any state assumptions of the original permission can be preserved. F-UP does the opposite but like F-JOIN-$\otimes$ it requires the fraction on the new root node to be 1. Similarly, P-UP can be used to weaken a pure permission by moving its root up in the state space. Finally, FORGET allows a permission to "forget" its state assumption. This rule is used to drop temporary state assumptions.

Our splitting and joining rules will maintain a consistent set of permissions for each object. Permissions to a subtree in the state space of a runtime object are consistent if there exists at most one full permission and an arbitrary number of pure permissions to the subtree. Moreover, an arbitrary number of share permissions is allowed to exist if and only if no full permission exists. Fractions k of all permissions to the subtree must sum up to (at most) 1. Furthermore, all other permissions to the object refer to parts of the state space that are orthogonal to the subtree (e.g. in a different state dimension). Finally, fraction functions of all permissions to an object sum up to (at most) 1.

6 Breaking an Invariant in Java Buffered Input Streams

To illustrate how verification proceeds, figure 16 shows a simplified version of the `fill` method in `java.io.BufferedInputStream` written in our core language. `BufferedInputStream` buffers characters from an underlying stream to make reading more efficient. In Sun's current Java standard library implementation (Java 5 and 6), `fill` is responsible for retrieving more characters from the underlying stream if its character buffer is depleted.

As can be seen we need an intermediate state reads and a marker field `reading` that indicate an ongoing call to the underlying stream. We also need an additional state refinement to specify the internal methods that implement reading from the underlying stream. (We assume that $\mathit{this}_{\mathsf{fr}}$ permissions can be used for calls to `private` methods.)

Maybe surprisingly, we have to re-assign field values after `super.read()` returns. The reason is that when calling `super` we loose temporary state information for *this*. Assignment re-establishes this information and lets us pack properly before calling `doFill` recursively or terminating in the case of a full buffer or a depleted underlying stream.

It turns out that these re-assignments are *not* just an inconvenience in our method but point to a real problem in the Java standard library implementation. It is possible to break an invariant in `BufferedInputStream` through a reentrant callback. In a nutshell, if an underlying stream calls back into the buffer to read then the following happens:

1. The underlying stream is called again, potentially overriding buffer content. If the underlying stream keeps calling back then the program goes into an infinite loop.

2. The second call into read advances the buffer's *pos* field to $pos > 0$. Later, the buffer's *count* field will be set as $pos + buffer.length$, thereby violating the invariant that $count < buffer.length$. Ultimately, the buffer will try to read behind the end of its buffer array, causing an undocumented `ArrayIndexOutOfBoundsException`.

The following implementation of an underlying stream exposes this problem (tested with Java 6, build 1.6.0_b105, and two versions of Java 5).

```
package test.java.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.BufferedInputStream;

/**
 * @author Kevin Bierhoff
 *
 */
public class MaliciousStream extends InputStream {

    private BufferedInputStream loop;
    private int callCount = 0;

    public MaliciousStream() {
        super();
    }

    @Override
    public int read() throws IOException {
        // never called
        return -1;
    }

    @Override
    public int read(byte b[], int off, int len) throws IOException {
        int calls = ++callCount;
        if(calls < 2) {
            System.out.println("Recursive read: " + loop.read());
        }
        System.out.println("Fill " + calls + " to " +
            b.hashCode() + "[" + off + ".." + (off+len) + "]");
        if(b[0] != 0)
            System.out.println("Overriding content");
        for(int i = off; i < off + len; i++) {
            b[i] = (byte) (calls & 0xFF);
        }
        return len;
    }

    public void setLoop(BufferedInputStream loop) {
        this.loop = loop;
    }

    public static void main(String[] args) {
        MaliciousStream h = new MaliciousStream();
        BufferedInputStream b = new BufferedInputStream(h);
        h.setLoop(b);

        int i = 0;
        try {
            int c, oldc = -1;
            for(; i < 30000; i++) {
                c = b.read();
                if(c != oldc) {
                    System.out.println("Character " + i +
                        " switches to " + c);
                    oldc = c;
                }
            }
        }
        catch (Exception e) {
            System.err.println("Exception in iteration " + i);
            e.printStackTrace();
        }
    }
}
```

Running the `main` method will terminate the program with an `ArrayIndexOutOfBoundsException` and produce the following output:

```
Fill 2 to 17523401[0..8192]
Recursive read: 2
Fill 1 to 17523401[0..8192]
Overriding content
Character 0 switches to 1
Exception in iteration 8191
java.lang.ArrayIndexOutOfBoundsException: 8192
    at java.io.BufferedInputStream.read(BufferedInputStream.java:239)
    at test.java.io.MaliciousStream.main(MaliciousStream.java:57)
```

The buffer array is filled twice, first with 2's and then with 1's. The buffer's client (the `main` method) never sees the 2's because they are immediately overridden with 1's. The exception is thrown when all characters from the buffer array were read and the buffer attempts to read the first cell behind the end of the array. The buffer attempts to read behind the end of the array because its field *count*—which indicates how many characters are currently buffered—is larger than the length of the buffer array, as explained above. We submitted this issue to Sun on 9 March 2007.

In our approach, this problem is avoided. Because `fill` operates on a share permission, our verification approach forces taking into account possible field changes through reentrant calls with other share permissions. (This is precisely what our malicious stream does.) We could avoid field reassignments by having `read` require a full permission, thereby documenting that (modifying) reentrant calls are not permitted for this method.

7 Soundness

This section proves soundness for a fragment of the system presented in the previous sections. To this end, we define a simple core language with an instrumented dynamic semantics that tracks fractions for references and states for objects. Since our approach guarantees protocol compliance at compile time it is not actually necessary to track fractions or states during execution; we only do this for the purpose of proving soundness.

The dynamic semantics is given as a small-step evaluation semantics that modifies a heap. Heaps track fractions for references and states for objects. In order to properly manipulate heaps we need to include fractions into our expressions, e.g., when passing an object as an argument to a method. It is assumed that typechecking ensures that the used fraction is available according to the current permission set. One could actually insert fractions into expressions based on typechecking.

Heaps track fractions both for stack and field references. Objects are represented as $k \cdot o \mapsto C(\overline{f = k \cdot o})@S$, which is read as, "object o of class C with field values $\overline{o}$ for its fields $\overline{f}$. $\overline{f}$ are the fields defined in class C. The initial k indicates the fraction that is currently available on the stack for accessing o. The fractions $\overline{k}$ are fractions of the referenced objects that are held by o's fields.

Compared to the full system, the fragment proven sound has the following limitations.

- Inheritance and subtyping are not supported.
- Only permissions for objects as a whole are supported and expressed as $k \cdot r@s$. The fraction k distinguishes pure ($k = 0$), shared ($0 < k < 1$) and full ($k = 1$) access. Thus, permissions for dimensions are not supported. Therefore, we do not include state dimensions into the formal system as they can be easily encoded with separate states for each element of the cross product of states from different dimensions.
- Specifications are deterministic.
- We assume that all expressions can have effects. This means that temporary state information is almost immediately forgotten.

In future work we plan to extended this fragment to support structural subtyping, non-deterministic specifications, and effect tracking for expressions similar to the formal system presented in preceding sections. We plan to encode inheritance, state dimensions, and permissions for subtrees of the state space of an object in this extended fragment.

Despite the simplified syntax, typing rules of the core language are largely unchanged. The inv_C judgment discussed before is dramatically simplified in the absence of permissions for subtrees. To simplify the dynamic semantics we assume that $\mathsf{inv}_C(s, k)$ refers to fields f with *this*$.f$. (As before, if $k = 0$ then the state invariant is purified, see figure 13.) We allow unpacking arbitrary objects, not just the receiver as in the system presented before. We keep insisting that only one object is unpacked at a time and that objects be packed before any methods are called. This ensures that effects are only collected for one object at a time (proven in a separate lemma). In order to so, the typechecking judgment for our core system keeps track of what object is currently unpacked in a separate context.

7.1 Core Language Syntax

terms	t	$::=$	x	*variable*
		$\mid$	o	*object*
expressions	e	$::=$	$k \cdot t$	*term*
		$\mid$	$\texttt{new}\, C(\overline{k \cdot t})$	*object construction*
		$\mid$	$k \cdot t.m(\overline{k \cdot t})$	*method call*
		$\mid$	$\texttt{let}\, x = e_1 \,\texttt{in}\, e_2$	*binding*
		$\mid$	$\texttt{unpack}\, k \cdot t@s \,\texttt{in}\, e$	*unpack*
		$\mid$	$\texttt{pack}\, t \,\texttt{to}\, s \,\texttt{in}\, e$	*pack*
		$\mid$	$k \cdot t.f$	*field read*
		$\mid$	$t_1.f := k \cdot t_2$	*assignment*
expr. types	E	$::=$	$\exists x : C.P$	
class decls.	*CL*	$::=$	$\texttt{class}\, C\, \{\, \texttt{states}\, \overline{s}\, \texttt{refine}\, \mathsf{alive}; \overline{C\ f}\ I\ \overline{N}\ \overline{M} \}$	
methods	M	$::=$	$C_r\ m(\overline{C\ x}) : P \multimap \exists \mathit{result} : C_r.P = e$	
initialization	I	$::=$	$\texttt{initially}\, \langle \exists \overline{f : C}.P, s \rangle$	
invariants	N	$::=$	$s = P$	
references	r	$::=$	t	*terms*
		$\mid$	$t.f$	*fields*
predicates	P	$::=$	$k \cdot r@s$	*permission*
		$\mid$	$P_1 \otimes P_2$	*conjunction*
valid contexts	Γ	$::=$	$\cdot \mid \Gamma, x : C$	
stores	Σ	$::=$	$\cdot \mid \Sigma, o : C$	
linear contexts	Δ	$::=$	$\cdot \mid \Delta, P$	
heaps	μ	$::=$	$\cdot \mid \mu, k \cdot o \mapsto C(\overline{f = k \cdot o})@S$	
object states	S	$::=$	s	*packed in state*
		$\mid$	$\mathsf{Unpacked}(k)$	*unpacked modifying*
		$\mid$	$\mathsf{Unpacked}(s)$	*unpacked read-only in state*
packing flags	u	$::=$	$-$	*no object unpacked*
		$\mid$	$k \cdot t@s$	*unpacked object*
effects	$\mathcal{E}$	$::=$	$\emptyset \mid \{t.f\} \mid \mathcal{E}_1 \cup \mathcal{E}_2$	
fractions	k	$\in$	$[0, 1]$	
class names	C			
method names	m			
variable names	x			
field names	f			
state names	s			
object locations	o			

