Verifying Stateful Programs with Substructural State and Hoare Types

Johannes Borgström *
Department of Information Technology, Uppsala University, Sweden
borgstrom@acm.org

Juan Chen
Microsoft Research, Redmond
juanchen@microsoft.com

Nikhil Swamy
Microsoft Research, Redmond
nswamy@microsoft.com

Abstract

A variety of techniques have been proposed to verify stateful functional programs by developing Hoare logics for the state monad. For better automation, we explore a different point in the design space: we propose using affine types to model state, while relying on refinement type checking to prove assertion safety.

Our technique is based on verification by translation, starting from FX, an imperative object-based surface language with specifications including object invariants and Hoare triple computation types, and translating into FINE, a functional language with dependent refinements and affine types. The core idea of the translation is the division of a stateful object into a pure value and an affine token whose type mentions the current state of the object. We prove our methodology sound via a simulation between imperative FX programs and their functional FINE translation.

Our approach enables modular verification of FX programs supported by an SMT solver. We demonstrate its versatility by several examples, including verifying clients of stateful APIs, even in the presence of aliasing, and tracking information flow through side-effecting computations.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Security, Verification, Languages, Theory

Keywords Security type systems, dependent types, affine types

1. Introduction

Several recent papers propose a verification methodology for stateful functional programs by developing Hoare logics for the state monad (Borgström et al. 2010; Nanevski et al. 2006; Swierstra 2009). Tools based on this approach are known to be powerful. For example, the Ynot tool has been used to carry out interactive proofs of correctness for programs that manipulate B+ trees (Malecha et al. 2010), a pointer structure with tricky sharing properties.

While such successes make a good case for interactive machine-assisted proofs in a Hoare logic for the state monad, this paper explores a different point in the design space of a program logic for stateful functional programs. Our primary motivation is to automate verification of functional programs that use local state, such that program components can be verified independently. We aim to develop a verification methodology based on classical first-order logic, for which powerful off-the-shelf solvers already exist. (In contrast, tools like Ynot work with separation logic to recover modularity. Although the situation is improving (Chlipala et al. 2009), automation for separation logic remains a significant challenge.)

Our work considers a well-known alternative to monadic state as the basis of a verification methodology: substructural state, by which we mean state modeled using linear (use-once) or affine (use-at-most-once) types (Wadler 1990). Our insight is that the problem of verifying stateful functional programs can be factored into two pieces. First, we can use an affine type system to model the stateful behavior of the program in a purely functional style. Then, to prove the safety of assertions in the program, we can rely on automated refinement type checking for a first-order classical logic. Affine types partition the state of the program into a number of disjoint pieces, thus yielding a modular verification procedure without necessitating the use of the separation logic connectives.

In addition to improved automation, modeling state using affine types is attractive since it enables combinations of Hoare logic with other program verification disciplines. We work out one such example in detail in §5.1, where we combine our basic approach with a discipline of fractional capabilities to control aliasing.

Concretely, our work is based on FINE (Swamy et al. 2010a), a purely functional language with a type system containing dependent refinements and affine types. Our contributions include:

- We present FX (§3), an extension of the term syntax of FINE with stateful commands including object allocation, deletion and mutation. We also extend the type language with Hoare types (Nanevski et al. 2006) to give specifications (in the style of Hoare triples) to stateful code in FX.
- We show how to translate FX programs to FINE (§4), using FINE's affine types to model state, and dependent refinements for the translation of Hoare types. The core idea is to divide a stateful object into a pure value and an affine token whose type mentions the current state of the object. We prove that our translation is a simulation, which implies that refinement type checking in FINE guarantees assertion safety for FX.
- We demonstrate the effectiveness of our approach on a variety of examples (§5), which, although simpler than the B+ trees verified interactively in Ynot, are still known to be challenging. Our examples include a conference management program previously verified for authorization properties by Swamy et al. (2010a); a client of a stateful API of collections and iterators studied by Bierhoff and Aldrich (2007); and an encoding of in-

* The first author was employed by Microsoft Research, Cambridge, for the duration of his work on this paper.

PLPV'11, January 29, 2011, Austin, Texas, USA.

formation flow tracking suitable for use with programs that may leak information via side effects (an enhancement of a technique studied by Swamy (2008)).

Although the results of this paper are limited (primarily) to first-order programs, we discuss the extension to the higher-order case in §5.3. The remainder of this section presents a short overview of our work and some technical challenges that we solve.

1.1 Marrying substructural and dependent types

Linear types have been known as an effective tool for modeling state for the last 20 years (Wadler 1990), roughly as long as the monadic approach. Yet, with a few notable exceptions, verification tools and programming languages have only seldom adopted linear types as model of state. One reason for this, particularly in the type-based approach to verification, is the subtle interaction between linear and dependent types. The metatheory of an arbitrary mixture of these types is usually considered intractable. For example, Linear LF (Cervesato and Pfenning 2002), perhaps the canonical presentation of linearity and dependency, restricts linear values from appearing as indices in types. FINE has a similar restriction on affine indices, placing seemingly severe limitations on expressiveness.

For a sense of the difficulty, consider implementing a function incr = λx:aInt. x+1 that increments an affine integer x:aInt. Since x is affine, its storage might be reclaimed when read, and $x+1$ can then either be written into a freshly allocated variable or into the old location of x. A more precise type of incr could be x:aInt → {y:aInt | y>x}. This is the type of a dependent function from affine integers x to affine integers y, where the refinement type to the right of the arrow states that the returned value y is greater than x. However, this type is illegal in Linear LF and FINE, because the refinement formula y>x is interpreted as a type, and the affine values x and y cannot be used at the type level.

A key technical contribution of our work is to surmount this difficulty by combining value-indexed types and affine capabilities (Walker et al. 2000) to permit unrestricted refinements of stateful values. In our solution, the type of an increment function, say incr2, could be x:int → Token x → (y:{y:int | $y > x$} ∗ Token y), where Token :: int ⇒ A is a dependent type constructor that constructs an affine type from an int-typed value. (The kind of affine types is A.) The type of incr2 indicates that to increment x, client code must present a token for x. The incr2 function destructs the token for x, produces y, and a new token for y. By making Token x affine, we ensure that the client can no longer use x in a context that expects a mutable integer. By indexing the type of the token with the value of x, a token of one value cannot be confused with a token for another. Finally, by separating x from its token, we can use x in refinement formulas, e.g., in the formula $y > x$ in the return type of incr2.

1.2 Shielding programmers from affinity

Clearly, requiring the programmer to directly manipulate tokens is far from optimal—mutable values and their tokens have to be threaded through the program in a functional style. Additionally, exposing affinity to the programmer is also questionable, since it limits the use of library code—data structures that contain affine values become affine themselves, making it hard to use, say, a standard non-affine lists to hold affine values.

To counter these difficulties, we present FX, a surface syntax with imperative commands that hides the low-level plumbing of tokens and affinity from the programmer. To illustrate FX, consider a snippet of an FX implementation of ConfWeb, a verified conference management tool we had previously implemented directly in FINE.

```
1 type review = {|txt:string; u:user|}
2 val upd_review: r:review → x:string →
3                 {(s1) ⊤} _:unit {(s2) s2(r).txt=x && s2(r).u=s1(r).u}
4 let upd_review r x = r.txt := x
```

Here, we define a mutable object type review of reviews, with fields for review text and the reviewer u. Line 4 shows the implementation of a function upd_review that updates a review r by assigning to its txt field. FX programmers can use imperative commands like assignment rather than threading mutable objects through the program. Mutable objects can also be placed in standard data structures, without having to re-implement libraries to deal with affinity.

Specifications in FX are written using Hoare types, a notation resembling Hoare triples. Lines 2-3 show the specification of upd_review, with the form x:t1 → {(s1) Φ} r:t2 {(s2) Ψ}, where the formula Φ is a pre-condition on the function argument x:t1 and pre-state s1; r:t2 is the name and type of the return value; and Ψ is a post-condition on the argument x, the pre-state s1, the return value r, and the post-state s2.

The main technical development of this paper is a translation from FX to FINE that introduces affine tokens and threads stateful values and their tokens through the program. The main results are Theorem 1, which establishes that the translation is a simulation, and Corollary 2, which establishes progress for FX programs that translate to well-typed FINE programs.

2. Overview

We begin with a brief review of FINE, then discuss ConfWeb, a conference management application we previously implemented in FINE. This application motivates, in part, the need for better support for imperative features. §2.2 presents the design of FX using examples from ConfWeb. §2.3 presents informally our methodology of verifying FX programs via translation to FINE.

2.1 A brief review of FINE

FINE is an experimental, purely functional programming language on the .NET platform. The principal novelty of FINE is in its type system, which is designed to support static verification of safety properties via a mixture of refinement and substructural types. FINE has, thus far, focused primarily on verifying security properties, including authorization and information flow controls. However, by design, FINE has little in it that is security-specific. This paper is a step towards putting FINE to use for the general-purpose verification of programs that mix both functional and imperative idioms. We discuss several elements of FINE's design next—for details, consult our prior papers (Chen et al. 2010; Swamy et al. 2010a).

Value-indexed types. Types in FINE can be indexed both by types (e.g., list int) as well as by values. For example, array int 17 could represent the type of an array of 17 integers, where the index 17:nat is a natural number value. FINE prohibits non-value expressions from appearing as type indices, in effect forbidding type-level computations. This restriction limits expressiveness, but considerably simplifies FINE's metatheory and implementation.

Dependent function types. Functions in FINE are, in general, given dependent function types, i.e., their range type depends on their argument. Dependent function types are written x:t → t', where the formal name x of the parameter of type t is in scope in t'. For example, the type of a function that allocates an array of n integers can be given the type n:nat → array int n. When a function is non-dependent, we simply drop the formal name.

Refinement types. A refinement type in FINE is written {x:t | ϕ}, where ϕ is a formula in which x may appear free. Formulas are drawn from the same syntactic category as types, although, for readability, we typeset formulas differently and use distinct meta-variables (ϕ versus t). In practice, types are refined by formulas from a first-order logic with equality, extended with user-defined predicates. For example, we may give the (partial) specification ∀α.l:list α → {m:list α | ∀x:α. *In x l ⇔ In x m*} to a list permutation. Universal quantifiers in formulas are represented using

dependent arrows and existential quantifiers using dependent pairs; In is a user-defined predicate for list membership; and connectives like $\Leftrightarrow$ are represented using indexed types (we elaborate below).

Refinement type checking. A refinement type $\{x{:}t \mid \phi\}$ is inhabited by values $v{:}t$, for which $\phi[v/x]$ is derivable. Derivability is defined with respect to assumptions induced by the program context (e.g., equalities due to pattern matching) as well as a set of assumptions that axiomatize user-provided predicates. We formalize this using an LCF-style (Milner 1979) kernel for FINE, which contains the inference rules for a classical first-order logic with equality, extended with constructors corresponding to user-provided axioms. As such, user-provided axioms must be used with care, since faulty axioms compromise soundness. Derivability is decided by relying on Z3 (de Moura and Bjorner 2008), an SMT solver. Formally, refinement types are viewed as Σ-types. However, we include a program transformation (somewhat similar to the coercions used by Sozeau (2006)) that systematically inserts pack/unpack operations so as to equip refinement types with a subtyping relation that allows programmers to view $\{x{:}t \mid \phi\}$ as a subtype of t.

