# Defining and Enforcing Referential Security

Jed Liu    Andrew C. Myers

Department of Computer Science
Cornell University
Ithaca, New York, United States
liujed@cs.cornell.edu   andru@cs.cornell.edu

**Abstract.** Referential integrity, which guarantees that named resources can be accessed when referenced, is an important property for reliability and security. In distributed systems, however, the attempt to provide referential integrity can itself lead to security vulnerabilities that are not currently well understood. This paper identifies three kinds of *referential security* vulnerabilities related to the referential integrity of distributed, persistent information. Security conditions corresponding to the absence of these vulnerabilities are formalized. A language model is used to capture the key aspects of programming distributed systems with named, persistent resources in the presence of an adversary. The referential security of distributed systems is proved to be enforced by a new type system.

## 1 Introduction

To make programming manageable, distributed systems are increasingly being implemented using high-level languages and libraries that present distributed resources as language-level objects. This approach goes back to research platforms such as Argus [14], Emerald [4], and Network Objects [3], but is now applied widely in commercial programming using middleware platforms such as CORBA [19], in more recent object-relational mapping (ORM) systems such as Hibernate [10] and other Java Persistence API (JPA) [6] implementations, and in modern JavaScript ORM libraries [5].

Distributed systems naturally cross trust domains; it is often why they are distributed in the first place. Running a program on a federated platform composed of differently trusted distributed nodes creates security vulnerabilities that are not immediately apparent at the high level of abstraction at which the programmer is operating. Some of these vulnerabilities have been addressed by prior work; for example, the Fabric system [15] provides a high-level, Java-like abstraction for distributed programming, while using information-flow control to enforce both confidentiality and integrity properties.

In this paper, we identify three new security goals relating to the security of references that cross trust domains. Cross-domain references are a common feature not only of high-level distributed programming models, but of distributed systems in general. For example, web pages hyperlink to other pages, and relational-database tuples can contain foreign keys referring to other tuples. Regardless of the kind of system, security and reliability vulnerabilities are created when references cross trust boundaries, because they introduce dependencies between different parts of the system. This paper identifies some of these *referential vulnerabilities,* formally characterizes them, and explores a language-based approach to modeling, analyzing, and preventing them.

The first goal is *referential integrity.* A system has referential integrity if a reference can be relied upon to continue pointing to the same object. Referential integrity fails when that object is deleted while the reference still exists, resulting in a *dangling reference*, or when the reference points to a different object altogether.

Referential integrity appears in many guises. We use the term in a more general sense than in the database literature, where referential integrity is an important aspect of the relational model [7]. For example, the web lacks referential integrity: the referent of a hyperlink can be deleted, leading to the familiar "404" error. Referential integrity is also an important property for programming languages. In languages such as C that lack referential integrity, dangling pointers are a serious problem. In other languages, automatic garbage collection reclaims memory while preserving referential integrity.

While absolute referential integrity sounds ideal, it cannot be achieved in a federated system: referential integrity is necessarily limited by the trustworthiness of the node (or nodes) storing the referent object. Therefore, this paper generalizes referential integrity to systems where nodes are partially trusted.

Our second goal is *intentional persistence.* With referential integrity, a reference to an object is a promise that the object will not move or disappear: it must be persistent. Therefore, reachability implies persistence, as in various object-oriented databases (e.g., [1, 17]) and in marshaling mechanisms such as Java serialization. However, if all reachable objects are persistent, objects can become *accidentally persistent* because they are unexpectedly reachable. This can inflate resource consumption, leading to poor performance and system failure. This problem is familiar to those who have used Java serialization. Intentional persistence entails the absence of accidental persistence.

The third goal of this paper is immunity against *storage attacks*. Referential integrity prevents discarding reachable objects. But this gives an adversary a means to mount a denial-of-service attack. The adversary creates references to objects intended to be discarded, preventing reclamation and perhaps exhausting available storage space.

This paper formalizes these three goals as *referential security properties*, corresponding to the absence of referential vulnerabilities. This is done in the context of a simple programming language that captures the key elements of distributed programming in a federated system with persistent information and pointers. A novel type system is defined and is proved to enforce these security properties. Details of these proofs are found in an accompanying technical report [16].

The rest of this paper is structured as follows. Section 2 describes the language model. Section 3 presents security policies for reasoning about the three vulnerabilities. Section 4 introduces the programming language $\lambda_{persist}$, which abstractly describes distributed programming with persistence and distrust. The language is defined formally in Sections 5 and 6. Section 7 defines the adversary model. Section 8 formalizes the desired security conditions, and sketches the proofs that the type system of $\lambda_{persist}$ soundly enforces them. Related work is discussed in Section 9, and Section 10 concludes.

## 2 Language model

### 2.1 Modeling distributed computing as a language

We model distributed computing using a core programming language that we call $\lambda_{persist}$. In $\lambda_{persist}$, persistence, distribution, and communication are implicit but are constrained

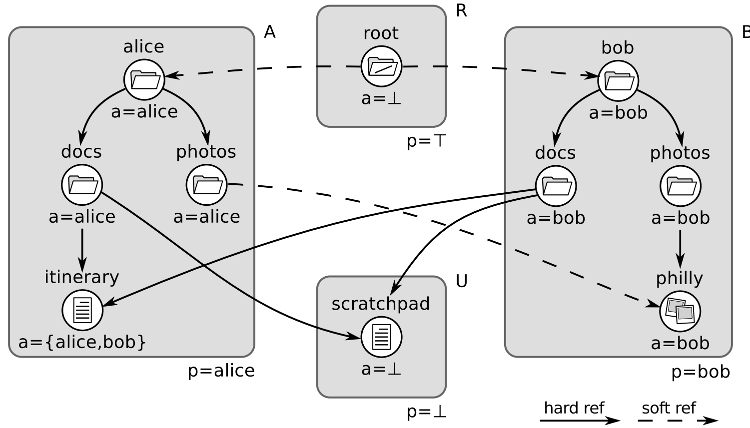


Fig. 1. Directory example

by policy annotations. Programs in $\lambda_{persist}$ are assumed to be mapped onto distributed host nodes in some way that agrees with these annotations. This mapping could be done manually by the programmer, or automatically by a compiler, à la Jif/split [23].

This implicit translation to a distributed implementation means that some apparently ordinary source-level operations may be implemented using distributed communication and computation. For example, function application may be implemented as a remote procedure call. Similarly, following references at the language level may involve communication between nodes to fetch referenced objects.

Although the concrete mapping from source-level constructs onto host nodes is left implicit, we can nevertheless faithfully evaluate the security of source-level computations. The key is to ensure that the system is secure under *any* possible concrete mapping that is consistent with the policy annotations in the source program. That is, any given computation or information might be located on any host that satisfies the source-level security constraints. A technical contribution of this paper is to develop an effective system of such source-level constraints, expressed as a type system.

Although we refer to $\lambda_{persist}$ as a source language, little attempt is made to make it congenial to actual programming. In particular, the type annotations introduced would be onerous in practice. They could be inferred automatically using standard constraint-solving techniques for inequations over $\mathcal{L}$, but we leave this to future work. One can view the type system as describing a program (or system) analysis, and the formal results of this paper as a demonstration that this analysis achieves its security goals.

### 2.2 Objects and references

Persistent objects are modeled in $\lambda_{persist}$ as records with mutable fields. The fields of an object can point to other objects through references. References contain the names of these mutable objects. References are not assignable as in ML [18]; imperative updates are achieved by assigning to mutable fields.

The language has two types of references: hard and soft. A *hard reference* is one with referential integrity: a promise that the referenced object will not be destroyed if its host is trustworthy. A *soft reference* does not create an obligation to maintain

the referenced object. Hard links in Unix and references in Java are examples of hard references. URLs, Unix symbolic links, and Java SoftReference objects are examples of soft references. The language models a garbage collector that may destroy objects reachable only via soft references. When following a soft reference or an untrusted hard reference, a program must be prepared to handle a failure in case the referenced object no longer exists.

This simple data model can represent many different kinds of systems, such as distributed objects, databases, and the web. The shared directory structure shown in Figure 1 serves as a running example. Alice and Bob are traveling together and are using the system to share photos and itineraries. The root directory is kept on a host R. Alice and Bob keep their directory objects on their own hosts, A and B, respectively. To share sightseeing ideas, they use a common scratchpad stored on host U. Solid arrows in the figure represent hard references, and dashed arrows are soft references. The a and p annotations are policies, which we now explain.