7.2 Judgment Forms

Judgment	Judgment form	Explanation
Evaluation	$e\|\mu \longmapsto e'\|\mu'$	In heap μ, expression e evaluates to e', changing the heap to μ', in one step.
Expression typing	$\Gamma\|\Sigma\|\Delta\|u \vdash e : E \setminus \mathcal{E}\|u'$	In variable context Γ, store Σ, linear context Δ, and unpacking flag u, expression e has type E and may assign to fields in $\mathcal{E}$ and and changes unpacking to u'.
Store typing (definition 1)	$\Sigma\|\Delta\|u \vdash \mu$	With store Σ, linear context Δ, and packing flag u, heap μ is well-typed.
Linear logic entailment (figure 14)	$\Gamma\|\Sigma\|\Delta \vdash P$	In variable context Γ and store Σ, linear context Δ proves P.
Runtime property check (definition 5)	$\mu\|\overline{k \cdot o} \vdash P$	Heap μ restricted to stack permissions $\overline{k \cdot o}$ satisfies property P.

7.3 Preservation

Definition 1 (Store Typing) *If*

- $\textsf{dom}(\Sigma) = \textsf{dom}(\mu)$
- *If $u = -$ then all objects in μ are packed and no permissions in Δ refer to fields*
- *If $u = k_u \cdot o_{\textsf{unp}}@s_u$ then $k_o \cdot o_{\textsf{unp}} \mapsto C(\ldots)@S$ is the only object unpacked in μ and all permissions to fields in Δ refer to fields of $o_{\textsf{unp}}$ and either (a) $k_u = 0$ and $S = \textsf{Unpacked}(s')$, where $s' \leq s_u$, or (b) $k_u > 0$ and $S = \textsf{Unpacked}(k_u)$.*
- $\forall o \in \textsf{dom}(\Sigma)$*: If $k \cdot o \mapsto C(\overline{f = k \cdot o})@S \in \mu$ then*
  - $(o : C) \in \Sigma$
  - *Either $S = \textsf{Unpacked}(k)$ or $S = \textsf{Unpacked}(s)$ and $[o/\textit{this}]\textsf{inv}_C(s, 1)$ is satisfied by o's fields or $S = s$ and $[o/\textit{this}]\textsf{inv}_C(s, 1)$ is satisfied by o's fields*
  - $w(o, \Delta) \leq k$
  - *If $\cdot|\Sigma|\Delta \vdash k' \cdot o@s' \otimes \top$ then $S \leq s'$ and $k' \leq k$.*
  - *If $\cdot|\Sigma|\Delta \vdash k'_i \cdot o.f_i@s \otimes \top$ then $k'_i \leq k_i$ (and $o = o_{\textsf{unp}}$) and $k_o \cdot o_i \mapsto C_o(\ldots)@s_o \in \mu$ and $s_o \leq s$ and either $S = \textsf{Unpacked}(s')$, which implies $k'_i = 0$, or $S = \textsf{Unpacked}(k')$*

then $\Sigma|\Delta|u \vdash \mu$

Definition 2 (Heap Manipulations) *For a given heap μ,*

- *$\mu[k \cdot o \mapsto C(\overline{f = k \cdot o})@S]$ replaces the entry for o in μ with the information given in $[\ldots]$*
- *If $k' \cdot o \mapsto C(\ldots)@S \in \mu$ then*

  – *If* $k' - k \geq 0$ *then* $\mu - k \cdot o = \mu[(k' - k) \cdot o \mapsto C(\ldots)@S$
  – *If* $k' + k \leq 1$ *then* $\mu + k \cdot o = \mu[(k' + k) \cdot o \mapsto C(\ldots)@S$

Definition 3 (Object Weight) *For any* o, μ, *and* Δ*:*

- $w(o, \mu) = \sum_{k \cdot o \in \mu} k$
- $w(o, \Delta) = \sum_{k \cdot o \in \Delta} k$
- $w(o, u) = \begin{cases} k \text{ if } u = k \cdot o@s \\ 0 \text{ otherwise} \end{cases}$

Definition 4 (State Ordering) *The relation* $S \leq S'$ *is defined with the following rules:*

$$\frac{}{S \leq S} \qquad \frac{}{S \leq \textsf{\textit{alive}}} \qquad \frac{}{\textsf{\textit{Unpacked}}(s) \leq s} \qquad \frac{}{s \leq \textsf{\textit{Unpacked}}(s)}$$

Note: These rules equate any state s with read-only unpacking in that state ($\textsf{Unpacked}(s)$). Moreover, modifying unpacked ($\textsf{Unpacked}(k)$) is considered a substate of $\textsf{alive}$. This convention simplifies store typing because it makes a state information $\textsf{alive}$ about an object valid even if that object is unpacked.

Definition 5 (Property Satisfied at Runtime) *If*

- $\overline{k' \cdot o \mapsto C(\ldots)@s} \subseteq \mu$
- $\cdot|\overline{o : C}|\overline{k \cdot o@s} \vdash P$ *(an instance of* $\Gamma|\Sigma|\Delta \vdash P$*)*
- $\overline{k \leq k'}$

then $\mu|\overline{k \cdot o} \vdash P$

Lemma 1 (Inversion) *If* $\Gamma|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u'$ *then*

- *If* e *is* $\textsf{\textit{new}}\ C(\overline{k \cdot t})$ *then* $\mathcal{E} = \emptyset$ *and* $E = \exists x : C.1 \cdot x@s$ *and* $\Gamma|\Sigma|\Delta \vdash [\overline{o}/\overline{f}]P$ *and* $u = u'$.
- *If* e *is* $k \cdot t.m(\overline{k \cdot t})$ *then* $\mathcal{E} = \emptyset$ *and* $E = E'$ *and* $\Gamma|\Sigma|\Delta \vdash [t/\textit{this}][\overline{t}/\overline{x}]P$ *and* $\Gamma|\Sigma \vdash t : C$ *and* $\Gamma|\Sigma \vdash \overline{t : C}$ *and* $\textsf{\textit{mtype}}(C, m) = \forall \overline{x : C}.P \multimap E$ *and* $\overline{x : C}, \textit{this} : C|\cdot|P|- \vdash e_m : E \setminus \emptyset|-$, *where* $\textsf{\textit{mbody}}(C, m) = \overline{x}.e_m$, *and* $u = u' = -$.
- $\ldots$

Lemma 2 (Substitution) *If* $\Gamma, \overline{x : C}|\Sigma|\Delta, P|u \vdash e : E \setminus \mathcal{E}|u'$, *where variables* $\overline{x}$ *do not occur in* Δ *and* $\mathcal{E}$, *and these exists* Δ' *and objects* $\overline{o}$ *such that* $\Gamma|\Sigma|\Delta' \vdash [\overline{o}/\overline{x}]P$ *then* $\Gamma|\Sigma|\Delta, [\overline{o}/\overline{x}]P|u \vdash e : E \setminus \mathcal{E}|u'$.

Lemma 3 *If* $\Gamma|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u'$ *then either (a)* $u = -$ *and* $\mathcal{E} = \emptyset$ *or (b)* $u = k \cdot t@s$ *and* $\mathcal{E}$ *contains only fields of* t.

Proof: *(a) $u = -$ is not a valid precondition for producing effects (using assignment or packing). (b) By induction on typing derivations, using (a). Only one object can be unpacked at a time, permission for unpacked object is needed for assignments and packing, and effect of `unpack` expression is $\emptyset$.*

Lemma 4 (Compositionality) *If $\Sigma|\Delta|u \vdash \mu$ and $\Delta = \Delta_1, \Delta_2$ then $\Sigma|\Delta_1|u \vdash \mu$ and $\Sigma|\Delta_2|u \vdash \mu$.*

Proof: *Immediate from the definition of store typing.*

Our preservation theorem is strengthened to preserve a "frame" of potential additional permissions around the expression being evaluated. This frame property is needed when appealing to the induction hypothesis. Since expressions are in let-normal form, only one case, E-LET-C, represents a congruence rule that appeals to the induction hypothesis.

The dynamic semantics relies on fractions being part of expressions in order to modify the heap accordingly, e.g., when reading a field. In typechecking expressions, we tacitly assume that permissions used for typechecking an expression have the fractions prescribed in the expression. A surface syntax could omit fractions in expressions and instead insert them automatically based on the permissions used in typechecking.

