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 [15], 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 [11] 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 [16] 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-language design; in programming languages that lack referential integrity, such
as C, dangling pointers are a serious problem. Today, many languages have automatic garbage collection, allowing the automatic reclamation of memory while preserving referential integrity.

While absolute referential integrity is desirable, 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 \textit{intentional persistence}. With referential integrity, a reference to an object is a promise to the referrer 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 \textit{accidentally persistent} because they are unexpectedly reachable. Accidental persistence 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 \textit{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 \textit{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.

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 $\lambda_{\text{persist}}$, 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 proves that the type system of $\lambda_{\text{persist}}$ 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 $\lambda_{\text{persist}}$. In $\lambda_{\text{persist}}$, persistence, distribution, and communication are implicit but are constrained by policy annotations. Programs in $\lambda_{\text{persist}}$ 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 λ persists 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 how 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 $\lambda$ persists 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. Because of this promise, hard references can only be created by trusted code. A soft reference does not create an obligation to maintain the referenced object. 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. 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.

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

Referential integrity ensures that a pointer can be followed to its referent—that there are no dangling pointers. 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 $(\mathcal{L}, \preceq, 1, \top)$ of policy levels. If $p_1 \preceq p_2$ 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 $p_1 \preceq p_2$, 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 $a$. 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 $\perp$.

Persistence policies are integrated into the type system of $\lambda$ persists. 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 $\lambda_{\text{persis}}$ will ensure their code handles such a
failure. Any reference to the scratchpad must have a type with $\bot$ persistence, because it can be no higher than the $\bot$
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 $\lambda_{\text{persis}}$, 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., $\alpha \not\leq p$), 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 $\lambda_{\text{persis}}$ 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 $(\mathcal{L}, \leq)$ 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, $a \leq a_p$.

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 $\text{bob}$; 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 $\{\text{alice, bob}\}$ 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 \( (L, \preceq) \) 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 \( \preceq \) corresponds to increasing integrity. If \( w_1 \preceq w_2 \), an information flow from level \( w_2 \) to \( w_1 \) would be secure: more-trusted information would be affecting less-trusted information.\(^1\) In \( \lambda_{\text{persist}} \), 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 interaction with 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 its interpretation 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 the reference. In Figure 1, if Alice follows the reference from her docs directory to the scratchpad, she obtains an untrusted result; the untrusted, adversarial 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.

---

\(^1\)This ordering is the opposite of the “upside-down” ordering typically seen in work on information-flow security [2], and corresponds to the trust ordering [16].
Figure 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.

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

4 Types for persistent programming

To formalize the ideas presented in the previous section, we introduce the $\lambda_{\text{persist}}$ language, an extension to the simply typed lambda calculus. Figure 4 gives part of the formal syntax of $\lambda_{\text{persist}}$. 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 $\lambda_{\text{persist}}$ program with respect to a particular adversary if following hard references whose persistence 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 $(\mathcal{L}, \leq, \bot, \top)$ 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 $\lambda_{\text{persist}}$ with recursive and dependent types; for simplicity, these features have been omitted from $\lambda_{\text{persist}}$ because they do not appear to add interesting issues. However, we can capture the security of a general directory structure by using $\lambda_{\text{persist}}$ 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 $\lambda_{\text{persist}}$ are about objects and references to them. Therefore, $\lambda_{\text{persist}}$ extends the lambda calculus with records that represent the content of objects. The record $\{x_i = v_i\}$ comprises a set of fields $x_i$ with corresponding values $v_i$. Records are not values in the language; instead, they are accessed via references $m^\delta$, where $m$ is the identity of the object and $S = \{x_i: \tau_i\}$, 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 $\{x_i: \tau_i\}_r$. 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 $\lambda_{\text{persist}}$ language is to model a distributed system in which code is running at different host nodes. A single program written in $\lambda_{\text{persist}}$ 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 $e_i$) executing at different nodes, the individual computations are composed in parallel $(e_1 \parallel \cdots \parallel e_n)$ into a single $\lambda_{\text{persist}}$ 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 $\lambda_{\text{persist}}$ in two phases. In this section, we present $\lambda_{\text{persist}}^0$, a simplified subset of $\lambda_{\text{persist}}$ that prevents accidental persistence and storage attacks.

5.1 Syntax of $\lambda_{\text{persist}}^0$

Figure 4 gives the syntax of $\lambda_{\text{persist}}^0$. The names $x$ and $y$ range over variable names $\text{Var}$; $m$ ranges over a space of memory addresses $\text{Mem}$; $w$, $a$, $p$, 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 $\lambda_{\text{persist}}^0$ consist of base types with an integrity label $(b_w)$, and the unit type $1$, which needs no integrity label. Base types include booleans, functions, and two kinds of references to mutable records: hard ($R$) and soft ($S$). The metavariable $R$ denotes a labeled reference type.

The type $\tau_1 \rightarrow_{\text{pc}} \tau_2$ is a function type with a $\text{pc}$ annotation that is a lower bound on the $\text{pc}$ label of the caller. It gives an upper bound on the integrity of data the function affects, on the authority level of references the function creates, and on the authority level of references in the function body and in the closure environment.

Values include variables $x$, booleans true and false, the unit value $\bot$, record-typed memory locations (references) $m^\delta$, soft references soft $m^\delta$, and functions $\lambda(x: \tau)[pc], e$. A function $\lambda(x: \tau)[pc], e$ has one argument $x$ with type $\tau$. The $\text{pc}$ component has the same meaning as that in function types. At run time, $p$-persistence failures $\bot_p$ can also appear as values.

Terms include values $v$ and $u$, applications $v_1 v_2$, if expressions if $v_1$ then $e_2$ else $e_3$, record constructors $\{x_i = v_i\}_x$, field selections $v.x$, field assignments $v_1.x := v_2$, soft references soft $e$, parallel composition $e_1 \parallel e_2$, soft-reference tests exists $v \exists x : e_1$ else $e_2$, and let expressions let $x = e_1$ in $e_2$. 

7
Figure 5 gives the small-step operational semantics of $\lambda_{\text{persist}}^0$. 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.

2To 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 $M$ represent a memory: a finite partial map from typed memory locations $m^S$ to closed record values. Let $(e, M)$ be a system configuration. A small evaluation step is a transition from $(e, M)$ to another configuration $(e', M')$, written $(e, M) \rightarrow (e', M')$.

To avoid using undefined memory locations, we restrict the form of $(e, M)$. Let $\text{locs}(e)$ 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 $\text{dom}(M)$, and for any location $m^S$ in $\text{dom}(M)$, $\text{locs}(M(m^S)) \subseteq \text{dom}(M)$. A configuration $(e, M)$ is well-formed only if $M$ is well-formed, $\text{locs}(e) \subseteq \text{dom}(M)$, and $e$ has no free variables. Evaluation preserves well-formed configurations (see Lemma 10 in Section 8.3).

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 $\{x_i = v_i\}^S$ (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 $\text{newloc}(M)$ deterministically generates a fresh memory location. If address-space($M$) represents the set of location names in $M$ (i.e., $\{m : \exists S. m^S \in \text{dom}(M)\}$), then $\text{newloc}(M) \not\in \text{address-space}(M)$ and $\text{newloc}(M') = \text{newloc}(M)$ if address-space($M'$) = address-space($M$).

Parallel-composition expressions $e_1 \parallel e_2$ evaluate to the unit value (rule PARALLEL-RESULT).

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 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 $p = \text{persist}(m^S)$.

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 $p = \text{persist}(m^S)$). The notation $M[m^S.x_e \rightarrow v]$ denotes the memory resulting from updating with value $v$ the field $x_e$ of the record at location $m^S$.

Persistence failures propagate outward dynamically (FAIL-PROP) until the whole program fails. The full $\lambda_{\text{persist}}$ language, defined in Section 6, handles these failures.

In rule EVAL-CONTEXT, $E$ represents an ordinary evaluation context, whereas in rule FAIL-PROP, $F$ gives the contexts from which persistence failures propagate. Contexts are given as a term with a single hole (denoted by $[\_]$) in redex position. The syntax of $E$ specifies the evaluation order.

The soft-reference expression $\text{soft } e$ evaluates $e$ to a hard reference and turns it into a soft reference. The soft-reference test $(\exists x : e_1 \text{ else } e_2)$ 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 $\text{gc}(G, (e, M))$ 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 $G$. This is defined formally as follows.

**Definition 1** (GC roots). A location $m^S$ is a GC root in an expression $e$, written $\text{root}(m^S, e)$, if it is a hard reference in $e$. This is formally defined by the following inference rules:

\[
\begin{align*}
[R1] & \quad \frac{\text{root}(m^S_1, m^S_2)}{\text{root}(m^S_1, m^S_2, m^S_e)} \\
[R2] & \quad \frac{\text{root}(m^S, e)}{\forall m^S_0. \text{root}(m^S_0, \text{soft } e)} \\
[R3] & \quad \frac{\exists i. \text{root}(m^S, v_i)}{\text{root}(m^S, (x_i = v_i)^S)} \\
[R4] & \quad \frac{\text{root}(m^S, v)}{\text{root}(m^S, v, x)} \\
[R5] & \quad \frac{\exists i. \text{root}(m^S, v_i)}{\text{root}(m^S, v, x := v_2)} \\
[R6] & \quad \frac{\text{root}(m^S, e)}{\text{root}(m^S, \lambda(x : \tau)[pc]. e)} \\
[R7] & \quad \frac{\exists i. \text{root}(m^S, v_i)}{\text{root}(m^S, v_1, v_2)} \\
[R8] & \quad \frac{\exists i. \text{root}(m^S, e_i)}{\text{root}(m^S, \text{let } x = e_1 \text{ in } e_2)} \\
[R9] & \quad \frac{\exists i. \text{root}(m^S, e_i)}{\text{root}(m^S, e_1 \parallel e_2)} \\
[R10] & \quad \frac{\exists i. \text{root}(m^S, e_i)}{\text{root}(m^S, \text{if } e_1 \text{ then } e_2 \text{ else } e_3)} \\
[R11] & \quad \frac{\exists i. \text{root}(m^S, e_i)}{\text{root}(m^S, \text{exists } x \text{ as } e : e_2 \text{ else } e_3)} \\
[R12] & \quad \frac{\exists i. \text{root}(m^S, e_i)}{\text{root}(m^S, \text{try } e_1 \text{ catch } p : e_2)} \\
[R13] & \quad \frac{\text{root}(m^S, e)}{\text{root}(m^S, [e])}
\end{align*}
\]

$\lambda_{\text{persist}}$, defined in Section 6, adds R12; and $[\lambda_{\text{persist}}]$, defined in Section 7, adds R13.
The subtyping judgment

\[ \frac{n > m}{\vdash \{x_1:\tau_1, \ldots, x_n:\tau_n\} \tau \{x_1:\tau_1, \ldots, x_m:\tau_m\} \tau} \]

Subtyping in \( \lambda_{\text{Persist}} \)

Typing rules for \( \text{pc} \)

Subtyping on functions; the additional \( \text{this} \) is omitted for simplicity. Rule S3 gives contravariant subtyping on integrity labels. Rule S4 gives standard (\( \lambda \)) subtype are at least as precise as those of the supertype. Rule S5 gives subtyping for labeled reference types. Subtyping is covariant on the \( \text{component is covariant. These are the opposite of the rules typically seen in work on information-flow security, accounting for our use of the trust ordering.} \)

Rule S5 gives subtyping for labeled reference types. Subtyping is covariant on the \( a^+ \) component of the reference label and contravariant on the other two components. This ensures the bounds specified by the reference label of the subtype are at least as precise as those of the supertype.

\[ \Gamma \vdash e : \tau \]

Subtyping in \( \lambda_{\text{Persist}} \)

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 \( \lambda_{\text{Persist}} \) is the least reflexive and transitive relation consistent with the rules given in Figure 6. Rule S1 gives standard width subtyping on records. Because records are mutable, there is no depth subtyping.

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 \( \text{pc} \) component is covariant. These are the opposite of the rules typically seen in work on information-flow security, accounting for our use of the trust ordering.

Rule S5 gives subtyping for labeled reference types. Subtyping is covariant on the \( a^+ \) component of the reference label and contravariant on the other two components. This ensures the bounds specified by the reference label of the subtype are at least as precise as those of the supertype.

\[ \Gamma \vdash e : \tau \]

Static semantics of \( \lambda_{\text{Persist}} \)

Typing rules for \( \lambda_{\text{Persist}} \) are given in Figure 7. The notation \( \text{auth}^+(r) \), \( \text{auth}^-(r) \), and \( \text{persist}(r) \) give the upper authority \( (a^+) \), lower authority \( (a^-) \), and persistence \( (p) \) component of a reference label \( r \), respectively. The notation \( \text{auth}^+(s) \) and \( \text{persist}(s) \) give the authority and persistence component of a storage label, respectively. The notation \( \text{auth}^+(\tau) \), defined below, gives the authority level needed to create a hard reference to a value of type \( \tau \). The integrity of \( \tau \) is written \( \text{integ}(\tau) \), and \( \tau \cap \ell \) denotes the type obtained by tainting (meeting) the integrity of \( \tau \) with \( \ell \):

\[
\begin{align*}
\text{auth}^+(\text{bool}) &= \text{auth}^+(1) = \text{auth}^+(\text{soft } R) = 1 \\
\text{integ}(b_w) &= w \quad (b_w) \cap \ell = b_w \cap \ell \quad \text{auth}^+(\tau_1 \xrightarrow{pc} \tau_2) = \text{pc} \\
\text{integ}(1) &= T \quad 1 \cap \ell = 1 \quad \text{auth}^+(\{x_i:\tau_i\}) = \text{auth}^+(s)
\end{align*}
\]

The typing context includes a type assignment \( \Gamma \) and the program-counter label \( \text{pc} \). \( \Gamma \) is a finite partial map from variables \( x \) to types \( \tau \), expressed as a finite list of \( x : \tau \) entries. We write \( x : \tau \in \Gamma \) and \( \Gamma(x) = \tau \) interchangeably. For an expression \( e \) that is well-typed in a context \( \Gamma ; \text{pc} \), the type checker produces a type \( \tau \). The typing assertion \( \Gamma ; \text{pc} \vdash e : \tau \), therefore, means that the expression \( e \) has type \( \tau \) under type assignment \( \Gamma \) with program-counter label \( \text{pc} \).

Most of the typing rules are standard rules, extended to ensure that the \( \text{pc} \) is sufficiently high to obtain any hard references that may result from evaluating subexpressions (e.g., premise \( \vdash \text{auth}^+(\tau) \leq \text{pc} \) in Rule T-IF), and that the \( \text{pc} \) is suitably tainted.
Figure 7: Typing rules for $\lambda^0_{\text{persist}}$

Rule T-BOT says persistence failures can have any well-formed type.

Rule T-ABS checks function values. It ensures that the function’s program-counter label $pc'$ accurately summarizes the authority levels of the references contained in the closure, and that the $pc$ is high enough to create this closure. The body is checked with program-counter label $pc'$, so in rule T-APP, the function can only be used by code with sufficient integrity.

Rule T-REC checks the creation of records. It requires that the annotation $S$ be well-formed. Also, 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 $\vdash \text{auth}^+(r) \leq pc$ 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 tint 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 $\vdash \text{auth}^+(r) \leq pc$).
The rules for determining the well-formedness of types are given in Figure 8. In rule WT6, a reference type \((\{x_i : \tau_i\}(a^+, a^-, p))\) is well-formed only if the upper authority label \(a^+= \) 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. To ensure hosts are able to create hard references to the objects they store, we also require \(\text{auth}^+(r)\) 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 \(\lambda_{\text{persist}}\) with persistence-failure handlers to obtain the full \(\lambda_{\text{persist}}\) language (see Appendix A.1 for its full syntax). The type system of \(\lambda_{\text{persist}}\) forces the programmer to be aware of and to handle all potential failures.

One approach is to handle the failure immediately upon using a broken reference. However, because low-persistence references may be used frequently, this would result in lots of duplicated failure-handling code.

Instead, \(\lambda_{\text{persist}}\) factors out failure-handling code from ordinary code by treating failures as a kind of exception. The value of \((\text{try} \; e_1 \; \text{catch} \; p : \; e_2)\) 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. 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.

Formally, \(\mathcal{H}\) is drawn from the bounded meet-semilattice given by the upper powerdomain [22] of persistence levels. The elements of the powerdomain are the finitely generated subsets\(^3\) of \(\mathcal{L}\), modulo equivalence relation (1) below. The ordering on these elements is given by (2). If we choose maximal sets to represent the equivalence classes, then the meet operation is set union.

\[
A \sim B \iff \forall \ell \in \mathcal{L}. (\exists a \in A. a \preceq \ell) \leftrightarrow (\exists b \in B. b \preceq \ell) \tag{1}
\]

\[
A \preceq_p B \iff \forall b \in B. \exists a \in A. a \preceq b \tag{2}
\]

\(^3\)Non-empty subsets that either are finite or contain \(\perp\).
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.

```javascript
let pad = docs.scratchpad
in try pad.add "Rodin Museum" catch 1: ...
```

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 λPersist

The small-step operational semantics of λPersist extends that of λ0Persist with the rules in 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. Appendix A.2 gives the full operational semantics for λPersist.

The subtyping rules are the same as for λ0Persist, except that function subtyping is also contravariant on the H component. Full subtyping rules are given in Appendix A.3.

The typing rules for λPersist extend those for λ0Persist. They augment the typing context with a handler environment H, indicating the set of persistence failures the evaluation context can handle. For an expression e that is well-typed in a context Γ;pc;H, typing judgments additionally produce an effect X, which is a set indicating the persistence failures that can occur during evaluation. The typing assertion Γ;pc;H ⊢ e : τ, X, therefore, means that the expression e has type τ and effect X under type assignment Γ, current program-counter label pc, and handler environment H.

The typing rules for λ0Persist are converted straightforwardly to thread H and X through typing judgments. Rules T-SEL and T-ASGN gain premises to ensure the context has a suitable handler in case dereferences fail. Appendix A.4 gives the full set of 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 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 H when checking e1. The value w in the typing rule is a conservative summary of the persistence errors that can occur while evaluating e1 and are not handled by the p-persistence handler. Because evaluation of e2 depends on the result of e1, the pc label for evaluating e2 is tainted by w. In this rule, the notation X/p denotes the subset of persistence errors X
not handled by \( p \). Formally, this is defined as follows.

\[
\mathcal{H} / p = \{ p' \in \mathcal{H} : p \not\prec p' \}
\]

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, \( \lambda_{\text{persist}} \) is also augmented to include bracketed expressions, resulting in the language \( [\lambda_{\text{persist}}] \). 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.)