## 3 Policies for persistent programming

### 3.1 Persistence policies

In a federated system, referential integrity cannot be absolute, because the referenced object may be located on an untrusted, perhaps maliciously controlled, host machine. Therefore, referential integrity must be constrained by the degree of trust in the referenced host. This constraint is expressed by assigning each object a *persistence policy* describing how much it can be trusted to remain in existence.

The precise form of the persistence policy is left abstract in this paper. Persistence policies $p$ are assumed to be drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq, \bot, \top)$ of *policy levels*. If $p_1 \preccurlyeq p_2$ for two persistence policies $p_1$ and $p_2$, then $p_2$ describes objects that are at least as persistent as those described by $p_1$.

Persistence policies have a simple, concrete interpretation. Absent replication, objects are located only on host nodes that are trusted to enforce their persistence policies, so a persistence policy $p$ corresponds to a set of sufficiently trusted host nodes $H_p$. Therefore, if $p_1 \preccurlyeq p_2$, then $p_2$ must be enforceable by a smaller set of hosts: $H_{p_1} \supseteq H_{p_2}$. In fact, it is reasonable to think of a policy $p$ as simply a set of hosts.

In Figure 1, the root directory has persistence policy $\top$, which only host R is trusted to enforce. Alice has a user directory and a persistence policy alice. While R is trusted to enforce this policy, she has chosen to use her own host A. Similarly, Bob's directory is on host B. The shared scratchpad is kept on an untrusted host U, which can only enforce the persistence policy $\bot$.

Persistence policies are integrated into the type system of $\lambda_{persist}$. The type of an object reference includes a lower bound on the persistence policy of its referent; the type system ensures that the persistence of an object is always at least as high as that of any reference pointing to it. Programs can therefore use the persistence of a reference to determine whether the reference can be trusted to be intact. This rule enables sound reasoning about persistence and referential integrity as the graph of objects is traversed.

For example, in Figure 1, while Alice and Bob both have a hard reference to the scratchpad, they must be prepared for a persistence failure when using the references. The type system of $\lambda_{persist}$ will ensure their code handles such a failure. Any reference

to the scratchpad must have a type with $\perp$ persistence, because it can be no higher than the $\perp$ persistence of the scratchpad itself.

Whether a hard reference can be trusted to be intact depends on context. In Figure 1, Alice and Bob both have a hard reference to the itinerary. Because Alice trusts her own persistence level, if either reference is typed with alice persistence, then she can use it without worrying about a persistence failure. However, unless Bob trusts Alice, he would need to be prepared for such a failure when using the references.

Soft references also have types with persistence levels, and hence might be trusted. Trusted soft references can be promoted to trusted hard references. Therefore, soft references are distinct from untrusted hard references.

In $\lambda_{persist}$, persistence is defined not by reachability, but by policy. This resolves by fiat one of the three problems identified earlier: accidental persistence. Accidents are avoided by allowing programmers to express their intention explicitly. An object that is not intended to be persistent is prevented from being treated as a persistent object.

### 3.2 Characterizing the adversary

Security involves an adversary, and is always predicated on assumptions about the power of the adversary. In the kind of decentralized, federated system under consideration, the adversary is assumed to control some of the nodes in the system.

Different participants in a distributed system may have their own viewpoints about who the adversary is, yet all participants need security assurance. Therefore, a given adversary is modeled as a point $\alpha$ in the lattice of persistence policy levels. In the host-set interpretation of persistence policies, $\alpha$ defines the set of trusted hosts that the adversary does not control. The adversary is assumed to have the power to delete (i.e., violate the persistence of) an object if its persistence is not $\alpha$ or higher (i.e., $\alpha \not\preccurlyeq p$), because the object might be stored at a host node controlled by the adversary. Other actions by the adversary are modeled by special evaluation rules (see Section 7).

The formal results for the security properties enforced by $\lambda_{persist}$ treat the adversary as an arbitrary parameter. Therefore, these properties hold for any adversary.

### 3.3 Storage attacks and authority policies

We introduce the idea of *storage attacks*, in which a malicious adversary tries to prevent reclamation of object storage by exploiting the enforcement of referential integrity. For example, in Figure 1, Bob has shared with Alice an album containing the photos he has so far taken during their trip. Bob does not consider the album to be private, so others may create references to his album, as Alice has done. However, an adversary that creates a hard reference to this album can prevent Bob from reclaiming its storage.

To prevent such storage attacks, we ensure that hard references can be created only in sufficiently trusted code. We introduce *creation authority* to abstractly define this power to create new references. This is the only action requiring some form of authority in this paper, so for brevity, we refer to creation authority simply as *authority*.

Like persistence policies, authority policies $a$ are assumed to be drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of policy levels. Without loss of expressive power, they are assumed to be drawn from the same lattice as persistence policies. Authority prevents storage attacks because hard references can only be created to objects whose authority policy $a$ is less than or equal to the authority $a_p$ of the process; that is, $a \preccurlyeq a_p$.

A hard reference is a reference that should have referential integrity, so creating hard references requires authority. The adversary is assumed to have some ability to create hard references, described by its authority level $\alpha$. Soft references do not keep an object alive, so no creation authority is required to create a soft reference.

In Figure 1, the root directory has the authority policy $\bot$, so anyone can create a hard reference to it. Bob's philly album is large, so he has given it the authority policy bob; only he can create hard references that prevent the album from being deleted. Therefore, Alice's reference to the album must be soft. Alice has drafted an itinerary, giving it the authority policy $\{\mathsf{alice}, \mathsf{bob}\}$ to indicate she will persist the document for as long as Bob requires. Bob's reference to the itinerary, therefore, can be hard.

It may sound odd to posit control over creation of references. But a reference with referential integrity is a contract between the referrer and the referent. For example, the node containing the referent is obligated to notify the referrer if the object moves. Entering into a contract requires agreement by both parties, so it is reasonable for the node containing the referent to refuse the creation of a reference.

### 3.4 Integrity

Thus far, the powers of the adversary include creating references to low-authority objects and destroying objects with low persistence. Because the adversary may control some nodes, the adversary can also change the state of objects located at these nodes. This may in turn affect code running on nodes not controlled by the adversary, if the adversary supplies inputs to that code, or if it affects the decision to run that code.

Integrity policies describe limitations on these effects of the adversary. Integrity policies $w$ are drawn from a bounded lattice $(\mathcal{L}, \preccurlyeq)$ of policy levels; without loss of expressive power, it is assumed to be the same lattice as for persistence and authority policies. In fact, we can think of the persistence and authority levels of an object as the integrity of other, implicit attributes of the object. For persistence, this implicit attribute is the existence of the object itself. For authority, the attribute is the set of incoming references to the object. This unifying view of different policies as different aspects of integrity explains why all three kinds of policies can come from the same lattice.

The ordering $\preccurlyeq$ corresponds to increasing integrity. If $w_1 \preccurlyeq w_2$, an information flow from level $w_2$ to $w_1$ would be secure: more-trusted information would be affecting less-trusted information.[1] In $\lambda_{persist}$, each variable and each field of an object has an associated integrity level describing how trusted it is, and hence how powerful an adversary must be to damage it. The integrity of a reference is the integrity of the field or variable it was read from.

Figure 2 summarizes the interpretation of the three kinds of policies.