Affine types. Dependent refinements in FINE bear close resemblance to constructs found in related languages like F7 (Bengtson et al. 2008) and Sage (Flanagan 2006), despite several technical differences. A substantial difference in FINE however, is the addition of affine types. The interaction between dependent and affine types in FINE is strictly regulated by the *no-affine-indices* restriction—this prohibits the use of values with affine types as type indices. As such, FINE is closely related to Linear LF (Cervesato and Pfenning 2002), which integrates linear and dependent types, with a similar restriction that prevents linear resources being mentioned in types. This paper shows that even with this restriction, the combination of dependent and affine types available in FINE provides a powerful set of primitives for verifying programs that use mutable local state.

Kind language. To enforce the no-affine-indices restriction (as well as to keep account of type constructors) FINE employs a system of kinds. The (slightly simplified) syntax of kinds is shown below:

$$\text{kinds} \quad k \quad ::= \quad \star \mid \mathtt{A} \mid \alpha{::}k \Rightarrow k \mid x{:}t \Rightarrow k$$

Types in FINE are divided into two basic kinds: $\star$, the kind of normal non-affine types, and $\mathtt{A}$, the kind of affine types. A $\star$-kinded type can be coerced to the $\mathtt{A}$ universe using the modality ¡, e.g., $\mathsf{int}{::}\star$, while $¡\mathsf{int}{::}\mathtt{A}$. Type constructors are given arrow kinds, which come in two flavors. The first, $\alpha{::}k \Rightarrow k'$ is the kind of type functions that construct a k'-kinded type from a k-kinded type α. Just as at the term level, type-level arrows are dependent—the type variable α can appear free in k'. Type functions that construct value-indexed types are given a kind $x{:}t \Rightarrow k$, where x names the index of type t and x can appear free in k. In both cases, when the kind is non-dependent, we simply drop the name of the index. The no-affine-indices manifests itself as a restriction on kinds $x{:}t \Rightarrow k$, where the domain type t must have kind $\star$. For example, the kind of list is $\star \Rightarrow \star$; the kind of the value-indexed array constructor is $\star \Rightarrow \mathsf{nat} \Rightarrow \star$; the kind of the propositional connective And is $\star \Rightarrow \star \Rightarrow \star$; the kind of the user-defined predicate In is $\alpha{::}\star \Rightarrow \alpha \Rightarrow \mathsf{list}\ \alpha \Rightarrow \star$.

Module system. Aside from the core type system, FINE has a simple module system to define certain types private to a module, which in effect forces clients of the module to view values of these private types abstractly. FINE uses the module system to prove secrecy and authenticity properties. In this paper, the module system comes in handy for the definition of unforgeable capabilities and the corresponding authenticity property plays a crucial role in proving the correctness of our translation from FX to FINE.

Proof-carrying compilation. Finally, it is worth mentioning that FINE is compiled in a proof-carrying style to DCIL, .NET bytecode enhanced with dependent and affine types in a backwards-compatible way. In addition to certification, compilation to DCIL allows FINE to interoperate easily with the other .NET languages. By virtue of the translation of the forthcoming sections, FX too enjoys a proof-carrying translation to DCIL (Chen et al. 2010).

```
1  type user=int
2  type NoConflict :: user ⇒ user ⇒ ⋆
3  assume nc_sym: ∀u1, u2. NoConflict u1 u2 ⇒ NoConflict u2 u1
4  val checkNC: u1:user → u2:user → {b:bool | b=true ⇔ NoConflict u1 u2}
5  type review = {|txt:string; u:user|}
6  type paper = {|authors:list user; revs:list review | ∀u:user, r:review.
7                  (In u self.authors && In r self.revs) ⇒ NoConflict u r.u|}
8  (* Allocation function with a Hoare type *)
9  val mk_review: txt:string → u:user → {(s1) ⊤} r:review
10                                      {(s2) s2(r).txt=txt && s2(r).u=u}
11 let mk_review txt u = new review{|txt=txt; u=u|}
12 (* Pure library function can operate on a list of mutable objects *)
13 val for_all: ∀α::⋆, P::α ⇒ ⋆.
14                 f:(x:α → {b:bool | b=true ⇔ P x}) → l:list α →
15                 {b:bool | b=true ⇔ ∀x:α. In x l ⇒ P x}
16 (* Impure function mutates a paper; also given a Hoare type *)
17 val add_review: p:paper → r:review →
18                 {(s) not(In s(r) s(p).revs)} o:option review
19                 {(t) (In s(r) t(p).revs || t(o)=Some s(r))&& ...}
20 let add_review p r =
21   if for_all<user,NoConflict r.u> (checkNC r.u) p.authors
22   then let tl = (p.revs :=: Nil) in p.revs :=: Cons r tl; None
23   else Some r
```

Figure 1. Invariants on mutable objects in ConfWeb

2.2 ConfWeb: A first taste of FX

Our prior work on FINE included the development of an application ConfWeb, a conference management tool based on the Continue server (Krishnamurthi 2003). Continue's behavior is governed by an authorization policy, modeled with liberal use of object-orientation and mutable state in Alloy (Jackson 2002). A basic data structure in the model is a *paper*, an object with mutable fields holding a paper submission and list of its associated reviews. An invariant on *paper* ensures that the reviews of a paper are from reviewers not conflicted with the paper's authors. Our prior implementation of ConfWeb modeled many of these stateful features, but some features (including invariants on mutable objects like *paper*) proved to be too cumbersome to implement directly in FINE.

In this section, we introduce FX, a surface syntax for FINE with direct support for mutable objects. FX considerably simplifies programming with mutable objects and makes an implementation of ConfWeb more faithful to its original specification. Subsequent sections shows how mutable state can be systematically translated away via translation to purely functional FINE programs.

Figure 1 shows a fragment of ConfWeb written in FX. The key data structure in ConfWeb is paper (line 6), a mutable object with fields containing the authors of a paper and its reviews (in addition to other fields, not shown). We verify that this program preserves the no-conflict invariant on paper. The full program translates to a 109-line FINE program that is verified automatically in 8 seconds.

Mutable objects and invariants. FX extends FINE with a notation for records with mutable fields—we call these types *objects*, although our core calculus for FX does not include other OO features like inheritance. Object types are of the form $\{|f1{:}t1;\ldots;fn{:}tn \mid \phi|\}$, where each fi is a mutable field of type ti, and the formula ϕ is an object invariant in which the name *self* refers to the object itself. Values of non-object types like string and int are immutable.

The paper object and its invariant. Line 6 defines paper, an object with a field authors (a list of immutable integers, standing

for user ids); and a field revs, a list of objects representing the reviews of a paper. The invariant on paper is a formula which uses two user-defined predicates. The first predicate, *In*, stands for list membership. The predicate *NoConflict* is a binary predicate to reflect the absence of conflict-of-interest between a pair of users. Line 2 shows the kind of *NoConflict* and line 3 shows a user-provided axiom that asserts that *NoConflict* is symmetric. Rather than provide further axioms to define *NoConflict*, at line 4 we show the signature of a trusted external function checkNC, where we use dependent refinement types to assert that checkNC decides the *NoConflict* relation. The implementation of checkNC consults a database that records conflict-of-interest declarations by authors and PC members—interoperability between FINE and the rest of .NET allows us to implement checkNC in F# and to call it from FX.

Impure functions and Hoare types. At line 11, we show the implementation of a function mk_review, which allocates a new review object. Its specification at lines 9-10 shows the type of an impure function, which, in general, has the form $x{:}t \to \{(s1)\ \Phi\}\, r{:}t' \ \{(s2)\ \Psi\}$, as explained in §1.2. For mk_review, the pre-condition Φ is trivial—$\top$, which always holds. The post-condition Ψ contains terms of the form *s2(r)*—this stands for the value of the mutable object r in the state *s2*. In this case, *s2(r)* is a record value of the object type review. The post-condition of mk_review states that the fields of the returned object r contain the arguments passed to the function.

Interacting with libraries. Next, at lines 13-15, we show the type of for_all, a library function on lists. This is a higher-order function whose argument f is a (pure) boolean-valued function over the α-typed elements of its second argument l, where f decides some property *P* of the list elements. The type we give to for_all is polymorphic in both α and *P*. The return type indicates that for_all returns true if, and only if, f returns true on each element of l. The type also indicates that both f and for_all are pure, since neither of them has a Hoare type. This pure function can be programmed in FINE and verified against its specification. The implementation is entirely standard—we omit it here for brevity.

Mutation and controlled aliasing. Finally, at lines 17-23, we show the type and implementation of a function add_review that tries to add a review to a paper. Informally, the implementation checks that the review's author is not conflicted with any of the paper's authors and, if the check succeeds, adds the review to the paper and returns None. Otherwise, add_review leaves the paper unchanged and returns the review to the caller. The type shows an impure function, where the precondition requires the review r to not be present in the paper p's revs list; the returned value is an option review, where the post-condition shows that the review r has been stored in the paper p, or that r is contained in the returned option value.

The implementation of add_review reveals several subtleties of FX. First, to properly model stateful invariants on objects, FX forbids constructing aliases to mutable objects. (§5.1 shows how to relax this restriction.) If a mutable object o is stored in another object c's f field, then a field projection c.f constructs an alias to the object o. As a result, FX does not permit unrestricted field projection. Instead, we provide other operations to emulate field projections (which get compiled to a small set of function calls in FINE).

At line 22, we call for_all, providing explicit instantiations for its type and predicate parameters, α and *P* respectively. We apply checkNC to r.u; project p.authors; and pass both to for_all. The projections here are safe since the type of for_all is pure, and hence does not create any unsafe aliases to p. Additionally, since the user type is immutable, it is always safe to project r.u. If the no-conflict check fails, we return r to the caller. Otherwise, we update the paper, adding the new review r to p.revs and returning None. The update uses FX's swap command, c.f :=: v, which replaces the f field of c with the value v and returns the original contents of c.f. We first swap out the original list of reviews from p; add our new review to it; and then swap the extended list back in. Note that it here would have been safe to project p.revs and use it in a normal assignment p.revs := Cons r p.revs, since this creates no aliases. Projections of fields that contain primitive or immutable types are also safe. However, swap is the appropriate primitive for languages with linearity (Ahmed et al. 2007) or affinity. Our technical report discusses these and other conditions under which projections and assignments are acceptable, and shows how they can be encoded in FINE.

In both branches of add_review, the mutable object r is stored within another object—in the **then**-branch, within p; in the **else**-branch, in the returned option object. Since creating unrestricted aliases to mutable objects is forbidden, and the caller retains a reference to r, we need to indicate to the caller that its reference r is invalidated. We use the absence of a sub-term *t(r)* in the post-condition to indicate that r is invalidated in the post-state, i.e., that it may no longer be used by the caller. By contrast, the type of upd_review shows that the review r is still accessible to the caller by mentioning *s2(r)* in its post-condition. Currently, FX cannot express that a reference has been conditionally consumed, e.g., returning a boolean success code b from add_review to indicate in the post-condition that *r* is consumed only if *b=true*.

2.3 Verifying FX programs via translation to FINE

This section sketches the translation of FX to FINE. The main idea is to thread record values corresponding to mutable objects through the program in a purely functional style. For this functional translation to be sound, we also thread affine capabilities for each object to render stale record values unusable. As discussed in §1, by making the capabilities affine, while giving record values $\star$-kinded types, we circumvent the no-affine-indices restriction and give precise refinement types (corresponding to object invariants, pre- and post-conditions) to the functional translation of imperative FX code. As a result, we can rely on FINE's automated refinement type checking system to verify FX programs.