Figure 10: Small-step operational semantics extensions for ordinary execution of \( [\lambda_{\text{persist}}] \)

The extended syntax and the rule for typing bracketed terms appear below. Extensions for the operational semantics appear in Figure 10.
\[ \alpha \text{-CREATE} \]
\[
\begin{array}{c}
m = \text{newloc}(M) \\
\frac{\alpha \vdash_{\text{wf}} M[m^S \Rightarrow \{x_i = [v_i]\}] \quad \alpha \not\in \text{persist}(S)}{\langle e, M \rangle \rightarrow_{\alpha} \left\{ e, M[m^S \Rightarrow \{x_i = [v_i]\}] \right\}}
\end{array}
\]

\[ m^S \in \text{dom}(M) \quad \alpha \not\in \text{persist}(S) \]
\[
\frac{S = (x_i : \tau_i), \quad \emptyset ; \tau ; \tau \vdash [v] : \tau_i, \tau}{\langle e, M \rangle \rightarrow_{\alpha} \left\{ e, M[m^S \Rightarrow [v]] \right\}}
\]

\[ \alpha \text{-ASSIGN} \]
\[
\frac{\alpha \vdash_{\text{wf}} M[m^S \Rightarrow [x_i = [v_i]]]}{\langle e, M \rangle \rightarrow_{\alpha} \left\{ e, M[m^S \Rightarrow [v_i]] \right\}}
\]

\[ \alpha \text{-FORGET} \]
\[
\frac{\alpha \not\in \text{persist}(S)}{\langle e, M \rangle \rightarrow_{\alpha} \left\{ e, M[m^S \Rightarrow [v]] \right\}}
\]

Figure 11: Effects caused by the $\alpha$-adversary

auto-bracketing function $e \bowtie_{\alpha} \ell$, which is formally defined below. The notation $e \bowtie_{\alpha} \tau$ is shorthand for $e \bowtie_{\alpha} \integ(\tau)$.

\[ e \bowtie_{\alpha} \ell = \begin{cases} e, & \text{if } \alpha \leq \ell \text{ or } \exists e'. \ e = [e']; \\ [e], & \text{otherwise.} \end{cases} \]

Also, rules TRY-VAL and LET are amended to prevent a transition when $v$ is bracketed, and rules that disallow transitions on bracketed bottom values are amended to prevent transitions on bracketed bottom values.

The adversary is powerful, as shown by the transitions defined in Figure 11. Adversaries may create new records, modify existing records, or remove records from memory altogether, but their ability is bounded by an integrity label $\alpha \in \mathcal{L}$. Such an $\alpha$-adversary has all creation authority except $\alpha$ and higher, can modify any record field except those with $\alpha$ (or higher) integrity, and can delete any record except those with $\alpha$ (or higher) persistence. A small evaluation step taken in the presence of an $\alpha$-adversary is a transition from a machine configuration $\langle e, M \rangle$ to another configuration $\langle e', M' \rangle$, written $\langle e, M \rangle \rightarrow_{\alpha} \langle e', M' \rangle$.

It is important to know that any evaluation of a program in the original language can be simulated in the amended 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 (see Lemma 1 in Section 8).

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.

Rule $\alpha$-CREATE lets the adversary create records at new memory locations. The premise $\emptyset ; \tau ; \tau \vdash \{x_i = [v_i]\}^S : R\tau$, $\tau$ ensures that the records are well-typed values and that new hard references satisfy the restrictions on the adversary. The premise $\alpha \vdash_{\text{wf}} M[m^S \Rightarrow \{x_i = [v_i]\}]$ ensures that the resulting memory is well-formed (formally defined in Section 8.1), so the adversary cannot create references to unknown memory locations.

Rule $\alpha$-ASSIGN lets the adversary modify existing records. The premise $M(m^S) \not= \bot$ ensures that the record being modified still exists in memory. The premise $\emptyset ; \tau ; \tau \vdash [v] : \tau_i$, $\tau$ ensures that the assignment is well-typed and that new hard references satisfy the restrictions on the adversary. The premise $\alpha \vdash_{\text{wf}} M[m^S, x_i \mapsto [v]]$ ensures that the resulting memory is well-formed, so the adversary cannot create references to unknown memory locations.

Rule $\alpha$-FORGET lets the adversary drop records from memory. The premise $\alpha \not\in \text{persist}(S)$ restricts the persistence level of dropped records.

### 8 Results

The goal of $\lambda_{\text{persist}}$ 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 prove that $\lambda_{\text{persist}}$ satisfies these properties.

#### 8.1 Well-formedness

A well-formed $\lambda_{\text{persist}}$ memory $M$, written $\vdash_{\text{wf}} M$, maps typed locations to record values with the same type. This is defined formally as follows.
Definition 3 (Well-formed \(\lambda_{\text{persist}}\) memory). A \(\lambda_{\text{persist}}\) memory \(M\) is well-formed, written \(\vdash_{\text{wf}} M\), if each record stored in \(M\) satisfies two conditions: the record’s type corresponds to the type of the record’s location in \(M\), and every location mentioned in the record is valid in \(M\).

More precisely, whenever \(M\) maps a reference \(m^S\) to a record value \(\{x_i = v_i\}\),

- \(S\) is well-formed,
- each \(v_i\) has the appropriate type (as specified by \(S\)),
- the locations mentioned in the field values \(v_i\) are in the domain of \(M\):

\[
\vdash_{\text{wf}} M \iff (M(m^S)) = \{x_i = v_i\} \land S = \{x_i : \tau_i\}_{(a,p)} \Rightarrow \vdash_{\text{wf}} S : \text{rectype} \land (\forall i. \emptyset : T \vdash v_i : \tau_i, T) \land \text{loc}(\{x_i = v_i\}) \subseteq \text{dom}(M)
\]

Our notion of a well-formed configuration relies on knowing when a location is noncollectible (cannot be garbage collected), which we now define.

Definition 4 (Noncollectible locations). A memory location \(m^S\) is noncollectible in a configuration \((e,M)\), written \(\text{nc}(m^S, (e,M))\), if it is reachable from a GC root of \(e\) through a path of hard references.

This is defined formally by the following induction rules:

\[
\begin{align*}
\text{root}(m^S,e) & \quad \text{nc}(m^S, (e,M)) \\
\text{nc}(m^S_1,e) & \quad \text{M}(m^S_1) = \{x_i = v_i\} \quad \exists c. \text{nc}(m^S_1, (v_c,M))
\end{align*}
\]

Well-formedness of configurations, defined below, is parameterized on an adversary \(\alpha\).

Definition 5 (Well-formed \(\lambda_{\text{persist}}\) configuration). A \(\lambda_{\text{persist}}\) configuration \((e,M)\) is well-formed, written \(\vdash_{\text{wf}} \alpha (e,M)\), if the following all hold:

- \(M\) is well-formed;
- the locations mentioned in \(e\) are valid in \(M\);
- no noncollectible high-persistence location is deleted; and
- if \(G\) is a minimal collectible group in which a high-persistence location is deleted, then all locations in \(G\) are also deleted.

Formally,

\[
\vdash_{\text{wf}} \alpha (e,M) \equiv (M \land \text{loc}(e) \subseteq \text{dom}(M)) \land (\forall m^S. \text{nc}(m^S, (e,M)) \land \vdash_{\text{wf}} M \land \alpha \leq \text{persist}(S)) \Rightarrow M(m^S) = \bot \land (\forall G. \text{gc}(G, (e,M)) \land (\exists G' \subseteq G. \text{gc}(G', (e,M))) \land (\exists m^S_{0} \in G. \vdash_{\text{wf}} M(m^S_{0}) = \bot) \Rightarrow \forall m^S \in G. M(m^S) = \bot)
\]

A \(\lambda_{\text{persist}}\) configuration is well-formed in a nonadversarial setting, written \(\vdash_{\text{wf}} (e,M)\), if it is well-formed with respect to the \(\bot\) adversary.

Corresponding well-formedness conditions are defined similarly for \([\lambda_{\text{persist}}]\) memories (written \(\vdash_{\text{wf}} M\) and \(\vdash_{\text{wf}} M_{\bot}\)) and for \([\lambda_{\text{persist}}]\) configurations (written \(\vdash_{\text{wf}} (e,M)\) and \(\vdash_{\text{wf}} (e,M)_{\bot}\)). Well-formedness of \([\lambda_{\text{persist}}]\) memories is parameterized on an \(\alpha\)-adversary, because values appearing in low-integrity record fields must be bracketed.
Definition 6 (Well-formed \([\lambda_{\text{persist}}]\) memory). A \([\lambda_{\text{persist}}]\) memory \(M\) is well-formed with respect to an adversary \(\alpha\) (written \(\vdash_{\alpha} M\)) if it is a well-formed \(\lambda_{\text{persist}}\) memory and all low-integrity field values are bracketed:

\[
\vdash_{\alpha} M \iff \vdash_{\text{wf}} M \land (M(m^S) = \{x_i = \bar{v}_i\} \land S = \{x_i: \tau_i\}, \land \alpha \notin \text{integ}(\tau_i) \\
\implies \exists v'. v_i = [v'])
\]

Definition 7 (Well-formed \([\lambda_{\text{persist}}]\) configuration). A \([\lambda_{\text{persist}}]\) configuration \(\langle e, M \rangle\) is well-formed if it is a well-formed \(\lambda_{\text{persist}}\) configuration with a well-formed \([\lambda_{\text{persist}}]\) memory. A configuration is well-formed in a nonadversarial setting if it is well-formed in the presence of a ⊥ adversary.

\[
\vdash_{\alpha} \langle e, M \rangle \iff \vdash_{\text{wf}} \langle e, M \rangle \land \vdash_{\alpha} \langle e, M \rangle
\]

8.2 Completeness of \([\lambda_{\text{persist}}]\) evaluation

For \([\lambda_{\text{persist}}]\) to be adequate for reasoning about referential security properties of \(\lambda_{\text{persist}}\), we must be able to simulate any \(\lambda_{\text{persist}}\) execution in \([\lambda_{\text{persist}}]\). This is a completeness property. To do this, we first need to define what it means for a \([\lambda_{\text{persist}}]\) configuration to simulate a \(\lambda_{\text{persist}}\) configuration. This involves defining a correspondence on expressions and heaps between the two languages. We write \(e_1 \leq e_2\) to denote that the \(\lambda_{\text{persist}}\) term \(e_1\) corresponds to the \([\lambda_{\text{persist}}]\) term \(e_2\). The terms correspond if erasing brackets from \(e_2\) produces \(e_1\).

Definition 8 (Correspondence between \([\lambda_{\text{persist}}]\) and \(\lambda_{\text{persist}}\) expressions). A \([\lambda_{\text{persist}}]\) expression \(e\) is related to a \(\lambda_{\text{persist}}\) expression \(e'\), written \(\vdash e \leq e'\), if the two are equal when brackets in \(e\) are removed. This is formally defined by the following induction rules.

This correspondence induces one on heaps. Together, the two give the correspondence on configurations.

Definition 9 (Correspondence between \([\lambda_{\text{persist}}]\) and \(\lambda_{\text{persist}}\) memories). A \([\lambda_{\text{persist}}]\) memory \(M_1\) is related to a \(\lambda_{\text{persist}}\) memory \(M_2\), written \(\vdash M_1 \leq M_2\), if they map the same set of locations, and the memories map each location to values that are related.

\[
\vdash M_1 \leq M_2 \iff \text{dom}(M_1) = \text{dom}(M_2) \\
\land \forall m^S \in \text{dom}(M_1). M_1(m^S) = M_2(m^S) = ⊥ \\
\lor \vdash M_1(m^S) \leq M_2(m^S)
\]

Definition 10 (Correspondence between \([\lambda_{\text{persist}}]\) and \(\lambda_{\text{persist}}\) configurations). A \([\lambda_{\text{persist}}]\) configuration \(\langle e_1, M_1 \rangle\) is related to a \(\lambda_{\text{persist}}\) configuration \(\langle e_2, M_2 \rangle\), written \(\vdash \langle e_1, M_1 \rangle \leq \langle e_2, M_2 \rangle\), if both the expressions and the memories are related:

\[
\vdash \langle e_1, M_1 \rangle \leq \langle e_2, M_2 \rangle \iff \vdash e_1 \leq e_2 \land \vdash M_1 \leq M_2
\]

We can now state and prove the completeness of \([\lambda_{\text{persist}}]\).
Lemma 1 (Completeness of $\left[\lambda_{\text{persist}}\right]$ with respect to $\lambda_{\text{persist}}$). Let $\langle e_1, M_1 \rangle$ be a well-formed $\lambda_{\text{persist}}$ configuration with $e_1$ well-typed. Let $\langle e_2, M_2 \rangle$ be a well-formed $\left[\lambda_{\text{persist}}\right]$ configuration corresponding to $\langle e_1, M_1 \rangle$, with $e_2$ well-typed. If $\langle e_1, M_1 \rangle$ takes a $\rightarrow$ transition to $\langle e_1', M_1' \rangle$, then there exists a configuration $\langle e_2', M_2' \rangle$ corresponding to $\langle e_1', M_1' \rangle$ such that $\langle e_2, M_2 \rangle \rightarrow_\alpha \langle e_2', M_2' \rangle$.

\[ \vdash^\alpha_\rho \langle e_1, M_1 \rangle \land \emptyset; \rho; H \vdash e_1 : \tau, \chi \]
\[ \land \vdash^\alpha_\rho \langle e_2, M_2 \rangle \land \emptyset; \rho; H \vdash e_2 : \tau, \chi' \]
\[ \land \vdash \langle e_1, M_1 \rangle \triangleq \langle e_2, M_2 \rangle \land \langle e_1, M_1 \rangle \rightarrow \langle e_1', M_1' \rangle \]
\[ \Rightarrow \exists e_2', M_2'. \langle e_2, M_2 \rangle \rightarrow^* \langle e_2', M_2' \rangle \land \langle e_1', M_1' \rangle \triangleq \langle e_2', M_2' \rangle \]

Proof. Induction on the derivation of $\langle e_1, M_1 \rangle \rightarrow_\alpha \langle e_1', M_1' \rangle$. □

8.3 Soundness of $\left[\lambda_{\text{persist}}\right]$ type system

We prove the type system sound with the usual method, via type preservation (Lemma 9) and progress (Lemma 13). Because we are only concerned with well-formed configurations, it is important to know that they are preserved by the operational semantics. This is captured by Lemma 10.

We first show some preliminary results for weakening the typing context.

Lemma 2 (Type-environment weakening). Extra type assumptions can be safely added to typing contexts: $\Gamma; \rho; H \vdash e : \tau, \chi \land x \not\in FV(e) \Rightarrow \Gamma, x : \tau' ; \rho; H \vdash e : \tau, \chi$.

Proof. By induction on the derivation of $\Gamma; \rho; H \vdash e : \tau, \chi$. □

Lemma 3 (pc weakening). The pc label in a typing context can be safely raised.

\[ \vdash \rho \equiv \rho' \land \Gamma; \rho; H \vdash e : \tau, \chi \Rightarrow \Gamma; \rho'; H \vdash e : \tau, \chi \]

Proof. By induction on the derivation of $\Gamma; \rho; H \vdash e : \tau, \chi$.

Rules T-BOOL, T-UNIT, T-LOC, T-BOTTOM, and T-VAR are trivial base cases. Rule T-ABS follows from the transitivity of $\equiv$. Rules T-SOFT, T-PARALLEL, and T-SUBSUME follow from the induction hypothesis.

Rules T-RECORD, T-SELECT, T-SOFT-SELECT, T-ASSIGN, T-SOFT-ASSIGN, T-EXISTS, T-APP, T-LET, T-TRY, T-IF, and T-BRACKET follow from the induction hypothesis and the transitivity of $\equiv$. In rules T-ASSIGN and T-SOFT-ASSIGN, we need to show $\vdash \tau \triangleq \rho \land p \leq \tau_c$. By assumption, we have $\vdash \tau \triangleq \rho \land p \leq \tau_c$. Let $b_w = \tau$. Then using S3, we can show
\[ \vdash b \leq b \quad \vdash w \land \rho \land p \leq w \land \rho \land p \leq \tau_c \quad \vdash \tau \land \rho \land p \leq \tau \land p \land p \leq \tau \land p \land p \leq \tau_c , \]
and the result follows from transitivity of subtyping. □

Lemma 4 (Handler weakening). Extra handler assumptions can be safely added to the typing context.

\[ \vdash H' \equiv H \land \Gamma; \rho; H \vdash e : \tau, \chi \Rightarrow \Gamma; \rho; H' \vdash e : \tau, \chi \]

Proof. By induction on the derivation of $\Gamma; \rho; H \vdash e : \tau, \chi$.

Rules T-BOOL, T-UNIT, T-LOC, T-BOTTOM, T-VAR, T-ABS, and T-PARALLEL are trivial base cases.


Rule T-TRY follows from the induction hypothesis and the fact that $\forall \cup \{ p \} \equiv H \lor \{ p \}$. □

Corollary 5 summarizes these results.

Corollary 5 (Context weakening). Suppose $\Gamma; \rho; H \vdash e : \tau, \chi$ with $x \not\in FV(e)$, $\rho \equiv \rho'$ and $H' \equiv H$. Then
\[ \Gamma, x : \tau' ; \rho; H' \vdash e : \tau, \chi \]

We can now prove the substitution lemma.
Lemma 6 (Substitution). $\Gamma, x: \tau'; pc; H \vdash e : \tau, X \land \emptyset; pc; H \vdash v : \tau', \tau \Rightarrow \Gamma; pc; H \vdash e\{v/x\} : \tau, X$.

Proof. By induction on the derivation of $\Gamma, x: \tau'; pc; H \vdash e : \tau, X$. Note that since $v$ was typed in an empty type assignment ($\Gamma = \emptyset$), we must have $\text{FV}(v) = \emptyset$.

Rules T-BOOL, T-UNIT, T-LOC, and T-BOTTOM are trivial base cases.


Case T-VAR:

Suppose $e = x$. Then $e\{v/x\} = v$ and $\tau' = \tau$ and the result holds in this case via Corollary 5 and T-SUBSUME. Alternatively, suppose $e = y \neq x$. In this case, $e\{v/x\} = e$ and the result holds trivially.