### 3.5 Integrity of dereferences and garbage collection

An adversary can directly affect the result of a dereference in two ways. First, if the reference has low integrity, the adversary can alter it to point to a different object. Second, if the referent has low persistence, the adversary can delete it. Therefore, the integrity of any dereference can be no higher than the integrity and persistence annotations on

[1] This ordering is the opposite of the "upside-down" ordering typically seen in work on information-flow security [2].

|  | Integrity | Authority | Persistence | Set of hosts |
|---|---|---|---|---|
| ⊤ "High" | Trusted, Untainted: No one can affect data | "superuser": No one can make a hard reference | Persistent: No one can delete object | No host nodes |
| ⊥ "Low" | Untrusted, Tainted: Anyone can affect data | "anyone": Anyone can make a hard reference | Transient: Anyone can delete object | All host nodes |

Fig. 2. Interpretations of the extremal policy labels

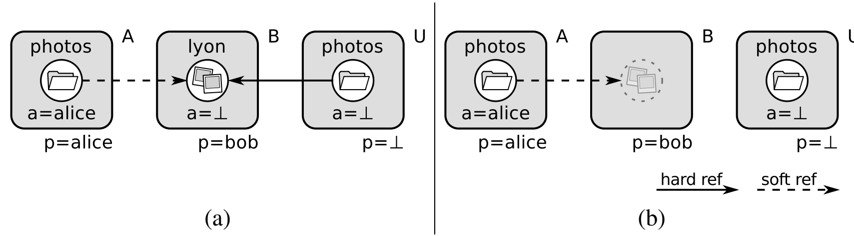


Fig. 3. Authority affects integrity of dereferences. Alice is following her soft reference to the lyon album. An adversary can affect the outcome of the dereference, because the album has low authority. (a) The untrusted host U has a hard reference preventing lyon from being garbage collected; Alice's dereference succeeds. (b) Host U has removed its hard reference, allowing lyon to be garbage collected; Alice's dereference fails.

the reference. In Figure 1, if Alice follows the reference from her docs directory to the scratchpad, she obtains an untrusted result; the untrusted host U influences the result by choosing whether to delete the scratchpad object.

More subtly, the adversary can manipulate hard references to influence the garbage collector, and thereby *indirectly* affect the result of a dereference. For example, in Figure 3a, Alice is following her soft reference to Bob's lyon album. Bob has marked lyon as only requiring low authority, allowing the untrusted, adversarial host U to create a hard reference, and thereby preventing lyon from being garbage-collected. Therefore, Alice's dereference must succeed.

However, in Figure 3b, the adversary U has removed its reference. Subsequently, lyon has been garbage-collected, and Alice's dereference fails. The adversary has indirectly affected the outcome of the dereference. To account for this, the integrity of Alice's dereference must be no higher than the authority required by lyon.

## 4 Types for persistent programming

To formalize the ideas presented in the previous section, we introduce the $\lambda_{persist}$ language, an extension to the simply typed lambda calculus. Figure 4 gives part of the formal syntax of $\lambda_{persist}$. Its type system prevents referential vulnerabilities by integrating policies for persistence, authority, and integrity into types. Accidental persistence is prevented because persistence is determined by policies expressing the programmer's intent, rather than by reachability. Referential integrity is maintained by a $\lambda_{persist}$ program with respect to a particular adversary if following hard references whose persis-

| | | |
|---|---|---|
| Variables | $x, y \in \mathsf{Var}$ | Policy levels $\quad w, a, p, \ell \in \mathcal{L}$ |
| Memory locations | $m \in \mathsf{Mem}$ | PC labels $\quad pc ::= w$ |
| Labeled record types | $S ::= \{\overrightarrow{x_i : \tau_i}\}_s$ | Storage labels $\quad s ::= (a, p)$ |
| Labeled ref types | $R ::= \{\overrightarrow{x_i : \tau_i}\}_r$ | Reference labels $\quad r ::= (a^+, a^-, p)$ |
| Base types | $b ::= \mathsf{bool} \mid \tau_1 \xrightarrow{pc} \tau_2 \mid R \mid \mathsf{soft}\ R$ | Types $\quad \tau ::= b_w \mid \mathbf{1}$ |
| Values | $v, u ::= x \mid \mathsf{true} \mid \mathsf{false} \mid * \mid m^S \mid \mathsf{soft}\ m^S \mid \lambda(x:\tau)[pc].e\ \ (\mid \bot_p)$ | |
| Terms | $e ::= v \mid v_1\ v_2 \mid \mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \{\overrightarrow{x_i = v_i}\}^S \mid v.x$ | |
| | $\mid v_1.x := v_2 \mid \mathsf{soft}\ e \mid e_1 \parallel e_2 \mid \mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ | |

Fig. 4. Syntax of $\lambda^0_{persist}$. Parenthesized productions only appear at run time.

tence and integrity are above the level of the adversary never leads to an object that has been destroyed by the adversary or garbage-collected. Storage attacks are prevented if the adversary is unable to change the set of high-authority objects that are reachable through hard references.

### 4.1 Labels

We assume a bounded lattice $(\mathcal{L}, \preccurlyeq, \bot, \top)$ of *policy levels*, from which integrity ($w$), authority ($a$), and persistence policies ($p$) are drawn.

Objects and reference values are annotated with *storage labels* consisting of a creation authority policy and a persistence policy. All non-unit types $\tau$ consist of a base type $b$ along with an integrity policy annotation $w$; fields and variables thereby acquire integrity policies, because they are part of their types. Objects do not have their own integrity labels because all of their state is in their fields, which do have labels.

The program-counter label $pc$ [9] is an integrity level indicating the degree to which the program's control flow has been tainted by untrusted data. This label restricts the side effects of code.

### 4.2 Example

Suppose we want to create a hierarchical, distributed directory structure, such as in Figure 1. Each directory maps names to either strings, representing ordinary files, or to other directories, and contains a reference to its parent directory (elided in the figure). To faithfully model ordinary filesystems, directories higher in the hierarchy should be more persistent: if they are destroyed, so is everything below.

A fully general directory structure would require augmenting $\lambda_{persist}$ with recursive and dependent types; for simplicity, these features have been omitted from $\lambda_{persist}$ because they do not appear to add interesting issues. However, we can capture the security of a general directory structure by using $\lambda_{persist}$ records to build a fixed-depth directory structure with a fixed set of entry names for each directory.

### 4.3 Modeling objects and references

The security policies of $\lambda_{persist}$ are about objects and references to them. Therefore, $\lambda_{persist}$ extends the lambda calculus with records that represent the content of objects. The record $\{\overrightarrow{x_i = v_i}\}$ comprises a set of fields $\overrightarrow{x_i}$ with corresponding values $\overrightarrow{v_i}$. Records are not values in the language; instead, they are accessed via references $m^S$, where $m$ is

the identity of the object and $S = \{\overrightarrow{x_i : \tau_i}\}_s$ gives its base record type. The *storage label* $s$ is a pair $(a, p)$. The *authority label* $a$ is an upper bound on the authority required to create a new reference to the referent object.

References to objects have labeled reference types $\{\overrightarrow{x_i : \tau_i}\}_r$. A reference label $r$ is a triple $(a^+, a^-, p)$ that gives upper and lower bounds on the authority required by the referent, and a lower bound on the persistence of the referent. The *upper authority label* $a^+$ restricts reference copying to prevent storage attacks. The *lower authority label* $a^-$ prevents the adversary from exploiting garbage collection to damage integrity (Section 3.5), by tainting the integrity of dereferencing soft references.

### 4.4 Modeling distributed systems

The goal of the $\lambda_{persist}$ language is to model a distributed system in which code is running at different host nodes. A single program written in $\lambda_{persist}$ is intended to represent such a system. The key to modeling distributed, federated computation faithfully is that different parts of the program can be annotated with different integrity labels, representing the trust that has been placed in that part of the code. To model a set of computations (subprograms $\overrightarrow{e_i}$) executing at different nodes, the individual computations are composed in parallel ($e_1 \parallel \cdots \parallel e_n$) into a single $\lambda_{persist}$ program.

From the viewpoint of a given principal in the system, code with a low integrity label, relative to that principal, can be replaced by any code at all. For the purposes of evaluating the security of the system, this code is in effect erased and replaced by the adversary. Therefore the single-program representation faithfully models a distributed system containing an adversary.

## 5 Accidental persistence and storage attacks

We present $\lambda_{persist}$ in two phases. In this section, we present $\lambda^0_{persist}$, a simplified subset of $\lambda_{persist}$ that prevents accidental persistence and storage attacks.

### 5.1 Syntax of $\lambda^0_{persist}$

Figure 4 gives the syntax of $\lambda^0_{persist}$. The names $x$ and $y$ range over variable names $\mathsf{Var}$; $m$ ranges over a space of memory addresses $\mathsf{Mem}$; $w$, $a$, $p$, and $\ell$ range over the lattice $\mathcal{L}$ of policy levels; and $s$ and $r$ range over the space of storage labels $\mathcal{L}^2$ and reference labels $\mathcal{L}^3$, respectively.

Types in $\lambda^0_{persist}$ consist of base types with an integrity label ($b_w$), and the unit type $\mathbf{1}$. Base types include booleans, functions, and two kinds of references to mutable records: hard ($R$) and soft ($\mathsf{soft}\ R$). The metavariable $R$ denotes a labeled reference type.