Theorem 1 (Preservation) *If*

- $\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ *and*
- $\Sigma|\Delta, \Delta^*|u \vdash \mu$, *where Δ^* are extra permissions that contain no temporary state information and no permissions for fields in $\mathcal{E}$, and*
- $e|\mu \longmapsto e'|\mu'$

then there exists

- $\Sigma' \supseteq \Sigma$ *and*
- u' *and*
- $\mathcal{E}'$, *where either (a) $e|\mu \longmapsto e'|\mu'$ unpacks an object o, i.e., $u = -$ and $u' = k \cdot o@s$ and $\mathcal{E}' - \mathcal{E}$ only mentions fields of o or (b) $\mathcal{E}' \subseteq \mathcal{E}$ and*
- Δ', *where Δ' contains no permissions for fields in $\mathcal{E} - \mathcal{E}'$*

such that

- $\cdot|\Sigma'|\Delta'|u' \vdash e' : E \setminus \mathcal{E}'|u''$ *and*
- $\Sigma'|\Delta', \Delta^* \vdash \mu'$ *and*
- $\forall o \in \textsf{dom}(\mu) : w(o,\mu) - w(o,\Delta) - w(o,u) \leq w(o,\mu') - w(o,\Delta') - w(o,u')$.

Proof: By structural induction on the derivation of $e|\mu \longmapsto e'|\mu'$.

CASE E-NEW

$$\frac{\mu|\overline{k \cdot o} \vdash [\overline{o}/\overline{f}]P \quad \mu'' = \mu - \overline{k \cdot o} \quad \mathsf{init}(C) = \langle \exists \overline{f : C}.P, s\rangle \quad (o^* \notin \mathsf{dom}(\mu))}{\mathtt{new}\ C(\overline{k \cdot o})|\mu \longmapsto 1 \cdot o^*|\mu'', 1 \cdot o^* \mapsto C(\overline{f = k \cdot o})@s}$$

Thus, e is $\mathtt{new}\ C(\overline{k \cdot o})$, e' is $1 \cdot o^*$, and $\mu' = \mu'', 1 \cdot o^* \mapsto C(\overline{f = k \cdot o})@s$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ Assumption
$u = u''$ and $\mathcal{E} = \emptyset$ and $E = \exists x : C.1 \cdot x@s$ and $\cdot|\Sigma|\Delta \vdash [\overline{o}/\overline{f}]P$ Inversion
Define $\Sigma' = (\Sigma, o^* : C)$ and $u' = u$ and $\mathcal{E}' = \emptyset = \mathcal{E}$ and $\Delta' = 1 \cdot o^*@S$
$\Sigma' \supseteq \Sigma$ By definition
$\cdot|\Sigma'|\Delta'|u \vdash 1 \cdot o^* : \exists x : C.1 \cdot x@s \setminus \emptyset|u$ By rule T-LOC
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u)$
$\overline{k \cdot o}$ move from stack to memory
$\Sigma'|\Delta', \Delta^*|u' \vdash \mu'$
Δ' only holds permission to o^*, existing objects do not change state, fractions in Δ^* remain valid

CASE E-CALL

$$\frac{\begin{array}{c}\mathsf{mbody}(C, m) = \overline{x}.e_m \quad \mathsf{mtype}(C, m) = \forall \overline{x : C}.P \multimap E' \\ \mu|k \cdot o, \overline{k \cdot o} \vdash [o/\textit{this}][\overline{o}/\overline{x}]P \quad \textit{all objects packed in } \mu\end{array}}{k \cdot o.m(\overline{k \cdot o})|\mu \longmapsto [o/\textit{this}][\overline{o}/\overline{x}]e_m|\mu}$$

Thus, e is $k \cdot o.m(\overline{k \cdot o})$, e' is $[o/\textit{this}][\overline{o}/\overline{x}]e_m$, and $\mu' = \mu$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ Assumption
$\mathcal{E} = \emptyset$ and $u = u'' = -$ and $E = E'$ and $\cdot|\Sigma|\Delta \vdash [o/\textit{this}][\overline{o}/\overline{x}]P$ Inversion
$\overline{x : C}, \textit{this} : C|\cdot|P|- \vdash e_m : E \setminus \emptyset|-$ Inversion (cont.)
Define $\Sigma' = \Sigma$ and $u' = -$ and $\mathcal{E}' = \emptyset = \mathcal{E}$ and $\Delta' = \Delta$
$\cdot|\Sigma|\Delta|- \vdash [o/\textit{this}][\overline{o}/\overline{x}]e_m : E \setminus \emptyset|-$ Substitution
$\Sigma|\Delta, \Delta^*|- \vdash \mu$ Given
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu) - w(o, \Delta) - w(o, u)$ No changes

CASE E-LET-C

$$\frac{e_1|\mu \longmapsto e_1'|\mu'}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\mu \longmapsto \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2|\mu'}$$

Thus, e is $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ and e' is $\mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2$.

$\cdot\|\Sigma\|\Delta\|u \vdash e : E \setminus \mathcal{E}\|u''$	Assumption
$\Sigma\|\Delta, \Delta^*\|u \vdash \mu$	Assumption
$\Delta = (\Delta_1, \Delta_2)$ and $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$ and no temporary states or fields from $\mathcal{E}_1$ in Δ_2	Inversion
$\cdot\|\Sigma\|\Delta_1\|u \vdash e_1 : \exists x : C.P \setminus \mathcal{E}_1\|u_2$ and $x : C\|\Sigma\|\Delta_2, P\|u_2 \vdash e_2 : E \setminus \mathcal{E}_2\|u''$	Inversion (cont.)
$\Sigma\|\Delta_1\|u \vdash \mu$ and $\Sigma\|\Delta_2\|u \vdash \mu$	Compositionality
Apply induction hypothesis, using (Δ_2, Δ^*) as additional linear context.	
Ex. $\Sigma' \supseteq \Sigma$ and u' and $\mathcal{E}_1'$ and Δ_1', where fields $\mathcal{E}_1 - \mathcal{E}_1'$ do not occur in Δ_1'	From i.h.
Either (a) $u = -$ and $u' = k \cdot o@s$ and $\mathcal{E}_1' - \mathcal{E}_1$ only contains fields of o or (b) $\mathcal{E}_1' \subseteq \mathcal{E}_1$	i.h. (cont.)
$\cdot\|\Sigma'\|\Delta_1'\|u' \vdash e_1' : \exists x : C.P \setminus \mathcal{E}_1'\|u_2$ and $\Sigma'\|\Delta_1', \Delta_2, \Delta^*\|u' \vdash \mu'$	i.h. (cont.)
$\forall o \in \mathsf{dom}(\mu') : w(o, \mu) - w(o, \Delta_1) - w(o, u) \leq w(o, \mu') - w(o, \Delta_1') - w(o, u')$	i.h. (cont.)
Define $\Sigma' = \Sigma$ and $\mathcal{E}' = \mathcal{E}_1' \cup \mathcal{E}_2$ and $\Delta' = (\Delta_1', \Delta_2)$	
Fields $\mathcal{E} - \mathcal{E}'$ do not occur in Δ'	$\mathcal{E} - \mathcal{E}' \subseteq \mathcal{E}_1$ and fields $\mathcal{E}_1$ do not occur in Δ_2
$\Sigma'\|\Delta', \Delta^*\|u' \vdash \mu'$	From i.h.
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) \leq w(o, \mu') - w(o, \Delta') - w(o, u')$	From i.h.: fractions in Δ_2 unchanged

SUBCASE: $u = -$ and $u' = k \cdot o@s$ and $\mathcal{E}_1' - \mathcal{E}_1$ only contains fields of o

Δ_2, Δ^* do not contain permissions for fields of o	Definition of $\Sigma\|\Delta, \Delta^*\|u \vdash \mu$
Δ_2, Δ^* do not contain permissions for fields in $\mathcal{E}_1'$	$\mathcal{E}_1' - \mathcal{E}_1$ contains only fields of o
$\cdot\|\Sigma\|\Delta'\|u' \vdash e' : E \setminus \mathcal{E}'\|u''$	By rule T-LET

SUBCASE: $\mathcal{E}_1' \subseteq \mathcal{E}_1$

Δ_2, Δ^* do not contain permissions for fields in $\mathcal{E}_1'$	$\mathcal{E}_1' \subseteq \mathcal{E}_1$
$\cdot\|\Sigma\|\Delta'\|u' \vdash e' : E \setminus \mathcal{E}'\|u''$	By rule T-LET

CASE E-LET-V

$$\frac{k' \cdot o \mapsto C(\ldots)@S \in \mu \quad k \leq k'}{\texttt{let}\ x = k \cdot o\ \texttt{in}\ e_2|\mu \longmapsto [o/x]e_2|\mu}$$

Thus, e is $\texttt{let}\ x = k \cdot o\ \texttt{in}\ e_2$, e' is $[o/x]e_2$, and $\mu' = \mu$.

$\cdot\|\Sigma\|\Delta\|u \vdash e : E \setminus \mathcal{E}\|u''$	Assumption
$\Sigma\|\Delta, \Delta^*\|u \vdash \mu$	Assumption
$\Delta = (\Delta_1, \Delta_2)$ and $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$	Inversion on e
$\cdot\|\Sigma\|\Delta_1\|u \vdash k \cdot o : \exists x : C.P \setminus \mathcal{E}_1\|u_2$ and $x : C\|\Sigma\|\Delta_2, P\|u_2 \vdash e_2 : E \setminus \mathcal{E}_2\|u''$	Inversion (cont.)
$\cdot\|\Sigma\|\Delta_1 \vdash [o/x]P$ and $u = u_1$ and $\mathcal{E}_1 = \emptyset$ thus $\mathcal{E}_2 = \mathcal{E}$	Inversion on $k \cdot o$
Define $\Sigma' = \Sigma$ and $u' = u$ and $\mathcal{E}' = \mathcal{E}$ and $\Delta' = \Delta$	
$\cdot\|\Sigma\|\Delta\|u \vdash e' : E \setminus \mathcal{E}\|u''$	Substitution
$\Sigma\|\Delta, \Delta^*\|u \vdash \mu$	Given
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu) - w(o, \Delta) - w(o, u)$	No changes

CASE E-UNPACK-MODIFYING

$$\frac{k' \cdot o \mapsto C(\ldots)@s' \in \mu \quad 0 < k \leq k' \quad s' \leq s}{\texttt{unpack}\ k \cdot o@s\ \texttt{in}\ e'|\mu \longmapsto e'|\mu[(k' - k) \cdot o \mapsto C(\ldots)@\mathsf{Unpacked}(k)]}$$

Thus, e is $\texttt{unpack}\ k \cdot o@s\ \texttt{in}\ e'$ and $\mu' = \mu[(k' - k) \cdot o \mapsto C(\ldots)@\textsf{Unpacked}(k)]$.