Case T-ABS ($e = \lambda(y: \tau_1)[pc_1; \hat{H}_1]. e_1$):

If $y = x$, then $e\{v/x\} = e$ and the result holds trivially. Alternatively, suppose $y \neq x$. We have $\text{FV}(v) = \emptyset$, so $e\{v/x\} = \lambda(y: \tau_1)[pc_1; \hat{H}_1]. e_1\{v/x\}$. Let $\Gamma' = \Gamma, x: \tau'$. From the typing of $e$, we have

$$
\begin{align*}
\Gamma, x: \tau'; pc; H &\vdash e_1 : \tau_2, \hat{H}_1 \\
\Gamma, x: \tau'; \hat{H} &\vdash \lambda(y: \tau_1)[pc_1; \hat{H}_1]. e_1 : \tau_1 \to (pc_1 \leq pc) \\
\Gamma, x: \tau'; pc; H &\vdash e_1\{v/x\} : \tau_2, \hat{H}_1
\end{align*}
$$

where $\tau = (pc_1, \hat{H}_1, \tau_2), \text{ and } X = \tau$. So, we know $\Gamma, x: \tau'; y: \tau_1; pc_1; \hat{H}_1 \vdash e_1 : \tau_2, \hat{H}_1$. Therefore, by the induction hypothesis, we have $\Gamma, y: \tau_1; pc_1; \hat{H}_1 \vdash e_1\{v/x\} : \tau_2, \hat{H}_1$. So the result holds in this case via an application of T-ABS:

$$
\begin{align*}
\Gamma, y: \tau_1; pc_1; \hat{H}_1 &\vdash e_1\{v/x\} : \tau_2, \hat{H}_1 \\
\Gamma, y: \tau_1; \hat{H} &\vdash \lambda(y: \tau_1)[pc_1; \hat{H}_1]. e_1 : (\tau_1 \to (pc_1 \leq pc)) \\
\Gamma, y: \tau_1; pc_1; \hat{H}_1 &\vdash \lambda(y: \tau_1)[pc_1; \hat{H}_1]. e_1\{v/x\} : (\tau_1 \to (pc_1 \leq pc))
\end{align*}
$$

Case T-EXISTS (exists $u$ as $y$: $e_1$ else $e_2$):

From the typing of $e$, we have

$$
\begin{align*}
\Gamma, x: \tau'; pc; H &\vdash u : (\text{soft } \{\bar{x}_i : \tau_i\})_w, \tau \\
\Gamma, x: \tau'; y: (\{\bar{x}_i : \tau_i\})_w; pc; H &\vdash e_1 : \tau'', X_1 \\
\Gamma, x: \tau'; pc; H &\vdash e_2 : \tau'', X_2 \\
\Gamma, x: \tau'; pc; H &\vdash \exists u \text{ as } y : (e_1 \text{ else } e_2) : \tau'' \land w', X_1 \land X_2
\end{align*}
$$

where $pc' = pc \land w'$, $\tau = \tau'' \land w'$, and $X = X_1 \land X_2$. By the induction hypothesis, we therefore have

$$
\begin{align*}
\Gamma; pc; H &\vdash u\{v/x\} : (\text{soft } \{\bar{x}_i : \tau_i\})_w, \tau \\
\Gamma; pc; H &\vdash e_1\{v/x\} : \tau'', X_1 \\
\Gamma; pc; H &\vdash e_2\{v/x\} : \tau'', X_2 \\
\Gamma; pc; H &\vdash \exists u\{v/x\} \text{ as } y : (e_1 \text{ else } e_2)\{v/x\} : \tau'' \land w', X_1 \land X_2
\end{align*}
$$

Suppose $y = x$. Then $e\{v/x\} = \exists u\{v/x\}$ as $y : e_1$ else $e_2\{v/x\}$, so from (3) and (4), the result holds in this case via an application of T-EXISTS:

$$
\begin{align*}
\Gamma; pc; H &\vdash u\{v/x\} : (\text{soft } \{\bar{x}_i : \tau_i\})_w, \tau \\
\Gamma, y: (\{\bar{x}_i : \tau_i\})_w; pc; H &\vdash e_1 : \tau'', X_1 \\
\Gamma; pc; H &\vdash e_2\{v/x\} : \tau'', X_2 \\
\Gamma; pc; H &\vdash \exists u\{v/x\} \text{ as } y : (e_1 \text{ else } e_2)\{v/x\} : \tau'' \land w', X_1 \land X_2
\end{align*}
$$

Alternatively, suppose $y \neq x$. We have $\text{FV}(v) = \emptyset$, so $e\{v/x\} = \exists u\{v/x\}$ as $y : e_1\{v/x\}$ else $e_2\{v/x\}$. From the typing of $e$, we know $\Gamma, x: \tau'; y: (\{\bar{x}_i : \tau_i\})_w; pc; H \vdash e_1 : \tau'', X_1$. By the induction hypothesis, we have $\Gamma, y: (\{\bar{x}_i : \tau_i\})_w; pc; H \vdash e_1\{v/x\} : \tau'', X_1$. From this, (3), and (4), the result holds in this case via an application of T-EXISTS:

$$
\begin{align*}
\Gamma; pc; H &\vdash u\{v/x\} : (\text{soft } \{\bar{x}_i : \tau_i\})_w, \tau \\
\Gamma, y: (\{\bar{x}_i : \tau_i\})_w; pc; H &\vdash e_1\{v/x\} : \tau'', X_1 \\
\Gamma; pc; H &\vdash e_2\{v/x\} : \tau'', X_2 \\
\Gamma; pc; H &\vdash \exists u\{v/x\} \text{ as } y : e_1\{v/x\} \text{ else } e_2\{v/x\} : \tau'' \land w', X_1 \land X_2
\end{align*}
$$
Lemma 8 (Effect bound). The effect of a well-typed expression is bounded from below by its handler environment.

\[ \Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \Rightarrow \mathcal{H} \leq \mathcal{X} \]

Proof. By induction on the derivation of \( \Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \). Rule T-TRY relies on the easily proved fact that if \( \mathcal{H} \cup \{ p \} \leq \mathcal{X}_1 \), then \( \mathcal{H} \leq \mathcal{X}_1/p \). □

Lemma 8 (Value typing). The handler environment is irrelevant for typing non-bottom values. Such a value \( v \) can also be typed with any \( pc \) that has the authority of the references that appear in \( v \), and can have any effect bounded from below by the handler environment. Formally,

\[ \Gamma; pc; \mathcal{H} \vdash v : \tau, \mathcal{X} \land (\forall p. \; v \neq \bot_p \land v \neq [\bot_p]) \]
\[ \Rightarrow \Gamma; pc; \mathcal{H}' \vdash v : \tau, \mathcal{X}' \]

Proof. By induction on the derivation of \( \Gamma; pc; \mathcal{H} \vdash v : \tau, \mathcal{X} \). □

We are now ready to prove type preservation for \( \lambda_{persis} \).

Lemma 9 (Type preservation). Let \( M \) be a well-formed memory. Let \( e \) be an expression with type \( \tau \) and effect \( \mathcal{X} \). Let \( \alpha \in L \). If the configuration \( (e, M) \) takes an \( \Rightarrow_{\alpha} \) transition, then the new expression \( e' \) will also have type \( \tau \) and effect \( \mathcal{X} \):

\[ \Gamma; \Rightarrow_{\alpha}^{q} M \land \emptyset; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \]
\[ \Rightarrow \Gamma; \emptyset; pc; \mathcal{H} \vdash e' : \tau, \mathcal{X} \]

Proof. By induction on the derivation of \( \emptyset; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \).

Given \( (e, M) \Rightarrow_{\alpha} (e', M') \), the proof proceeds by cases according to the evaluation rules. For cases GC, \( \alpha\)-CREATE, \( \alpha\)-ASSIGN, and \( \alpha\)-FORGET, we have \( e = e' \), so the result follows trivially.

By Lemma 7, we have \( \mathcal{H} \leq \mathcal{X} \).
Case \textsc{create} \(\{(x_i = v_i) \in S, M\} \rightarrow_a \{m^5, M[m^5 \rightarrow \{(x_i = v_i')\}]\}\), where \(m\) is fresh, \(S = \{(x_i = v_i)\}_{(a,p)}\), and \(v_i' = v_i \triangleright_a x_i\):

We have \(\emptyset; pc; H \vdash (x_i = v_i) \in S, R, \top\) with \(R = \{(x_i = v_i)\}_{(a,p)}\) and \(\vdash_{nf} S : \text{rectype}\). We need to show \(\emptyset; pc; H \vdash m^5 : R, \top\). This is given by trivial application of T-Loc.

Case \textsc{apply} \(\{(\lambda(x) : \tau_1)(pc_1, H_1), e_1\} \rightarrow_a \{e_1 \{v/x\}, M\}\):

From the typing of \(e\), we have

\[
\emptyset; pc; H \vdash \lambda(x : \tau_1)(pc_1, H_1); e_1 : (\tau_1 \rightarrow (pc_1, H_1))_{\top, \top} \tag{6}
\]

and

\[
\emptyset; pc; H \vdash v : \tau_1, \top \tag{7}
\]

with \(\vdash \vdash H \leq H_1\) and \(H_1 = \mathcal{X}\). We need to show \(\emptyset; pc; H \vdash e_1 \{v/x\} : \tau, H_1\). From the derivation of (6), we know

\[
x : \tau_1; pc_1; H_1 \vdash e_1 : \tau, H_1
\]

with \(\vdash pc_1 \leq pc\). Applying Corollary 5 to this, we get

\[
x : \tau_1; pc; H \vdash e_1 : \tau, H_1.
\]

The result follows from this and (7) via Lemma 6.

Case \textsc{select} \(\{m^5, x_c, M\} \rightarrow_a \{v_c, \triangleright_a p, M\}\), where \(M(m^5) = \{(x_i = v_i)\}\) and \(S = \{(x_i = v_i)\}_{(a,p)}\):

We have \(\emptyset; pc; H \vdash m^5 : x_c \cap p, p\) and need to show \(\emptyset; pc; H \vdash v_c, \triangleright_a p : \tau_c \cap p, p\).

From the typing of \(e\), we know \(\emptyset; pc; H \vdash m^5 : (\{(x_i = v_i)\})_{(a,a,p)} ; \top\) and \(\vdash a \leq pc\) with \(\vdash_{nf} S : \text{rectype}\). Therefore, the following holds:

\[
\vdash \vdash \text{auth}^+ (\tau_c) \leq a \leq pc\text{ and}
\]

\[
\vdash \vdash \text{auth}^+ (\tau_c) \leq p.
\]

So, we know \(\vdash \vdash \text{auth}^+ (\tau_c) \leq pc \cap p\).

By Lemma 7, we have \(\vdash \vdash H \leq p\). Since \(M\) is well-formed, we also know that \(\emptyset; \top, \top \vdash v_c : \tau_c, \top\), so by Lemma 8, we have

\[
\emptyset; pc \cap p; H \vdash v_c : \tau_c, p. \tag{8}
\]

Suppose \(\vdash \vdash \alpha \leq p\) or \(v_c\) is bracketed. Then \(v_c, \triangleright_a p = v_c\) and the result follows from (8) via Corollary 5 and T-Subsume.

Otherwise, \(\alpha \nleq p\) and \(v_c\) is unbracketed. So \(v_c, \triangleright_a p = [v_c]\), and the result follows via T-Bracket:

\[
\frac{\emptyset; pc \cap p; H \vdash v_c : \tau_c, p \quad \alpha \nleq p}{\emptyset; pc; H \vdash [v_c] : \tau_c \cap p, p}.
\]

Case \textsc{dangle-select} \(\{m^5, x_c, M\} \rightarrow_a \{\bot, \triangleright_a p, M\}\), where \(M(m^5) = \bot\) and \(S = \{(x_i = v_i)\}_{(a,p)}\):

We have \(\emptyset; pc; H \vdash m^5 : x_c \cap p, p\) and need to show \(\emptyset; pc; H \vdash \bot, \triangleright_a p : \tau_c \cap p, p\).

From the typing of \(e\), we know \(\emptyset; pc; H \vdash m^5 : (\{(x_i = v_i)\})_{(a,a,p)} ; \top\) and \(\vdash a \leq pc\) with \(\vdash_{nf} S : \text{rectype}\). Therefore, the following holds:

\[
\vdash \vdash \text{auth}^+ (\tau_c) \leq a \leq pc\text{ and}
\]

\[
\vdash \vdash \text{auth}^+ (\tau_c) \leq p.
\]
So, we know $\vdash \text{auth}^+(\tau_c) \leq pc \cap p$.

By Lemma 7, we know $\vdash \mathcal{H} \leq p$, so by T-BOTTOM, we have

$$\emptyset;pc \cap p; \mathcal{H} \vdash \bot_p: \tau_c,p.$$  (9)

Suppose $\vdash \alpha \leq p$. Then $\bot_p \bullet_a p = \bot_p$ and the result follows from (8) via Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\vdash p$. So $\bot_p \bullet_a p = [\bot_p]$, and the result follows via T-BRACKET:

$$\emptyset;pc \cap p; \mathcal{H} \vdash \bot_p: \tau_c,p \quad \alpha \not\vdash p \quad \vdash \text{auth}^+(\tau_c) \leq pc \cap p$$

$$\emptyset;pc; \mathcal{H} \vdash [\bot_p]: \tau_c \cap p,p$$

Case SOFT-SELECT ($(\text{soft } m^3)(x_c,M) \rightarrow_{\alpha} (v \bullet_a (a \cap n)(p),M)$, where $S = \{x_i: \tau_j\}_{(a,p)}$ and $(m^3)(x_c,M) \rightarrow (v,M)$):

From the typing of $e$, we have $\emptyset;pc; \mathcal{H} \vdash (\text{soft } m^3)(x_c: \tau_c \cap p',p)$ and $\vdash \text{auth}^+(\tau_c) \leq pc$, where $p' = a \cap n$. So $\tau_c = \tau_c \cap p'$ and $X' = p$. We need to show $\emptyset;pc; \mathcal{H} \vdash v \bullet_a p' = \tau_c \cap p',p$.

From the typing of $e$, it follows that $\vdash_{\text{off } S} \text{rectype}$, and therefore, we know $\vdash \text{auth}^+(\tau_c) \leq p'$. Since we also know $\vdash \text{auth}^+(\tau_c) \leq pc$, we therefore have $\vdash \text{auth}^+(\tau_c) \leq pc \cap p'$.

We proceed by cases according to the evaluation rules for $(m^3)(x_c,M) \rightarrow (v,M)$.

Sub-case SELECT ($(m^3)(x_c,M) \rightarrow (v \bullet_a p,M)$, where $M(m^3) = \{x_i = v_i\}$):

We have $v = v_c \bullet_a p$.

By Lemma 7, we have $\vdash \mathcal{H} \leq p$. Since $M$ is well-formed, we also know that $\emptyset;\top;\top \vdash v_c: \tau_c;\top$, so by Lemma 8, we have

$$\emptyset;pc \cap p'; \mathcal{H} \vdash v_c: \tau_c,p.$$  (10)

Suppose $\vdash \alpha \leq p'$. Then $\vdash \alpha \leq pc$, so $v \bullet_a p' = v = v_c \bullet_a p = v_c$. Similarly, if $v_c$ is bracketed, then $v \bullet_a p' = v_c$.

In these cases, the result follows from (10) via Corollary 5 and T-SUBSUME.

Otherwise, $\alpha \not\vdash p'$ and $v_c$ is unbracketed. So $v \bullet_a p' = [\bot_p]$ and the result follows via T-BRACKET:

$$\emptyset;pc \cap p'; \mathcal{H} \vdash v_c: \tau_c,p \quad \alpha \not\vdash p' \quad \vdash \text{auth}^+(\tau_c) \leq pc \cap p'$$

$$\emptyset;pc; \mathcal{H} \vdash [\bot_p]: \tau_c \cap p',p$$

Sub-case DANGLE-SELECT ($(m^3)(x_c,M) \rightarrow (\bot_p \bullet_a p,M)$):

We have $v = \bot_p \bullet_a p$.

By Lemma 7, we know $\vdash \mathcal{H} \leq p$, and we have $\vdash p' \leq p$ by definition, so by T-BOTTOM, we have $\emptyset;pc \cap p'; \mathcal{H} \vdash \bot_p: \tau_c \cap p',p$.

Suppose $\vdash \alpha \leq p'$. Then $\alpha \leq pc$, so $v \bullet_a p' = v = \bot_p \bullet_a p = \bot_p$, and the result follows via Corollary 5.

Otherwise, $\alpha \not\vdash p'$, so $v \bullet_a p' = [\bot_p]$, and the result follows via T-BRACKET:

$$\emptyset;pc \cap p'; \mathcal{H} \vdash \bot_p: \tau_c \cap p',p \quad \alpha \not\vdash p' \quad \vdash \text{auth}^+(\tau_c) \leq pc \cap p'$$

$$\emptyset;pc; \mathcal{H} \vdash [\bot_p]: \tau_c \cap p',p$$

Case ASSIGN ($(m^3)(x_c := v,M) \rightarrow (\ast \bullet_a p,M[m^3](x_c := v')]$, where $S = \{x_i: \tau_j\}_{(a,p)}$):

From the typing of $e$, we have $\emptyset;pc; \mathcal{H} \vdash m^3(x_c := v): 1,p$ and need to show $\emptyset;pc; \mathcal{H} \vdash \ast_a p : 1,p$. By Lemma 7, we have $\vdash \mathcal{H} \leq p$, so from T-UNIT and T-SUBSUME, we have $\emptyset;pc \cap p; \mathcal{H} \vdash \ast : 1,p$.

If $\vdash \alpha \leq p$, then $\ast \bullet_a p = \ast$, and the result follows by Corollary 5.

Otherwise, $\alpha \not\vdash p$ and $\ast_a p = [\ast]$, and the result follows via T-BRACKET.

Case DANGLE-ASSIGN ($(m^3)(x_c := v,M) \rightarrow (\bot_p \bullet_a p,M)$, where $S = \{x_i: \tau_j\}_{(a,p)}$):

From the typing of $e$, we have $\emptyset;pc; \mathcal{H} \vdash m^3(x_c := v): 1,p$ and need to show $\emptyset;pc; \mathcal{H} \vdash \bot_p \bullet_a p : 1,p$. By Lemma 7, we have $\vdash \mathcal{H} \leq p$, so by T-BOTTOM, we have $\emptyset;pc \cap p; \mathcal{H} \vdash \bot_p : 1,p$.

If $\vdash \alpha \leq p$, then $\bot_p \bullet_a p = \bot_p$, and the result follows by Corollary 5.