The type $\tau_1 \xrightarrow{pc} \tau_2$ is a function type with a $pc$ annotation that is a lower bound on the $pc$ label of the caller. It gives an upper bound on the authority level of references the function creates and on the authority level of references held in the closure environment.

Values include variables $x$, booleans $\mathsf{true}$ and $\mathsf{false}$, the unit value $*$, record-typed memory locations (references) $m^S$, soft references $\mathsf{soft}\ m^S$, and functions $\lambda(x{:}\tau)[pc].e$. The $pc$ component of a function $\lambda(x{:}\tau)[pc].e$ has the same meaning as that in function types. At run time, $p$-persistence failures $\perp_p$ can also appear as values.

Most terms are standard. The unusual features are record constructors $\{\overrightarrow{x_i = v_i}\}^S$, soft references soft $e$, parallel composition $e_1 \parallel e_2$, and soft-reference tests exists $v$ as $x$ : $e_1$ else $e_2$.

### 5.2 Example

Returning to the directory example in Figure 1, Bob can add to the itinerary with the code below. It starts at the root of the directory structure, traverses down to the itinerary, and invokes an add method to add a museum.

```
let home = root.bob
in exists home as bob:
     let docs = bob.docs
     in docs.itinerary.add "Rodin Museum"
   else: ...
```

The garbage collector may have snapped the soft reference home to Bob's home directory, so exists is used to determine whether the reference is still valid. If so, the body of the exists is evaluated with bob bound to a hard reference to the home directory.[2] (This reference can be created because the *pc* label at this point has sufficient creation authority.) The second select expression, bob.docs, dereferences the hard reference.

### 5.3 Operational semantics of $\lambda^0_{persist}$

Figure 5 gives the small-step operational semantics of $\lambda^0_{persist}$, omitting standard rules. The notation $e\{v/x\}$ denotes capture-avoiding substitution of value $v$ for variable $x$ in expression $e$. A failed or garbage-collected memory location contains value $\bot$. Most of the operational semantics rules are straightforward, but a few deserve more explanation.

Let $M$ represent a memory: a finite partial map from typed memory locations $m^S$ to closed record values. Let $\langle e, M\rangle$ be a system configuration. A small evaluation step is a transition from $\langle e, M\rangle$ to another configuration $\langle e', M'\rangle$, written $\langle e, M\rangle \to \langle e', M'\rangle$.

Let $\mathsf{locs}(e)$ represent the set of locations appearing explicitly in $e$. A memory $M$ is well-formed only if every address $m$ appears at most once in $\mathsf{dom}(M)$, and for any location $m^S$ in $\mathsf{dom}(M)$, $\mathsf{locs}(M(m^S)) \subseteq \mathsf{dom}(M)$. A configuration $\langle e, M\rangle$ is well-formed only if $M$ is well-formed, $\mathsf{locs}(e) \subseteq \mathsf{dom}(M)$, and $e$ has no free variables.

Though the operational semantics refer to complete record types, only their persistence labels are needed at run time. These labels are only used to determine the level of persistence failure that occurs when dereferencing a dangling reference (rules DANGLE-SELECT and DANGLE-ASSIGN), so run-time overhead should be small.

The record constructor $\{\overrightarrow{x_i = v_i}\}^S$ (rule CREATE) creates a new memory location $m^S$ to hold the record. The component $S$ specifies the base type and storage label of the record. The storage label governs at what nodes the object can be created. The function $\mathsf{newloc}(M)$ deterministically generates a fresh memory location.

The field-selection expression $v.x$ (rules SELECT and DANGLE-SELECT) evaluates $v$ to a memory location $m^S$. If the location has not failed, the result of the selection is

[2] To avoid a race with the garbage collector, an implementation of exists should first optimistically create the hard reference, then check its validity before exposing it to the program.

[LET] $$\frac{\forall p.\ v \neq \bot_p}{\langle \mathsf{let}\ x = v\ \mathsf{in}\ e, M\rangle \xrightarrow{\mathsf{e}} \langle e\{v/x\}, M\rangle}$$

[CREATE] $$\frac{m = \mathsf{newloc}(M)}{\left\langle \{\overrightarrow{x_i = v_i}\}^S, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle m^S, M[m^S \mapsto \{\overrightarrow{x_i = v_i}\}]\right\rangle}$$

[PARALLEL-RESULT] $$\langle v_1 \parallel v_2, M\rangle \xrightarrow{\mathsf{e}} \langle *, M\rangle$$

[SELECT] $$\frac{M(m^S) = \{\overrightarrow{x_i = v_i}\}}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle v_c, M\rangle}$$

[ASSIGN] $$\frac{M(m^S) \neq \bot \qquad \forall p.\ v \neq \bot_p}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle *, M[m^S.x_c \mapsto v]\right\rangle}$$

[DANGLE-SELECT] $$\frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle}$$

[DANGLE-ASSIGN] $$\frac{M(m^S) = \bot \qquad p = \mathsf{persist}(m^S)}{\left\langle m^S.x_c := v, M\right\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle}$$

[EXISTS-TRUE] $$\frac{M(m^S) \neq \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \left\langle e_1\{m^S/x\}, M\right\rangle}$$

[EXISTS-FALSE] $$\frac{M(m^S) = \bot}{\left\langle \mathsf{exists\ soft}\ m^S\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2, M\right\rangle \xrightarrow{\mathsf{e}} \langle e_2, M\rangle}$$

Evaluation contexts
$$E ::= \mathsf{soft}\ [\cdot] \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e \mid [\cdot] \parallel e \mid e \parallel [\cdot]$$

[FAIL-PROP] $$\langle F[\bot_p], M\rangle \xrightarrow{\mathsf{e}} \langle \bot_p, M\rangle \qquad F ::= \mathsf{soft}\ [\cdot] \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e$$

[PROG-STEP] $$\frac{\langle e, M\rangle \xrightarrow{\mathsf{e}} \langle e', M'\rangle}{\langle e, M\rangle \to \langle e', M'\rangle}$$

[GC] $$\frac{\mathsf{gc}(G, \langle e, M\rangle)}{\langle e, M\rangle \to \langle e, M[G \mapsto \bot]\rangle}$$

Fig. 5. Small-step operational semantics for nonadversarial execution of $\lambda^0_{persist}$. Rules that are standard have been elided.

[S1] $$\frac{n > m}{\vdash \{x_1 : \tau_1, \ldots, x_n : \tau_n\}_r \leq \{x_1 : \tau_1, \ldots, x_m : \tau_m\}_r}$$

[S2] $$\frac{\vdash R_1 \leq R_2}{\vdash \mathsf{soft}\ R_1 \leq \mathsf{soft}\ R_2}$$

[S3] $$\frac{\vdash b_1 \leq b_2 \qquad \vdash w_2 \preccurlyeq w_1}{\vdash (b_1)_{w_1} \leq (b_2)_{w_2}}$$

[S4] $$\frac{\vdash \tau_2 \leq \tau_1 \qquad \vdash \tau_1' \leq \tau_2' \qquad \vdash pc_1 \preccurlyeq pc_2}{\vdash \tau_1 \xrightarrow{pc_1} \tau_1' \leq \tau_2 \xrightarrow{pc_2} \tau_2'}$$

[S5] $$\frac{\vdash a_1^+ \preccurlyeq a_2^+ \qquad \vdash a_2^- \preccurlyeq a_1^- \qquad \vdash p_2 \preccurlyeq p_1}{\vdash \{\overrightarrow{x_i : \tau_i}\}_{(a_1^+, a_1^-, p_1)} \leq \{\overrightarrow{x_i : \tau_i}\}_{(a_2^+, a_2^-, p_2)}}$$

Fig. 6. Subtyping rules for $\lambda^0_{persist}$

the value of the field $x$ of the record at that location. Otherwise, a $p$-persistence failure occurs, where $p$ is the persistence level of $m^S$, written $p = \mathsf{persist}(m^S)$.

The field-assignment expression $v_1.x := v_2$ evaluates $v_1$ to a memory location $m^S$ (rules ASSIGN and DANGLE-ASSIGN) If the location has not failed, $v_2$ is assigned into the field $x$ of the record at that location; otherwise, a $p$-persistence failure occurs (where $p = \mathsf{persist}(m^S)$). The notation $M[m^S.x_c \mapsto v]$ denotes the memory resulting from updating with value $v$ the field $x_c$ of the record at location $m^S$.