A module for FX primitives. The translation is organized around a FINE module, Prims, which collects definitions of FX primitives. Certain types will be private to Prims, and the metatheory of FINE (specifically, the value abstraction theorem provided by its module system) guarantees that clients of Prims treat these types abstractly.

Objects and constructors. An object with mutable fields in FX, say r:review, where review = {|txt:string; u:user|} from Figure 1, is represented (roughly) in FINE as a pair of 1) a record value r:review where review is a record type in FINE, {txt:string; u:user}; and 2) a token value v:Token r, used as a capability for r, where $\text{Token}{::}\alpha{::}\star \Rightarrow \alpha \Rightarrow A$, is a constructor of an affine, value-indexed type. Given an object type declaration in FX, we generate several public functions in Prims, corresponding to constructors, field updates, etc., as shown below. Imperative commands in FX get translated into calls to these functions.

```
1 private type Token:: α::⋆ ⇒ α ⇒ A = | MkToken: x:α → Token x
2 private type review = {txt:string; u:user}
3 val mkReview: txt:string → u:user → unit →
4                 (r:review ∗ Token r ∗ {_:unit | r.txt=txt && r.u=u})
5 val reviewUpdTxt: r:review → txt:string → (Token r ∗ unit) →
6                 (r':review ∗ Token r' ∗ {_:unit | r'.txt=txt && r'.u=r.u})
7 val reviewReadTxt: r:review → (Token r ∗ unit) →
8                  (txt:string ∗ Token r ∗ {_:unit | r.txt=txt})
```

The function mkReview is the constructor function of review. In addition to the constructed record value r and its token, it returns a unit value refined with a formula recording information about the contents of r. The reviewUpdTxt function updates the txt field. Its arguments include r the object to be updated and its token; the return type shows r' and its token, where the refined unit shows that

r' differs from r in only its txt field. We also show reviewReadTxt, a pure function that serves as a projection function for the txt field. Note that all three functions take seemingly redundant unit-typed arguments and return refined units—we include these because, as we shall see, they help keep our translation uniform.

Finally, the Token type, which has a single constructor MkToken, is declared private to Prims to ensure that its values cannot be manufactured directly by clients. Likewise, the review record type is also private. Stateful operations on r:review (e.g., updating one of its fields) require a client to present a v:Token r attesting to the validity of r; the operation consumes the token, rendering r invalid for further use in stateful operations. Pure operations like reviewReadTxt thread the token back to the caller for further use.

Hoare types. The FINE type below is the translation of the FX type r:review → {*(s1) s1(r).u=1*} x:int {*(s2) s2(r).u=s1(r).u+x*}.

```
1 r:review →(Token r ∗ {_:unit | r.u=1}) →
2         (x:int ∗ r':review ∗ Token r' ∗ {_:unit | r'.u=r.u+x})
```

This type shows a function that takes as arguments tokens for all the variables x where *s1(r)* is a subterm in the pre- or post-condition; in this case a token for r, since only *s1(r)* is mentioned. In addition to the token, we have a unit argument refined with the translation of the pre-condition. The return type of the function shows x:int the value returned by the impure FX computation. We also include in the return type all variables y where *s2(y)* is a subterm of the post-condition (i.e., r,); tokens for these values; and, the translation of the post-condition formula expressed as a refinement on a unit value. The scoping rules of dependent functions, pairs (aka sums), and refinement types naturally allow the post-condition to describe properties of the initial value of r in the pre-state, the returned value x, and the value of r in the post-state, (i.e., r').

Adapting libraries to work with mutable objects. A significant advantage of our token-based translation is that it lends itself naturally to the re-use of types and libraries that were originally constructed for purely functional programs. For example, we show below how the standard list library can be adapted to work with lists of mutable objects. The list type shown below is the standard type for a list of non-affine values. FX programs are free to use the constructors of list directly to store immutable values within them, and to pattern match on lists to extract immutable values as well. But, we require a bit more care to store mutable objects within lists, since the constructed list would capture a reference to the object. For this, we provide functions in Prims that construct and destruct lists while managing tokens appropriately.

```
1 type list :: ⋆⇒ ⋆= Nil : list α
2                  | Cons : α→list α→list α
3 val mkCons: hd:α →tl:list α→(Token hd ∗ Token tl ∗ unit) →
4                              (l:list α∗ Token l ∗ {_:unit | l=Cons hd tl})
5 val unCons: l:list α→(Token l ∗ {_:unit | ∃hd,tl. l=Cons hd tl}) →
6         (hd:α ∗ tl: list α∗ Token hd ∗ Token tl ∗ {_:unit | l=Cons hd tl})
7 val mkNil: unit →(l:list α∗ Token l ∗ {_:unit | l=Nil})
```

The function mkCons takes an object value hd:α and a list of object values tl:list α; tokens for each of them; consumes the tokens for hd and tl and returns a new list l, a token for l, and a refined unit capturing the structure of l. While it is possible for a client program to pattern match against l to extract its components, to mutate any object in l, a client must destruct l by presenting l and its token to unCons; proving a pre-condition that l is indeed a Cons-cell; and receiving the components of l, tokens for each of them, and a post-condition relating the components to the original l.

By adopting a token-based approach, not only have we been able to circumvent the no-affine-indices restriction, but we have been able to store mutable objects in standard lists, operate over these lists using the standard implementations of map, fold, for_all, etc., since these functions are pure. Impure functions that need to explicitly manipulate the objects in a list, need to be written and verified in FX. Even if the no-affine-indices restriction were to somehow be lifted, representing mutable objects using affine types directly would lead to significant difficulties. For example, if our translation were to translate the mutable object rev in FX to an affine record arev::A, then we would need an alternative type of list, say alist :: A →A, in which to store arev::A values and a duplicate standard library to operate over alist α values (which would be much more cumbersome to write, since we would have to pay attention to the affinity of both the list and of the α-typed values it contains). Even in a system like F° (Mazurak et al. 2010), which supports sub-kinding between normal and linear kinds, we would have to construct separate libraries to work with linear and non-linear lists. Tokens make life much easier!

Nested objects and object invariants. Translating objects with nested objects does not add any further complexity. Object invariants too are easily translated using refinement types. The translation of the FX type paper (from Figure 1) is shown below. The constructor and destructor are similar to those for list and thus omitted.

```
1 private type paper = {self:{authors:list user; revs: list rev} |
2         ∀u:user, r:rev. In u self.authors && In r self.revs ⇒ NoConflict u r.u }
3 val swapRevs: p:paper →newrl:list rev →(Token p ∗ Token newrl ∗
4       {_:unit | ∀u:user, r:rev. In u p.authors && In r newrl ⇒ NoConflict u r.u}) →
5     (oldrl:list rev ∗ p':paper ∗ Token oldrl ∗ Token p' ∗
6     {_:unit | p'.authors=p.authors && p'.revs=newrl && p.revs=oldrl})
```

The refinement on the paper record shows the invariant from FX. Swapping a new value newrl into the revs field of a paper p calls swapRevs and has to prove that newrl satisfies the object invariant. The tokens of the old record p and the new field value newrl are consumed. The token for the old field value oldrl is available again.

As an example, we show how to translate an FX expression p.revs :=: Cons r tl; None using swapRevs:

```
1 let newrl, tok_newrl = mkCons r tl (tok_r, tok_tl, ()) in
2 let orl, p', tok_orl, tok_p', _ = swapRevs p newrl (tok_p, tok_newrl, ()) in
3 let none, tok_none, () = mkNone () in
4 none, p', tok_none, tok_p', ()
```

At line 1, we construct the new review list newrl by calling mkCons. We then call swapRevs to swap newrl in and the old field value orl out, creating a new record p'. We create a record none corresponding to None at line 3. At line 4, in addition to returning none, we thread out p' and its token, since it represents an updated object that may be accessed later.

3. Formalization of core FX

This section formalizes the syntax and the dynamic semantics of a core subset of FX. As mentioned previously, FX is considered a surface syntax for FINE. FX has no static semantics of its own. Instead, we introduce the notion of *type shapes* for FX and formalize a simple syntactic analysis of FX to compute these shapes. Type shapes serve only to facilitate the translation of FX to FINE (§4), where the types of an FX program can be interpreted using FINE's type system. We show that this interpretation is sound by proving a simulation between FX programs and their FINE counterparts.

Core FX is a lambda calculus augmented with imperative commands for object allocation, deletion, and mutation. It uses a monomorphic typing discipline based around dependent functions, Hoare types and objects with mutable fields. Its syntax is below.

Core syntax of FX

$\tau ::= TC \mid x{:}\tau \to C \mid \{\!\{\overline{f{:}\tau} \mid \phi\}\!\}$	types
$C ::= \{(s)\Phi\}\, r{:}\tau\, \{(s')\Psi\}$	Hoare types
$\phi, \Phi, \Psi ::= P\ \overline{p} \mid \forall x{:}\tau.\Phi \mid \exists x{:}\tau.\Phi$	formulas
$\quad \mid \Phi \wedge \Phi' \mid \Phi \vee \Phi' \mid \neg\Phi \mid \top \mid \bot$	
$S ::= \textbf{assume}\ \phi \mid P :: k \mid S; S'$	signature

$$
\begin{array}{lll}
p ::= v \mid s(\eta) \mid p.f & & \text{atoms} \\
v ::= \eta \mid c \mid \textbf{error} \mid \lambda x{:}\tau.e & & \text{values} \\
\eta ::= x \mid \ell & & \text{names} \\
e ::= v \mid v\ v' \mid \textbf{let}\ x = e\ \textbf{in}\ e' \mid e{:}C \mid \textbf{assert}\ (s)\Phi & & \text{terms} \\
\quad \mid \textbf{new}_\tau\ \{\!|\overline{f = v}|\!\} \mid \textbf{destruct}\ v\ \textbf{as}\ \{\!|\overline{f = x}|\!\}\ \textbf{in}\ e \mid v.f :=: v' &&
\end{array}
$$

FX kinds are the same as those in FINE (see §2.1) and omitted here. FX types τ include nullary type constants TC, like unit. Abstractions are given impure, dependent function types, $x{:}\tau \to C$, where x names the formal and is bound in the computation type C, whose syntax was described in §1.2. (Pure function types $x{:}\tau \to \tau'$ are sugar for $x{:}\tau \to \{(s)\top\}\,\tau'\,\{(t)\top\}$, since the pre- and post-conditions $\top$ do not mention any objects.) We also have object types $\{\!|\overline{f{:}\tau} \mid \phi|\!\}$, which can be thought of as records with mutable fields $f_1{:}\tau_1, \ldots, f_n{:}\tau_n$. The formula ϕ specifies an object invariant, referring to each of the fields using self.$f_1, \ldots,$ self.f_n.

Formulas are ranged over by ϕ, Φ, and Ψ, and are first order formulas defined over a set of n-ary base predicates P over atoms p. The atoms include FX values v; field projections (which are not themselves values in FX); and terms of the form $s(\eta)$, which stands for the value of an object named η in the state s. We use ϕ for object invariants, which cannot refer to state variables, while Φ and Ψ may contain state variables.

A signature S collects top-level user-specified assumptions **assume** ϕ and declarations of predicates $P :: k$ of an FX program. The assumptions are global and do not refer to state variables. The predicate declarations specify kinds (k) of predicates (P).