Otherwise, $\alpha \not\vdash p$ and $\bot_p \bullet_a p = [\bot_p]$, and the result follows via T-BRACKET.
Case \textbf{SOFT-ASSIGN} \((\langle (\text{soft } m^S).x_c := v.M \rangle \rightarrow_{\alpha} \langle v'.\alpha(a \cap p), M' \rangle)\), where \(\langle m^S.x_c := v.M \rangle \xrightarrow{\alpha} \langle v', M' \rangle\) and \(S = \{x_i : \tau_i\}_{(a,p)}\):

From the typing of \(e\), we have \(\emptyset;pc;H \vdash (\text{soft } m^S).x_c := v : 1.p\). By Lemma 7, we have \(\vdash H \not\preceq p\). Let \(p' = a \cap p\). We need to show \(\emptyset;pc;H \vdash v'.\alpha p' : 1.p\). We proceed by cases according to the evaluation rules for \(\langle m^S.x_c := v.M \rangle \xrightarrow{\alpha} \langle v', M' \rangle\).

Sub-case \textbf{ASSIGN} \((\langle m^S.x_c := v.M \rangle \xrightarrow{\alpha} \langle * \alpha p, M' \rangle)\):

We have \(v' = * \alpha p\).

If \(\vdash \alpha \not\preceq p'\), then \(\vdash \alpha \not\preceq p\), so \(v' = * \alpha p = \ast\). The result follows from T-UNIT and T-SUBSUME.

Otherwise, \(\alpha \not\not\preceq p'\), so \(v' = * \alpha p = \ast\). The result follows from T-UNIT, T-BRACKET, and T-SUBSUME.

Sub-case \textbf{DANGLE-ASSIGN} \((\langle m^S.x_c := v.M \rangle \xrightarrow{\alpha} \langle 1.p \alpha p, M' \rangle)\):

We have \(v' = 1.p \alpha p\).

If \(\vdash \alpha \not\preceq p'\), then \(\vdash \alpha \not\preceq p\), so \(v' = 1.p \alpha p = 1.p\). The result follows from T-BOTTOM.

Otherwise, \(\alpha \not\not\preceq p'\), so \(v' = 1.p \alpha p = 1.p\). The result follows from T-BOTTOM and T-BRACKET.

Case \textbf{EXISTS-TRUE} \((\{\text{exists soft } m^S \text{ as } x : e_1 \text{ else } e_2.M \rangle \rightarrow_{\alpha} \langle (e_1\{m^S/x\}) \bullet \alpha w, M \rangle, \text{ where } S = \{x_i : \tau_i\}_{(a,p)}\) and \(w = a \cap p\):

From the typing of \(e\), we have

\[
\emptyset;pc;H \vdash \text{soft } m^S : (\text{soft } \{x_i : \tau_i\}_{(a,p)}) \tau, \tau \quad \vdash \alpha \not\preceq pc \quad w = a \cap p
\]

\[
x : (\{x_i : \tau_i\}_{(a,p)}) \tau;pc \cap w;H \vdash e_1 : \tau', X_1 \quad \emptyset;pc \cap w;H \vdash e_2 : \tau' \cap w, X_2 \quad \vdash \text{auth}^\ast(\tau') \leq pc \cap w
\]

\[
\emptyset;pc;H \vdash \text{exists soft } m^S \text{ as } x : e_1 \text{ else } e_2 : \tau' \cap w, X_1 \cap X_2
\]

where \(\tau = \tau' \cap w\) and \(X = X_1 \cap X_2\). We need to show

\[
\emptyset;pc;H \vdash (e_1\{m^S/x\}) \bullet \alpha w : \tau, X.
\]

By T-\textbf{LOC}, we have \(\emptyset;pc \cap w;H \vdash m^S : (\{x_i : \tau_i\}_{(a,p)}) \tau, \tau\). Therefore, by Lemma 6, we know \(\emptyset;pc \cap w;H \vdash e_1\{m^S/x\} : \tau', X_1\). Also, by Lemma 7, we know \(\vdash H \not\preceq X_1 \cap X_2\), so by T-SUBSUME, we know

\[
\emptyset;pc \cap w;H \vdash e_1\{m^S/x\} : \tau, X.
\]

Suppose \(\vdash \alpha \not\preceq w\). Then \((e_1\{m^S/x\}) \bullet \alpha w = e_1\{m^S/x\}\). The result therefore follows by Corollary 5.

Otherwise, we have \(\alpha \not\not\preceq w\), so \((e_1\{m^S/x\}) \bullet \alpha w = [e_1\{m^S/x\}]\). The result therefore follows via T-BRACKET:

\[
\emptyset;pc \cap w;H \vdash e_1\{m^S/x\} : \tau' \cap w, X \quad \alpha \not\preceq w \quad \vdash \text{auth}^\ast(\tau') \leq pc \cap w
\]

\[
\emptyset;pc;H \vdash [e_1\{m^S/x\}] : \tau' \cap w, X
\]

Case \textbf{EXISTS-FALSE} \((\{\text{exists soft } m^S \text{ as } x : e_1 \text{ else } e_2.M \rangle \rightarrow_{\alpha} \langle e_2 \bullet \alpha w, M \rangle, \text{ where } S = \{x_i : \tau_i\}_{(a,p)}\) and \(w = a \cap p\):

From the typing of \(e\), we have

\[
\emptyset;pc;H \vdash \text{soft } m^S : (\text{soft } \{x_i : \tau_i\}_{(a,p)}) \tau, \tau \quad \vdash \alpha \not\preceq pc \quad w = a \cap p
\]

\[
x : (\{x_i : \tau_i\}_{(a,p)}) \tau;pc \cap w;H \vdash e_1 : \tau', X_1 \quad \emptyset;pc \cap w;H \vdash e_2 : \tau' \cap w, X_2 \quad \vdash \text{auth}^\ast(\tau') \leq pc \cap w
\]

\[
\emptyset;pc;H \vdash \text{exists soft } m^S \text{ as } x : e_1 \text{ else } e_2 : \tau' \cap w, X_1 \cap X_2
\]

where \(\tau = \tau' \cap w\) and \(X = X_1 \cap X_2\). We need to show \(\emptyset;pc;H \vdash e_2 \bullet \alpha w : \tau, X\).

Suppose \(\vdash \alpha \not\preceq w\). Then \(\emptyset;pc \vdash e_2 \bullet \alpha w = e_2\). The result follows from Corollary 5 and T-SUBSUME.

Otherwise, \(\alpha \not\not\preceq w\), so \(e_2 \bullet \alpha w = [e_2]\). By Lemma 7, we know \(\vdash H \not\preceq X_1 \cap X_2\). The result follows via T-BRACKET and T-SUBSUME:

\[
\emptyset;pc \cap w;H \vdash e_2 : \tau' \cap w, X_2 \quad \alpha \not\not\preceq w \quad \vdash \text{auth}^\ast(\tau') \leq pc \cap w
\]

\[
\emptyset;pc;H \vdash [e_2] : \tau' \cap w, X_1 \cap X_2
\]

\[
\emptyset;pc;H \vdash [e_2] : \tau' \cap w, X_1 \cap X_2 \quad \vdash H \not\preceq X_1 \cap X_2 \quad \vdash X_1 \cap X_2 \not\preceq X_2
\]
Case **TRY-VAL** \((\text{try } v \text{ catch } p : e_1, M) \to \alpha (v, M)\), where \(v \neq \bot, p\) and \(v \neq [\bot_p]\) for all \(p'):\n
From the typing of \(e\), we have

\[
\emptyset ; p_c ; \mathcal{H}, p \vdash v : \tau, \mathcal{X}_1,
\]

for some \(\mathcal{X}_1\). Since \(v\) is a non-bottom value, the result \(\emptyset ; p_c ; \mathcal{H} \vdash v : \tau, \mathcal{X}\) follows via Lemma 8.

Case **TRY-CATCH** \((\text{try } \bot_p \text{ catch } p' : e_2, M) \to \alpha (e_2, M)\), where \(\top p' \leq p\):

We have \(p' \leq p\), so from the typing of \(e\), we have

\[
\emptyset ; p_c \cap p \cap \text{integ}(\tau_1) ; \mathcal{H} \vdash e_2 : \tau_1, \mathcal{X},
\]

where \(\tau = \tau_1 \cap p\). We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash v : \tau, \mathcal{X}\). This follows from Corollary 5 and T-SUBSUME.

Case **TRY-ESC** \((\text{try } \bot_p \text{ catch } p' : e_2, M) \to \alpha (e_2, M)\), where \(p' \neq p\):

We have \(p' \neq p\), so from the typing of \(e\), it follows that \(\mathcal{H} \leq \mathcal{X} \leq p\). We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash \bot : \tau, \mathcal{X}\). This follows from T-BOTTOM and T-SUBSUME.

Case **PARALLEL-RESULT** \((\{v_1 \parallel v_2, M\} \to \alpha (*, M))\):

From the typing of \(e\), we have \(\tau = 1\) and \(\mathcal{X} = \tau\). The result \(\emptyset ; p_c ; \mathcal{H} \vdash * : 1, \tau\) follows trivially from T-UNIT.

Case **IF-TRUE** \((\text{if true then } e_1 \text{ else } e_2, M) \to \alpha (e_1, M)\):

From the typing of \(e\), we have \(\emptyset ; p_c ; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}_1\), where \(\mathcal{X} \leq \mathcal{X}_1\). We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}\), which follows from Lemma 7 and T-SUBSUME.

Case **IF-FALSE** \((\text{if false then } e_1 \text{ else } e_2, M) \to \alpha (e_2, M)\):

From the typing of \(e\), we have \(\emptyset ; p_c ; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}_2\), where \(\mathcal{X} \leq \mathcal{X}_2\). We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash e_2 : \tau, \mathcal{X}\), which follows from Lemma 7 and T-SUBSUME.

Case **LET** \((\{\text{let } x = v \text{ in } e_1, M\} \to \alpha (e_1 \{v/x\}, M))\), where \(v \notin \{\bot_p, [\bot_p]\}\) for all \(p)\):

Since \(v\) is a non-bottom value, by Lemma 8, it can be typed with \(\tau\) effect. From the typing of \(e\), then, we have

\[
\emptyset ; p_c ; \mathcal{H} \vdash v : \tau_1, \tau
\]

and

\[
x : \tau_1 ; p_c \cap \text{integ}(\tau_1) ; \mathcal{H} \vdash e_1 : \tau, \mathcal{X}.
\]

We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash e_1 \{v/x\} : \tau, \mathcal{X}\), which follows from Corollary 5 and Lemma 6.

Case **EVAL-CONTEXT** \((\{E[e_1], M\} \to \alpha (E[e'_1], M'))\), where \((e_1, M) \rightarrow (e'_1, M')\):

We need to show \(\emptyset ; p_c ; \mathcal{H} \vdash E[e'_1] : \tau, \mathcal{X}\). We proceed by cases according to the structure of \(E[e_1]\).

Case soft \(e_1\):

From the typing of \(e\), we have \(\emptyset ; p_c ; \mathcal{H} \vdash e_1 : R_w, \mathcal{X}\), where \(\tau = (\text{soft } R)\). By the induction hypothesis, we have \(\emptyset ; p_c ; \mathcal{H} \vdash e'_1 : R_w, \mathcal{X}\), so the result follows via T-SOFT.

Cases \(e_1 \parallel e_2\) and \(e_2 \parallel e_1\):

We show case \(e_1 \parallel e_2\). The other case follows symmetrically. From the typing of \(e\), we have \(\emptyset ; p_c ; \tau \vdash e_1 : \tau_1, \tau\). By the induction hypothesis, we have \(\emptyset ; p_c ; \tau \vdash e'_1 : \tau_1, \tau\), so the result follows via T-PARALLEL.

Case try \(e_1\) catch \(p : e_2\):

From the typing of \(e\), we have \(\emptyset ; p_c ; \mathcal{H}, p \vdash e_1 : \tau_1, \mathcal{X}_1\), where \(\tau = \tau_1 \cap w\) with \(w = \bigcap_{p' \in \mathcal{X}_1} (p \cup p')\) and \(\mathcal{X} = (\mathcal{X}_1/p) \cap \mathcal{X}_2\). By the induction hypothesis, we have \(\emptyset ; p_c ; \mathcal{H}, p \vdash e'_1 : \tau_1, \mathcal{X}_1\), so the result follows via T-TRY.

Case let \(x = e_1\) in \(e_2\):

From the typing of \(e\), we have \(\emptyset ; p_c ; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}_1\), where \(\vdash \text{auth}^\ast (\tau_1) \leq pc\) and \(\vdash \mathcal{X} \leq \mathcal{X}_1\). By the induction hypothesis, we have \(\emptyset ; p_c ; \mathcal{H} \vdash e'_1 : \tau_1, \mathcal{X}_1\), so the result follows via T-LET.
Case **FAIL-PROP** \((\{ F[\perp_p].M \} \rightarrow_\alpha (\perp_p.M))\):

We have \(\emptyset; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}\) and need to show \(\emptyset; pc; \mathcal{H} \vdash \perp_p : \tau, \mathcal{X}\). From the typing of \(e\), it follows that \(\mathcal{H} \equiv \mathcal{X} \equiv p \neq \tau\), so the result follows from T-BOTTOM and T-SUBSUME.

Case **BRACKET-SELECT** \((\{ m^S \}_{x_c}.M) \rightarrow_\alpha (\{ m^S \}_{x_c}.M)\):

Without loss of generality, assume \(S = \{ x_i : \tau_i \}_{(a,p)}\). Then, from the typing of \(e\), we have

\[
\emptyset; pc; \mathcal{H} \vdash m^S : \{ x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash m^S : \{ x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc
\]

\[
\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c} : \tau_c \cap w', p
\]

where \(\tau = \tau_c \cap w'\) and \(\mathcal{X} = p\). We need to show \(\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c} : \tau_c \cap w', p\).

By the typing of \(m^S\), we know \(\vdash_w e : e\) and so, \(\vdash \text{auth}^+(\tau_c) \leq a \leq pc \cap w\). Therefore, the result follows from an application of T-SELECT, followed by T-BRACKET:

\[
\emptyset; pc; \mathcal{H} \vdash m^S : \{ x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c} : \tau_c \cap w', p
\]

Case **BRACKET-SOFT-SELECT** \((\{ \text{soft } m^S \}_{x_c}.M) \rightarrow_\alpha (\{ \text{soft } m^S \}_{x_c}.M)\):

Without loss of generality, assume \(S = \{ x_i : \tau_i \}_{(a,p)}\). Then, from the typing of \(e\), we have

\[
\emptyset; pc; \mathcal{H} \vdash \text{soft } m^S : \{ \text{soft } x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [\text{soft } m^S]_{x_c} : \{ \text{soft } x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash \text{auth}^+(\tau_c) \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [\text{soft } m^S]_{x_c} : \tau_c \cap w', p
\]

where \(\tau = \tau_c \cap w'\) and \(\mathcal{X} = p\). We need to show \(\emptyset; pc; \mathcal{H} \vdash [\text{soft } m^S]_{x_c} : \tau_c \cap w', p\).

By the typing of \(m^S\), we know \(\vdash_w e : e\) and so, \(\vdash \text{auth}^+(\tau_c) \leq a \leq pc \cap w\). Therefore, the result follows from T-SOFT-SELECT and T-BRACKET:

\[
\emptyset; pc; \mathcal{H} \vdash \text{soft } m^S : \{ \text{soft } x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [\text{soft } m^S]_{x_c} : \tau_c \cap w', p
\]

\[
\alpha \not\vdash w \quad \vdash \text{auth}^+(\tau_c) \leq pc \cap w
\]

Case **BRACKET-ASSIGN** \((\{ m^S \}_{x_c := v}.M) \rightarrow_\alpha (\{ m^S \}_{x_c := v}.M)\):

Without loss of generality, assume \(S = \{ x_i : \tau_i \}_{(a,p)}\). Then, from the typing of \(e\), we have

\[
\emptyset; pc; \mathcal{H} \vdash m^S : \{ x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c} : \{ x_i : \tau_i \}_{(a,p)}, \top
\]

\[
\alpha \not\vdash w \quad \vdash a \leq pc
\]

\[
\emptyset; pc; \mathcal{H} \vdash \vdash_\tau \cap pc \cap w \leq \tau_c
\]

\[
\alpha \not\vdash w \quad \vdash \text{auth}^+(\tau') \leq pc \cap w
\]

\[
\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c} := v : \top, p
\]

where \(\tau = \top\) and \(\mathcal{X} = p\). We need to show \(\emptyset; pc; \mathcal{H} \vdash [m^S]_{x_c := v} : \top, p\).
From \( \vdash \text{auth}^+(\tau') \leq pc \cap w \) and Lemma 8, we know \( \varnothing; pc \cap w; \mathcal{H} \vdash v: \tau', \top \). The result then follows from an application of T-ASSIGN followed by T-BRACKET:

\[
\begin{align*}
\varnothing; pc \cap w; \mathcal{H} \vdash m^S : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{\tau, \top} & \vdash a \leq pc \\
\varnothing; pc \cap w; \mathcal{H} \vdash v: \tau', \top & \vdash \tau' \cap pc \cap w \leq \tau_c \\
& \vdash \text{auth}^+(\tau') \leq pc \cap w & \vdash \mathcal{H} \leq p \\
\end{align*}
\]

Case BRACKET-SOFT-ASSIGN

\((\mathbf{[soft~m^S\times x_c := v,M]} \rightarrow_{\alpha} (\mathbf{[soft~m^S\times x_c := v,M]})):\)

Without loss of generality, assume \( S = \langle x_i : \tau_i \rangle_{(a,a,p)} \). Then, from the typing of \( e \), we have

\[
\begin{align*}
\varnothing; pc \cap w; \mathcal{H} \vdash \text{soft} m^S : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{\tau, \top} \\
\varnothing; pc; \mathcal{H} \vdash [\text{soft} m^S] : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top} & \vdash \tau' \cap pc \cap w \leq \tau_c \\
& \vdash \text{auth}^+(\tau') \leq pc \cap w & \vdash \mathcal{H} \leq p \\
\end{align*}
\]

where \( \tau = 1 \) and \( \mathcal{X} = p \). We need to show \( \varnothing; pc; \mathcal{H} \vdash [\text{soft} m^S \times x_c := v] : 1, p \).