$\cdot\vert\Sigma\vert\Delta\vert u \vdash e : E \setminus \mathcal{E}\vert u''$	Assumption
$\Sigma\vert\Delta, \Delta^*\vert u \vdash \mu$	Assumption
$\Delta = (k \cdot o@s, \Delta'')$ and $u = u'' = -$ and $\mathcal{E} = \emptyset$	Inversion
$\cdot\vert\Sigma\vert\Delta'', [o/\textit{this}]\textsf{inv}_C(s, k)\vert k \cdot o@s \vdash e' : E \setminus \mathcal{E}'\vert-$	Inversion (cont.)
Define $\Sigma' = \Sigma$ and $u' = k \cdot o@s$ and $\Delta' = (\Delta'', [o/\textit{this}]\textsf{inv}_C(s, k))$	
o was unpacked	$u = -$ and $u' = k \cdot o@s$
$\mathcal{E}' - \mathcal{E} = \mathcal{E}'$ only contains fields of o	Lemma 3
Δ' does not contain fields in $\mathcal{E} - \mathcal{E}'$	$\mathcal{E} - \mathcal{E}' = \emptyset$
$\cdot\vert\Sigma\vert\Delta'\vert u' \vdash e' : E \setminus \mathcal{E}'\vert-$	From above
$\Sigma\vert\Delta', \Delta^*\vert u' \vdash \mu'$	o was unpacked, Δ' differs from Δ in $[o/\textit{this}]\textsf{inv}_C(s, k)$
$\forall o \in \textsf{dom}(\mu) : w(o, \mu) - w(o, \Delta)w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u')$	$k \cdot o$ moves from μ to u'

Case E-Unpack-Readonly

$$\frac{k' \cdot o \mapsto C(\ldots)@s' \in \mu \quad s' \leq s}{\texttt{unpack}\ 0 \cdot o@s\ \texttt{in}\ e'\vert\mu \longmapsto e'\vert\mu[k' \cdot o \mapsto C(\ldots)@\textsf{Unpacked}(s')]}$$

Thus, e is $\texttt{unpack}\ 0 \cdot o@s\ \texttt{in}\ e'$ and $\mu' = \mu[k' \cdot o \mapsto C(\ldots)@\textsf{Unpacked}(s')]$.

$\cdot\vert\Sigma\vert\Delta\vert u \vdash e : E \setminus \mathcal{E}\vert u''$	Assumption
$\Sigma\vert\Delta, \Delta^*\vert u \vdash \mu$	Assumption
$\Delta = (0 \cdot o@s, \Delta'')$ and $u = u'' = -$ and $\mathcal{E} = \emptyset$	Inversion
$\cdot\vert\Sigma\vert\Delta'', [o/\textit{this}]\textsf{inv}_C(s, 0)\vert 0 \cdot o@s \vdash e' : E \setminus \mathcal{E}'\vert-$	Inversion (cont.)
Define $\Sigma' = \Sigma$ and $u' = 0 \cdot o@s$ and $\Delta' = (\Delta'', [o/\textit{this}]\textsf{inv}_C(s, 0))$	
o was unpacked	$u = -$ and $u' = 0 \cdot o@s$
$\mathcal{E}' - \mathcal{E} = \mathcal{E}'$ only contains fields of o	Lemma 3
Δ' does not contain fields in $\mathcal{E} - \mathcal{E}'$	$\mathcal{E} - \mathcal{E}' = \emptyset$
$\cdot\vert\Sigma\vert\Delta'\vert u' \vdash e' : E \setminus \mathcal{E}'\vert-$	From above
$\Sigma\vert\Delta', \Delta^*\vert u' \vdash \mu'$	o was unpacked, Δ' differs from Δ in $[o/\textit{this}]\textsf{inv}_C(s, 0)$
$\forall o \in \textsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u')$	No changes: 0 fraction unpacked

Case E-Read

$$\frac{k_o \cdot o \mapsto C(\ldots, f = k' \cdot o'@s', \ldots)@\textsf{Unpacked}(k'') \in \mu \quad k \leq k'}{k \cdot o.f\vert\mu \longmapsto k \cdot o'\vert(\mu + k \cdot o')[k_o \cdot o \mapsto C(\ldots, f = (k' - k) \cdot o'@s', \ldots)@\textsf{Unpacked}(k'')]}$$

Thus, e is $k \cdot o.f$ and e' is $k \cdot o'$ and $\mu' = (\mu + k \cdot o')[k_o \cdot o \mapsto C(\ldots, f = (k' - k) \cdot o', \ldots)@\textsf{Unpacked}(k'')]$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ Assumption
$u = u'' = k \cdot o@s$, where $k > 0$, and $\mathcal{E} = \emptyset$ and $E = \exists x : C_f.P$ and $\cdot|\Sigma|\Delta \vdash [o.f/x]P$ Inversion
Define $\Sigma' = \Sigma$ and $u' = u$ and $\mathcal{E}' = \emptyset = \mathcal{E}$ and $\Delta' = [o'/x]P$
$\cdot|\Sigma|\Delta'|u \vdash k \cdot o' : E \setminus \emptyset|u$ By rule T-LOC
$\Sigma|\Delta', \Delta^*|u \vdash \mu'$ $\Delta' = [o'/o.f]\Delta$
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u)$
$k \cdot o'$ moves from field to stack

CASE E-READ-PURE

$$\frac{k_o \cdot o \mapsto C(\ldots, f = k' \cdot o'@s', \ldots)@\mathsf{Unpacked}(s'') \in \mu}{0 \cdot o.f|\mu \longmapsto 0 \cdot o'|\mu}$$

Thus, e is $0 \cdot o.f$ and e' is $0 \cdot o'$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ Assumption
$u = u'' = k' \cdot o@s$ and $\mathcal{E} = \emptyset$ and $E = \exists x : C_f.P$ and $\cdot|\Sigma|\Delta \vdash [o.f/x]P$ Inversion
$k' = 0$ Store typing
Define $\Sigma' = \Sigma$ and $u' = u$ and $\mathcal{E}' = \emptyset = \mathcal{E}$ and $\Delta' = [o'/x]P$
$\cdot|\Sigma|\Delta'|u \vdash 0 \cdot o' : E \setminus \emptyset|u$ By rule T-LOC
$\Sigma|\Delta', \Delta^*|u \vdash \mu$ $\Delta' = [o'/o.f]\Delta$, μ is unchanged
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu) - w(o, \Delta') - w(o, u)$ No changes

CASE E-ASSIGN

$$\frac{\begin{array}{c} k_1 \cdot o_1 \mapsto C(\ldots, f = k' \cdot o', \ldots)@\mathsf{Unpacked}(k'') \in \mu \quad k_2 \cdot o_2 \mapsto C(\ldots)@S_2 \in \mu \quad k \leq k_2 \quad k'' > 0 \\ \mu' = ((\mu - k \cdot o_2) + k' \cdot o')[k_o \cdot o \mapsto C(\ldots, f = k \cdot o_2, \ldots)@\mathsf{Unpacked}(k'')] \end{array}}{o_1.f := k \cdot o_2|\mu \longmapsto k' \cdot o'|\mu'}$$

Thus, e is $o_1.f := k \cdot o_2$ and e' is $k' \cdot o'$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$ Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ and no permissions for fields in $\mathcal{E}$ in Δ^* Assumption
$\mathcal{E} = \{o_1.f\}$ and $u = u'' = k_u \cdot o_1@s_u$, where $k_u > 0$, and $\Delta = (\Delta_1, \Delta_2)$ Inversion
$E = \exists x'.P' \otimes [o_1.f/x]P$ Inversion (cont.)
$\cdot|\Sigma|\Delta_1 \vdash [o_1.f/x']P'$ and $\cdot|\Sigma|\Delta_2|u \vdash k \cdot o_2 : \exists x : C_f.P|u$ Inversion (cont.)
Define $\Sigma' = \Sigma$ and $u' = u$ and $\mathcal{E}' = \emptyset \subset \mathcal{E}$ and $\Delta' = [o'/x']P' \otimes [o_1.f/x]P$
Δ' does not contain permissions for $\mathcal{E} - \mathcal{E}' = \{o_1.f\}$ By definition of Δ' and assumption above
$\cdot|\Sigma|\Delta'|u \vdash e' : E \setminus \emptyset|u$ By rule T-LOC
$\Sigma|\Delta', \Delta^*|u \vdash \mu'$ $\Delta' = [o_1.f/o_2]([o'/o_1.f]\Delta)$ and no permissions for $o_1.f$ in Δ^*
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u)$
$k \cdot o_2$ and $k' \cdot o'$ move between field and stack