Values in FX are names (variables x or memory locations ℓ); constants c, where the type of each constant c is denoted TC_c; **error**, used only during execution to indicate failures; or lambda abstractions. Expressions e are required to be in A-normal form (Flanagan et al. 1993). Dependent typing in FX and in FINE only permits values to index types—A-normal form helps ensure that expressions do not escape into the type level. The expression forms include the standard function application $v\ v'$ and let-binding **let** $x = e$ **in** e'; the form $e{:}C$ ascribes a computation type C to e; and **assert** $(s)\Phi$ asserts formula Φ of the current state s.

The imperative fragment of FX includes two core constructs for object allocation and deletion. Allocation is $\textbf{new}_\tau\ \{\!|\overline{f = v}|\!\}$, which creates a new object of type τ initializing its fields $f_1, \ldots, f_n$ to $v_1, \ldots, v_n$. Expression **destruct** v **as** $\{\!|\overline{f = x}|\!\}$ **in** e destructs an object v, and binds the contents of its fields $f_1, \ldots, f_n$ to fresh variables $x_1, \ldots, x_n$ in e. The fresh variables x_i are in scope within e, while v is no longer accessible in e or later in the program. Using these two constructs alone, we can encode field updates, field swaps, etc. However, we include field swaps, $v.f :=: v'$, as a primitive as well, since we use this in our examples. This instruction assigns the new value v' to the field f of v and returns the old value of the field.

3.1 Dynamic semantics

Runtime configurations in FX are expressions e paired with a store σ, a finite map from memory locations ℓ to object values $\{\!|\overline{f = v}|\!\}$. We show the key rules in the dynamic semantics of FX below, omitting standard rules for beta and let reduction—these can be found in our technical report. Congruence rules are elided.

The semantics has the form $(e, \sigma) \to (e', \sigma')$, and is mostly as one would expect, except for two subtleties. First, to specify progress for FX, it is convenient to ensure that FX programs never get stuck. Instead, we mark certain evaluation steps as erroneous and define safe FX programs as those that are guaranteed to reduce without taking erroneous steps. Specifically, when evaluating **assert** $(s)\Phi$ we check whether the current values in the store satisfy the formula Φ, under the assumptions in the signature S. Subterms $s(\ell)$ in Φ are replaced with $\hat{v}_\ell$, which is a location-free object value corresponding to $\sigma(\ell)$ (where $\sigma \vdash \ell \downarrow \hat{v}_\ell$ is in §4). Assertions evaluate to the error value if the check fails (the last rule). Although some care has to be taken to translate the two-state predicate Ψ properly, the translation of ascriptions $(e{:}C)$ to assertions is straightforward.

The other subtlety is that bindings in the store σ are tagged with a superscript ι, indicating whether a location represents allocated ($\oplus$) or freed ($\ominus$) memory. These tags have no impact on the reduction of FX programs, but the translation of FX to FINE uses these tags to prove that if an FX program is translated to a well-typed FINE program, it never references freed memory. The reduction rule for **new** creates a new memory location ℓ and tags it with $\oplus$. The rule for **destruct** substitutes the bound variables with the object contents in e and tags the destructed location with $\ominus$. Finally, the rule for swap updates the object and returns the old value.

Expressions with failures such as assertion failures or missing fields evaluate to the error value.

Selected dynamic semantics of FX: $(e, \sigma) \to (e', \sigma')$

$$
\begin{array}{l}
\sigma ::= \sigma + \{\ell \mapsto \{\!|\overline{f = v}|\!\}\}^\iota \mid \bullet \quad \text{where } \iota ::= \oplus \mid \ominus \\
(\textbf{assert}\ (s)\Phi, \sigma) \to ((), \sigma) \qquad \text{if } S \models \Phi[\hat{v}_\ell/s(\ell)], \forall \ell \in FV(\Phi) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \sigma \vdash \ell \downarrow \hat{v}_\ell \\
((e{:}\{(s)\Phi\}\, r{:}\tau\, \{(t)\Psi\}), \sigma) \to \\
\qquad (\textbf{let}\ \_ = \textbf{assert}\ (s)\Phi\ \textbf{in let}\ x = e\ \textbf{in} \\
\qquad\ \textbf{let}\ \_ = \textbf{assert}\ (t)\Psi[x/r, \sigma(\ell)/s(\ell)]\ \textbf{in}\ x, \sigma) \\
(\textbf{new}_\tau\ \{\!|\overline{f = v}|\!\}, \sigma) \to (\ell, \sigma + \{\ell \mapsto \{\!|\overline{f = v}|\!\}\}^\oplus) \quad \ell \notin \mathrm{dom}(\sigma) \\
(\textbf{destruct}\ \ell\ \textbf{as}\ \{\!|\overline{f = x}|\!\}\ \textbf{in}\ e, \sigma + \{\ell \mapsto \{\!|\overline{f = v}|\!\}\}^\iota) \to \\
\qquad (e[{}^{v_i}/_{x_i}]_{i=1}^n, \sigma + \{\ell \mapsto \{\!|\overline{f = v}|\!\}\}^\ominus) \\
(\ell.f_i :=: v_i', \sigma + \{\ell \mapsto \{\!|\overline{f = v}|\!\}\}^\iota) \to \\
\qquad (v_i, \sigma + \{\ell \mapsto \{\!|\cdots; f_i = v_i'; \cdots|\!\}\}^\iota) \\
(e, \sigma) \to (\textbf{error}, \sigma) \qquad e \neq v \text{ otherwise}
\end{array}
$$

3.2 Computing type shapes

In preparation for the translation of FX to FINE, we define a simple syntactic analysis of FX programs to compute *type shapes*. Type shapes describe an equivalence relation on computation types, where types that describe expressions with the same footprint (i.e, the set of locations read or written) are placed in the same equivalence class. The footprint of an expression (and hence the shape of its type) guides the translation of §4 since it determines the values that we must thread into and out of a translated FX expression.

Formally, we define two functions, iv (input variables) and ov (output variables), from computations types C to sets of names $X, Y \subseteq 2^\eta$. Given a computation type $C = \{(s)\Phi\}\, r{:}\tau\, \{(t)\Psi\}$, $\mathrm{iv}(C) = \{\eta \mid s(\eta) \in \mathrm{Subterms}(\Phi, \Psi)\}$ and $\mathrm{ov}(C) = \{\eta \mid t(\eta) \in \mathrm{Subterms}(\Psi)\} \setminus \{r\}$. We write $C \sim \{X\}\, r{:}\tau'\, \{Y\}$, when $X = \mathrm{iv}(C)$, $Y = \mathrm{ov}(C)$, and $\tau \sim \tau'$. The last relation, $\tau \sim \tau'$ is a structural congruence on type shapes lifted into function types, i.e., $\tau \sim \tau$ and $x{:}\tau_1 \to C_1 \sim x{:}\tau_2 \to C_2$ when $\tau_1 \sim \tau_2$ and $C_1 \sim C_2$, where we write $C_1 \sim C_2$ when $C_1 \sim \{X\}\, r{:}\tau\, \{Y\}$ and $C_2 \sim \{X\}\, r{:}\tau\, \{Y\}$.

Intuitively, the input variables of an expression e (strictly, e's type C) represent the objects that may be read, modified, deleted, or stored inside other objects by e; the output variables represent the subset of the input variables (excluding the returned value r) that are still accessible after e has been evaluated.

The first judgment $\Gamma \vdash v \sim \tau$ says that the value v (if it type checks in FINE) has a type that is in the same shape equivalence class as τ, where Γ maps names to their types. Likewise, the judgment $\Gamma \vdash e \sim C$ says that an expression (if it type checks in FINE) has a type that is in the same equivalence class as C. In some cases, we overload notation and write $\Gamma \vdash e \sim \{X\}\, r{:}\tau\, \{Y\}$, with the obvious meaning, i.e., the type of e is in the equivalence class described by $\{X\}\, r{:}\tau\, \{Y\}$. We also write $\eta{:}\tau, X$ to conditionally add a name to an input or output variable set—$\eta{:}\tau, X$ means η, X when τ is an object type, and X otherwise.

Computing type shapes: $\Gamma \vdash v \sim \tau$ and $\Gamma \vdash e \sim C$

$$\frac{}{\Gamma \vdash \eta \sim \Gamma(\eta)} \qquad \frac{}{\Gamma \vdash c \sim TC_c}$$

$$\frac{\Gamma, x{:}\tau \vdash e \sim C}{\Gamma \vdash \lambda x{:}\tau.e \sim x{:}\tau \to C} \qquad \frac{\Gamma \vdash v_1 \sim (x{:}\tau' \to C) \quad \Gamma \vdash v_2 \sim \tau'}{\Gamma \vdash v_1\ v_2 \sim C[{}^{v_2}/_{x}]}$$

$$\frac{\Gamma \vdash v \sim \tau}{\Gamma \vdash v \sim \{v{:}\tau\}\, r{:}\tau\, \{\}} \qquad \frac{X = \{\eta \mid s(\eta) \in \text{Subterms}(\Phi)\}}{\Gamma \vdash \textbf{assert}\ (s)\Phi\ \sim \{X\}\ \_ : \mathsf{unit}\ \{X\}}$$

$$\frac{\Gamma \vdash e \sim C' \quad C' \sim C}{\Gamma \vdash (e{:}C) \sim C} \qquad \frac{\tau' = \{\!|\overline{f{:}\tau} \mid \phi|\!\} \quad \Gamma \vdash \overline{v} \sim \overline{\tau}}{\Gamma \vdash (\textbf{new}_{\tau'}\ \{\!|\overline{f = v}|\!\}) \sim \{\overline{v{:}\tau}\}\, r{:}\tau'\, \{\}}$$

$$\frac{\Gamma \vdash v_1 \sim \{\!|\overline{f{:}\tau} \mid \phi|\!\} \quad \Gamma \vdash v_2 \sim \tau_k}{\Gamma \vdash (v_1.f_k :=: v_2) \sim \{v_1, v_2{:}\tau_k\}\, r{:}\tau_k\, \{v_1\}}$$

$$\frac{\Gamma \vdash v \sim \{\!|\overline{f{:}\tau} \mid \phi|\!\} \quad \Gamma, \overline{z{:}\tau} \vdash e \sim \{X\}\, r{:}\tau\, \{Y\}}{\Gamma \vdash \textbf{destruct}\ v\ \textbf{as}\ \{\!|\overline{f = z}|\!\}\ \textbf{in}\ e \sim \{X \setminus \{\overline{z}\} \cup v\}\, r{:}\tau\, \{Y \setminus \overline{z}\}}$$

$$\frac{\Gamma \vdash e_1 \sim \{X_1\}\, x{:}\tau_1\, \{Y_1\} \quad \Gamma, x{:}\tau_1 \vdash e_2 \sim \{X_2\}\, r{:}\tau_2\, \{Y_2\}}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \sim \{X_1 \cup (X_2 \setminus \{x\})\}\, r{:}\tau_2\, \{Y_2 \cup (Y_1 \setminus X_2) \setminus \{x\}\}}$$

The first four rules compute shapes for values and function application. An object value v, when used in an expression context, can be given a computation type that includes itself as an input variable, and no output variables. We give assert expressions **assert** $(s)\Phi$ a shape where the input and output variables are the same, namely those referenced by Φ, i.e., all the objects mentioned in an assertion are still accessible after the assertion has been evaluated. An expression e can be ascribed the type C, (using $e{:}C$) so long as the ascription does not change the shape of the type.