From \( \vdash \text{auth}^+(\tau') \leq pc \cap w \) and Lemma 8, we know \( \varnothing; pc \cap w; \mathcal{H} \vdash v: \tau', \top \). The result then follows from T-SOFT-ASSIGN and T-BRACKET:

\[
\begin{align*}
\varnothing; pc \cap w; \mathcal{H} \vdash \text{soft} m^S : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{\tau, \top} \\
\varnothing; pc \cap w; \mathcal{H} \vdash v: \tau, \top & \vdash \tau' \cap pc \cap w \leq \tau_c \\
& \vdash \text{auth}^+(\tau') \leq pc \cap w & \vdash \mathcal{H} \leq p \\
\end{align*}
\]

Case BRACKET-SOFT ((\text{soft} m^S,M) \rightarrow_{\alpha} (\text{soft} m^S,M)):

Without loss of generality, assume \( S = \langle x_i : \tau_i \rangle_{(a,a,p)} \). Then, from the typing of \( e \), we have

\[
\begin{align*}
\varnothing; pc \cap w; \mathcal{H} \vdash m^S : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{\tau, \top} \\
\varnothing; pc; \mathcal{H} \vdash [m^S] : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top}, \\
\end{align*}
\]

where \( \tau = (\text{soft} \langle x_i : \tau_i \rangle_{(a,a,p)})_w \) and \( \mathcal{X} = \top \). We need to show \( \varnothing; pc; \mathcal{H} \vdash [\text{soft} m^S] : (\text{soft} \langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top} \). This follows via an application of T-SOFT followed by T-BRACKET:

\[
\begin{align*}
\varnothing; pc \cap w; \mathcal{H} \vdash m^S : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{\tau, \top} & \alpha \notin w \\
\varnothing; pc; \mathcal{H} \vdash [m^S] : (\langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top}, \\
\end{align*}
\]

Case BRACKET-EXISTS

\((\mathbf{[exists~v~as~x:~e_1~else~e_2,M]} \rightarrow_{\alpha} (\mathbf{[exists~v~as~x:~e_1~else~e_2,M]})):\)

From the typing of \( e \), we have

\[
\begin{align*}
\varnothing; pc \cap \ell; \mathcal{H} \vdash v : (\text{soft} \langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top} \\
\alpha \notin \ell & \vdash \text{auth}^+(r) \leq pc \cap \ell \\
\end{align*}
\]

\[
\begin{align*}
\varnothing; pc; \mathcal{H} \vdash [v] : (\text{soft} \langle x_i : \tau_i \rangle_{(a,a,p)})_{w, \top} & \vdash \text{auth}^+(r) \leq pc \cap \ell & w' = \text{auth}^+(r) \cap \text{exist}(\tau) \cap \text{pc} \cap \ell \\
\end{align*}
\]

\[
\begin{align*}
\end{align*}
\]
where $\tau = \tau' \cap w'$ and $X = X_1 \cap X_2$. We need to show $\varnothing; pc; H \vdash [\exists x : e_1 \text{ else } e_2] : \tau' \cap w', X_1 \cap X_2$. To do this, we need to know that $x : (\{x : \tau_1\}, \tau_1) \cap w' ; H \vdash e_1 : \tau', X_1$, which can be demonstrated by an easy induction on the derivation of $x : (\{x : \tau_1\}, \tau_1) \cap w' ; H \vdash e_1 : \tau', X_1$. The result therefore follows via an application of T-Exists and T-Bracket.

$$\varnothing; pc \cap \ell ; H \vdash \nu : (\text{soft } \{x : \tau_1\}, \tau_1) \cap w, \top \quad \vdash \text{auth}^+(r) \leq pc \cap w \cap \ell \quad w'' = \text{auth}^-(r) \cap \text{persist}(r) \cap w$$

$x : (\{x : \tau_1\}, \tau_1) \cap w ; \varnothing; pc \cap w \cap w', H \vdash e_1 : \tau', X_1$

$$\varnothing ; pc \cap \ell ; H \vdash \exists x : e_1 \text{ else } e_2 : \tau' \cap w'', X_1 \cap X_2$$

$$\alpha \notin \ell \quad \vdash \text{auth}^+(\tau' \cap w'') \leq pc \cap w \cap w'' \leq pc \cap \ell$$

$$\varnothing ; pc ; H \vdash [\exists x : e_1 \text{ else } e_2] : \tau' \cap w', X_1 \cap X_2$$

(Noted that $w' = \ell \cap w''$.)

Case Bracket-Apply

$$\langle [\lambda(x : \tau_1) [pc_1 H_1] . e_1], v_2, M \rangle \rightarrow_\alpha \langle [\lambda(x : \tau_1) [pc_1 H_1] . e_1], v_2, M \rangle$$

From the typing of $e$, we have

$$x : \tau_1 ; pc_1 H_1 \vdash e_1 : \tau', H_1$$

$$\vdash_{wf} (\tau_1 \vdash p_{c_1 H_1} \rightarrow \tau') : \text{type} \quad \vdash p_{c_1} \leq pc \cap w$$

$$\varnothing ; pc \cap w ; H \vdash \lambda(x : \tau_1) [pc_1 H_1] . e_1 : (\tau_1 \vdash p_{c_1 H_1} \rightarrow \tau') \cap w, \top$$

$$\alpha \notin w \quad \vdash p_{c_1} \leq pc \cap w$$

$$\varnothing ; pc ; H \vdash [\lambda(x : \tau_1) [pc_1 H_1] . e_1] : (\tau_1 \vdash p_{c_1 H_1} \rightarrow \tau') \cap w, \top$$

$$\varnothing ; pc ; H \vdash v_2 ; \tau_1, \top \quad \vdash p_{c_1} \leq pc \cap w \quad H \vdash H_1$$

$$\alpha \notin w \quad \vdash \text{auth}^+(\tau') \leq pc \cap w$$

$$\varnothing ; pc ; H \vdash \langle [\lambda(x : \tau_1) [pc_1 H_1] . e_1], v_2, \tau', H_1 \rangle$$

where $\tau = \tau' \cap w$ and $X = H_1$. We need to show $\varnothing ; pc ; H \vdash \langle [\lambda(x : \tau_1) [pc_1 H_1] . e_1], v_2 \rangle : \tau' \cap w, H_1$.

By WT2, from $\vdash_{wf} (\tau_1 \vdash p_{c_1 H_1} \rightarrow \tau') : \text{type}$, we know $\vdash \text{auth}^+(\tau_1) \cup \text{auth}^+(\tau') \leq pc_1$. Since we also know from the above derivation that $\vdash p_{c_1} \leq pc \cap w$, it therefore follows that

$$\vdash \text{auth}^+(\tau_1) \leq pc \cap w$$

(11)

and $\vdash \text{auth}^+(\tau') \leq pc \cap w$. From the above derivation, we also have $\varnothing ; pc ; H \vdash v_2 ; \tau_1, \top$. Applying Lemma 8 to this and (11), we have $\varnothing ; pc \cap w ; H \vdash v_2 ; \tau_1, \top$. The result then follows from an application of T-App followed by T-Bracket:

$$\varnothing ; pc \cap w ; H \vdash [\lambda(x : \tau_1) [pc_1 H_1] . e_1] v_2 ; \tau', H_1$$

$$\alpha \notin w \quad \vdash \text{auth}^+(\tau') \leq pc \cap w$$

Case Bracket-Try

$$\langle \text{try } v \text{ catch } p ; e_2, M \rangle \rightarrow_\alpha \langle \text{try } v \text{ catch } p ; e_2, M \rangle$$

From the typing of $e$, we have

$$\varnothing ; pc \cap \ell ; H ; p \vdash v ; \tau', X_1$$

$$\alpha \notin \ell \quad \vdash \text{auth}^+(\tau') \leq pc \cap \ell$$

$$\varnothing ; pc ; H \vdash [v : \tau' \cap \ell, X_1] \quad w = \bigcap_{p \in X_1} (p \cup p')$$

$$\varnothing ; pc \cap w \cap \text{integ}(\tau' \cap \ell) ; H \vdash e_2 ; \tau' \cap \ell, X_2$$

$$\alpha \notin \ell \quad \vdash \text{auth}^+(\tau' \cap \ell) \leq pc$$

$$\varnothing ; pc ; H \vdash \text{try } v \text{ catch } p ; e_2 : \tau' \cap \ell \cap w, (X_1/p) \cap X_2$$
Case B

where $\tau = \tau' \cap \ell \cap w$ and $X = (X_1/p) \cap X_2$.

We need to show

$$\emptyset; pc; H + [v \text{ try } catch p : e_2] : \tau' \cap \ell \cap w, (X_1/p) \cap X_2.$$  

Note that $\text{integ}(\tau' \cap \ell)$ and $\text{auth}^+(\tau') = \text{auth}^+(\tau' \cap \ell) = \text{auth}^+(\tau' \cap \ell \cap w)$. The result then follows from S3, T-SUBSUME, T-TRY, and T-BRACKET:

$$\emptyset; pc \cap \ell; H + v : \tau', X_1 \quad \text{integ}(\tau') \cap \ell \leq \text{integ}(\tau') \quad \frac{\emptyset; pc \cap \ell; H + v : \tau', X_1 \quad \text{integ}(\tau' \cap \ell) \leq \text{integ}(\tau')}{w = \prod_{p \in X_1} (p \cup p')}
$$

$$\emptyset; pc \cap \ell \cap w \cap \text{integ}(\tau' \cap \ell); H + e_2 : \tau' \cap \ell, X_2 \quad \text{auth}^+(\tau' \cap \ell) \leq \text{pc} \cap \ell
$$

Case Bracket-If ((if $v$ then e1 else e2, M) $\rightarrow_\alpha$ ([if $v$ then e1 else e2], M)):

From the typing of $e$, we have

$$\emptyset; pc \cap \ell; H + v : \text{bool}_w, \top \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \quad \frac{\emptyset; pc; H + [v] : \text{bool}_w, \top \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \cap w}{w = \text{integ}(\tau') \cap \ell \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \cap w}
$$

where $\tau = \tau' \cap \ell \cap w \cap X = X_1 \cap X_2$. We need to show $\emptyset; pc; H + [v \text{ then e1 else e2}] : \tau' \cap \ell \cap w, X_1 \cap X_2$.

Since $\text{auth}^+(\tau' \cap w) = \text{auth}^+(\tau')$ and $\text{pc} \cap \ell \cap w \leq \text{pc} \cap \ell$, we therefore have $\text{auth}^+(\tau' \cap w) \leq \text{pc} \cap \ell$. The result then follows from T-IF and T-BRACKET:

$$\emptyset; pc \cap \ell; H + v : \text{bool}_w, \top \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \quad \frac{\emptyset; pc; H + [v] : \text{bool}_w, \top \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \cap w}{w = \text{integ}(\tau') \cap \ell \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \cap w}
$$

Case Bracket-Let ((let $x = [v]$ in e1, M) $\rightarrow_\alpha$ ([e1[[v]/x]], M), where $v \neq \bot_p$ for all $p$):

By Lemma 8, we know $[v]$ can be typed with $\tau$ effect. Therefore, from the typing of $e$, we have

$$\emptyset; pc \cap \ell; H + v : \tau'', \top \quad \text{auth}^+(\tau'') \leq \text{pc} \cap \ell \quad \frac{\emptyset; pc; H + [v] : \tau'', \top \quad \text{auth}^+(\tau'') \leq \text{pc} \cap \ell \cap w}{w = \text{integ}(\tau'' \cap \ell) \quad x : \tau'' \cap \ell; pc \cap w; H + e_1 : \tau', X \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \quad \text{auth}^+(\tau') \leq \text{pc} \cap \ell \cap w}
$$

where $\tau = \tau' \cap w$. We need to show $\emptyset; pc; H + [e_1[[v]/x]] : \tau' \cap w, X$.

From the derivation above, we know $\emptyset; pc; H + [v] : \tau'' \cap \ell, \tau$. Since $\text{auth}^+(\tau'' \cap \ell) = \text{auth}^+(\tau'')$, from $\text{auth}^+(\tau'') \leq \text{pc} \cap \ell$ above, we have $\text{auth}^+(\tau'' \cap \ell) \leq \text{pc} \cap \ell$. Therefore, by Lemma 8, we know

$$\emptyset; pc \cap \ell; H + [v] : \tau'' \cap \ell, \tau.$$  

(12)

From the derivation above, we also know $x : \tau'' \cap \ell; pc \cap w; H + e_1 : \tau', X$. By definition, $\tau \not\leq \ell$, so by Corollary 5, we know $x : \tau'' \cap \ell; pc \cap \ell; H + e_1 : \tau', X$. Applying Lemma 6 to this and (12), we have

$$\emptyset; pc \cap \ell; H + e_1[[v]/x] \cap \ell \cap w, X.$$
Since \( \text{auth}^+(\tau' \cap w) = \text{auth}^+(\tau') \) and \( \vdash w \leq \ell \), from \( \vdash \text{auth}^+(\tau') \leq pc \cap w \) above, we have

\[ \vdash \text{auth}^+(\tau' \cap w) \leq pc \cap \ell. \]

Finally, by Lemma 7, we have \( \vdash \mathcal{H} \leq \mathcal{X} \).

The result, then, follows from \text{T-SUBSUME} and \text{T-BRACKET}:

\[
\frac{\frac{\varnothing;pc \ni \ell; \vdash e_1[\{v\}/x] : \tau' \cap w, \mathcal{X}}{\varnothing;pc \ni \ell; \vdash \alpha \# \ell \quad \vdash \text{auth}^+(\tau' \cap w) \leq pc \cap \ell}}{\varnothing;pc; \vdash [e_1[\{v\}/x]] : \tau' \cap w \cap \ell, \mathcal{X}}.
\]

Case \text{DOUBLE-BRACKET} (\( [[v], M] \to_{\alpha} [[e], M] \)):

From the typing of \( e \), we have

\[
\frac{\frac{\varnothing;pc \cap \ell; \vdash [v] : \tau', \mathcal{X} \quad \alpha \# \ell \quad \vdash \text{auth}^+(\tau') \leq pc \cap \ell}{\varnothing;pc; \vdash [\{v\}] : \tau' \cap \ell, \mathcal{X}}}{\varnothing;pc; \vdash [v] : \tau' \cap \ell, \mathcal{X}}.
\]

where \( \tau' = \tau' \cap \ell \). We need to show \( \varnothing;pc; \vdash [v] : \tau' \cap \ell, \mathcal{X} \). By Corollary 5, we know \( \varnothing;pc; \vdash [v] : \tau', \mathcal{X} \). The result then follows via S3 and \text{T-SUBSUME}:

\[
\frac{\frac{\varnothing;pc; \vdash [v] \quad \vdash \text{integ}^+(\tau') \cap \ell \leq \text{integ}^+(\tau')}{\varnothing;pc; \vdash [v] : \tau' \cap \ell, \mathcal{X}}}{\varnothing;pc; \vdash [v] : \tau' \cap \ell, \mathcal{X}}.
\]

Case \text{BRACKET-CONTEXT} (\( [[e_1], M] \to_{\alpha} [[e_1'], M'] \)), where \( \langle e_1, M \rangle \to_{\alpha} \langle e_1', M' \rangle \)):

From the typing of \( e \), we have

\[
\frac{\frac{\varnothing;pc \cap \ell; \vdash e_1[\{v\}/x] : \tau', \mathcal{X} \quad \alpha \# \ell \quad \vdash \text{auth}^+(\tau') \leq pc \cap \ell}{\varnothing;pc; \vdash [e_1[\{v\}/x]] : \tau' \cap \ell, \mathcal{X}}}{\varnothing;pc; \vdash [e_1] : \tau' \cap \ell, \mathcal{X}}.
\]

where \( \tau' = \tau' \cap \ell \). We need to show \( \varnothing;pc; \vdash [e_1] : \tau', \mathcal{X} \).

By the induction hypothesis, we have \( \varnothing;pc \cap \ell; \vdash e_1[\{v\}/x] : \tau', \mathcal{X} \). The result follows by \text{T-BRACKET}:

\[
\frac{\frac{\varnothing;pc \cap \ell; \vdash e_1[\{v\}/x] : \tau', \mathcal{X} \quad \alpha \# \ell \quad \vdash \text{auth}^+(\tau') \leq pc \cap \ell}{\varnothing;pc; \vdash [e_1[\{v\}/x]] : \tau' \cap \ell, \mathcal{X}}}{\varnothing;pc; \vdash [e_1] : \tau' \cap \ell, \mathcal{X}}.
\]

Case \text{BRACKET-FAIL} (\( \langle F[[\bot]], M \rangle \to_{\alpha} \langle [\bot], M \rangle \)):

We need to show \( \varnothing;pc; \vdash [\bot] : \tau', \mathcal{X} \). We proceed by cases according to the structure of \( F[[\bot]] \).

Case \text{soft} \( [\bot] \):

From the typing of \( e \), we have

\[
\frac{\frac{\varnothing;pc \ni \ell; \vdash \bot \leq p}{\varnothing;pc \cap w; \vdash [\bot] : R_\tau, p \quad \alpha \# w \quad \vdash \text{auth}^+(R_\tau) \leq pc \cap w}}{\varnothing;pc; \vdash [\bot] : R_\tau, p}
\]

\[
\frac{\frac{\varnothing;pc \ni \ell; \vdash \bot \leq p}{\varnothing;pc \cap w; \vdash [\bot] : (\text{soft } R)_\tau, p \quad \alpha \# w \quad \vdash \text{auth}^+((\text{soft } R)_\tau) \leq pc \cap w}}{\varnothing;pc; \vdash [\bot] : (\text{soft } R)_\tau, p}
\]

where \( \tau = (\text{soft } R)_w \) and \( \mathcal{X} = p \). The result follows via \text{T-BOTTOM} and \text{T-BRACKET}:
Case let $x = [1, p]$ in $e_2$:

From the typing of $e$, we have

$$
\alpha \not\in T \vdash \mathcal{H} \not\subseteq p \cdot \quad \alpha \not\in \ell \vdash \text{auth}^+(\tau_1) \subseteq p \cap \ell
$$

where $\tau = \tau_2 \cap w$ and $X = X_2 \cap p$. By construction, $w \not\subseteq e$, so we know $\alpha \not\subseteq w$. By Lemma 7, we also know $\vdash \mathcal{H} \not\subseteq \mathcal{X} \cap p$. The result follows via T-BOTTOM, T-BRACKET and T-SUBSUME:

$$
\vdash \mathcal{H} \not\subseteq \mathcal{X} \cap p \quad \vdash \mathcal{H} \not\subseteq \mathcal{X} \cap p
$$

We now show that well-formedness is also preserved during execution.