CASE E-PACK-MODIFYING

$$\frac{k_o \cdot o \mapsto C(\overline{f = k \cdot o})@\mathsf{Unpacked}(k) \in \mu \quad \mathsf{inv}_C(s) \textit{ is satisfied by o's fields}}{\texttt{pack}\ o\ \texttt{to}\ s\ \texttt{in}\ e'|\mu \longmapsto e'|\mu[(k_o + k) \cdot o \mapsto C(\overline{f = k \cdot o})@s]}$$

Thus, e is $\texttt{pack}\ o\ \texttt{to}\ s\ \texttt{in}\ e'$ and $\mu' = \mu[(k_o + k) \cdot o \mapsto C(\overline{f = k \cdot o})@s]$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$	Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ and no temporary states or permissions for $\mathcal{E}$ in Δ^*	Assumption
$u = k \cdot o@s'$ and $u'' = -$ and $\mathcal{E} = \{o.\overline{f}\}$ and $\Delta = (\Delta'', \Delta''')$	Inversion
No temporary states or permissions for $o.\overline{f}$ in Δ'''	Inversion (cont.)
$\cdot|\Sigma|\Delta'' \vdash [o/\textit{this}]\mathsf{inv}_C(s, k)$ and $\cdot|\Sigma|\Delta''', k \cdot o@s|- \vdash e' : E \setminus \emptyset|-$	Inversion (cont.)
Define $\Sigma' = \Sigma$ and $u' = -$ and $\mathcal{E}' = \emptyset \subset \mathcal{E}$ and $\Delta' = (\Delta''', k \cdot o@s)$	
No permissions for fields in $\mathcal{E} - \mathcal{E}' = \{o.\overline{f}\}$ in Δ'	No permissions for $\{o.\overline{f}\}$ in Δ''' from above
$\cdot|\Sigma|\Delta' \vdash e' : E \setminus \emptyset$	From above
$\Sigma|\Delta'''|u \vdash \mu$	Compositionality
$\Sigma|\Delta'''|- \vdash \mu'$	Δ''' not affected by packing since no temporary states
$\Sigma|\Delta', \Delta^*|- \vdash \mu'$	$k \cdot o@s$ comes from u, no temporary states or permissions for $\mathcal{E}$ in Δ^*
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u')$	$k \cdot o$ moves from u to μ'

CASE E-PACK-READONLY

$$\frac{k_o \cdot o \mapsto C(\overline{f = k \cdot o})@\mathsf{Unpacked}(s) \in \mu \quad \mathsf{inv}_C(s) \textit{ is satisfied by o's fields}}{\texttt{pack}\ o\ \texttt{to}\ s\ \texttt{in}\ e'|\mu \longmapsto e'|\mu[k_o \cdot o \mapsto C(\overline{f = k \cdot o})@s]}$$

Thus, e is $\texttt{pack}\ o\ \texttt{to}\ s\ \texttt{in}\ e'$ and $\mu' = \mu[k_o \cdot o \mapsto C(\overline{f = k \cdot o})@s]$.

$\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u''$	Assumption
$\Sigma|\Delta, \Delta^*|u \vdash \mu$ and no temporary states or permissions for $\mathcal{E}$ in Δ^*	Assumption
$u = k \cdot o@s'$ and $u'' = -$ and $\mathcal{E} = \{o.\overline{f}\}$ and $\Delta = (\Delta'', \Delta''')$	
No temporary states or permissions for $o.\overline{f}$ in Δ'''	Inversion
$\cdot|\Sigma|\Delta'' \vdash [o/\textit{this}]\mathsf{inv}_C(s, 0)$ and $\cdot|\Sigma|\Delta''', k \cdot o@s|- \vdash e' : E \setminus \emptyset|-$	Inversion (cont.)
$k = 0$ and $s' = s$	Store typing, inversion
Define $\Sigma' = \Sigma$ and $u' = -$ and $\mathcal{E}' = \emptyset \subset \mathcal{E}$ and $\Delta' = (\Delta''', 0 \cdot o@s)$	
No permissions for fields in $\mathcal{E} - \mathcal{E}' = \{o.\overline{f}\}$ in Δ'	No permissions for $\{o.\overline{f}\}$ in Δ''' from above
$\cdot|\Sigma|\Delta'|- \vdash e' : E \setminus \emptyset|-$	From above
$\Sigma|\Delta'''|u \vdash \mu$	Compositionality
$\Sigma|\Delta'''|- \vdash \mu'$	Only packing changes
$\Sigma|\Delta', \Delta^*|- \vdash \mu'$	$0 \cdot o@s$ replaces u, no temporary states or permissions for $\mathcal{E}$ in Δ^*
$\forall o \in \mathsf{dom}(\mu) : w(o, \mu) - w(o, \Delta) - w(o, u) = w(o, \mu') - w(o, \Delta') - w(o, u')$	No changes: 0 fraction packed

7.4 Progress

Lemma 5 (Canonical Form) *If e is a value then e has the form $k \cdot o$.*

Let-normal form for expressions simplifies proving progress: In closed terms, arguments to atomic expressions (new, method call, field read, etc.) are automatically values. We reflect this directly in the cases we prove.

Theorem 2 (Progress) *If $\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u'$—i.e., e is closed and well-typed—then either e is a value $k \cdot o$ or else, for any heap μ st. $\Sigma|\Delta|u \vdash \mu$, there exists an expression e' and a heap μ' with $e|\mu \longmapsto e'|\mu'$.*

Proof: By structural induction on the derivation of $\cdot|\Sigma|\Delta|u \vdash e : E \setminus \mathcal{E}|u'$.
CASE T-VAR N/A: Variable is not a closed term.
CASE T-LOC

$$\frac{(o : C) \in \Sigma \quad \Sigma|\Delta \vdash [o/x]P}{\cdot|\Sigma|\Delta|u \vdash k \cdot o : \exists x : C.P \setminus \emptyset|u}$$

$k \cdot o$ is a value.
CASE T-NEW

$$\frac{\overline{o : C} \subseteq \Sigma \quad \mathsf{init}(C) = \langle \exists \overline{f : C}.P, s\rangle \quad \cdot|\Sigma|\Delta \vdash [\overline{o}/\overline{f}]P}{\cdot|\Sigma|\Delta|u \vdash \mathsf{new}\, C(\overline{k \cdot o}) : \exists x : C.1 \cdot x@s \setminus \emptyset|u}$$

$\Sigma|\Delta \vdash \mu$ Assumption

$\overline{k' \cdot o \mapsto C(\ldots)@s} \subseteq \mu$ such that $\overline{k \leq k'}$ and $\mu|\overline{k \cdot o} \vdash [\overline{o}/\overline{f}]P$ Heap well-typed

Define $\mu' = (\mu - \overline{k \cdot o}), 1 \cdot o^* \mapsto C(\overline{f = k \cdot o@s})@s$ (where $o^* \notin \mathsf{dom}(\mu)$) and $e' = 1 \cdot o^*$

$e|\mu \longmapsto e'|\mu'$ By rule E-NEW

CASE T-CALL

$$\frac{\begin{array}{c}(o : C) \in \Sigma \quad \overline{o : C} \subseteq \Sigma \quad \cdot|\Gamma|\Delta \vdash [o/\mathit{this}][\overline{o}/\overline{f}]P \\ \mathsf{mtype}(C, m) = \forall \overline{x : C}.P \multimap E\end{array}}{\cdot|\Sigma|\Delta|- \vdash k \cdot o.m(\overline{k \cdot o}) : E \setminus \emptyset|-}$$

$\Sigma|\Delta \vdash \mu$ Assumption

$k' \cdot o \mapsto C(\ldots)@s \in \mu$ and $\overline{k' \cdot o \mapsto C(\ldots)@s} \subseteq \mu$ Heap well-typed

$k \leq k'$ and $\overline{k \leq k'}$ and $\mu|k \cdot o, \overline{k \cdot o} \vdash [o/\mathit{this}][\overline{o}/\overline{f}]P$ Heap well-typed (cont.)

Define $\mu' = \mu$ and e' as $[o/\mathit{this}][\overline{o/x}]e_m$ where $\mathsf{mbody}(C, m) = \overline{x}.e_m$

$e|\mu \longmapsto e'|\mu'$ By rule E-CALL

CASE T-LET

$$\frac{\begin{array}{c}\cdot|\Sigma|\Delta_1 u \vdash e_1 : \exists x : C.P \setminus \mathcal{E}_1|u_2 \quad x : C|\Sigma|\Delta_2, P|u_2 \vdash e_2 : E \setminus \mathcal{E}_2|u' \\ \textit{No temporary states or permissions for } \mathcal{E}_1 \textit{ in } \Delta_2\end{array}}{\cdot|\Sigma|\Delta_1, \Delta_2|u \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : E \setminus \mathcal{E}_1 \cup \mathcal{E}_2|u'}$$

$\Sigma|\Delta \vdash \mu$ Assumption
$\Sigma|\Delta_1 \vdash \mu$ Compositionality
SUBCASE: e_1 is a value
  $e_1 = k \cdot o$ Canonical form
  $e|\mu \longmapsto [o/x]e_2|\mu$ By rule E-LET-V

SUBCASE: e_1 makes a step
  Ex μ' st. $e_1|\mu \longmapsto e_1'|\mu'$ From i.h.
  $e|\mu \longmapsto \texttt{let}\ x = e_1'\ \texttt{in}\ e_2|\mu'$ By rule E-LET-C

CASE T-UNPACK

$$\frac{(o : C) \in \Sigma \quad \cdot|\Sigma|\Delta \vdash k \cdot o@s \quad \cdot|\Sigma|D', [o/\textit{this}]\mathsf{inv}_C(s,k)|k \cdot o@s \vdash e' : E \setminus \mathcal{E}|-}{\cdot|\Sigma|\Delta, \Delta'|- \vdash \texttt{unpack}\ k \cdot o@s\ \texttt{in}\ e' : E \setminus \emptyset|-}$$