Persistence failures propagate outward dynamically (Fail-Prop) until the whole program fails. The production $F$ gives the contexts from which persistence failures propagate. The full $\lambda_{persist}$ language, defined in Section 6, can handle these failures.

The soft-reference expression $\mathsf{soft}\ e$ evaluates $e$ to a hard reference and turns it into a soft reference. The soft-reference test $(\mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2)$ promotes the soft reference $v$ (if valid) to a hard reference bound to $x$ and evaluates $e_1$. If the reference is invalid, $e_2$ is evaluated instead.

In rule GC, the notation $\mathsf{gc}(G, \langle e, M\rangle)$ means that $G$ is a set of locations that is *collectible*. $G$ is considered collectible if it has no GC roots (i.e., hard references in $e$), and no location outside $G$ has a hard reference into $G$.

### 5.4 Subtyping in $\lambda^0_{persist}$

The subtyping judgment $\vdash \tau_1 \leq \tau_2$ states that any value of type $\tau_1$ can be treated as a value of type $\tau_2$. Subtyping in $\lambda^0_{persist}$ is the least reflexive and transitive relation consistent with the rules given in Figure 6.

Subtyping on soft references is covariant (rule S2). While hard references may be soundly used as soft references, this is omitted for simplicity. Rule S3 gives contravariant subtyping on integrity labels. Rule S4 gives standard subtyping on functions; the additional *pc* component is covariant. Rule S5 gives subtyping for labeled reference types. It ensures the bounds specified by the reference label of the subtype are at least as precise as those of the supertype.

### 5.5 Static semantics of $\lambda^0_{persist}$

Typing rules for $\lambda^0_{persist}$ are given in Figure 7. The notation $\mathsf{auth}^+(r)$ and $\mathsf{auth}^-(r)$ give the upper ($a^+$) and lower ($a^-$) authority component of a reference label $r$, respectively. The notation $\mathsf{auth}^+(\tau)$, defined below, gives the authority level needed to create a hard reference to a value of type $\tau$. The integrity of $\tau$ is written $\mathsf{integ}(\tau)$, and $\tau \sqcap \ell$ denotes the type obtained by tainting (meeting) the integrity of $\tau$ with $\ell$.

$$\mathsf{auth}^+(\mathsf{bool}) = \mathsf{auth}^+(\mathbf{1}) = \mathsf{auth}^+(\mathsf{soft}\ R) = \bot$$
$$\mathsf{auth}^+(\tau_1 \xrightarrow{pc} \tau_2) = pc \qquad \mathsf{auth}^+(\{\overrightarrow{x_i : \tau_i}\}_s) = \mathsf{auth}^+(s)$$

The typing context includes a *type assignment* $\Gamma$ and the program-counter label *pc*. We write $x{:}\tau \in \Gamma$ and $\Gamma(x) = \tau$ interchangeably. The typing assertion $\Gamma; pc \vdash e : \tau$ means that the expression $e$ has type $\tau$ under type assignment $\Gamma$ with program-counter label *pc*.

Most of the typing rules are standard rules, extended to ensure that the *pc* is sufficiently high to obtain any hard references that may result from evaluating subexpressions (e.g., premise $\vdash \mathsf{auth}^+(\tau) \preccurlyeq pc$ in Rule T-If), and that the *pc* is suitably tainted.

Rule T-Rec checks the creation of records. The *pc* must be high enough to create any hard references that appear in the fields, and to write to the fields themselves.

When using a hard reference $v_1$, the *pc* must have sufficient authority to possess $v_1$ (premise $\vdash \mathsf{auth}^+(r) \preccurlyeq pc$ in rules T-Sel and T-Asgn). When assigning through $v_1$, hard references contained in the assigned value $v_2$ also require authority. Since the integrity and persistence of $v_1$ can affect whether the assignment succeeds, we taint the *pc* with these labels before comparing with the authority requirement of $v_2$.

Rule T-Exists checks soft-reference validity tests. It ensures that the *pc* has the authority to promote the reference from soft to hard (premise $\vdash \mathsf{auth}^+(r) \preccurlyeq pc$).

$$[\text{T-Bool}]\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma; pc \vdash b : \mathsf{bool}_\top} \qquad [\text{T-Unit}]\ \Gamma; pc \vdash * : \mathbf{1}$$

$$[\text{T-Var}]\ \frac{\Gamma(x) = \tau}{\Gamma; pc \vdash x : \tau} \qquad [\text{T-Bot}]\ \frac{p \neq \top}{\Gamma; pc \vdash \bot_p : \tau}$$

$$[\text{T-Loc}]\ \frac{\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)}}{\Gamma; pc \vdash m^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top} \qquad [\text{T-Soft}]\ \frac{\Gamma; pc \vdash e : R_w}{\Gamma; pc \vdash \mathsf{soft}\ e : (\mathsf{soft}\ R)_w}$$

$$[\text{T-If}]\ \frac{\begin{array}{c}\Gamma; pc \vdash v : \mathsf{bool}_w \\ \Gamma; pc \sqcap w \vdash e_i : \tau\ ^{(\forall i)} \\ \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \sqcap w} \qquad [\text{T-Pll}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e_i : \tau_i\ ^{(\forall i)} \\ \vdash \mathsf{auth}^+(\tau_i) \preccurlyeq pc\ ^{(\forall i)}\end{array}}{\Gamma; pc \vdash e_1 \parallel e_2 : \mathbf{1}}$$

$$[\text{T-Abs}]\ \frac{\begin{array}{c}\Gamma, x{:}\tau'; pc' \vdash e : \tau \\ \vdash_{wf} (\tau' \xrightarrow{pc'} \tau)_\top : \mathsf{type} \qquad \vdash pc' \preccurlyeq pc\end{array}}{\Gamma; pc \vdash \lambda(x{:}\tau')[pc'].e : (\tau' \xrightarrow{pc'} \tau)_\top} \qquad [\text{T-App}]\ \frac{\begin{array}{c}\Gamma; pc \vdash v_1 : (\tau' \xrightarrow{pc'} \tau)_w \\ \Gamma; pc \vdash v_2 : \tau' \\ \vdash pc' \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash v_1\ v_2 : \tau \sqcap w}$$

$$[\text{T-Rec}]\ \frac{\begin{array}{c}\vdash_{wf} S : \mathsf{rectype} \qquad S = \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} \qquad \Gamma; pc \vdash v_i : \tau_i'\ ^{(\forall i)} \\ \vdash \tau_i' \leq \tau_i\ ^{(\forall i)} \qquad \vdash \mathsf{auth}^+(\tau_i') \preccurlyeq pc\ ^{(\forall i)} \qquad \vdash \mathsf{integ}(\tau_i) \preccurlyeq pc\ ^{(\forall i)} \qquad \vdash p \preccurlyeq pc\end{array}}{\Gamma; pc \vdash \{\overrightarrow{x_i = v_i}\}^S : (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top}$$

$$[\text{T-Sel}]\ \frac{\begin{array}{c}\Gamma; pc \vdash v : (\{\overrightarrow{x_i : \tau_i}\}_r)_w \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \\ w' = w \sqcap \mathsf{persist}(r)\end{array}}{\Gamma; pc \vdash v.x_c : \tau_c \sqcap w'} \qquad [\text{T-Asgn}]\ \frac{\begin{array}{c}\Gamma; pc \vdash v_1 : (\{\overrightarrow{x_i : \tau_i}\}_r)_w \\ \vdash \mathsf{auth}^+(r) \preccurlyeq pc \\ \Gamma; pc \vdash v_2 : \tau \\ \vdash \tau \sqcap pc \sqcap w \leq \tau_c \\ \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w\end{array}}{\Gamma; pc \vdash v_1.x_c := v_2 : \mathbf{1}}$$

$$[\text{T-Exists}]\ \frac{\begin{array}{c}\Gamma; pc \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w \qquad \vdash \mathsf{auth}^+(r) \preccurlyeq pc \sqcap w \\ w' = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \Gamma, x{:}(\{\overrightarrow{x_i : \tau_i}\}_r)_w; pc \sqcap w' \vdash e_1 : \tau \\ \Gamma; pc \sqcap w' \vdash e_2 : \tau \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap w'\end{array}}{\Gamma; pc \vdash \mathsf{exists}\ v\ \mathsf{as}\ x : e_1\ \mathsf{else}\ e_2 : \tau \sqcap w'}$$

$$[\text{T-Let}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e_1 : \tau' \qquad \vdash \mathsf{auth}^+(\tau') \preccurlyeq pc \\ w = \mathsf{integ}(\tau') \qquad pc' = pc \sqcap w \\ \Gamma, x{:}\tau'; pc' \vdash e_2 : \tau \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc'\end{array}}{\Gamma; pc \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \sqcap w} \qquad [\text{T-Sub}]\ \frac{\Gamma; pc \vdash e : \tau' \qquad \vdash \tau' \leq \tau}{\Gamma; pc \vdash e : \tau}$$