**Lemma 10** (Well-formedness preservation). If $(e, M)$ is a well-formed configuration wherein $e$ is well-typed, and $(e, M)$ takes an $\rightarrow_{\alpha}$ transition, then the new configuration $(e', M')$ will also be well-formed:

$$
\vdash_{[\text{wf}]} (e, M) \land \forall \alpha; pc; \mathcal{H} \vdash e \cdot \tau, X \cdot (e, M) \rightarrow_{\alpha} (e', M') \Rightarrow \vdash_{[\text{wf}]} (e', M')
$$

**Proof.** First, note that $\text{dom}(M) \subseteq \text{dom}(M')$. This can be shown by an easy induction on the derivation of $(e, M) \rightarrow_{\alpha} (e', M')$.

We prove the lemma by induction on the derivation of $(e, M) \rightarrow_{\alpha} (e', M')$. Given $(e, M) \rightarrow_{\alpha} (e', M')$, the proof proceeds by cases according to the evaluation rules.

Rules DANGLE-SELECT, DANGLE-ASSIGN, PARALLEL-RESULT, TRY-ESC, FAIL-PROP, and Bracket-Fail, are trivial base cases in which $\text{locs}(e') = \emptyset$ and $M' = M$. Rules SELECT, and SOFT-SELECT follow from the definition of $\vdash_{[\text{wf}]} M$.

Rules APPLY, EXISTS-TRUE, EXISTS-FALSE, TRY-VAL, TRY-CATCH, IF-NEW, IF-FALSE, LET, Bracket-SELECT, Bracket-Soft-SELECT, Bracket-Assign, Bracket-Soft-Assign, Bracket-Soft, Bracket-Exists, Bracket-Apply, Bracket-Try, Bracket-If, Bracket-Let, and Double-Bracket follow because in these cases, $\text{locs}(e') \subseteq \text{locs}(e)$, $M' = M$, and $\text{root}(m^3, e') \Rightarrow \text{root}(m^3, e)$.

Rule GC follows from the fact that $\vdash_{[\text{wf}]} M$ and that minimal collectible groups must be disjoint.

Rules $\alpha$-CREATE and $\alpha$-ASSIGN follow trivially from the transition rules.

Rule $\alpha$-FORGET follows from the fact that $\vdash_{[\text{wf}]} M$.

Case CREATE $(\langle \{x_i = v_i\}^S, M\rangle) \rightarrow_{\alpha} \langle m^S, M^S \rightarrow \{x_i = v_i, a, \tau_i\} \rangle$, where $m^S$ is fresh and $S = \{x_i : \tau_i\}$:

We show $\emptyset; \tau ; \tau \vdash v_i : \tau_i, \tau$ for arbitrary $i$; the rest of this case follows from $\vdash_{[\text{wf}]} (e, M)$.

From the typing of $e$, we have $\emptyset; pc; \mathcal{H} \vdash v_i : \tau_i, \tau$. From this, the result follows via Lemma 8 and T-BRACKET.

Case ASSIGN $(\langle m^S.x_c := v, M\rangle) \rightarrow_{\alpha} \langle \vdash_{\alpha} p, M^S \rightarrow \{v, v_a, \tau_c\} \rangle$, where $S = \{x_i : \tau_i\} (a, p)$:

From the typing of $e$ and by T-SUBSUME, we have $\emptyset; pc; \mathcal{H} \vdash v_i : \tau_c, \tau$. By Lemma 8, we have $\emptyset; \tau ; \tau \vdash v : \tau_c, \tau$, and the rest of this case follows from $\vdash_{[\text{wf}]} (e, M)$.

Case SOFT-ASSIGN $(\langle \text{soft } m^S.x_c := v, M\rangle) \rightarrow_{\alpha} \langle v', \alpha \cap p, M' \rangle$, where $\langle m^S.x_c := v, M\rangle \rightarrow_{\alpha} \langle v', M' \rangle$ and $S = \{x_i : \tau_i\} (a, p)$:

If $M(m^S) = 1$, then we have $\text{locs}(e') = \emptyset$ and $M' = M$, and the result follows.

Otherwise, $M(m^S) \neq 1$, so from the definition of $\vdash_{[\text{wf}]} (\langle \text{soft } m^S.x_c := v, M\rangle)$, we can obtain $\vdash_{[\text{wf}]} \langle m^S.x_c := v, M\rangle$, and the rest follows similarly to the previous case.
Case \textsc{Eval-Context} ($([E[e_1],M] \rightarrow A (E[e'],M'))$, where $\langle e_1, M \rangle \xrightarrow{\alpha} \langle e', M' \rangle$):

Since $\vdash_{[w]} \langle e, M \rangle$, we know $\vdash_{[w]} \langle e_1, M \rangle$. Proceeding by cases according to the structure of $E[e_1]$, in each case we have $\emptyset; p c_1; H_1 \vdash e_1 : \tau_1 \chi_1$ (for some $p c_1, H_1, \tau_1$, and $\chi_1$). So, by the induction hypothesis, we have $\vdash_{[w]} \langle e_1', M' \rangle$. From this and the fact that $\vdash_{[w]} \langle e, M \rangle$, the result follows.

Case \textsc{Bracket-Context} ($([e_1], M) \rightarrow A ([e'], M')$, where $\langle e_1, M \rangle \xrightarrow{\alpha} \langle e', M' \rangle$):

Since $\vdash_{[w]} \langle e, M \rangle$, we know $\vdash_{[w]} \langle e_1, M \rangle$. From the typing of $e$, we have $\emptyset; p c^e; H \vdash e : \tau^e \chi^e$ for some appropriate $p c^e$ and $\tau^e$. So, by the induction hypothesis, we have $\vdash_{[w]} \langle e_1', M' \rangle$, and therefore $\vdash_{[w]} \langle e', M' \rangle$.

\hfill $\Box$

\textbf{Corollary 11 (Preservation).}

$$\vdash_{[w]} \langle e, M \rangle \rightarrow \emptyset; p c; H \vdash e : \tau \chi \rightarrow \vdash_{[w]} \langle e', M' \rangle$$

\textbf{Proof.} This follows from Lemmas 9 and 10 by induction on the number of $\rightarrow_{\alpha}$ transitions taken. \hfill $\Box$

\textbf{Corollary 12 (Preservation (nonadversarial execution)).}

$$\vdash_{[w]} \langle e, M \rangle \rightarrow \emptyset; p c; H \vdash e : \tau \chi \rightarrow \vdash_{[w]} \langle e', M' \rangle$$

\textbf{Proof.} This follows by Corollary 11 and the definition of $\vdash_{[w]} \langle e, M \rangle$. \hfill $\Box$

\textbf{Lemma 13 (Progress).} Let $\langle e, M \rangle$ be a configuration wherein $e$ has type $\tau$ and effect $\chi$, and the locations appearing in $e$ are mapped by $M$. Then either $e$ is a value, or $\langle e, M \rangle$ can take an $\xrightarrow{\xi}$ transition:

$$\emptyset; p c; H \vdash e : \tau \chi \text{ and } \text{locs}(e) \subseteq \text{dom}(M) \Rightarrow e \text{ is a value } \lor \exists e', M' \text{. } \langle e, M \rangle \xrightarrow{\xi} \langle e', M' \rangle$$

(Doubly bracketed values are considered expressions and not values.)

\textbf{Proof.} By induction on the derivation of $\emptyset; p c; H \vdash e : \tau \chi$. We proceed by cases according to the syntax of $e$. Note that since $e$ is typed in an empty type context ($\Gamma = \emptyset$), we must have $\text{FV}(e) = \emptyset$.

Case $e = v$:

Trivial since $e$ is a value.

Case $e = v_1 \ \text{v}_2$:

From the typing of $e$, we know that $v_1$ is a value with an arrow type and $\top$ effect, so it is either an abstraction $\lambda(x : \tau_1) [pc_1; H_1], e_1$ or a bracketed value $[v'_1]$. In the former case, by \textsc{Apply}, we have

$$\langle e, M \rangle = \langle (\lambda (x : \tau_1) [pc_1; H_1], e_1), v_2, M \rangle \xrightarrow{\xi} \langle e_1 \{ v_2 / x \}, M \rangle.$$  

In the latter case, by \textsc{Bracket-Apply}, we have $\langle e, M \rangle = \langle [v'_1], v_2, M \rangle \xrightarrow{\alpha} \langle [v'_1], v_2, M \rangle$.

Case $e = \text{if } v_1 \text{ then } e_2 \text{ else } e_3$:

From the typing of $e$, we know that $v_1$ is a value with bool type and $\top$ effect, so either $v_1 = \text{true}$, $v_1 = \text{false}$, or $v_1$ is a bracketed value $[v'_1]$. If $v_1 = \text{true}$, then by \textsc{If-True}, we have $\langle e, M \rangle = \langle \text{if true then } e_2 \text{ else } e_3, M \rangle \xrightarrow{\xi} \langle e_2, M \rangle$. If $v_1 = \text{false}$, then by \textsc{If-False}, we have $\langle e, M \rangle = \langle \text{if false then } e_2 \text{ else } e_3, M \rangle \xrightarrow{\xi} \langle e_3, M \rangle$. Otherwise, $v_1 = [v'_1]$ and by \textsc{Bracket-If}, we have $\langle e, M \rangle = \langle [v'_1], e_2 \text{ else } e_3, M \rangle \xrightarrow{\xi} \langle [v'_1], e_2 \text{ else } e_3, M \rangle$.

Case $e = \{ x_1 = v_1 \}$:

Trivial by \textsc{Create}.
Case $e = \nu x.c$:

From the typing of $e$, we know $v$ is a value with record type and $\top$ effect, so it is either a bracketed hard reference ($v = [m^S]$), a bracketed soft reference ($v = [\text{soft } m^S]$), a hard reference ($v = m^S$), or a soft reference ($v = \text{soft } m^S$).

In the first case ($v = [m^S]$), by \textsc{Bracket-Select}, we have $\langle e, M \rangle = \langle [m^S].x_c := v', M \rangle \xrightarrow{\varepsilon} \langle [m^S].x_c := v', M \rangle$.

In the second case ($v = [\text{soft } m^S]$), by \textsc{Bracket-Soft-Select}, we have $\langle e, M \rangle = \langle [\text{soft } m^S].x_c := v', M \rangle \xrightarrow{\eta} \langle [\text{soft } m^S].x_c := v', M \rangle$.

In the third case ($v = m^S$) and fourth case ($v = \text{soft } m^S$), assume without loss of generality $S = \{ x_i : \tau \}$. Since $\text{locs}(e) \subseteq \text{dom}(M)$, we must have $m^S \in \text{dom}(M)$. There are two sub-cases to consider. In each sub-case, we will show $\langle m^S.x_c := v', M \rangle \xrightarrow{\varepsilon} \langle v'', M \rangle$, for some $v''$, thereby proving the case of a hard reference ($v = m^S$). The case of a soft reference ($v = \text{soft } m^S$) follows by \textsc{Soft-Select}: $\langle (\text{soft } m^S).x_c := v', M \rangle \xrightarrow{\eta} \langle v'' \bullet_{\alpha} (a \cap p), M \rangle$.

1. Case $M(m^S) \neq \bot$:
   Without loss of generality, assume $M(m^S) = \{ x_i = v_i \}$. Then, by \textsc{Select},
   $$\langle m^S.x_c := v', M \rangle \xrightarrow{\varepsilon} \langle v_c \bullet_{\alpha} p, M \rangle.$$  

2. Case $M(m^S) = \bot$:
   By \textsc{Dangle-Select}, $\langle m^S.x_c := v', M \rangle \xrightarrow{\varepsilon} \langle \bot_p \bullet_{\alpha} p, M \rangle$.

Case $e = \exists x. e_1$ else $e_2$:

From the typing of $e$, we know that $v$ is a value with soft reference type, so it is either a bracketed value ($v = [v']$) or a soft reference ($v = \text{soft } m^S$).

In the first case ($v = [v']$), by \textsc{Bracket-Exists}, we have

$$\langle e, M \rangle = \langle \exists v' \text{ as } x : e_1 \text{ else } e_2, M \rangle \xrightarrow{\varepsilon} \langle \exists v' \text{ as } x : e_1 \text{ else } e_2, M \rangle$$
The second case \((v = \text{soft } m^\delta)\) splits into two subcases. Assume \(S = \{x_i : \tau_i\}_{(a,p)}\) without loss of generality. If \(M(m^\delta) \neq \bot\), then by EXITS-TRUE, we have
\[
\langle e, M \rangle = \langle \text{exists soft } m^\delta \text{ as } x : e_1 \text{ else } e_2, M \rangle \xrightarrow{e} \langle e_1 \{m^\delta / x\} \bullet_a (a \cap p), M \rangle
\]
Otherwise, \(M(m^\delta) = \bot\), and by EXITS-FALSE, we have
\[
\langle e, M \rangle = \langle \text{exists soft } m^\delta \text{ as } x : e_1 \text{ else } e_2, M \rangle \xrightarrow{e_1} \langle e_2 \bullet_a (a \cap p), M \rangle
\]
Case \(e = \text{soft } e_1\):
From the typing of \(e\), we know that \(\emptyset : pc ; \mathcal{H} \vdash e_1 : R_w , \mathcal{X}_w\), where \(\tau = (\text{soft } R_w)\). Since \(\text{locs}(e) \in \text{dom}(M)\), we must also have \(\text{locs}(e_1) \in \text{dom}(M)\), so by the induction hypothesis, either \(e_1\) is a value or \(\langle e_1, M \rangle \xrightarrow{\mathcal{R}} \langle e_1', M' \rangle\), for some \(e_1'\) and \(M'\). There are four cases to consider:
1. \(e_1\) is a value \(\bot_p\).
2. \(e_1\) is a bracketed value \([v]\).
3. \(e_1\) is some other value, and
4. \(\langle e_1, M \rangle \xrightarrow{e} \langle e_1', M' \rangle\).
In case 1, by FAIL-PROP, we have \(\langle e, M \rangle = \langle \text{soft } \bot_p, M \rangle \xrightarrow{\mathcal{R}} \langle \bot_p, M \rangle\).
In case 2, by BRACKET-SOFT, we have \(\langle e, M \rangle = \langle \text{soft } [v], M \rangle \xrightarrow{\mathcal{R}} \langle [\text{soft } v], M \rangle\).
In case 3, \(e_1\) must be a hard reference \(m^\delta\), so \(e = \text{soft } m^\delta\) is also a value.
In case 4, by EVAL-CONTEXT, we have \(\langle e, M \rangle = \langle \text{soft } e_1, M \rangle \xrightarrow{\mathcal{R}} \langle \text{soft } e_1', M' \rangle\).
Case \(e = e_1 \parallel e_2\):
From the typing of \(e\), we have \(\emptyset : pc ; \mathcal{H} \vdash e_i : \tau_i, \mathcal{T}\) for \(i \in \{1, 2\}\). Since \(\text{locs}(e) \in \text{dom}(M)\), we must also have \(\text{locs}(e_1) \in \text{dom}(M)\), so by the induction hypothesis, either \(e_1\) and \(e_2\) are both values, or (without loss of generality) \(\langle e_1, M \rangle \xrightarrow{\mathcal{R}} \langle e_1', M' \rangle\), for some \(e_1'\) and \(M'\).
If \(e_1\) and \(e_2\) are both values, then by PARALLEL-RESULT, we have \(\langle e, M \rangle = \langle e_1 \parallel e_2, M \rangle \xrightarrow{\mathcal{R}} \langle \ast, M \rangle\).
Otherwise, by EVAL-CONTEXT, we have \(\langle e, M \rangle = \langle e_1 \parallel e_2, M \rangle \xrightarrow{\mathcal{R}} \langle e_1' \parallel e_2', M' \rangle\).
Case \(e = \text{try } e_1 \text{ catch } p : e_2\):
From the typing of \(e\), we have \(\emptyset : pc ; \mathcal{H}, p \vdash e_1 : \tau_1, \mathcal{X}_1\), where \(\tau = \tau_1 \cap w\) with \(w = \bigcap_{p' \notin \mathcal{X}_1} (p \cup p')\). Since \(\text{locs}(e) \in \text{dom}(M)\), we must also have \(\text{locs}(e_1) \in \text{dom}(M)\), so by the induction hypothesis, either \(e_1\) is a value or \(\langle e_1, M \rangle \xrightarrow{\mathcal{R}} \langle e_1', M' \rangle\), for some \(e_1'\) and \(M'\). There are five cases to consider:
1. \(e_1\) is a value \(\bot_{p'}\) and \(\vdash p \leq p'\).
2. \(e_1\) is a value \(\bot_{p'}\) and \(p \not\leq p'\).
3. \(e_1\) is a bracketed value \([v]\).
4. \(e_1\) is some other value \(v\), and
5. \(\langle e_1, M \rangle \xrightarrow{e} \langle e_1', M' \rangle\).
In case 1, by TRY-CATCH, we have \(\langle e, M \rangle = \langle \text{try } \bot_{p'}, \text{ catch } p : e_2, M \rangle \xrightarrow{e} \langle e_2, M \rangle\).
In case 2, by TRY-ESC, we have \(\langle e, M \rangle = \langle \text{try } \bot_{p'}, \text{ catch } p : e_2, M \rangle \xrightarrow{e} \langle \bot_{p'}, M \rangle\).
In case 3, by BRACKET-TRY, we have \(\langle e, M \rangle = \langle \text{try } [v], \text{ catch } p : e_2, M \rangle \xrightarrow{e} \langle [\text{try } v, \text{ catch } p : e_2], M \rangle\).
In case 4, by TRY-VAL, we have \(\langle e, M \rangle = \langle \text{try } v, \text{ catch } p : e_2, M \rangle \xrightarrow{e} \langle v, M \rangle\).
Finally, in case 5, by EVAL-CONTEXT, we have
\[
\langle e, M \rangle = \langle \text{try } e_1 \text{ catch } p : e_2, M \rangle \xrightarrow{e} \langle \text{try } e_1' \text{ catch } p : e_2', M' \rangle
\]
Case $e = \text{let } x = e_1 \text{ in } e_2$:

From the typing of $e$, we have $\emptyset ; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{X}_1$. Since $\text{locs}(e) \subseteq \text{dom}(M)$, we must also have $\text{locs}(e_1) \subseteq \text{dom}(M)$, so by the induction hypothesis, either $e_1$ is a value or $\langle e_1, M \rangle \xrightarrow{e} \langle e_1', M' \rangle$, for some $e_1'$ and $M'$. There are five cases to consider:

1. $e_1$ is a bottom value $\bot_{\mathcal{P}}$,
2. $e_1$ is a bracketed bottom value $[\bot_{\mathcal{P}}]$,
3. $e_1$ is some other bracketed value $[v]$,
4. $e_1$ is some other value $v$, and
5. $\langle e_1, M \rangle \xrightarrow{e} \langle e_1', M' \rangle$.