$\Sigma|\Delta \vdash \mu$ Assumption
$k' \cdot o \mapsto C(\ldots)@s' \in \mu$ st. $k \leq k'$ and $s' \leq s$ Heap well-typed
SUBCASE $k > 0$
  Define $\mu' = \mu[(k' - k) \cdot o \mapsto C(\ldots)@\mathsf{Unpacked}(k)]$
  $e|\mu \longmapsto e'|\mu'$ By rule E-UNPACK-MODIFYING

SUBCASE $k = 0$
  Define $\mu' = \mu[(k' - k) \cdot o \mapsto C(\ldots)@\mathsf{Unpacked}(s')]$
  $e|\mu \longmapsto e'|\mu'$ By rule E-UNPACK-READONLY

CASE T-READ

$$\frac{\cdot|\Sigma|\Delta \vdash [o.f_i/x]P \quad \mathsf{localFields}(C) = \overline{f : T} \quad k_u = 0\ \textit{implies}\ k = 0}{\cdot|\Sigma|\Delta|k_u \cdot o@s_u \vdash k \cdot o.f_i : \exists x : T_i.P \setminus \emptyset|k_u \cdot o@s_u}$$

$\Sigma|\Delta \vdash \mu$ Assumption
$k_o \cdot o \mapsto C(\ldots, f_i = k_i \cdot o_i, \ldots)@S \in \mu$ st. $k \leq k_i$ Heap well-typed
Define $\mu' = (\mu + k \cdot o_i)[k_o \cdot o \mapsto C(\ldots, f_i = (k_i - k) \cdot o_i, \ldots)@S]$ and e' as $k \cdot o_i$
SUBCASE $k_u > 0$
  $S = \mathsf{Unpacked}(k_u)$ Heap well-typed
  $e|\mu \longmapsto e'|\mu'$ By rule E-READ

SUBCASE $k_u = 0$
  $k = 0$ Implied by typing rule
  $S = \mathsf{Unpacked}(s)$ Heap well-typed
  $\mu' = \mu$ Moved fraction is 0
  $e|\mu \longmapsto e'|\mu$ By rule E-READ-PURE

CASE T-ASSIGN

$$\frac{\begin{array}{c}\cdot|\Sigma|\Delta \vdash k \cdot o : \exists x : C_i.P \quad \cdot|\Sigma|\Delta' \vdash [o'.f_i/x']P' \\ \mathsf{localFields}(C') = \overline{f : C} \quad (o' : C') \in \Sigma \quad k' > 0\end{array}}{\cdot|\Sigma|\Delta, \Delta'|k' \cdot o'@s' \vdash o'.f_i := k \cdot o : \exists x' : C_i.P' \otimes [o'.f_i/x]P \setminus \{o_1.f\}|k' \cdot o'@s'}$$

$\Sigma|\Delta \vdash \mu$ Assumption
$k'_o \cdot o' \mapsto C(\ldots, f_i = k_i \cdot o_i, \ldots)@\mathsf{Unpacked}(k') \in \mu$ Heap well-typed
$k_o \cdot o_i \mapsto C_i(\ldots)@S$ st. $k \leq k_o$ Heap well-typed
Define $\mu' = ((\mu - k \cdot o) + k_i \cdot o_i)[k'_o \cdot o' \mapsto C(\ldots, f_i = k \cdot o, \ldots)@\mathsf{Unpacked}(k')]$ and e' as $k_i \cdot o_i$
$e|\mu \longmapsto e'|\mu'$ By rule E-ASSIGN

CAST T-PACK

$$\frac{\begin{array}{c}\cdot|\Sigma|\Delta \vdash [o/\textit{this}]\mathsf{inv}_C(s,k) \quad \cdot|\Sigma|\Delta', k \cdot o@s|- \vdash e' : E \setminus \emptyset|- \quad k = 0 \textit{ implies } s' = s \\ \mathsf{localFields}(C) = \overline{f : C} \quad (o : C) \in \Sigma \quad \textit{No temporary states or permissions for } o.\overline{f} \textit{ in } \Delta'\end{array}}{\cdot|\Sigma|\Delta, \Delta'|k \cdot o@s' \vdash \texttt{pack } o \texttt{ to } s \texttt{ in } e' : E \setminus \{o.\overline{f}\}|-}$$

$\Sigma|\Delta \vdash \mu$ Assumption
SUBCASE $k > 0$
  $k_o \cdot o \mapsto C(\ldots)@\mathsf{Unpacked}(k) \in \mu$ Heap well-typed
  o's fields satisfy $[o/\textit{this}]\mathsf{inv}_C(s,k)$ $\cdot|\Sigma|\Delta \vdash [o/\textit{this}]\mathsf{inv}_C(s,k)$ and heap well-typed
  Define $\mu' = \mu[k \cdot o \mapsto C(\ldots)@s]$
  $e|\mu \longmapsto e'|\mu'$ By rule E-PACK-MODIFYING

SUBCASE $k = 0$
  $k_o \cdot o \mapsto C(\ldots)@\mathsf{Unpacked}(s'') \in \mu$ Heap well-typed
  $s'' \leq s$ and o's fields satisfy $[o/\textit{this}]\mathsf{inv}_C(s'', 1)$ Heap well-typed
  Define $\mu' = \mu[k \cdot o \mapsto C(\ldots)@s'']$
  $e|\mu \longmapsto e'|\mu'$ By rule E-PACK-READONLY

8 Related Work

In previous work we proposed more expressive typestate specifications [4] that can be verified with the approach presented in this paper. We also recently proposed full and pure permissions and applied our approach to specifying full Java iterators [3]. Verification of protocol compliance has been studied from many different angles including type systems, abstract interpretation, model checking, and verification of general program behavior. Aliasing is a challenge for all these approaches.

The system that is closest to our work is Fugue [11], the first modular typestate verification system for object-oriented software. Methods are specified with a deterministic state transition of the receiver and pre-conditions on arguments. Fugue's type system tracks objects as "not aliased" or "maybe aliased". Leveraging research on "alias types" [29] (see below), objects typically remain "not aliased" as long as they are only referenced on the stack. Only "not aliased" objects can change state; once an object becomes "maybe aliased" its state is permanently fixed although fields can be assigned to if the object's abstract typestate is preserved. There exists no soundness proof for Fugue.

Our work is greatly inspired by Fugue's abilities. Our approach supports more expressive method specifications based on linear logic [17]. Our verification approach is based on "access permissions" that permit state changes even in the presence of aliases. We extend several ideas from Fugue to work with access permissions including state invariants, packing, and frames. Fugue's specifications are expressible with our system [4]. Fugue's "not aliased" objects can be simulated with unique permissions for alive and "maybe aliased" objects correspond to shared permissions with state guarantees. There is no equivalent for state dimensions, temporary state assumptions, full, immutable, and pure permissions, or permissions for object parts in Fugue. We prove a core fragment of our system sound.

Verification of protocol compliance has also been described as "resource usage analysis" [21]. Protocol specifications are based on very different concepts including typestates [30, 10, 23], type qualifiers [15], size properties [8], direct constraints on ordering [21, 31], and effective refinements [27]. None of the above systems can verify implementations of object-oriented protocols like our approach and only one [31] targets an object-oriented language. Temporary state information, full, and pure permissions are not supported. Effective type refinements [27] employ linear logic reasoning but cannot reason about protocol implementations and do not support aliasing abstractions. Hob [23] verifies data structure implementations for a procedural language with static module instantiation based on typestate-like constraints using shape analyses. In Hob, data can have states, but modules themselves cannot. In contrast, we can verify the implementation of stateful objects that are dynamically allocated and support aliasing with permissions instead of shape analysis.

Because programming with linear types [32] is very inconvenient, a variety of relaxing mechanisms were proposed. Uniqueness, sharing, and immutability (sometimes called read-only) [6] have recently been put to use in resource usage analysis [21, 8]. Alias types [29] allow multiple variables to refer to the same object but require a linear token for object accesses that can be borrowed [6] during function calls. Focusing can be used for temporary state changes of shared objects [12, 15, 2]. Adoption prevents sharing from leaking through entire object graphs (as in Fugue [11]) and allows temporary sharing until a linear adopter is deallocated [12]. All these techniques need to be aware of all references to an object in order to change its state.

Access permissions allow state changes even if objects are aliased from unknown places. Moreover, access permissions give fine-grained access to individual data groups [24]. States and fractions [5] let us capture alias types, borrowing, adoption, and focus with a single mechanism. Sharing of individual data groups has been proposed before [6], but it has not been exploited for reasoning about object behavior. In Boyland's work [5], a fractional permission means immutability (instead of sharing) in order to ensure non-interference of permissions. We use permissions to keep state assumptions consistent but track, split, and join permissions in the same way as Boyland.

Global approaches are very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [1, 18, 13]) typically verify client conformance while the protocol implementation is assumed correct. Sound approaches rely on a global aliasing analysis [1, 13]. Likewise, most model checkers operate globally (e.g. [20]) or use assume-guarantee reasoning between coarse-grained static components [16]. The Magic tool checks individual C functions but has to inline user-provided state machine abstractions for library code in order to accommodate aliasing [7]. The above analyses typically run on the complete code base once a system is fully implemented and are very expensive. Our approach supports developers by checking the code at hand like a typechecker. Thus the benefits of our approach differ significantly from global analyses. It is interesting to note that protocols found by typestate inference in the presence of aliasing [28] are very similar to what we can enforce. These research directions could be fruitfully combined.