Fig. 7. Typing rules for $\lambda^0_{persist}$

The rules for determining the well-formedness of types are given in Figure 8. In rule WT6, a reference type $(\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w$ is well-formed only if the upper authority label $a^+$ is an upper bound on the authority levels of the field types $\tau_i$. This ensures that the upper authority label is an accurate summary of the authority required by the fields. We also require $a^+$ be bounded from above by the integrity $w$ of the reference, since low-integrity data should not influence the creation of high-authority references.

$$[\text{WT1}]\quad \vdash_{wf} \mathsf{bool}_w : \mathsf{type} \qquad [\text{WT2}]\quad \frac{\vdash pc \preccurlyeq w \quad \vdash_{wf} \tau_1 : \mathsf{type} \quad \vdash_{wf} \tau_2 : \mathsf{type} \quad \vdash \mathsf{auth}^+(\tau_1) \sqcup \mathsf{auth}^+(\tau_2) \preccurlyeq pc}{\vdash_{wf} (\tau_1 \xrightarrow{pc} \tau_2)_w : \mathsf{type}}$$

$$[\text{WT3}]\quad \vdash_{wf} \mathbf{1} : \mathsf{type} \qquad [\text{WT4}]\ \frac{\vdash_{wf} (\{\overrightarrow{x_i : \tau_i}\}_{(a,a,p)})_\top : \mathsf{type} \quad \vdash \mathsf{integ}(\tau_i) \preccurlyeq p\ ^{(\forall i)}}{\vdash_{wf} \{\overrightarrow{x_i : \tau_i}\}_{(a,p)} : \mathsf{rectype}}$$

$$[\text{WT5}]\ \frac{\vdash_{wf} R_\top : \mathsf{type}}{\vdash_{wf} (\mathsf{soft}\ R)_w : \mathsf{type}} \quad [\text{WT6}]\quad \frac{\vdash_{wf} \tau_i : \mathsf{type}\ ^{(\forall i)} \quad \vdash \mathsf{auth}^+(\tau_i) \preccurlyeq a^+\ ^{(\forall i)} \quad \vdash a^+ \preccurlyeq w \sqcap p \quad \vdash a^- \preccurlyeq a^+}{\vdash_{wf} (\{\overrightarrow{x_i : \tau_i}\}_{(a^+,a^-,p)})_w : \mathsf{type}}$$

Fig. 8. Well-formedness of types

To ensure hosts are able to create hard references to the objects they store, we also require $\mathsf{auth}^+(r)$ to be bounded from above by the persistence level $p$ of the record.

## 6 Ensuring referential integrity

In a distributed system, references can span trust domains, so to be secure and reliable, program code must in general be ready to encounter a dangling reference, one perhaps created by the adversary. Therefore, we extend $\lambda^0_{persist}$ with *persistence-failure handlers* to obtain the full $\lambda_{persist}$ language (see [16] for its full syntax). The type system of $\lambda_{persist}$ forces the programmer to be aware of and to handle all potential failures.

We might consider an approach in which failures must be handled immediately upon encountering a broken reference. However, because low-persistence references may be used frequently, this would likely result in much duplication of failure-handling code.

Instead, $\lambda_{persist}$ factors out failure-handling code from ordinary code by treating failures as a kind of exception. The value of ($\mathsf{try}\ e_1\ \mathsf{catch}\ p\colon e_2$) is the value of evaluating $e_1$. If a dangling reference at persistence level $p$ or higher is encountered, the failure handler $e_2$ is evaluated instead. A try expression creates a context ($e_1$) in which the programmer can write simpler code under the assumption that certain persistence failures are impossible, yet without sacrificing the property that all failures are handled.

### 6.1 Persistence handler levels

To track the failures that the current context can handle, a *set* of persistence levels $\mathcal{H}$ is used.[3] It provides lower bounds on the persistence levels of hard references that may be directly dereferenced. Functions $\lambda(x{:}\tau)[pc;\mathcal{H}].\,e$ and function types $\tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2$ are extended with an $\mathcal{H}$ component, which is an upper bound on the $\mathcal{H}$ levels of the caller.

### 6.2 Example

Returning to the directory example in Figure 1, Alice can add a place to the list of sightseeing ideas with the code below. This code starts at Alice's docs directory, traverses the reference to the scratchpad, and invokes an add method to add a museum.

```
let pad = docs.scratchpad
in try pad.add "Rodin Museum" catch ⊥: ...
```

[3] Formally, $\mathcal{H}$ is drawn from the upper powerdomain [22] of persistence levels.

$$\begin{bmatrix}\textsc{Try-}\\\textsc{Val}\end{bmatrix}\ \frac{\forall p'.\ v \neq \bot_{p'}}{\langle \mathsf{try}\ v\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle v, M\rangle} \qquad \begin{bmatrix}\textsc{Try-}\\\textsc{Catch}\end{bmatrix}\ \frac{p \preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle e, M\rangle}$$

$$\begin{bmatrix}\textsc{Try-}\\\textsc{Esc}\end{bmatrix}\ \frac{p \not\preccurlyeq p'}{\langle \mathsf{try}\ \bot_{p'}\ \mathsf{catch}\ p\colon e, M\rangle \xrightarrow{\mathsf{e}} \langle \bot_{p'}, M\rangle} \qquad E ::= \ldots \mid \mathsf{try}\ [\cdot]\ \mathsf{catch}\ p\colon e$$

---

$$\begin{bmatrix}\textsc{T-Soft-}\\\textsc{Select}\end{bmatrix}\ \frac{\Gamma;pc;\mathcal{H} \vdash v : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \qquad \vdash \mathsf{auth}^+(\tau_c) \preccurlyeq pc \qquad p = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \qquad \vdash \mathcal{H} \preccurlyeq p}{\Gamma;pc;\mathcal{H} \vdash v.x_c : \tau_c \sqcap p, p}$$

$$\begin{bmatrix}\textsc{T-Soft-}\\\textsc{Assign}\end{bmatrix}\ \frac{\begin{array}{c}\Gamma;pc;\mathcal{H} \vdash v_1 : (\mathsf{soft}\ \{\overrightarrow{x_i : \tau_i}\}_r)_w, \top \qquad p = \mathsf{auth}^-(r) \sqcap \mathsf{persist}(r) \sqcap w \\ \Gamma;pc;\mathcal{H} \vdash v_2 : \tau, \top \qquad \vdash \tau \sqcap pc \sqcap p \leq \tau_c \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap p \qquad \vdash \mathcal{H} \preccurlyeq p\end{array}}{\Gamma;pc;\mathcal{H} \vdash v_1.x_c := v_2 : \mathbf{1}, p}$$

$$[\textsc{T-Try}]\ \frac{\begin{array}{c}\Gamma;pc;\mathcal{H}, p \vdash e_1 : \tau, \mathcal{X}_1 \qquad w = \bigsqcap_{p' \in \mathcal{X}_1} (p \sqcup p') \\ \Gamma;pc \sqcap w \sqcap \mathsf{integ}(\tau);\mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2 \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc\end{array}}{\Gamma;pc;\mathcal{H} \vdash \mathsf{try}\ e_1\ \mathsf{catch}\ p\colon e_2 : \tau \sqcap w, (\mathcal{X}_1 / p) \sqcap \mathcal{X}_2}$$

Fig. 9. Additional small-step evaluation and typing rules for $\lambda_{persist}$

The expression pad.add follows a hard reference to the scratchpad. Despite the hard reference, a try is needed because Alice does not trust host U to persist the scratchpad.