In case 1, by $\text{FAIL-PROP}$, we have $\langle e, M \rangle = \langle \text{let } x = \bot_{\mathcal{P}} \text{ in } e_2, M \rangle \xrightarrow{e} \langle \bot_{\mathcal{P}}, M \rangle$.

In case 2, by $\text{BRACKET-FAIL}$, we have $\langle e, M \rangle = \langle \text{let } x = [\bot_{\mathcal{P}}] \text{ in } e_2, M \rangle \xrightarrow{e} \langle [\bot_{\mathcal{P}}], M \rangle$.

In case 3, by $\text{BRACKET-LET}$, we have $\langle e, M \rangle = \langle \text{let } x = [v] \text{ in } e_2, M \rangle \xrightarrow{e} \langle e_2\{[v]/x\}, M \rangle$.

In case 4, by $\text{LET}$, we have $\langle e, M \rangle = \langle \text{let } x = v \text{ in } e_2, M \rangle \xrightarrow{e} \langle e_2\{v/x\}, M \rangle$.

In case 5, by $\text{EVAL-CONTEXT}$, we have $\langle e, M \rangle = \langle \text{let } x = e_1 \text{ in } e_2, M \rangle \xrightarrow{e} \langle \text{let } x = e_1' \text{ in } e_2, M' \rangle$.

Case $e = [e']$:

From the typing of $e$, we have $\emptyset ; pc; \mathcal{H} \vdash e' : \tau', \mathcal{X}$. Since $\text{locs}(e) \subseteq \text{dom}(M)$, we must also have $\text{locs}(e') \subseteq \text{dom}(M)$, so by the induction hypothesis, either $e'$ is a value or $\langle e', M \rangle \xrightarrow{e'} \langle e'', M' \rangle$.

If $e'$ is a bracketed value $[v]$, then by $\text{DOUBLE-BRACKET}$, we have $\langle e, M \rangle = \langle [[v]], M \rangle \xrightarrow{e} \langle [v], M \rangle$.

If $e'$ is some other value $v$, then $e = [v]$ is a value.

Otherwise, by $\text{BRACKET-CONTEXT}$, we have $\langle e, M \rangle = \langle ([e']), M \rangle \xrightarrow{e} \langle ([e'']), M' \rangle$.

\[
\text{Corollary 14 (Soundness of } [\lambda_{\text{persist}}]\text{).}
\]

\[
\vdash^{\text{def}} \langle e, M \rangle \wedge \emptyset ; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \Rightarrow \langle e, M \rangle \parallel \forall v \in \text{Val}. M'. \langle e, M \rangle \xrightarrow{e} \langle v, M' \rangle
\]

\textbf{Proof.} This follows from Corollary 11 and Lemma 13 by induction on the number of $\xrightarrow{e}$ transitions taken. \hfill \square

\subsection*{8.4 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. This property is similar to noninterference [10], and similarly can be expressed using a security relation on configurations. Two configurations are related if they agree on all high-integrity parts of the program and of the memory.

The property states that for any execution influenced by the adversary, there is a corresponding, related execution in which the adversary is not present. Hence, the adversary’s influence is not significant. More precisely, 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 is a possibilistic security property, 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.
\[\phi(m_1^S) = m_2^S\]
\[\lambda(x: \tau)[pc; \mathcal{H}]. e \equiv \bigwedge \lambda(x: \tau)[pc; \mathcal{H}]. e'\]

**Definition 12** (Security relation on memories). Two memories, \(M_1\) and \(M_2\), are \(\phi\)-related, written \(M_1 \equiv \phi M_2\), if for any location \(m^S \in \text{dom}(\phi)\),

1. If \(m^S\) is not deleted, then \(\phi(m^S)\) maps to a \(\phi\)-related record; and
2. If \(m^S\) is a deleted high-authority, high-persistence location, then so is \(\phi(m^S)\).

\[
M_1 \equiv \phi M_2 \iff \forall m^S \in \text{dom}(\phi).
\]
\[
(M_1(m^S) \neq \bot \Rightarrow M_2(\phi(m^S)) \neq \bot \land M_1(m^S) \equiv \phi M_2(\phi(m^S)))
\]
\[
\land \neg \alpha \in \text{auth}(S) \land \text{persist}(S) \land M_1(m^S) = \bot \Rightarrow M_2(\phi(m^S)) = \bot
\]

\[\text{Definition 11 (High-integrity homomorphism).} \quad \text{An injective partial function } \phi : \text{dom}(M_1) \rightarrow \text{dom}(M_2) \text{ is a high-integrity homomorphism from } M_1 \text{ to } M_2 \text{ if it satisfies the following:}
\]

- **Injective:** \(m_1^S \neq m_2^S \land \{m_1^S, m_2^S\} \subseteq \text{dom}(\phi) \Rightarrow \phi(m_1^S) \neq \phi(m_2^S);\)

- **Type-preserving:** \(m_2^S = \phi(m_1^S) \Rightarrow S_1 = S_2;\) and

- **Isomorphic when the domain and range are restricted to the high-integrity and high-persistence locations in \(M_1\) and \(M_2\):**

\[\phi|_D : D \rightarrow R,\]

where \(D = \{m^S \in \text{dom}(M_1) | - \alpha \in \text{integ}(S) \lor - \alpha \in \text{persist}(S)\}\) and \(R = \{m^S \in \text{dom}(M_2) | - \alpha \in \text{integ}(S) \lor - \alpha \in \text{persist}(S)\}\). The notation \(\text{integ}(S)\) denotes the integrity of a record with type \(S\): the least upper bound of its fields' integrity.

\[
\text{integ}(\{x_i : \tau_i\}) = \bigsqcup \text{integ}(\tau_i)
\]

We are now ready to define our security relation on expressions. An expression \(e_1\) is considered to be related to \(e_2\) via a high-integrity homomorphism \(\phi\), written \(e_1 \equiv \phi e_2\), if \(e_1\) is equal to \(e_2\) (modulo bracketed expressions) when the memory locations in \(e_1\) are transformed via \(\phi\). This is defined formally in Figure 12.

To ensure that high-integrity dereferences yield related results, we also define a security relation on memories: \(M_1\) and \(M_2\) are related via \(\phi\), written \(M_1 \equiv \phi M_2\), if two conditions hold for each location \(m^S \in \text{dom}(\phi)\). 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 \(\phi(m^S)\).
These two security relations induce a security relation on configurations:

\[ \langle e_1, M_1 \rangle \rightarrow^*_{\alpha} \langle e_2, M_2 \rangle \ \overset{\text{def.}}{\iff} \ e_1 \rightarrow^*_{\alpha} e_2 \land M_1 \rightarrow^*_{\alpha} M_2. \]

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

8.5 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.

**Theorem 1** (Referential integrity). 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:

\[ \vdash_{\text{wf}} \langle e_1, M_1 \rangle \land \vdash_{\text{wf}} \langle e_2, M_2 \rangle \land \langle e_1, M_1 \rangle \rightarrow^*_{\alpha} \langle e_2, M_2 \rangle \land \exists \phi'. \langle e_2, M_2 \rangle \rightarrow^* \langle e'_{2}, M'_2 \rangle \land \langle e'_1, M'_1 \rangle \rightarrow^*_{\alpha} \langle e', M' \rangle \land \phi = \phi' \mid_{\text{dom}(\phi)} \]

To prove this, we first need to prove a few preliminary results about garbage collection, given by Lemmas 15–25.

**Lemma 15.** Let \(e\) be well-typed in a low-integrity context. Assume that if \(e\) is a memory location, then it is low-authority. If \(m^5\) is a GC root in \(e\), then \(m^5\) must be low-authority:

\[ \Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X} \land \alpha \not\in pc \land (e \in \text{dom}(M) \Rightarrow \alpha \not\in \text{auth}(\tau)) \land \text{root}(m^5, e) \]

\[ \Rightarrow \alpha \not\in \text{auth}^+(S) \]

**Proof.** By induction on the derivation of \(\text{root}(m^5, e)\).

Case R1 \((e = m^5)\):

Without loss of generality, assume \(\mathcal{S} = \{x_i : \tau_i\}_{(a, p)}\), where \(\alpha \not\in a\). We have \(\tau = (\{x_i : \tau_i\}_{(a, a, p)})\), so \(\text{auth}^+(\tau) = \text{auth}^+(S)\) and the result follows trivially.

Case R2 \((e = \text{soft } e', \forall m^5_1 . e' \neq m^5_1)\):

From the derivation of \(\text{root}(m^5, e)\), we have \(\text{root}(m^5, e')\). From the typing of \(e\), we have \(\Gamma; pc; \mathcal{H} \vdash e' : R_w, \mathcal{X}\). Therefore, the induction hypothesis applies, and the result follows.

Case R3 \((e = \{x_i = \overline{v_i}\})^5\):

From the derivation of \(\text{root}(m^5, e)\), we have \(\text{root}(m^5, v_i)\) for some \(c\).

From the typing of \(e\), we know \(\Gamma; vc; \mathcal{H} \vdash \tau' : \tau, \tau\), where \(\vdash \text{auth}^+(\tau') \subseteq pc\). From this, we also know \(\alpha \not\in \text{auth}^+(\tau')\).

Therefore, the induction hypothesis applies, and the result follows.
Case R4 ($e = v.x$):
From the derivation of root($m^\delta, e$), we have root($m^\delta, v$).
From the typing of $e$, we have $\Gamma; pc; \mathcal{H} \vdash v : \tau', \mathcal{T}$, where either $\tau' = (\{ x_i : \tau_i \} (a^+ a^- p))_w$ with $a^+ a^- p \leq pc$, or $\tau' = (soft R)_w$ for some $R$. We therefore know $\alpha \not\in \text{auth}^+ (\tau')$.
Therefore, the induction hypothesis applies, and the result follows.

Case R5 ($e = v_1, x := v_2$):
From the derivation of root($m^\delta, e$), we have root($m^\delta, v_i$) for some $i \in \{1,2\}$.
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash v_1 : \tau', \mathcal{T}$ (where either $\tau' = (\{ x_i : \tau_i \} (a^+ a^- p))_w$ with $a^+ a^- p \leq pc$, or $\tau' = (soft R)_w$ for some $R$) and $\Gamma; pc; \mathcal{H} \vdash v_2 : \tau'', \mathcal{T}$, where $\sim \text{auth}^+ (\tau'') \leq pc$. Therefore, $\alpha \not\in \text{auth}^+ (\tau')$ and $\alpha \not\in \text{auth}^+ (\tau'')$. So, the induction hypothesis applies for $i \in \{1,2\}$, and the result follows.

Case R6 ($e = \lambda (x : \tau') [pc'; \mathcal{H}']. e'$):
From the typing of $e$, we know $\sim w f \tau : \text{type}$, $\sim pc' \leq pc$, and $\Gamma, x : \tau'; pc'; \mathcal{H}' \vdash e' : \tau'', \mathcal{H}'$, where $\tau = \tau' \stackrel{pc'}{\rightarrow} \mathcal{H}' \rightarrow \tau''$. Since $\alpha \not\in pc$, we have $\alpha \not\in pc'$. From $\sim w f \tau : \text{type}$, we know $\sim \text{auth}^+ (\tau'') \leq pc'$, so we therefore know $\alpha \not\in \text{auth}^+ (\tau'')$. From the derivation of root($m^\delta, e$), we know root($m^\delta, e'$). Therefore, the induction hypothesis applies, and the result follows.

Case R7 ($e = v_1 v_2$):
From the derivation of root($m^\delta, e$), we have root($m^\delta, v_i$) for some $i \in \{1,2\}$.
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash v_1 : (\tau \stackrel{pc'}{\rightarrow} \mathcal{H}')_w, \mathcal{T}$, $\Gamma; pc; \mathcal{H} \vdash v_2 : \tau', \mathcal{T}$, and $\sim w f (\tau' \stackrel{pc'}{\rightarrow} \mathcal{H}') \vdash w : \text{type}$, with $\sim pc' \leq pc$. Therefore, $\alpha \not\in pc'$.
From the derivation of $\sim w f (\tau' \stackrel{pc'}{\rightarrow} \mathcal{H}')_w : \text{type}$, we have $\sim \text{auth}^+ (\tau') \leq pc$ and $\sim \text{auth}^+ (\tau') \leq pc$. We therefore know $\alpha \not\in \text{auth}^+ (\tau')$.
So, the induction hypothesis applies for $i \in \{1,2\}$, and the result follows.

Case R8 ($e = \text{let } x = e_1 \text{ in } e_2$):
From the derivation of root($m^\delta, e$), we have root($m^\delta, e_i$) for some $i \in \{1,2\}$.
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{T}_1$ and $\Gamma, x : \tau' : pc'; \mathcal{H} \vdash e_2 : \tau'', \mathcal{T}_2$, where $\sim \text{auth}^+ (\tau_1) \leq pc$, $pc' = pc \cap w$, and $\sim \text{auth}^+ (\tau'') \leq pc'$, for some $w$. Therefore, it follows that $\alpha \not\in pc'$, $\alpha \not\in \text{auth}^+ (\tau''')$, and $\alpha \not\in \text{auth}^+ (\tau'')$.
So, the induction hypothesis applies for $i \in \{1,2\}$, and the result follows.

Case R9 ($e = e_1 \parallel e_2$):
From the derivation of root($m^\delta, e$), we have root($m^\delta, e_i$) for some $i \in \{1,2\}$. Without loss of generality, assume $i = 1$.
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash e_1 : \tau_1, \mathcal{T}$ and $\sim \text{auth}^+ (\tau_1) \leq pc$. From this, we also know $\alpha \not\in \text{auth}^+ (\tau_1)$.
Therefore, the induction hypothesis applies, and the result follows.

Case R10 ($e = v$ if $v$ then $e_1$ else $e_2$):
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash v : \text{bool}, \mathcal{T}$ and $\Gamma; pc \cap w; \mathcal{H} \vdash e_1 : \tau', \mathcal{T}_i$ (for $i \in \{1,2\}$), where $\sim \text{auth}^+ (\tau') \leq pc$. From this, we therefore know $\alpha \not\in \text{auth}^+ (\tau')$.
From the derivation of root($m^\delta, e$), we have either root($m^\delta, v$) or root($m^\delta, e_j$), for some $j \in \{1,2\}$. In all cases, the induction hypothesis applies, and the result follows.

Case R11 ($e = \text{exists } x : e_1$ else $e_2$):
From the typing of $e$, we know $\Gamma; pc; \mathcal{H} \vdash v : (\text{soft } \{ x_i : \tau_i \})_w, \mathcal{T}$ and $\Gamma; pc \cap w'; \mathcal{H} \vdash e_1 : \tau', \mathcal{T}_i$ (for $i \in \{1,2\}$), where $\sim \text{auth}^+ (\tau') \leq pc \cap w'$. From this, we therefore know $\alpha \not\in \text{auth}^+ (\tau')$.
From the derivation of root($m^\delta, e$), we have either root($m^\delta, v$) or root($m^\delta, e_j$), for some $j \in \{1,2\}$. In all cases, the induction hypothesis applies, and the result follows.
Case R12 ($e = \text{try } e_1 \text{ catch } p: e_2$):
From the derivation of root($m^S, e$), we have root($m^S, e_i$), for some $i \in \{1, 2\}$.
From the typing of $e$, we know $\Gamma; pc; \mathcal{H}, p \vdash e_1 : \tau, \mathcal{X}_1$ and $\Gamma; pc; \mathcal{H}, \vdash e_2 : \tau', \mathcal{X}_2$, where $\vdash pc' \leq pc$ and $\vdash \mathcal{H}^+(\tau') \leq pc$. From this, we also know $\alpha \notin pc'$ and $\alpha \notin \mathcal{H}^+(\tau')$.
So, the induction hypothesis applies for $i \in \{1, 2\}$, and the result follows.

Case R13 ($e = [e']$):
From the derivation of root($m^S, e$), we have root($m^S, e'$).
From the typing of $e$, we know $\Gamma; pc; \mathcal{H}, \vdash e' : \tau', \mathcal{X}'$, where $\vdash pc' \leq pc$ and $\vdash \mathcal{H}^+(\tau') \leq pc$. From this, it follows that $\alpha \notin pc'$ and $\alpha \notin \mathcal{H}^+(\tau')$.
Therefore, the induction hypothesis applies, and the result follows.

Lemma 16. Suppose $m^S$ is a high-authority location that is a GC root in the well-typed value $v$. Then $v$ must have high-authority type.

\[ \vdash \alpha \leq \mathcal{H}^+(S) \land \text{root}(m^S, v) \land \emptyset ; \tau \vdash \tau, \top \Rightarrow \vdash \alpha \leq \mathcal{H}^+(\tau) \]

Proof. Since root($m^S, v$), the value $v$ must either be equal to $m^S$, be a lambda abstraction $\lambda(x : \tau)[pc; \mathcal{H}, e$, or be a bracketed value $[v']$.

Case $v = m^S$:
We therefore have $\emptyset ; \tau \vdash m^S : \tau, \top$. So $\vdash \mathcal{H}^+(S) \leq \mathcal{H}^+(\tau)$. Therefore, $\vdash \alpha \leq \mathcal{H}^+(\tau)$, as desired.

Case $v = \lambda(x : \tau)[pc; \mathcal{H}, e$:
From the derivation of root($m^S, v$), we have root($m^S, e$). So, we must have $\tau = \tau_1 \xrightarrow{pc; \mathcal{H}} \tau_2$, for some $\tau_1$ and some $\tau_2$. Therefore, $\mathcal{H}^+(\tau) = pc$.

Suppose $\alpha \notin pc$. From the typing of $v$, we know $x : \tau_1; pc; \mathcal{H}, \vdash e : \tau_2, \mathcal{H}$ and $\vdash \mathcal{H}^+(\tau_2) \leq \alpha$. Therefore, $\vdash \mathcal{H}^+(\tau_2)$, since root($m^S, e$), by Lemma 15, we must have $\alpha \notin \mathcal{H}^+(S)$, a contradiction. So, we must have $\vdash \alpha \leq pc = \mathcal{H}^+(\tau)$, as desired.