Finally, general approaches to program verification such as ESC/Java [14] and Boogie [2] can be used to specify and verify protocols. Our approach is strictly less expressive but supports protocols more directly, includes special-purpose aliasing abstractions, and therefore promises better automation.

9 Conclusions

This paper proposes a modular type system for verifying usage and implementation of typestate protocols that supports several forms of aliasing. It allows different references to control separate parts of an object's state, leveraging hierarchical state spaces based on state refinement. Multiple references can have access to the same part of the state either by uniformly sharing access or by giving one reference full access while the other references can only read but not change the state. We support expressive typestate protocols as previously proposed [4] and specify protocols from the Java standard library that were previously hard to capture [3].

We develop these ideas in a type system that tracks "access permissions" to objects with linear logic [17]. Permissions can be flexibly split and joined using fractions [5]. We extend ideas from Fugue [11] to connect protocol specifications to implementations. Other novel features include a principled approach to callbacks and dynamic tests and the interpretation of typestates as data groups.

In future work we hope to develop a practical system that avoids user annotations in method bodies. A challenge in this effort will be efficient reasoning about linear logic propositions. Like any sound static reasoning system, our approach will reject protocol-compliant programs due to reasoning imprecisions. Sharing and dynamic tests can be used to recover from imprecisions, but an interesting empirical question will be how often programmers will have to resort to these mechanisms.

Acknowledgments

We thank John Boyland, Frank Pfenning, Nels Beckman, Sebastian Bossung, and Jason Reed for fruitful discussions on this topic. The presentation of linear logic follows lecture notes by Frank Pfenning.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the Eighth SPIN Workshop*, pages 101–122, May 2001.

[2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[3] K. Bierhoff. Iterator specification with typestates. In *5th Int. Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.

[4] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conf. and ACM Symp. on the Foundations of Software Engineering*, pages 217–226, Sept. 2005.

[5] J. Boyland. Checking interference with fractional permissions. In *International Symp. on Static Analysis*, pages 55–72. Springer, 2003.

[6] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symp. on Principles of Programming Languages*, pages 283–295, Jan. 2005.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Int. Conference on Software Engineering*, pages 385–395, May 2003.

[8] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *Int. Conference on Software Engineering*, pages 186–195, May 2005.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conf. on Programming Language Design and Implementation*, pages 59–69, 2001.

[11] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[12] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conf. on Programming Language Design and Implementation*, pages 13–24, June 2002.

[13] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ACM Int. Symp. on Software Testing and Analysis*, pages 133–144, July 2006.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conf. on Programming Language Design and Implementation*, pages 234–245, May 2002.

[15] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conf. on Programming Language Design and Implementation*, pages 1–12, 2002.

[16] D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Int. Conference on Software Engineering*, pages 211–220, May 2004.

[17] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[18] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM Conf. on Programming Language Design and Implementation*, pages 69–82, 2002.

[19] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symp. on Principles of Programming Languages*, pages 58–70, 2002.

[21] A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM Symp. on Principles of Programming Languages*, pages 331–342, Jan. 2002.

[22] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999.

[23] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), Dec. 2006.

[24] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.

[25] P. Lincoln and A. Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.

[26] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[27] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM Int. Conf. on Functional Programming*, pages 213–225, 2003.

[28] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications*, pages 77–96, New York, NY, USA, 2005. ACM Press.

[29] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000.

[30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

[31] G. Tan, X. Ou, and D. Walker. Enforcing resource usage protocols via scoped methods. In *Int. Workshop on Foundations of Object-Oriented Languages*, 2003.

[32] P. Wadler. Linear types can change the world! In *Working Conf. on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

$$\frac{\texttt{class}\ C\ \{\ldots\ \overline{M}\ldots\} \in \overline{CL} \quad T_r\ m(\overline{T\ x}) : P \multimap \exists \textit{result} : T_r.P' = e \in \overline{M}}{\mathsf{mtype}(m, C) = \forall \overline{x : T}.P \multimap \exists \textit{result} : T_r.P'}$$

$$\frac{C \text{ extends } C' \quad \mathsf{mtype}(m, C') = \forall \overline{x : T}.MS' \textit{ implies } (\overline{x : T}, \texttt{this} : C); \cdot \vdash MS \multimap MS'}{\mathsf{override}(m, C, \forall \overline{x : T}.MS)}$$

$$\frac{\texttt{class}\ C\ \{\ldots\ n = P \ldots\} \in \overline{CL}}{\mathsf{pred}_C(n) = P} \qquad \frac{P = \bigotimes_{n' \leq n'' < n} \mathsf{pred}_C(n'')}{\mathsf{pred}_C(n', n) = P} \qquad \frac{\texttt{class}\ C \ldots\ \{\overline{F}\ \ldots\} \in \overline{CL}}{\mathsf{localFields}(C) = \overline{F}}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ldots\} \in \overline{CL}}{C \text{ extends } C'} \qquad \frac{}{\mathsf{init}(\texttt{Object}) = (\mathbf{1}, \mathsf{alive})}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{f : T\ \texttt{in}\ n}\ \overline{S}\ \texttt{initially}\ \langle \exists \overline{f' : T'}, \overline{f : T}.P' \otimes P, A\rangle\ \ldots\} \\ \mathsf{init}(C') = (\exists \overline{f' : T'}.P', A') \quad \cdot; (P, \mathsf{full}(\texttt{super}, \mathsf{alive}, [\mathsf{alive} \mapsto 1], A')) \vdash \mathsf{inv}_C(A) \otimes \top\end{array}}{\mathsf{init}(C) = \langle \exists \overline{f' : T'}, \overline{f : T}.P' \otimes P, A\rangle}$$

$$\frac{\mathsf{inv}_C(A) = P \Rightarrow n'}{\mathsf{inv}_C(n, A) = P \otimes \mathsf{pred}_C(n', n) \otimes \mathsf{pred}_C(n)} \qquad \frac{}{\mathsf{inv}_C(n) = \mathbf{1} \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \quad \mathsf{pred}_C(n_1, n) = P_i' \quad n_1 \otimes n_2 \ll n \quad (i \in 1, 2)}{\mathsf{inv}_C(A_1 \otimes A_2) = P_1 \otimes P_1' \otimes P_2 \otimes P_2' \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \quad \mathsf{pred}_C(n_i, n) = P_i' \quad n_1 \oplus n_2 \ll n \quad (i \in 1, 2)}{\mathsf{inv}_C(A_1 \oplus A_2) = (P_1 \otimes P_1') \oplus (P_2 \otimes \mathsf{pred}_C(n_2, n)) \Rightarrow n}$$

$$\frac{\textit{only } \mathsf{pure} \textit{ permissions in } P}{\mathsf{effectsAllowed}(P) = 0} \qquad \frac{\textit{exists } \mathsf{share} \textit{ or } \mathsf{full} \textit{ permission in } P}{\mathsf{effectsAllowed}(P) = 1}$$

Figure 10: Protocol verification helper judgments

$$\frac{\Gamma;\Delta \vdash_C \mathsf{access}(\textit{this}_{\mathsf{fr}}, n, g, k, A) \quad \textit{receiver packed} \qquad k = 0 \textit{ implies } i = 0 \quad \Gamma;(\Delta', \mathsf{inv}_C(n,g,k,A), \mathsf{unpacked}(n,g,k,A)) \vdash^i_C e : E \setminus \mathcal{E}}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{unpack}(n,k,A)\ \mathtt{in}\ e : E \setminus \mathcal{E}}\ \text{P-UNPACK}$$

$$\frac{\Gamma;\Delta \vdash_C \mathsf{inv}_C(n,g,k,A) \otimes \mathsf{unpacked}(n,g,k,A') \quad k = 0 \textit{ implies } A = A' \qquad \Gamma;(\Delta', \mathsf{access}(\textit{this}_{\mathsf{fr}}, n,g,k,A)) \vdash^i_C e : E \setminus \mathcal{E} \quad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \quad \overline{f} \notin \mathsf{dom}(\Delta')}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{pack}\ n\ \mathtt{to}\ A\ \mathtt{in}\ e : E \setminus \overline{f}}\ \text{P-PACK}$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad \Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \qquad \mathsf{mtype}(m, C_0) = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i t_0.m(\overline{t}) : [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}\ \text{P-CALL}$$

$$\frac{\Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \quad C \text{ extends } C' \qquad \mathsf{mtype}(m, C') = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i_C \mathtt{super}.m(\overline{t}) : [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}\ \text{P-SUPER}$$

$$\frac{\Gamma;\Delta \vdash t : \exists x : T_i.P \otimes p \quad \Gamma;\Delta' \vdash_C [f_i/x']P' \quad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad n_i \leq n \quad p = \mathsf{unpacked}(n,g,k,A), k \neq 0}{\Gamma;(\Delta,\Delta') \vdash^1_C \mathtt{assign}\ f_i := t : \exists x' : T_i.P' \otimes [f_i/x]P \otimes p \setminus f_i}\ \text{P-ASSIGN}$$

Figure 11: Permission checking for expressions (part 2)

$$\begin{array}{rcl}
\mathsf{inv}_C(n,g,k,A) & = & \mathsf{inv}_C(n,A) \otimes \mathsf{purify}(\mathsf{above}_C(n)) \\
\mathsf{inv}_C(n,g,0,A) & = & \mathsf{purify}\,(\mathsf{inv}_C(n,A) \otimes \mathsf{above}_C(n)) \\
\textit{where}\ \mathsf{above}_C(n) & = & \bigotimes_{n' : n < n' \leq \mathsf{alive}} \mathsf{pred}_C(n')
\end{array}$$