### 6.3 Static and dynamic semantics of $\lambda_{persist}$

The small-step operational semantics of $\lambda_{persist}$ extends that of $\lambda^0_{persist}$ with the rules at the top of Figure 9. Failures propagate outward dynamically (TRY-ESC) until either they are handled by a failure handler (TRY-CATCH), or the whole program fails. See [16] for the full operational semantics for $\lambda_{persist}$.

The subtyping rules are the same as for $\lambda^0_{persist}$, except that function subtyping is also contravariant on the $\mathcal{H}$ component. Full subtyping rules are also in [16].

The typing rules for $\lambda_{persist}$ extend those for $\lambda^0_{persist}$. They augment the typing context with a *handler environment* $\mathcal{H}$, indicating the set of persistence failures the evaluation context can handle. Typing judgments additionally produce an effect $\mathcal{X}$, which is a set indicating the persistence failures that can occur during evaluation.

The typing rules for $\lambda^0_{persist}$ are converted straightforwardly to thread $\mathcal{H}$ and $\mathcal{X}$ through typing judgments. Rules T-SEL and T-ASGN gain premises to ensure the context has a suitable handler in case dereferences fail. See [16] for the converted rules.

The bottom of Figure 9 gives three new typing rules. T-SOFT-SELECT and T-SOFT-ASSIGN check direct uses of soft references. They taint the integrity of the dereference with $\mathsf{auth}^-(r)$ because the result of the dereference is affected by those able to pin the referent in memory by creating a hard reference (Section 3.5). Rule T-TRY checks try expressions. To reflect the installation of a $p$-persistence handler, $p$ is added to the handler environment $\mathcal{H}$ when checking $e_1$. The value $w$ in the typing rule is a conservative summary of the persistence errors that can occur while evaluating $e_1$ and are not handled by the $p$-persistence handler. Because evaluation of $e_2$ depends on the result of $e_1$,

$$[\alpha\text{-CREATE}] \quad \frac{m = \mathsf{newloc}(M) \qquad \varnothing;\top;\top \vdash \{\overrightarrow{x_i = [v_i]}\}^S : R_\top, \top \qquad \vdash^{\alpha}_{[wf]} M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}] \qquad \alpha \not\leq \mathsf{persist}(S)}{\langle e, M\rangle \rightsquigarrow_\alpha \left\langle e, M[m^S \mapsto \{\overrightarrow{x_i = [v_i]}\}]\right\rangle}$$

$$[\alpha\text{-ASSIGN}] \quad \frac{m^S \in \mathsf{dom}(M) \qquad M(m^S) \neq \bot \qquad S = \{\overrightarrow{x_i : \tau_i}\}_s \qquad \varnothing;\top;\top \vdash [v] : \tau_c, \top \qquad \vdash^{\alpha}_{[wf]} M[m^S.x_c \mapsto [v]]}{\langle e, M\rangle \rightsquigarrow_\alpha \left\langle e, M[m^S.x_c \mapsto [v]]\right\rangle}$$

$$[\alpha\text{-FORGET}] \quad \frac{m^S \in \mathsf{dom}(M) \qquad \alpha \not\leq \mathsf{persist}(S)}{\langle e, M\rangle \rightsquigarrow_\alpha \left\langle e, M[m^S \mapsto \bot]\right\rangle}$$

Fig. 10. Effects caused by the α-adversary

the *pc* label for evaluating $e_2$ is tainted by $w$. In this rule, the notation $\mathcal{X}/p$ denotes the subset of persistence errors $\mathcal{X}$ not handled by $p$.

## 7 The power of the adversary

Possible actions of the adversary are modeled by extending the operational semantics of Figure 5 with more transitions. To support reasoning about what an adversary may have affected in a partially evaluated program, $\lambda_{persist}$ is also augmented to include bracketed expressions, resulting in the language $[\lambda_{persist}]$. The term $[e]$ represents an expression $e$ that may have been influenced by the adversary, and $[v]$ is an influenced value. The operational semantics is extended by adding rules that propagate these brackets in the obvious manner. (Doubly bracketed values are considered expressions, not values.)

The rule for typing bracketed expressions is as follows:

$$[\text{T-BRACKET}] \quad \frac{\Gamma; pc \sqcap \ell; \mathcal{H} \vdash e : \tau, \mathcal{X} \qquad \alpha \not\leq \ell \qquad \vdash \mathsf{auth}^+(\tau) \preccurlyeq pc \sqcap \ell}{\Gamma; pc; \mathcal{H} \vdash [e] : \tau \sqcap \ell, \mathcal{X}}$$

The adversary is powerful, as shown by the transitions defined in Figure 10. Adversaries may create new records (rule α-CREATE), modify existing records (rule α-ASSIGN), or remove records from memory altogether (rule α-FORGET), but their ability is bounded by an integrity label $\alpha \in \mathcal{L}$. Such an α-*adversary* has all creation authority except α and higher, can modify any record field except those with α (or higher) integrity, and can delete any record except those with α (or higher) persistence. A small evaluation step taken in the presence of an α-adversary is written $\langle e, M\rangle \to_\alpha \langle e', M'\rangle$.

It is important to know that any evaluation of a program in the original language can be simulated in the augmented language, which amounts to showing that the rules cover all the ways that brackets can appear. This is proved straightforwardly by induction on the evaluation rules.

The adversary's transitions embody a simplifying assumption that the adversary can only create well-typed values. While it is reasonable to allow the adversary to create ill-typed values, an implementation with run-time type checking can catch ill-typed values when they cross between hosts and replace them with well-typed default values.

## 8 Results

The goal of $\lambda_{persist}$ is to prevent accidental persistence and to ensure that the adversary cannot damage referential integrity or cause storage attacks. Accidental persistence is prevented by the use of persistence policies. We now show how to formalize the other security properties and sketch the proof that they are enforced.

### 8.1 Soundness and well-formedness

We have proved [16] the $[\lambda_{persist}]$ type system sound with the usual method, via preservation and progress. A well-formed $\lambda_{persist}$ memory $M$, written $\vdash_{wf} M$, maps typed locations to record values with the same type. In a $\lambda_{persist}$ configuration that is well-formed with respect to an $\alpha$-adversary (written $\vdash_{wf}^{\alpha} \langle e, M\rangle$), no noncollectible high-persistence location is deleted. $\lambda_{persist}$ configurations are well-formed in a nonadversarial setting ($\vdash_{wf} \langle e, M\rangle$) if they are well-formed with respect to the $\bot$-adversary.

Corresponding well-formedness conditions are defined similarly for $[\lambda_{persist}]$ and is written with brackets around the *wf* subscript. Well-formed $[\lambda_{persist}]$ memories additionally require that values appearing in low-integrity record fields must be bracketed.

### 8.2 Security relation

The key to proving both referential integrity and immunity to storage attacks is to show that the adversary cannot meaningfully influence the high-integrity parts of the program and memory. To do this, we define a *security relation* and show that each configuration $\langle e_1, M_1\rangle$ reached via the language augmented by adversarial transitions must be related to some configuration $\langle e_2, M_2\rangle$ reachable by purely nonadversarial execution. This security property is possibilistic, which is problematic for confidentiality properties [21] but is acceptable for integrity.

Because the two executions being compared operate on different heaps, with the adversary behaving differently in the two executions, the addresses chosen during record allocation may differ. However, the structure of the high-integrity part of the heap should still correspond. A *high-integrity homomorphism* $\phi$ is used to relate corresponding locations in the two heaps that are high-integrity or high-persistence. High-integrity homomorphisms are injective, preserve location types, and are isomorphisms on both high-integrity and high-persistence locations. This is defined formally in [16].

An expression $e_1$ is considered to be related to $e_2$ via a high-integrity homomorphism $\phi$, written $e_1 \approx_{\alpha}^{\phi} e_2$, if $e_1$ is equal to $e_2$ (modulo bracketed expressions) when the memory locations in $e_1$ are transformed via $\phi$.

We also define a security relation on memories: $M_1$ and $M_2$ are related via $\phi$, written $M_1 \approx_{\alpha}^{\phi} M_2$, if two conditions hold for each location $m^S \in \mathsf{dom}(\phi)$. If $m^S$ is not deleted, then $\phi(m^S)$ maps to a related record. Otherwise, if $m^S$ is deleted, high-authority, and high-persistence, then so is $\phi(m^S)$. The formal definition is given in [16]. These two security relations induce a security relation on configurations:

$$\langle e_1, M_1\rangle \approx_{\alpha}^{\phi} \langle e_2, M_2\rangle \overset{def.}{\iff} e_1 \approx_{\alpha}^{\phi} e_2 \wedge M_1 \approx_{\alpha}^{\phi} M_2.$$

A $[\lambda_{persist}]$ program has limited adversary influence if related initial configurations produce related final configurations. We now see that the language $[\lambda_{persist}]$ enforces security, because all well-formed programs do have limited adversary influence.