When computing type shapes for the imperative operations, we pay attention to the aliasing relationships induced by each operation. However, we do not attempt to check here that FX programs respect the aliasing constraints on the object graph necessary for assertion safety—all checking is left to FINE. With this principle in mind, consider the rule for **new**. It states that the input variables are the stateful field values $\overline{v}$, while the output variable set is empty. Since references to each of the $\overline{v}$ are captured by the newly allocated object, the empty output variable set indicates that the $\overline{v}$ should no longer be accessed directly by the program. The swap statement is similar. Its input variables include the object being updated v_1 and the new contents of the field v_2 (if its type τ_k is an object type); the output variables only include v_1 since the reference to v_2 has been captured by v_1. For a **destruct** expression, the input variables include v, the object being destructed, and e's input variables but not the $\overline{z}$ since these are locally bound; for the same reason, $\overline{z}$ are excluded from the output variables. Note that e (or some enclosing expression) may attempt to use v after it has been destructed—this situation will be trapped by the FINE type checker. Finally, we have **let**s. Here, the input variables are the union of the input variables of e_1 and e_2, excluding the let-bound variable x. The output variables are the outputs of e_2 (excluding x) and those outputs of e_1 that are not in the footprint of e_2.

4. Translating FX to FINE

As sketched in §2.3, FX programs are translated to functional FINE programs. We formalize this translation here, and prove that the translation of FX programs preserve their semantics. Since FINE programs can be verified using refinement type checking, we also obtain the result that the safety of FX programs can be established by the FINE type checker. Our description of the translation starts by describing the role of tokens and their types in our translation. We then describe the translation of types and finally the translation of terms. The section concludes with discussion of our main results.

4.1 FX objects, tokens, and ownership

For safety in the presence of mutation and aliasing, we require FX programs to respect an ownership discipline on the object store. Affine tokens play a central role in our translation, and serve primarily to enforce this ownership discipline. To illustrate this point, we first discuss how to represent FX object values in FINE. As we have seen, FX object values are locations ℓ, where a store σ maintains a mapping from locations to records holding the object's fields. In FINE, we represent object values as immutable records. The translation of objects is described by the judgment below.

$$\frac{\sigma(\ell) = \{\!|\overline{f = v}|\!\}^{\iota} \qquad \sigma \vdash v_i \downarrow \hat{v}_i\ \ \forall i}{\sigma \vdash \ell \downarrow \{f_1 = \hat{v}_1, \ldots, f_n = \hat{v}_n\}^{\ell}}$$

This rule looks up the binding for a location ℓ in σ, and produces a tuple of FINE values, where each $\hat{v}_i$, recursively, is the translation of the components of the object stored at ℓ.[1] FINE values that correspond to FX objects are tagged with the location ℓ from which the value was derived—we explain its significance shortly.

Now, given a store $\sigma = \ell_1 \mapsto \{\!|f = 0|\!\}^{\oplus}$, consider the simple FX program and its translation to FINE shown below (eliding irrelevant unit-values from the FINE versions for clarity).

FX (1) **let** x = **new**$_{\tau'}$ $\{\!|g = \ell_1|\!\}$ **in let** $y = \ell_1.f :=: 1$ **in** x

FINE (1) **let** $x, tok_x = \mathsf{mk}_{\tau'}\{f = 0\}^{\ell_1} tok_{\ell_1}$ **in**
let $y, tok_y = \mathsf{swap}_{\tau}\{f = 0\}^{\ell_1} 1\ tok_{\ell_1}$ **in** x, tok_x

The FX program allocates a new object x storing the object ℓ_1 inside it, then updates ℓ_1 and returns x. The program above violates ownership, since after a single step of execution, the store contains a new location $\ell_2 \mapsto \{\!|g = \ell_1|\!\}^{\oplus}$, meaning that a reference to ℓ_1 has been captured by ℓ_2, and the very next step mutates ℓ_1 directly.

The image of the FX program under the translation is the FINE program shown directly beneath it. The FINE program contains calls to the functions $\mathsf{mk}_{\tau'}$ and swap_{τ} (discussed in §2.3), passing in FINE values corresponding to the object ℓ_1 (i.e., $\{f = 0\}$), and, importantly, the variable tok_{ℓ_1} representing a token or capability to use the immutable record value in a stateful way. Tokens are, of course, affine and since tok_{ℓ_1} is used more than once, the FINE program fails to type check, and the FX program is dismissed as potentially unsafe.

Not all violations of ownership are as easily detected as in this first example. One problematic case is when a reference to an object is captured by a closure, and somewhere else in the program the captured object is updated via an alias, as in the example shown below (with the same initial store σ).

FX (2) **let** $g = \lambda \_{:}\mathsf{unit}.\ell_1$ **in let** $\_ = \ell_l.f :=: 1$ **in** g

FINE (2) **let** $g = \lambda \_{:}\mathsf{unit}.\lambda x{:}\mathtt{Token}\ \{f = 0\}^{\ell_1}.(\{f = 0\}^{\ell_1}, x)$ **in**
let $\_ = \mathsf{swap}_{\tau}\{f = 0\}^{\ell_1} 1\ tok_{\ell_1}$ **in** g

In this case, we translate the thunk g by adding an additional argument for the token of the captured object value $\{f = 0\}^{\ell_1}$. Notice that the type of the token variable ($\mathtt{Token}\ \{f = 0\}^{\ell_1}$) is indexed by the value for which it is a capability. Any operation that mutates ℓ_1 consumes its token, producing a new token with a type indexed by the new value. Thus, after an update the thunk g becomes unusable since a token with the appropriate type cannot be passed in. (Unless the update does not change the value of the object, in which case it is still perfectly safe to use g). Since this program never attempts to use g, it is safe in FX and is also type correct in FINE. In other words, in addition to tokens being affine, giving them value-indexed types is also crucial for safety.

[1] Throughout this section, syntactic elements from FINE are indicated using the hatted version of the corresponding elements in FX.

Token threading, for all its benefits (recall also the discussion of §2.3), also makes it harder to prove that our translation from FX to FINE is a simulation. Informally, we wish to show that when an FX program e_1 is translated to a well-typed FINE program $\hat{e}_1$, and if e_1 steps to e_2, then $\hat{e}_1$ steps (using one or more steps) to some $\hat{e}_2$, where $\hat{e}_2$ is semantically equivalent to the image of e_2 under the translation. If we inspect the example program FX (2) above, we see that after several steps it reduces to $\lambda\_{:}\mathsf{unit}.\ell_1$, in a store where $\ell_1 \mapsto \{\!|f=1|\!\}^{\oplus}$, which translates to $\lambda\_{:}\mathsf{unit}.\lambda x{:}\mathtt{Token}\,\{f=1\}^{\ell_1}.(\{f=1\}^{\ell_1},x)$. However, this FINE lambda term is syntactically distinct from the value of g in the program FINE (2) above—g contains a stale value for ℓ_1. In proving our simulation result we show that, despite the syntactic differences, these two values are in fact semantically equivalent. We use FINE's module system (formalized and proved sound previously using the colored brackets of Grossman et al. (2000)) to show that stale values that result from the reduction of FINE programs are semantically irrelevant since these values must always be treated abstractly. The ℓ-superscripts on values facilitate this proof. Our technical report includes the details.

4.2 Type translation

The translation of FX types τ and computation types C to FINE types $\hat{\tau}$ ($\hat{C}$ is a synonym for $\hat{\tau}$) is shown below. Throughout the translation, we use variables named tok_η for η's token. We also use $toks(X)$ to mean the set of tokens for variables in X, i.e., $toks(\eta,X) = tok_\eta, toks(X)$ with $toks(\cdot) = \cdot$.

Translation of types: $\Gamma \vdash \tau \rightsquigarrow \hat{\tau}$ and $\Gamma \vdash C \rightsquigarrow \hat{C}$

$$\frac{}{\Gamma \vdash TC \rightsquigarrow TC} \qquad \frac{\forall i.\Gamma \vdash \tau_i \rightsquigarrow \hat{\tau}_i \quad \Gamma \vdash \phi \rightsquigarrow \hat{\phi}}{\Gamma \vdash \{\!|\overline{f:\tau} \mid \phi|\!\} \rightsquigarrow \{self{:}\{\overline{f:\hat{\tau}}\} \mid \hat{\phi}\}}$$

$$\frac{\Gamma \vdash \tau \rightsquigarrow \hat{\tau} \quad \Gamma; x:\tau \vdash C \rightsquigarrow \hat{C} \quad C \sim \{X\}\, r{:}\tau\, \{Y\}}{\Gamma \vdash x{:}\tau \to C \rightsquigarrow x{:}\hat{\tau} \to (\{\mathtt{Token}\, x_j\}_{x_j \in X} * \{\_{:}unit \mid Pre(C)_\Gamma\}) \to \hat{C}}$$

$$\frac{C \sim \{X\}\, r{:}\tau\, \{Y\} \quad \Gamma \vdash \tau \rightsquigarrow \hat{\tau} \quad \forall y_j \in Y,\ \Gamma \vdash y_j : \tau_j \quad \Gamma \vdash \tau_j \rightsquigarrow \hat{\tau}_j}{\Gamma \vdash C \rightsquigarrow (r{:}\hat{\tau} * \{y'_j{:}\hat{\tau}_j\}_{y_j \in Y} * \{\mathtt{Token}\, y'_j\}_{y_j \in Y} * \{\_{:}unit \mid Post(C)_\Gamma\})}$$

where, for $C = \{(s)\Phi\}\, r{:}\tau\, \{(t)\Psi\}$

$Pre(C)_\Gamma = \hat{\Phi}$ if $\Gamma \vdash \Phi[\eta/s(\eta)] \rightsquigarrow \hat{\Phi}$

$Post(C)_\Gamma = \hat{\Psi}$ if $\Gamma \vdash \Psi[\eta/s(\eta)][\eta'/t(\eta)] \rightsquigarrow \hat{\Psi}$

Type constants TC remain the same. Object types in FX are translated to refined record types in FINE, with each field type translated to $\hat{\tau}_j$, and the invariant translated to $\hat{\phi}$ as described shortly. Function types $x{:}\tau \to C$ are translated to FINE function types that bind the same parameter x (with translated type $\hat{\tau}$), tokens for each input variable of C, and a unit value refined with the translated precondition $Pre(C)_\Gamma$. The function returns a value with the translated computation type $\hat{C}$. A computation type C is translated to a dependent tuple, consisting of the return value r (with the translated return type $\hat{\tau}$), the output variables y'_j and their tokens, and a unit value whose type is refined with the translated post-condition $Post(C)_\Gamma$. The output variables y_j are rebound to y'_j in the tuple type, meaning the values of y_j at the function exit point.

Translation of FX formulas is straightforward and is defined by the judgments $\Gamma \vdash \Phi \rightsquigarrow \hat{\Phi}$ and $\Gamma \vdash \phi \rightsquigarrow \hat{\phi}$. The rules in these judgments are congruences over the structure of formulas, where, in the base case, value-indexed predicates $P\bar{v}$ are translated to $P\hat{\bar{v}}$, for $\Gamma \vdash v_i \rightsquigarrow \hat{v}_i$, the value translation judgment of the next section. We use two macros $Pre(C)_\Gamma$ and $Post(C)_\Gamma$ for translating pre- and post-conditions of computations to refinement formulas in FINE. The refinement formula replaces the stateful terms in a formula, $s(\eta)$ and $t(\eta)$, with the corresponding name bindings for the immutable values in FINE that correspond to η.