Figure 12: Invariant construction

$$\frac{p = \mathsf{access}(r,n,g,k,A)}{\mathsf{purify}(p) = \mathsf{pure}(r,n,g,A)} \qquad \frac{\mathsf{purify}(P_1) = P_1' \quad \mathsf{purify}(P_2) = P_2' \quad \mathtt{op} \in \{\otimes, \&, \oplus\}}{\mathsf{purify}(P_1\ \mathtt{op}\ P_2) = P_1'\ \mathtt{op}\ P_2'}$$

$$\frac{\mathtt{unit} \in \{\mathbf{1}, \top, \mathbf{0}\}}{\mathsf{purify}(\mathtt{unit}) = \mathtt{unit}} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\exists z : H.P) = \exists z : H.P'} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\forall z : H.P) = \forall z : H.P'}$$

Figure 13: Permission purification

$$\frac{}{\Gamma; P \vdash P}\ \textsc{LinHyp} \qquad \frac{\Gamma; \Delta \vdash P' \quad P' \Rrightarrow P}{\Gamma; \Delta \vdash P}\ \textsc{Subst}$$

$$\frac{\Gamma; \Delta_1 \vdash P_1 \quad \Gamma; \Delta_2 \vdash P_2}{\Gamma; (\Delta_1, \Delta_2) \vdash P_1 \otimes P_2}\ \otimes I \qquad \frac{\Gamma; \Delta \vdash P_1 \otimes P_2 \quad \Gamma; (\Delta', P_1, P_2) \vdash P}{\Gamma; (\Delta, \Delta') \vdash P}\ \otimes E$$

$$\frac{}{\Gamma; \cdot \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma; \Delta \vdash \mathbf{1} \quad \Gamma; \Delta' \vdash P}{\Gamma; (\Delta, \Delta') \vdash P}\ \mathbf{1}E$$

$$\frac{\Gamma; \Delta \vdash P_1 \quad \Gamma; \Delta \vdash P_2}{\Gamma; \Delta \vdash P_1 \mathbin{\&} P_2}\ \& I \qquad \frac{\Gamma; \Delta \vdash P_1 \mathbin{\&} P_2}{\Gamma; \Delta \vdash P_1}\ \& E_L \qquad \frac{\Gamma; \Delta \vdash P_1 \mathbin{\&} P_2}{\Gamma; \Delta \vdash P_2}\ \& E_R$$

$$\frac{}{\Gamma; \Delta \vdash \top}\ \top I$$

no* $\top$ *elimination

$$\frac{\Gamma; \Delta \vdash P_1}{\Gamma; \Delta \vdash P_1 \oplus P_2}\ \oplus I_L \qquad \frac{\Gamma; \Delta \vdash P_2}{\Gamma; \Delta \vdash P_1 \oplus P_2}\ \oplus I_R$$

$$\frac{\Gamma; \Delta \vdash P_1 \oplus P_2 \quad \begin{array}{c}\Gamma; (\Delta', P_1) \vdash P \\ \Gamma; (\Delta', P_2) \vdash P\end{array}}{\Gamma; (\Delta, \Delta') \vdash P}\ \oplus E$$

no* $\mathbf{0}$ *introduction

$$\frac{\Gamma; \Delta \vdash \mathbf{0}}{\Gamma; (\Delta, \Delta') \vdash P}\ \mathbf{0}E$$

$$\frac{(\Gamma, z : H); \Delta \vdash P}{\Gamma; \Delta \vdash \forall z : H.P}\ \forall I \qquad \frac{\Gamma \vdash h : H \quad \Gamma; \Delta \vdash \forall z : H.P}{\Gamma; \Delta \vdash [h/z]P}\ \forall E$$

$$\frac{\Gamma \vdash h : H \quad \Gamma; \Delta \vdash [h/z]P}{\Gamma; \Delta \vdash \exists z : H.P}\ \exists I \qquad \frac{\Gamma; \Delta \vdash \exists z : H.P \quad (\Gamma, z : H), (\Delta', P) \vdash P'}{\Gamma; (\Delta, \Delta') \vdash P'}\ \exists E$$

Figure 14: Linear logic for permission reasoning

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Longleftrightarrow \mathsf{access}(r, n, g/2, k/2, A') \otimes \mathsf{access}(r, n, g/2, k/2, A'')}\ \textsc{Sym}$$

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Longleftrightarrow \mathsf{access}(r, n, g/2, k, A') \otimes \mathsf{pure}(r, n, g/2, A'')}\ \textsc{Asym}$$

$$\frac{\begin{array}{c} n_1 \# n_2 \quad A_1 \prec n_1 \leq n \quad A_2 \prec n_2 \leq n \\ p_i = \mathsf{full}(r, n_i, \{g, \mathsf{nodes}(n_i, n) \mapsto 1\}/2, A_i) \end{array}}{\mathsf{full}(r, n, g, A_1 \otimes A_2) \Rrightarrow p_1 \otimes p_2}\ \text{F-}\textsc{Split}\text{-}\otimes$$

$$\frac{\begin{array}{c} n_1 \# n_2 \quad A_1 \prec n_1 \leq n \quad A_2 \prec n_2 \leq n \\ p_i = \mathsf{full}(r, n_i, \{g, n \mapsto 1, \mathsf{nodes}(n_i, n) \mapsto 1\}/2, A_i) \end{array}}{p_1 \otimes p_2 \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A_1 \otimes A_2)}\ \text{F-}\textsc{Join}\text{-}\otimes$$

$$\frac{A_1 \# A_2}{\mathsf{full}(r, n, g, A_1 \oplus A_2) \Longleftrightarrow \mathsf{full}(r, n, g, A_1) \oplus \mathsf{full}(r, n, g, A_2)}\ \text{F-}\oplus$$

$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n, g, A) \Rrightarrow \mathsf{full}(r, n', \{g, \mathsf{nodes}(n', n) \mapsto 1\}, A)}\ \text{F-}\textsc{Down}$$

$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n', \{g, n \mapsto 1, \mathsf{nodes}(n', n) \mapsto 1\}, A) \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A)}\ \text{F-}\textsc{Up}$$

$$\frac{n' \leq n}{\mathsf{pure}(r, n, \{g, \mathsf{nodes}(n', n) \mapsto \overline{k}\}, A) \Rrightarrow \mathsf{pure}(r, n', g, A)}\ \text{P-}\textsc{Up}$$

$$\frac{}{\mathsf{access}(r, n, g, k, A) \Rrightarrow \mathsf{access}(r, n, g, k, n)}\ \textsc{Forget}$$

Figure 15: Splitting and joining of access permissions

```
class BufferedInputStream extends FilterInputStream {
states ready, reads refine open;
states within, eof refine ready;
states depleted, filled refine within;
states partial, complete refine filled;

$\text{reads} := \textit{reading} = \texttt{true}$
$\text{ready} := \textit{reading} = \texttt{false}$
$\text{depleted} := pos \geq count \otimes \text{unique}(\textit{super}, \text{alive}, \text{within})$
$\text{partial} := \textit{pos} < \textit{count} \otimes \textit{count} < \textit{buf.length} \otimes \text{unique}(\textit{super}, \text{alive}, \text{open})$
$\text{complete} := \textit{pos} < \textit{count} \otimes \textit{count} = \textit{buf.length} \otimes \text{unique}(\textit{super}, \text{alive}, \text{open})$

private boolean reading = false;
private int[] buf = new byte[8192];
private int pos = -1, count = 0;

public int read() $: \forall g : \{\text{alive}, \text{open}\} \mapsto \text{Fract}.\forall k : \text{Fract}. \ldots =$
unpack$(\text{open}, k, \text{open})$ in
  let r = reading in if(r == false, ...  fill() ...  )

private bool fill() $: \forall g : \{\text{alive}, \text{open}\} \mapsto \text{Fract}.\forall k : \text{Fract}.$
  $\text{share}(\textit{this}_{\text{fr}}, \text{open}, g, k, \text{depleted} \oplus \text{eof}) \multimap$
    $\text{share}(\textit{this}_{\text{fr}}, \text{open}, g, k, \text{available} \oplus \text{eof}) =$
unpack$(\text{open}, k, \text{depleted} \oplus \text{eof})$ in
  assign count = 0 in assign pos = 0 in
  assign reading = true in
  pack to reads in
    let b = super.read() in
    unpack$(\text{open}, k, \text{open})$ in
      let r = reading in assign reading = false in
      assign count = 0 in assign pos = 0 in
      if(r, if(b = -1, pack to eof in false,
              pack to depleted in doFill(b)),
            pack to eof in false)

private bool doFill(int b) $: \forall g : \{\text{alive}, \text{open}\} \mapsto \text{Fract}.\forall k : \text{Fract}.$
  $\text{share}(\textit{this}_{\text{fr}}, \text{open}, g, k, \text{depleted} \oplus \text{partial}) \multimap$
    $\text{share}(\textit{this}_{\text{fr}}, \text{open}, g, k, \text{partial} \oplus \text{complete}) =$
unpack$(\text{open}, k, \text{depleted} \oplus \text{partial})$ in
  let c = count in let buffer = buf in
  assign buffer[c] = b in assign count = c + 1 in
  let l = buffer.length in
  if(c + 1 >= l, pack to complete in true,
    assign reading = true in pack to reads in
      let b = super.read() in unpack$(\text{open}, k, \text{open})$ in
        let r = reading in assign reading = false in
        assign count = c + 1 in assign pos = 0 in
        pack to partial in
          if(r == false || b == -1, true, doFill(b))
```

Figure 16: Fragment of `java.io.BufferedInputStream` in core language