### 8.3 Referential integrity

Theorem 1 formalizes the referential integrity result, showing that the adversary has limited influence on program execution: execution in the presence of an adversary is $\phi$-related to a nonadversarial execution.

For the remainder of this paper, assume $\langle e_1, M_1 \rangle$ is a well-formed configuration and $\langle e_2, M_2 \rangle$ is a well-formed, nonadversarial, $\phi$-related configuration, such that $e_1$ and $e_2$ have type $\tau$ and $M_2$ is well-formed:

$$\vdash^{\alpha}_{[wf]} \langle e_1, M_1 \rangle \;\wedge\; \vdash_{[wf]} \langle e_2, M_2 \rangle \;\wedge\; \langle e_1, M_1 \rangle \approx^{\phi}_{\alpha} \langle e_2, M_2 \rangle$$
$$\wedge\; \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \;\wedge\; \varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \;\wedge\; \vdash^{\alpha}_{[wf]} M_2$$

**Theorem 1 (Referential integrity)** *Suppose $\langle e_1, M_1 \rangle$ takes some number of steps in the presence of an adversary to another configuration $\langle e'_1, M'_1 \rangle$. Then either $\langle e_2, M_2 \rangle$ diverges, or it can take some number of steps in the absence of an adversary to another configuration $\langle e'_2, M'_2 \rangle$ and there exists a high-integrity homomorphism $\phi'$ from $M'_1$ to $M'_2$ that extends $\phi$, such that $\langle e'_1, M'_1 \rangle$ is related to $\langle e'_2, M'_2 \rangle$ via $\phi'$:*

$$\langle e_1, M_1 \rangle \rightarrow^{*}_{\alpha} \left\langle e'_1, M'_1 \right\rangle \wedge \neg \langle e_2, M_2 \rangle \Uparrow$$
$$\Rightarrow \exists e'_2, M'_2, \phi'.\ \langle e_2, M_2 \rangle \rightarrow^{*} \left\langle e'_2, M'_2 \right\rangle \wedge \left\langle e'_1, M'_1 \right\rangle \approx^{\phi'}_{\alpha} \left\langle e'_2, M'_2 \right\rangle \wedge \phi = \phi'\big|_{\mathsf{dom}(\phi)}$$

*Proof:* Induction on the derivation of $\langle e_1, M_1 \rangle \rightarrow_{\alpha} \langle e'_1, M'_1 \rangle$.

### 8.4 Storage attacks

To formalize immunity to storage attacks, we first show that the adversary cannot cause more high-persistence locations to be allocated. Theorem 1 captures this via the security relation, since all high-persistence locations are mapped by the homomorphism.

We now show that the adversary cannot cause more high-authority locations to become *noncollectible*; that is, reachable through hard references. Lemma 1 says that this is also implied by Theorem 1. (We write $\mathsf{nc}(m^S, \langle e, M \rangle)$ to mean $m^S$ is noncollectible in $\langle e, M \rangle$. The formal, inductive definition is in [16].)

**Lemma 1.** *If $m^S$ is a high-authority, noncollectible location in $\langle e_1, M_1 \rangle$, then $\phi(m^S)$ is also noncollectible in $\langle e_2, M_2 \rangle$.*

$$\vdash \alpha \preccurlyeq \mathsf{auth}^{+}(S) \wedge \mathsf{nc}(m^S, \langle e_1, M_1 \rangle) \Rightarrow \mathsf{nc}(\phi(m^S), \langle e_2, M_2 \rangle)$$

*Proof:* By induction on the derivation of $\mathsf{nc}(m^S, \langle e_1, M_1 \rangle)$.

## 9 Related work

This paper identifies and addresses a new problem, referential security. As a result, little prior work is closely related.

Some prior work has tried to improve referential integrity through system mechanisms, for example improving the referential integrity of web hyperlinks [8, 11]. Systems mechanisms for improving referential integrity (and other aspects of trustworthiness) are orthogonal to the language model presented here, but could be used to justify assigning persistence, integrity, and authority levels to nodes.

Liblit and Aiken [12] develop a type system for distributed data structures. Its explicit two-level hierarchy distinguishes between local pointers that are meaningful only to a single processor, and global pointers that are valid everywhere. The type system ensures that local pointers do not leak into a global context. This work was extended in [13] to add types for dealing with private vs. shared data. However, this line of work does not consider security properties that require defense against an adversary.

Riely and Hennessey study type safety in a distributed system of partially trusted mobile agents [20] but do not consider referential security.

This paper builds on prior work on language-based information-flow security, much of which is summarized by [21]. The Fabric system [15] is programmed in a high-level language that includes integrity annotations and abstracts away the locations of objects, as $\lambda_{persist}$ does. Its type system does not enforce referential security, however, so adding the features described here is an obvious next step.

## 10 Conclusions

Complex distributed information systems are being integrated across different organizations with only partial trust, often in the context of cloud computing. But the security properties that are desirable in distributed computing are poorly understood, and the options for enforcing security are murkier still. In fact, the desirable referential security properties are actually in tension with each other. The result is that programmers have little guidance in designing distributed systems to be secure and reliable.

This paper makes several contributions that aid in resolving this situation. The paper newly identifies and formalizes some important referential security properties. It introduces a high-level language for modeling referential security issues in a distributed system. The language introduces a way to express referential security requirements through label annotations for persistence and creation authority, which can be viewed as different aspects of integrity. The paper demonstrates how to enforce referential security, through static analysis expressed as a type system in the language. The type system is validated by formal proofs that $\lambda_{persist}$ programs enforce the new security properties.

While this paper is a useful first step, clearly there is more to be done. The type system could be enriched with more features such as parametric polymorphism, recursive and dependent types. With such extensions, an implementation would then help evaluate how well these types guide programmers designing distributed computing systems.

## Acknowledgments

This research was supported in part by ONR Grants N00014-09-1-0652 and N00014-13-1-0089; by MURI grant FA9550-12-1-0400, administered by the U.S. Air Force; by NSF Grants 0541217, 0627649, and 0964409; and by a grant from Microsoft Corporation. The views and conclusions here are those of the authors and do not necessarily reflect those of ONR, the Navy, the Air Force, NSF, or Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon.

We also thank Aslan Askarov, Danfeng Zhang, Owen Arden, Barbara Liskov, Mike George, and David Schulz for their suggestions about this work or its presentation.

## References


1. Malcom Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proc. International Conference on Deductive Object Oriented Databases*, Kyoto, Japan, December 1989.
2. K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.
3. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP '93*, pages 217–230, December 1993.
4. Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *OOPSLA '86*, pages 78–86, November 1986.
5. Breeze, 2013. `www.breezejs.com`.
6. Heiko Böck. *Java Persistence API*. Springer, 2011.
7. E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, December 1979.
8. Hugh C. Davis. Referential integrity of links in open hypermedia systems. In *Proc. 9th ACM Conference on Hypertext and Hypermedia*, pages 207–216, 1998.
9. Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.
10. Hibernate. `www.hibernate.org`.
11. Frank Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *Journal of Universal Computer Science*, 1(2), 1995.
12. Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *POPL*, pages 199–213, January 2000.
13. Ben Liblit, Alexander Aiken, and Katherine A. Yelick. Type systems for distributed data sharing. In *Proc. 10th International Static Analysis Symposium*, volume 2694 of *LNCS*, San Diego, California, June 2003. Springer-Verlag.
14. Barbara H. Liskov. The Argus language and system. In *Distributed Systems: Methods and Tools for Specification*, volume 150 of *Lecture Notes in Computer Science*, pages 343–430. Springer-Verlag Berlin, 1985.
15. Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009.
16. Jed Liu and Andrew C. Myers. A language for securely referencing persistent information in a federated system. Technical Report 1813-35150, Computing and Information Science Department, Cornell University, January 2014.
17. D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
18. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
19. OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.
20. James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL '99*, pages 93–104, 1999.
21. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
22. Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.
23. Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.