Translation of signatures. Predicate declarations $P :: k$ are translated to type constructors in FINE with the same name P and the same kind k. Assumptions **assume** ϕ are translated to data constructors in FINE representing proofs of $\hat{\phi}$ if $\Gamma \vdash \phi \rightsquigarrow \hat{\phi}$. These data constructors are included in FINE's LCF-style proof kernel, along with the standard first-order logic inference rules (see §2.1).

4.3 Value and expression translation

This section describes the translation of FX expressions. An FX expression e is translated to a FINE terms $\hat{e}$, where the free variables of $\hat{e}$ correspond to the object locations ℓ in e. For translation of source code the translated term $\hat{e}$ is closed, since store locations do not occur in the source code of a program.

The value translation judgment $\Gamma \vdash v \rightsquigarrow \hat{v}$ states that an FX value v is translated to a FINE value $\hat{v}$. The translation of FX names η (variables or locations) is trivial—these are translated to FINE variables with the same names, which may be substituted away when the FINE expression is closed. The rule for translation of functions is shown below.

$$\frac{\Gamma, x{:}\tau_1; \bullet \vdash (e{:}C) \rightsquigarrow \hat{e} \qquad \Gamma \vdash \tau_1 \rightsquigarrow \hat{\tau}_1 \quad \hat{\tau}_2 = (\{\mathtt{Token}\ \eta\}_{\eta \in \mathrm{iv}(C)} * \{\_{:}unit \mid Pre(C)_\Gamma\})}{\Gamma \vdash \lambda x{:}\tau_1.(e{:}C) \rightsquigarrow \lambda x{:}\hat{\tau}_1.\lambda y{:}\hat{\tau}_2.\mathbf{let}\ \overline{tok_\eta}, \_ = y\ \mathbf{in}\ \hat{e}}$$

This rule follows the translation of function types. We add a parameter $y{:}\hat{\tau}_2$ to receive the tokens for the function's input variables. The function body $e{:}C$ is translated to $\hat{e}$, preceded by a **let** that unpacks the components of the tuple y, giving token variables their required names and types, $tok_\eta{:}\mathtt{Token}\ \eta$. Our formalization does not attempt to infer types, or pre- and post-conditions for functions. As such, the rule above expects both the formal parameter x to be decorated with a type and for the function body to be ascribed a computation type. In practice, we expect weakest precondition inference to alleviate some of this difficulty, although, of course, annotations will be expected at least on recursive functions. Additionally, while our simulation result holds for translations of arbitrary FX programs (that are well-typed in FINE), in practice we expect FX programs to be closure-converted prior to translation.

The judgment $\Gamma; \dot{Y} \vdash e \rightsquigarrow \hat{e}$ (below) translates an FX expression e to a FINE expression $\hat{e}$. The environment includes $\dot{Y}$, either empty ($\bullet$) or a set of names Y representing the output variables of a computation. We define an operation $\oslash$ on output sets: $\dot{Y} \oslash Y'$ returns Y' if $\dot{Y} = \bullet$, and $\dot{Y}$ otherwise. Intuitively, $\dot{Y}$ corresponds to locations that may have been modified in expressions preceding e in a block of **let**s. FINE values (and tokens) corresponding to locations in this set need to be "threaded out" in the translation.

Translation of expressions, $\Gamma; \dot{Y} \vdash e \rightsquigarrow \hat{e}$

$$\text{T-Return}\ \frac{\Gamma \vdash v \sim \tau \quad \Gamma \vdash v \rightsquigarrow \hat{v}}{\Gamma; Y \vdash v \rightsquigarrow (\hat{v}, Y, toks(v{:}\tau, Y), ())}$$

$$\text{T-Let}\ \frac{\Gamma \vdash e_1 \sim \{X_1\}\, x{:}\tau_1\, \{Y_1\} \qquad \Gamma; \bullet \vdash e_1 \rightsquigarrow \hat{e}_1 \quad \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \sim C \quad \Gamma, x{:}\tau_1; \dot{Y} \oslash \mathrm{ov}(C) \vdash e_2 \rightsquigarrow \hat{e}_2}{\Gamma; \dot{Y} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x, Y_1, tok_x, tok_{Y_1}, \_ = \hat{e}_1\ \mathbf{in}\ \hat{e}_2}$$

$$\text{T-Assert}\ \frac{\Gamma \vdash \Phi[\eta/s(\eta)] \rightsquigarrow \hat{\Phi} \qquad X = \mathrm{iv}_s(\Phi)}{\Gamma; \bullet \vdash \mathbf{assert}\ (s)\Phi \rightsquigarrow \mathrm{ASRT}_X(\hat{\Phi})}$$

$$\text{T-Chk}\ \frac{\begin{array}{c}\Gamma;\bullet \vdash e \rightsquigarrow \hat{e} \qquad C \sim \{X\}\, r{:}\tau\, \{Y\} \\ C = \{(s)\Phi\}\, r{:}\tau\, \{(t)\Psi\} \qquad X_1 = \mathrm{iv}_s(\Phi) \qquad X_2 = \mathrm{iv}_t(\Psi)\end{array}}{\begin{array}{l}\Gamma;\bullet \vdash (e{:}C) \rightsquigarrow \textbf{let}\ \_, X_1, tok_{X_1}, \_ = \mathrm{ASRT}_{X_1}(Pre(C)_\Gamma)\ \textbf{in} \\ \quad \textbf{let}\ r, Y, toks(r{:}\tau, Y), \_ = \hat{e}\ \textbf{in} \\ \quad \textbf{let}\ \_, X_2, tok_{X_2}, \_ = \mathrm{ASRT}_{X_2}(Post(C)_\Gamma)\ \textbf{in} \\ \quad (r, Y, toks(r{:}\tau, Y), ())\end{array}}$$

$$\text{T-App}\ \frac{\Gamma \vdash v_1\ v_2 \sim C \qquad \Gamma \vdash v_1 \rightsquigarrow \hat{v}_1 \qquad \Gamma \vdash v_2 \rightsquigarrow \hat{v}_2}{\Gamma;\bullet \vdash v_1\ v_2 \rightsquigarrow \hat{v}_1\ \hat{v}_2\ (toks(\mathrm{iv}(C)), ())}$$

$$\text{T-New}\ \frac{\Gamma \vdash (\textbf{new}_\tau\ \{\!|\overline{f = v}|\!\}) \sim C \qquad \forall i.\, \Gamma \vdash v_i \rightsquigarrow \hat{v}_i}{\Gamma;\bullet \vdash (\textbf{new}_\tau\ \{\!|\overline{f = v}|\!\}) \rightsquigarrow \mathsf{mk}_\tau\ \hat{v}_1\ \cdots\ \hat{v}_n\ (toks(\mathrm{iv}(C)), ())}$$

$$\text{T-Upd}\ \frac{\Gamma \vdash (v_1.f_k :=: v_2) \sim C \qquad \Gamma \vdash v_1 \rightsquigarrow \hat{v}_1 \qquad \Gamma \vdash v_2 \rightsquigarrow \hat{v}_2}{\Gamma;\bullet \vdash (v_1.f_k :=: v_2) \rightsquigarrow \mathsf{upd}_{f_k}(\hat{v}_1, \hat{v}_2, toks(\mathrm{iv}(C)), ())}$$

$$\text{T-Destr}\ \frac{\Gamma \vdash v \sim \{\!|\overline{f{:}\tau}|\!\} \qquad \Gamma \vdash v \rightsquigarrow \hat{v} \qquad \Gamma, z_i{:}\tau_i; \dot{Y} \vdash e \rightsquigarrow \hat{e}}{\begin{array}{l}\Gamma; \dot{Y} \vdash \textbf{destruct}\ v\ \textbf{as}\ \{\!|\overline{f = z}|\!\}\ \textbf{in}\ e \rightsquigarrow \\ \quad \textbf{let}\ \overline{z_i}, \overline{tok_{z_i}}, \_ = \mathsf{destr}_\tau\ \hat{v}\ tok_v\ \textbf{in}\ \hat{e}\end{array}}$$

To illustrate the significance of $\dot{Y}$, consider the FX expression **let** $y = (\ell.f :=: 1)$ **in** y. The type shape computed for this expression is $\{\ell\}\, r{:}\mathsf{int}\, \{\ell\}$, indicating that ℓ was modified in the expression and is still accessible after the expression has been evaluated. When translating the body of the **let** (i.e., y), we need to return y as well as the value and token corresponding to the updated value of ℓ. The rule (T-Return) does just this. It translates values v that appear in expression contexts to tuples containing the translated value $\hat{v}$, any additional values dictated by the output set Y (if any), tokens for the object values, and finally a unit for the post-condition.

The other aspect of the $\dot{Y}$ environment is illustrated in the translation of let expressions **let** $x = e_1$ **in** e_2. The first premise computes the shape of the let-bound expression e_1 for its set of output variables Y_1, and binds names and tokens for these in the translation. To translate the body e_2, we compute the shape C of the entire let expression for its set of outputs $Y = \mathrm{ov}(C)$—these are locations that may have been modified in either e_1 or e_2 and are still live, and so, must be threaded out when translating e_2. We do this by translating e_2 in a context $\dot{Y} \oslash Y$, which ensures that Y is threaded out of e_2, unless $\dot{Y}$ is non-empty, in which case the output variables of some enclosing let are threaded out of e_2.

The remaining rules are mostly straightforward. To translate assertions, we define the following macros: $\mathrm{iv}_s(\Phi) = \{\eta \mid s(\eta) \in \mathrm{Subterms}(\Phi)\}$ and $\mathrm{ASRT}_X(\hat{\Phi}) = ((), X, toks(X), (() : \{\_ : unit \mid \hat{\Phi}\}))$. Assertions reduce to unit in FX—the first unit value in the tuple corresponds to this. The last unit value with a refined type corresponds to a check of the assertion formula. Additionally, we include values and tokens for each object value $\ell \in X$. This ensures that assertions in FINE never refer to stale values, thereby keeping the behavior of FINE assertions in correspondence with FX. The ascription form $(e{:}C)$ is translated, as expected, using a pair of assertions. The tokens used in the assertion are rebound so that the capabilities they stand for are not consumed.

Applications $(v_1\ v_2)$ are translated to pass in tokens (computed using the type shape judgment) for the input variables of v_1, as well as a unit value for the precondition (T-App). The FINE type checker must prove that the unit value can be refined with the precondition of v_1. The translation of **new** uses the constructor mk_τ of the record type τ (T-New). (The function mkRev in §2.3 is an instance of such a constructor.) The constructor takes the initial values $\hat{v_1}, \ldots, \hat{v_n}$ for the fields, consumes the tokens for the input variables (again computed using type shapes), and returns the new record r and its token tok_r. The translation of swaps using (T-Upd) is similar. The translation of a destruct expression calls the destructor destr_τ of the record type τ (T-Destr). The destructor consumes the token tok_v for the record to destruct, and returns the field values and their tokens. These are let-bound and in scope for $\hat{e}$, the translated body.

Our main result is Theorem 1, which shows that the translation from FX to FINE preserves the semantics of the FX program, if the resulting FINE program is well-typed. More precisely, the translation is a weak simulation modulo strong bisimilarity. We do not give meaning to FX programs that translate to ill-typed FINE programs since ill-typed FINE programs are never executed.

From this result we derive Corollary 2, a progress result for FX that shows that FX programs that translate to well-typed FINE programs do not reach the **error** state. An additional result is that the computation types given to FX programs describe small footprints. Our technical report gives formalizations of all lemmas and proofs.

Setting up for the simulation. Using the rules of §4.3, an FX expression e produces $\hat{e}$, a FINE program that contains free variables corresponding to locations ℓ and tokens for locations tok_ℓ. Given a store σ that maps locations ℓ to values v^ℓ, free location variables ℓ are eliminated via substitution with the corresponding translated FINE values $\hat{v}^\ell$.

The semantics of FINE also makes it convenient to handle free token variables, since it permits affine values to be held in a memory, M, rather than inlined in the program. We translate the FX store σ to a FINE memory M, which maps free token variables tok_ℓ in $\hat{e}$ to FINE values $\texttt{MkToken}\ \hat{v}^\ell$. The reduction of FINE programs has the form $(M, \hat{e}) \xrightarrow{\text{fine}} (M', \hat{e}')$. Reads from M are destructive, and FINE's metatheory establishes that affine values are used at most once.

We present a simplified version of our simulation lemma below, and illustrate its structure graphically. To type check the FINE programs $\hat{e}_1$ and $\hat{e}_2$ in the statement below, we use an environment including a signature $\hat{S}_{base}$ for FX primitives (corresponding to the Prims module of §2.3), and affine capabilities for every token in M.

$$\begin{array}{ccccc} \text{FX} & & & & \text{FINE} \\ (e_1, \sigma_1) & & \rightsquigarrow & & (M_1, \hat{e}_1) \\ \downarrow & & & & \downarrow_+ \\ (e_2, \sigma_2) & \rightsquigarrow & (M_2, \hat{e}_2') & \cong & (M_2, \hat{e}_2) \end{array}$$

Theorem 1 (Fine programs simulate FX programs)**.** *If an* FX *runtime configuration* (e_1, σ_1) *translates to a well-typed* FINE *configuration* $(M_1, \hat{e}_1)$*, and* (e_1, σ_1) *steps to* (e_2, σ_2)*, then* (e_2, σ_2) *translates to a well-typed* FINE *configuration* $(M_2, \hat{e}_2')$*, and* $(M_1, \hat{e}_1)$ *takes multiple steps to a configuration* $(M_2, \hat{e}_2)$ *such that* $\hat{e}_2'$ *is strongly bisimilar to* $\hat{e}_2$*.*

Theorem 1 establishes that our translation relation is a simulation up to strong bisimilarity, between $\hat{e}_2$ and $\hat{e}_2'$ above. The relation $M_2 \vdash \hat{e}_2 \cong \hat{e}_2'$ states that $\hat{e}_2$ and $\hat{e}_2'$ reduce in "lock step", and is derived from a value abstraction theorem provided by FINE's module system. In addition to this core simulation result, the full version of Theorem 1 establishes several other key properties. Among these are 1. that FX programs that translate to well-typed FINE programs respect aliasing and ownership constraints on the object graph; and 2. that the types given to effectful expressions only mention locations that they access or modify.

Theorem 1 also yields an important corollary. Since the **error** state cannot be translated to FINE, FX programs that translate to well-typed FINE programs do not reach the **error** state.

Corollary 2 (Progress for FX)**.** *Given a closed* FX *program e. If* $\cdot;\bullet \vdash e \rightsquigarrow \hat{e}$*, where* $\hat{S}_{base}; \cdot; \cdot \vdash \hat{e} : \hat{\tau}$*, then for any reduction sequence* $(e, \cdot) \xrightarrow{fx}{}^{*} (e', \sigma')$*,* e' *is guaranteed to not be* ***error****.*

```
1  (* Fragment of a library implementing fractional permissions *)
2  private type trk α = {|v:α; p:rat|}
3  type Aliases :: (α::⋆ ⇒ α ⇒ α ⇒ ⋆)
4  assume A_refl: ∀x:trk α, y:trk α. x.v=y.v ⇔ Aliases x y
5  val split: x:trk α → {(s) ⊤} y:trk α
6                {(t) Aliases t(x) s(x) && Aliases t(x) t(y) && t(y).p=t(x).p=s(x).p/2 }
7  val join: x:trk α → y:trk α → {(s) Aliases s(x) s(y)} unit
8                                  {(t) Aliases t(x) s(x) && t(x).p=s(x).p+s(y).p}
9  (* Fragment of a safe wrapper for the Collections API *)
10 type collection α
11 type iterator α
12 type coll α = trk (collection α)
13 type istate = HasMore | Unknown
14 type iter α = trk {|i:iterator α; c:coll α; st:istate|}
15 val newColl: unit → {(s) ⊤} c:coll α {(t) t(c).p=1}
16 val add: c:coll α → y:trk α → {(s) s(c).p=1 && s(y).p > 0} unit
17                                {(t) t(c).p=s(c).p && Aliases t(c) s(c) && t(y).p=s(y).p/2 }
18 val iterator: c:coll α → {(s) s(c).p>0} i:iter α
19                              {(t) t(c).p=s(c).p/2 && Aliases t(c) s(c) &&
20                                t(i).p=1 && Aliases t(c) t(i).v.c && t(i).v.c.p = t(c).p }
21 val finalize: c:coll α → i:iter α → {(s) s(i).p=1 && Aliases c s(i).v.c} unit
22                          {(t) t(i).p=0 && Aliases t(c) s(c) && t(c).p = s(c).p + s(i).v.c.p}
23 val next: i:iter α → {(s) s(i).p=1 && s(i).v.st=HasMore} y:trk α
24             {(t) t(i).p=s(i).p && t(y).p > 0 && Aliases s(i) t(i) && t(i).v.i=s(i).v.i}
25 val hasNext: i:iter α → {(s) s(i).p>0 } b:bool
26             {(t) t(i).p=s(i).p && Aliases t(i) s(i) && (b=true ⇔ t(i).v.st=HasMore) }
27 (* A client of the Collections API *)
28 val client: c:coll α → {(s) s(c).p=1} unit {(t) t(c).p = s(c).p && t(c).v=s(c).v}
29 let client c =
30   let it1 = iterator c in
31   let rec loop1 (c:coll α) (it1:iter) :
32     {(s) Aliases s(it1).v.c s(c) && s(it1).p=1 && s(c).p> 0} unit
33     {(t) t(it1).p=s(it1).p && t(it1).v.c=s(it1).v.c && t(c).p=s(c).p} =
34     if hasNext it1 && ... then
35       let rec loop2 (c:coll α) (it2:iter) :
36         {(s) Aliases s(it2).v.c s(c) && s(it2).p=1} unit
37         {(t) t(c).p=s(c).p+s(it2).v.c.p} =
38         if hasNext it2 then let a = next it2 in ... loop2 c it2
39         else finalize c it2 in
40       let it2 = iterator c in loop2 c it2;
41       let b = next it1 in ... loop1 c it1
42     else () in
43   loop1 c it1; finalize c it1
```

Figure 2. Controlled aliasing using fractional permissions

5. Examples

This section illustrates the use of FX using further examples. We have applied the translation of §4 (manually, at present) to extended versions of these examples (and the ConfWeb example of §2), and verified the resulting programs using the FINE type checker.

5.1 Verifying stateful APIs in the presence of aliasing

In this section, we show how to verify two properties of clients of a stateful API of collections and iterators, even in the presence of aliasing. First, we ensure that the collection underlying an iterator is never modified while an iteration over the collection is in progress. Second, we ensure that a client never attempts to extract elements from an iterator that has been exhausted. Our example is adapted from the work of Bierhoff and Aldrich (2007), who develop a special-purpose type system that uses linear logic to check type-state properties in the presence of aliasing, and apply it to clients of the Java collections library. A variant of this example has also been studied by Krishnaswami et al. (2009), who verify a similar program using interactive proofs in a higher-order separation logic. In contrast, we enforce an aliasing discipline by developing a library of fractional permissions (Boyland 2003), and rely on an SMT solver for assertion checking.

We highlight three elements of our solution. First, the library-based approach to aliasing illustrates the flexibility of our approach.This library also illustrates the value of substructural state—affine tokens introduced by the translation ensure proper use of the library. Next, we leverage local state, a feature of FX. Permissions are mutable fields within structured, alias-controlled objects. A similar approach using monadic state would involve explicitly modeling a map from alias-controlled objects to their permissions. Finally, we show how the local state of permissions for collections and iterators can be easily combined, and for these permissions to be used with a state machine that tracks whether or not an iterator has been exhausted. The modularity afforded by our approach makes these combinations natural.

The program in Figure 2 contains three parts. First, we have the interface of our permissions library. Next, we show the interface of a wrapper to an underlying implementation of the .NET collections library. Although not shown, the wrapper is programmed in FX and contains calls to the permissions library to manage aliases. Finally, we show a client program that uses the wrapped collections library. Although we have yet to formalize this, we conjecture that for clients that verify against the wrapper, calls to the wrapper can be replaced with direct calls to the underlying library.

Permissions library. Lines 1-9 of Figure 2 show the interface of our fractional permissions library. At a high level, our library defines a type trk α that associates a permission p:rat (a rational number) with an α-typed value. Using split, a client can construct an alias y to a tracked object x, but the permission previously associated with x is split between x and y. The join function allows a client to destroy y, an alias of x, coalescing the permission previously associated with y into x's new permission. The library implementation (not shown) directly manipulates the private permission field, and uses A_refl to generate its post-condition Aliases. Two tracked values are Aliases if they refer to the same underlying object.

Wrapper of the collections library. Lines 10-27 show the interface of our wrapper of the .NET collection API. The abstract types collection α and iterator α correspond to the types implemented by the .NET library. We then define coll α, the tracked version of a collection α. The type iter α is the tracked version of an iterator and associates a permission with a record of three fields, and illustrates how local state in FX can be easily composed. The first field, i, is the iterator itself; the second field c:coll α holds a tracked reference to the collection from which the iterator was derived; the last field represents a state machine to track when an iterator is exhausted.

At line 16 we show the signature of newColl, a function that allocates a new collection. Its post-condition shows that the returned collection has a full permission. The add function (17-18) allows an object y to be added to the collection c. Since this mutates the underlying collection, the pre-condition of add requires c to hold a full permission. Since c captures a reference to y, the post-condition shows the permission of y halved.

Lines 19-20 shows show the signature of iterator, which produces an iterator i over a collection c. Calling iterator requires only a non-zero permission on c, but the post-condition shows that it consumes half of c's permission, since the iterator captures a reference to the collection from which it was derived. By halving the permission of c, we ensure that c cannot be modified while the iterator is still live. The implementation of iterator (not shown) calls split to stash a reference to c in the returned object. The finalize operation (22-23) allows an iterator i to be destroyed, so long as there are no other aliases to i. Its post-condition restores the permission to the collection from which the iterator was derived—the implementation of finalize calls to join.

The function next (24-25) allows a client to extract a tracked object from an iterator. The pre-condition of next requires the iterator to be in a state where it has more elements. In the post-

```
1  module InfoFlow
2  type level = High:level | Low:level | J:level → level → level
3  type LEQ :: level ⇒ level ⇒ ⋆
4  assume Lattice_assumptions: LEQ Low High, ...
5  (* The program counter sensitivity level, initially Low *)
6  private type pc = {|v:level|}
7  let pc=new pc{| v=Low|}
8  (* An abstract monad for leveled data *)
9  private type L :: ⋆⇒ level ⇒ ⋆= MkL : α→l:level → L α l
10 val return: l:level →α→ L α l
11 val bind: l:level →m:{m:level | LEQ l m} →pc1:level →x:L α l →
12           f:(α →{(s2) s2(pc).v=J l pc1 } L β m {(s2') s2'(pc)=s2(pc)}) →
13           {(s1) LEQ s1(pc).v pc1 } L β m {(s1') s1'(pc)=s1(pc)}
14 let bind l m pc1 (MkL x _) f = let tmp = pc.v in pc.v := J l pc1;
15                               let r = f x in pc.v := tmp; r
16 (* Channels on which to send data (side effects) *)
17 type Ch :: ⋆⇒ level ⇒ ⋆
18 val write: α→l:level →Ch α l →{(s1) LEQ s1(pc).v l} unit
19                              {(s2) s2(pc)=s1(pc)}
20 end
21 val client: L str Low →L str High →Ch str High →
22           {(s1) s1(pc).v=Low } L str High {(s2) s2(pc)=s1(pc)}
23 let client lx1 lx2 chan = bind Low High Low lx1 (fun x1 →
24     bind High High High lx2 (fun x2 →
25         let x = strcat x1 x2 in (write x High chan); return High x))
```

Figure 3. Tracking information flow through side-effects

condition, the state of the iterator is updated to indicate that it may or may not have more elements. In order to establish the pre-condition of next, clients can call and test the result of hasNext.

A client program. Lines 29-44 show a client function that takes a collection c with a full permission, extracts an iterator it1 (line 31), and iterates over it1 using loop1 and finalizes the iterator afterward (line 44), leaving c with a full permission. At each iteration of loop1, we extract another iterator it2 from c (line 41) and iterate over it in loop2 and finalize it2 before exiting loop2 (line 40). Concurrent modifications to the collection c (say, by calling add) would fail to type check, since c has less than a full permission.

Notice that aside from the annotation with loop invariants, the client function is fairly direct. Pleasingly, client contains no explicit calls to split or join. All these operations are factored into the implementation of the wrapped collection API, and the specifications of this API in effect manage the implicit aliasing behavior of the client program. Clients can also call split and join directly to explicitely manage aliases, should the need arise.

Translating to FINE**.** Translating Figure 2 to FINE is relatively straightforward. We first closure-convert the nested loops, hoisting them to the top-level. Thereafter, the translation follows the rules of §4 directly (which generalizes readily to handle type abstraction and application). Our implementation uses a more complex model for permissions that maintains separate fractions for read and write permissions; a larger API for collections, including functions to remove elements from collections and to query the size of a collection; and finally, a more complex client program. The resulting FINE program is 244 lines long, and its verification involves proving 29 goals, which are discharged in 15 seconds.

5.2 Tracking implicit information flows through impure code

Our second example develops an information-flow tracking library suitable for programs with side-effects. The basic idea is to combine a monadic treatment of dependency (as in DCC (Abadi et al. 1999)) with the program-counter technique used (originally) by Fenton (1973) in his Data Mark Machine. In short, we represent α-typed data protected at security level l as values of the type L α l, where the type L represents a family of monads arranged in a lattice according to the ordering on security levels. Additionally, we maintain a global stateful value pc that accounts for the influence of protected data on reaching the current program point. Effectful operations (such as writing a message to a channel) have preconditions (expressed as constraints on pc) to ensure that they are not control dependent on secret data.

We prove two properties of client programs: Low channels never carry data marked High-security; and Low channels are never used at High security-level program points. Although we have not proved noninterference for the program of Figure 3, a similar encoding was proved noninterferent by Swamy (2008).

Note that this program uses no local states, but a single global state pc. As such, this program could easily be modeled and verified using a monadic treatment of state. However, this example highlights two features of FX. First, it illustrates the ability of FX to handle programs that manipulate global state. Second, it serves as initial evidence that our approach generalizes to higher order programs. Our information flow library example uses a second-order function bind, where the type of its function argument is parameterized by the preceding arguments to the bind function. We discuss some further generalizations to higher order programs in §5.3.

The API. Figure 3 shows a module InfoFlow (lines 1-20) which defines the monad L α l and exports it abstractly to the client (21-25). InfoFlow begins with a definition of the security levels using the type level, which stands for a two-point lattice, ordered by the relation *LEQ*. The ordering is axiomatized by user-provided assumptions—we show one such assumption at line 4; the others are standard. At line 6, we define the type pc, a mutable object with a single field, v, which will hold the sensitivity of the program counter—we initialize pc to Low at line 7. We define the L α l type at line 9 and the type of function return at line 10, which allows any value to be injected into the monad using any level.

To appreciate the type of bind in Figure 3, it is instructive to consider a type of bind for flow controls in purely functional code: l:level →m:{m:level |*LEQ l m*} →x:L α l →f:(α →L β m) →L β m This version of bind allows f to view the protected data x:L α l at the unprotected type α, but the constraint *LEQ l m* ensures that the result f computes from x is at least as secret as x itself. However, this model is only sound if f is a pure function—nothing prevents f from writing its unprotected α-typed argument to a public channel.

The program-counter pc serves to prevent such leaks, as seen in two parts of InfoFlow. First, at lines 16-19 we define Ch α l, the type of a channel on which to send (or receive) α-typed data to parties privileged to view data at secrecy level l. To protect against leaks via implicit flows, the pre-condition of write states that the sensitivity of the program counter pc must not be greater than l. Next, consider the type of bind at line 11—we give the function f passed to bind the type $\alpha \rightarrow$ {*(s2) s2(pc).v=J l pc1*} L β m {*(s2') s2'(pc)=s2(pc)*}. This type allows f to view the protected data x:L α l at its underlying type α. But, the pre-condition on f's type says that the state of pc is elevated to be the join of pc1 (the value of pc at the time write is called) and l (the level on the secret data x). This ensures that f cannot satisfy the pre-condition of write if f calls write using a channel c:Ch α l2, for some l2 less secret than l.

A proof of noninterference for this scheme relies crucially on f treating pc values abstractly—we achieve this by marking the pc type private. This ensures, for example, that f cannot mutate the pc itself. The only mutation of pc occurs in the implementation of bind, which elevates the pc before calling f; then restores it before returning the value computed by f.

The client program concatenates Low and High strings, writes the result on a High channel, and returns a High string. The explicit level arguments to bind (e.g., the three occurrences of High on line 24) lead to some syntactic noise—this could easily be eliminated with some inference for implicit parameters.

Translating to FINE**.** As previously, translating Figure 3 to FINE begins with a closure-conversion step, turning the global variable pc into an argument of every stateful function. The resulting FINE program is 81 lines long, and produces 7 verification goals that are discharged automatically in 6 seconds.

5.3 Extension to higher-order programs

Although our token-threading translation works equally well for both first- and higher-order programs, we have yet to formalize an extension of the assertion language of FX to work with higher-order stateful code. However, this extension is fairly natural, since the kind language of FINE allows us to write types for higher-order programs. §2.2 shows such an example where the type of for_all abstracted over the predicate ***P*** decided by its argument f. However, in for_all, the argument f was required to be a pure function. Our technical report (Borgström et al. 2010) contains a detailed example showing how to extend this scheme to higher-order stateful code.

6. Related work, conclusions and future work

Our work is perhaps most closely related to the work of Charguéraud and Pottier (2008), who show how to functionalize imperative programs using a translation based on linear capabilities. Like us, they prove their translation sound via a simulation argument. However, their work does not include a program logic, although motivated by the desire to verify programs; our work includes the use of Hoare types and their translation to refinement types. Additionally, in the absence of a logic, Charguéraud and Pottier embed a specific aliasing discipline into their calculus, (based on the adoption and focus constructs of Fähndrich and DeLine (2002)). In contrast, because of the expressiveness of refinement types, we show how aliasing controls can be encoded using a library of fractional permissions.

Linear capabilities for accessing aliased objects are also treated in the linear core calculus L^3 (Ahmed et al. 2007), where a higher level surface language is left as future work. Our use of tokens in the translation of FX to FINE, is closely related to Walker et al.'s calculus of capabilities (Walker et al. 2000). Whereas Walker et al. prove a syntactic type soundness property for the capability calculus (which is sufficient for their domain of safe memory management), we prove the correctness of our token-based translation using a simulation argument. The ATS language (Zhu and Xi 2005) combines stateful views with indexed types for full verification, where stateful views are described using linear logic. This gives ATS the ability to reason directly about pointer manipulations, but at a price, since linear logic is hard to automate. FX is also related to HOOP (Flanagan et al. 2006), a language that uses dependent types to express refinements on imperative objects. However, unlike FX, refinements in HOOP can only mention immutable data.

Finally, our work is closely related to the work of Borgström et al. (2010), who use Hoare types for a state monad to verify stateful computations. Here a monolithic state is threaded through the entire program, making it difficult (without resorting to separation logic, as in Ynot), to reason locally about parts of the state or to mix stateful idioms. In contrast, FX programs use local state in the form of mutable objects, and, as illustrated by the example of §5.1, permits multiple stateful idioms to be combined in a modular way.

In summary, we have presented FX, a functional language with support for mutable objects. We show how FX programs can be translated to functional FINE programs, using affine types to model state and refinement type checking for verification. Future work includes an implementation and a source language type system. We also intend to work on applying FX to higher-order programs, building on a recent enhancement of FINE to support abstraction over predicates (Swamy et al. 2010b).

References

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.

A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. *OOPSLA*, 2007.

J. Borgström, J. Chen, and N. Swamy. Hoare-types for programming with affinity. Technical Report TR-2010-95, MSR, 2010.

J. Borgström, A. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. *J. Funct. Program.*, 2010. To appear.

J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72. Springer, 2003.

I. Cervesato and F. Pfenning. A linear logical framework. *Inf. Comput.*, 179 (1), 2002.

A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ICFP '08*, 2008.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI '10*. ACM, 2010.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.

J. Fenton. *Information Protection Systems*. PhD thesis, U. Cambridge, 1973.

C. Flanagan. Hybrid type checking. In *POPL*, 2006.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL/WOOD '06*, 2006.

D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000. ISSN 0164-0925.

D. Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2), 2002.

S. Krishnamurthi. The Continue server. In *PADL*, 2003.

N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI*, 2009.

G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.

K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in System F°. In *TLDI*, 2010.

R. Milner. LCF: A way of doing proofs with a machine. In *MFCS*, 1979.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.

M. Sozeau. Subset coercions in Coq. In *TYPES*. Springer-Verlag, 2006.

N. Swamy. *Language-based Enforcement of User-defined Security Policies*. PhD thesis, University of Maryland, College Park, August 2008.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*, 2010a.

N. Swamy, J. Chen, C. Fournet, K. Bharagavan, and J. Yang. Security programming with refinement types and mobile proofs. Technical Report MSR-TR-2010-149, Microsoft Research, 2010b.

W. Swierstra. A Hoare logic for the state monad. In *TPHOLs*, 2009.

P. Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM TOPLAS*, 22(4), 2000.

D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *PADL*, 2005.