Case $v = [v']$:
From the derivation of root($m^S, v$), we have root($m^S, v'$). From the typing derivation of $v$, we know $\emptyset ; \ell \vdash v' : \tau', \top$ and $\vdash \mathcal{H}^+(\tau') \leq \ell$, where $\alpha \notin \ell$. Therefore, $\vdash \mathcal{H}^+(\tau')$. Since root($m^S, v'$), by Lemma 15, we must have $\alpha \notin \mathcal{H}^+(S)$, a contradiction. This case therefore holds vacuously.

Lemma 17. Suppose $m^S$ is a high-authority location that is noncollectible in the configuration $\langle v, M \rangle$, where $M$ is well-formed and the value $v$ has type $\tau$. Then $\tau$ must be high-authority.

\[ \vdash^{[\text{nc}]} M \land \vdash \alpha \leq \mathcal{H}^+(S) \land \text{nc}(m^S, \langle v, M \rangle) \land \emptyset ; \tau \vdash \tau, \top \Rightarrow \vdash \alpha \leq \mathcal{H}^+(\tau) \]

Proof. By induction on the derivation of nc($m^S, \langle v, M \rangle$).

Case NC1:
We have root($m^S, v$). The result follows via Lemma 16.
Case NC2:

We have root($m^S_1, v$), $M(m^S_1) = \{x_i = \bar{v}_i\}$, and $nc(m^S, \langle v_c, M \rangle)$ for some $c$. Without loss of generality, assume $S_1 = \{x_i : \tau_i\}_{a,p}$. From $\vdash^*_{\alpha} M$, we have $\emptyset; T \vdash v_c : \tau_c, T$. Therefore, we can apply the induction hypothesis to obtain $\vdash \alpha \leq \text{auth}^+(\tau_c)$. From $\vdash^*_{\alpha} M$, we know $\vdash_{wf} S_1 : \text{rectype}$, and therefore, $\vdash \text{auth}^+(\tau_c) \leq a$. Hence, $\vdash \alpha \leq a = \text{auth}^+(S_1)$. The result follows via Lemma 16.

Corollary 18. Suppose $m^S$ is a high-authority location that is noncollectible in the configuration $\{m^S_1, M\}$, where $M$ is well-formed. Then $m^S_1$ is high-integrity, high-authority, and high-persistence.

Proof. Since $\vdash^*_{\alpha} M$, we must have $\vdash_{wf} S_1 : \text{rectype}$, and so, $\vdash \text{auth}^+(S_1) \leq \text{integ}(S_1) \cap \text{auth}^+(S_1) \cap \text{persist}(S_1)$. It therefore suffices to show that $m^S_1$ is high-authority. This follows from Lemma 17.

Lemma 19. Let $m^S$ be part of a group $G$ that is collectible in $\langle e, M \rangle$. If $m^S$ is noncollectible in $\{m^S_1, M\}$, then $m^S_1$ must also be in $G$.

$$\text{gc}(G, \langle e, M \rangle) \wedge m^S \in G \Rightarrow nc(m^S, \{m^S_1, M\}) \Rightarrow m^S_1 \in G$$

Proof. By induction on the derivation of $nc(m^S, \{m^S_1, M\})$.

Case NC1:

We have root($m^S, m^S_1$). From this, it follows that $m^S_1 = m^S$, and the result follows trivially.

Case NC2:

We have $M(m^S_1) = \{x_i = \bar{v}_i\}$, and $nc(m^S, \langle v_c, M \rangle)$ for some $c$. Suppose root($m^S_2, v_c$) for some $m^S_2 \in G$. From this, we know root($m^S_2, \langle v_c, M \rangle$), and from the definition of $\text{gc}(G, \langle e, M \rangle)$, we have $m^S_1 \in G$, as desired.

We proceed by cases according to the derivation of $nc(m^S, \langle v_c, M \rangle)$ to find such an $m^S_2$.

Case NC1:

We have root($m^S, v_c$), so choose $m^S_2 = m^S$.

Case NC2:

We have root($m^S_1, v_c$), $M(m^S_1) = \{x_i = \bar{u}_i\}$, and $nc(m^S, \langle u_c, M \rangle)$ for some $m^S_3$ and some $c'$. It therefore follows that $nc(m^S, \{m^S_1, M\})$. So, we can apply the induction hypothesis to obtain $m^S_3 \in G$. Therefore, we can choose $m^S_2 = m^S_3$.

Lemma 20. All locations in a collectible group are collectible:

$$\text{gc}(G, \langle e, M \rangle) \wedge m^S \in G \Rightarrow \neg nc(m^S, \langle e, M \rangle).$$

Proof. By contradiction. Let $m^S \in G$ be such that $nc(m^S, \langle e, M \rangle)$. We proceed by cases according to the derivation of $nc(m^S, \langle e, M \rangle)$.

Case NC1:

We have root($m^S, e$). But from the definition of $\text{gc}(G, \langle e, M \rangle)$, no such $m^S$ can exist; a contradiction.
Case NC2:

We have \( \text{root}(m^S_1, e), M(m^S_1) = \{ \overline{x_i} = \overline{v_i} \} \), and \( \text{nc}(m^S, \langle \nu_v, M \rangle) \) for some \( c \). From this, it follows that \( \text{nc}(m^S, \{(m^S_1, M)\}) \).

Therefore, by Lemma 19, we have \( m^S_1 \in G \). So, from the definition of \( \text{gc}(G, \langle e, M \rangle) \), we have \( \neg \text{root}(m^S_1, e) \), a contradiction.

\( \square \)

Lemma 21. Let \( C \subseteq \text{dom}(M) \) be a set of locations that are collectible in a configuration \( \langle e, M \rangle \):

\[
\forall m^S \in C. \neg \text{nc}(m^S, \langle e, M \rangle).
\]

Then there is a collectible group that contains \( C \). In particular, let \( G \) be the largest superset of \( C \) such that from every location in \( G \), some location in \( C \) is reachable through a chain of hard references:

\[
\forall m^S_0 \in G. \exists m^S_1 \in C. \text{nc}(m^S_1, \{(m^S_0, M)\}).
\]

Then \( G \) is a collectible group: \( \text{gc}(G, \langle e, M \rangle) \).

Proof. Suppose \( G \) is not a collectible group. Then either \( G \) contains a GC root in \( e \), or there is a location outside \( G \) with a hard reference into \( G \).

Suppose \( G \) contains a GC root \( m^S \) in \( e \): \( m^S \in G \Rightarrow \text{root}(m^S, e) \). By construction of \( G \), let \( m^S_1 \in C \) be a location reachable through a chain of hard references from \( m^S \): \( \text{nc}(m^S_1, \{(m^S, M)\}) \). If \( M(m^S) = \bot \), then from the derivation of \( \text{nc}(m^S_1, \{(m^S, M)\}) \), we must have \( m^S = m^S_1 \), and so from \( \text{root}(m^S, e) \), we know \( \text{nc}(m^S_1, \langle e, M \rangle) \), a contradiction. Otherwise, assume \( M(m^S) \neq \bot \) and \( m^S \neq m^S_1 \). Let \( \overline{v_i} \) be such that \( M(m^S) = \{ \overline{x_i} = \overline{v_i} \} \). From the derivation of \( \text{nc}(m^S_1, \{(m^S, M)\}) \), we know there exists a \( c \) such that \( \text{nc}(m^S_1, \langle \nu_v, M \rangle) \). Therefore, by NC2, we have \( \text{nc}(m^S_1, \langle e, M \rangle) \), a contradiction.

Otherwise, let \( m^S \notin G \) be such that \( M(m^S) \) has a hard reference to some \( m^S_0 \in G: \text{root}(m^S_0, M(m^S)) \). By construction of \( G \), let \( m^S_1 \in C \) be a location reachable through a chain of hard references from \( m^S_0 \): \( \text{nc}(m^S_1, \{(m^S_0, M)\}) \).

We presently show that \( \text{nc}(m^S_1, \{(m^S, M)\}) \). If \( M(m^S_0) = \bot \), then from the derivation of \( \text{nc}(m^S_1, \{(m^S_0, M)\}) \), we must have \( m^S_0 = m^S_1 \), and so from \( \text{root}(m^S_0, M(m^S)) \), we know \( \text{nc}(m^S_1, \{(m^S, M)\}) \). Otherwise, assume \( M(m^S_0) \neq \bot \) and \( m^S_0 \neq m^S_1 \). Let \( \overline{v_i} \) be such that \( M(m^S_0) = \{ \overline{x_i} = \overline{v_i} \} \). From the derivation of \( \text{nc}(m^S_1, \{(m^S_0, M)\}) \), we know there exists a \( c \) such that \( \text{nc}(m^S_1, \langle \nu_v, M \rangle) \). Therefore, by NC2, we have \( \text{nc}(m^S_1, \{(m^S, M)\}) \).

So, we know \( \text{nc}(m^S_1, \{(m^S, M)\}) \). It therefore follows that \( \text{nc}(m^S_1, \langle m^S, M \rangle) \). So, \( G \cup \{m^S\} \) is a set larger than \( G \) satisfying property (13), a contradiction.

\( \square \)

Lemma 22. Let \( e_1 \) and \( e_2 \) be well-typed related expressions, and suppose \( \phi(m^S) \) is a high-authority GC root of \( e_2 \). Then \( m^S \) is a GC root of \( e_1 \).

\[
e_1 \equiv_a e_2 \land \Gamma; pc; H \vdash e_1 : \tau \land \Gamma; pc; H \vdash e_2 : \tau \land \Gamma
\]

\( \land \vdash \alpha \equiv \text{auth}^\tau(S) \land \text{root}(\phi(m^S), e_2) \)

\( \Rightarrow \text{root}(m^S, e_1) \)

Proof. By induction on the derivation of \( \text{root}(\phi(m^S), e_2) \). Since \( e_2 \) is well-typed, by Lemma 15, we know that case R13 holds vacuously.

\( \square \)

Lemma 23. Suppose \( M_1 \) and \( M_2 \) are well-formed memories. Let \( \phi \) be a high-integrity homomorphism such that \( M_1 \equiv_{\phi} M_2 \). Let \( m^S \) and \( m^S_1 \) be locations in \( M_1 \) mapped by \( \phi \). Assume \( m^S_1 \) is high-authority and high-persistence. If \( \text{nc}(\phi(m^S_1), \{(\phi(m^S), M_2)\}) \), then \( \text{nc}(m^S_1, \{(m^S, M_1)\}) \):

\[
\vdash^a_{\text{uf}} M_1 \land \vdash^a_{\text{uf}} M_2 \land M_1 \equiv_{\phi} M_2
\]

\( \land \vdash \alpha \equiv \text{auth}^\tau(S_1) \cap \text{persist}(S_1) \land \text{nc}(\phi(m^S_1), \{(\phi(m^S), M_2)\}) \)

\( \Rightarrow \text{nc}(m^S_1, \{(m^S, M_1)\}) \).

40
Proof. By induction on the derivation of $\text{nc}(\phi(m_{1}^{S}), \{\phi(m^{S}), M_{2}\})$.

Case NC1:

We must have $\phi(m^{S}) = \phi(m_{1}^{S})$. Since $\phi$ is injective, we know $m^{S} = m_{1}^{S}$, so the result follows by NC1.

Case NC2:

We know there exists some $\overline{u}$ and some $c$ such that $M_{2}(\phi(m^{S})) = \{\overline{x}_{i} = \overline{u}_{i}\}$ and $\text{nc}(\phi(m_{1}^{S}), (u_{c}, M_{2}))$. Assume $\phi(m^{S}) = \phi(m_{1}^{S})$. (Otherwise, the argument for Case NC1 applies.) Since $\phi(m_{1}^{S})$ is high-authority and $\text{nc}(\phi(m_{1}^{S}), (\phi(m^{S}), M_{2}))$, by Corollary 18, we know that $m^{S}$ is high-integrity, high-authority and high-persistence. Since $M_{1} \equiv_{\mathcal{A}} M_{2}$ and $M_{2}(\phi(m^{S})) \neq \bot$, then we must have $M_{1}(m^{S}) \neq \bot$. So let $\overline{v}_{i}$ be such that $M_{1}(m^{S}) = \{\overline{x}_{i} = \overline{v}_{i}\}$.

We show that $\text{nc}(m_{1}^{S}, (v_{c}, M_{1}))$. From this, the result follows via an application of NC2:

$$\begin{align*}
\root{(m^{S}, m^{S})} & \quad M_{1}(m^{S}) = \{\overline{x}_{i} = \overline{v}_{i}\} & \quad \text{nc}(m_{1}^{S}, (v_{c}, M_{1}))
\end{align*}$$

Consider the two cases in the derivation of $\text{nc}(\phi(m_{1}^{S}), (u_{c}, M_{2}))$:

Sub-case NC1:

We have $\root{(\phi(m_{1}^{S}), u_{c})}$. From $M_{1} \equiv_{\mathcal{A}} M_{2}$, we know $v_{c} \equiv_{\mathcal{A}} u_{c}$. From $\vdash_{\mathcal{U}_{\mathcal{A}}} M_{1}$ and $\vdash_{\mathcal{U}_{\mathcal{A}}} M_{2}$, we know $\emptyset; \top \vdash v_{c} : \tau_{c}, \top$ and $\emptyset; \top \vdash u_{c} : \tau_{c}, \top$, for some $\tau_{c}$. Therefore, we can apply Lemma 22 to get $\root{(m_{1}^{S}, v_{c})}$, and the result follows via NC1.

Sub-case NC2:

Assume $\neg\root{(\phi(m_{1}^{S}), u_{c})}$. (Otherwise, the argument in sub-case NC1 applies.)

We know there exists some $m_{2}^{S} \in \text{dom}(M_{2})$, some $\overline{u}'_{i}$ and some $c'$ such that $\root{(m_{2}^{S}, u_{c})}$, $M_{2}(m_{2}^{S}) = \{\overline{x}_{i} = \overline{u}'_{i}\}$, and $\text{nc}(\phi(m_{1}^{S}), (\overline{u}'_{c}, M_{2}))$. From this, it follows that $\text{nc}(\phi(m_{1}^{S}), (m_{2}^{S}, M_{2}))$. Since $\phi(m_{1}^{S})$ is high-authority and $\text{nc}(\phi(m_{1}^{S}), (m_{2}^{S}, M_{2}))$, by Corollary 18, we know that $m_{2}^{S}$ is high-integrity, high-authority, and high-persistence.

Since $\phi$ is a high-integrity homomorphism, there exists an $m_{3}^{S}$ such that $m_{2}^{S} = \phi(m_{3}^{S})$. Therefore, we can apply the induction hypothesis to get $\text{nc}(m_{1}^{S}, (m_{3}^{S}, M_{1}))$.

We have $\root{(\phi(m_{1}^{S}), u_{c})}$. From $M_{1} \equiv_{\mathcal{A}} M_{2}$, we know $v_{c} \equiv_{\mathcal{A}} u_{c}$. From $\vdash_{\mathcal{U}_{\mathcal{A}}} M_{1}$ and $\vdash_{\mathcal{U}_{\mathcal{A}}} M_{2}$, we know $\emptyset; \top \vdash v_{c} : \tau_{c}, \top$ and $\emptyset; \top \vdash u_{c} : \tau_{c}, \top$, for some $\tau_{c}$. Therefore, we can apply Lemma 22 to get $\root{(m_{3}^{S}, v_{c})}$. Since $m_{3}^{S} = \phi(m_{1}^{S})$, from $\neg\root{(\phi(m_{1}^{S}), u_{c})}$ and $\root{(m_{2}^{S}, u_{c})}$, we know $m_{3}^{S} \neq m_{1}^{S}$. Therefore, from the derivation of $\text{nc}(m_{1}^{S}, (m_{3}^{S}, M_{1}))$, we know there exists some $\overline{v}_{i}'$ and some $c''$ such that $M_{1}(m_{3}^{S}) = \{\overline{x}_{i} = \overline{v}_{i}'\}$ and $\text{nc}(m_{1}^{S}, (v_{c}', M_{1}))$.

The result follows via NC2:

$$\begin{align*}
\root{(m_{3}^{S}, v_{c})} & \quad M_{1}(m_{3}^{S}) = \{\overline{x}_{i} = \overline{v}_{i}'\} & \quad \text{nc}(m_{1}^{S}, (v_{c}', M_{1}))
\end{align*}$$

Lemma 24. Let $G$ be a collectible group in $(e, M)$ with $m_{1}^{S} \in G$. If $\text{nc}(m_{1}^{S}, (m^{S}, M))$, then $m^{S} \in G$:

$$\text{gc}(G, (e, M)) \land m_{1}^{S} \in G \land \text{nc}(m_{1}^{S}, (m^{S}, M)) \Rightarrow m^{S} \in G$$

Proof. By induction on the derivation of $\text{nc}(m_{1}^{S}, (m^{S}, M))$. 

41
By induction on the derivation of \( m^S \), then we must have \( m^S = m_1^{S_1} \), so the result follows trivially.

Case NC2:

We know there exists some \( v_i \) and some \( e \) such that \( M(m^S) = \{ x_i = v_i \} \) and \( nc(m_1^{S_1},(v,c,M)) \). Consider the two cases in the derivation of \( nc(m_1^{S_1},(v,c,M)) \):

Sub-case NC1:

We have \( \text{root}(m_1^{S_1},v,c) \), so we therefore know \( \text{root}(m_1^{S_1},M(m^S)) \). The result then follows from the definition of \( gc(G,(e,M)) \).

Sub-case NC2:

We know there exists some location \( m_2^{S_2} \) such that \( \text{root}(m_2^{S_2},v,c) \) and \( nc(m_1^{S_1},m_2^{S_2},M(m^S)) \). So, by the induction hypothesis, we know \( m_2^{S_2} \in G \). From \( \text{root}(m_2^{S_2},v,c) \), we know \( \text{root}(m_2^{S_2},M(m^S)) \). The result then follows from the definition of \( gc(G,(e,M)) \).

\[ \Box \]

**Lemma 25.** Let \( M_1 \) and \( M_2 \) be well-formed memories. Suppose \( (e_1,M_1) \approx_{\alpha} (e_2,M_2) \). Let \( G \) be a collectible group in \( (e_1,M_1) \), and let \( \phi(G) \) denote \( \{ \phi(m^S) : m^S \in G \cap \text{dom}(\phi) \} \). Let \( C \) represent the high-authority, high-persistence members of \( G \cap \text{dom}(\phi) \):

\[ C = \{ m^S \in G \cap \text{dom}(\phi) : \alpha \in \text{auth}^+(S) \cap \text{persist}(S) \} \]

Then there exists a set \( G' \) such that \( \phi(G') \) is a subset of \( \phi(G) \), is a collectible group in \( (e_2,M_2) \), and contains all members of \( \phi(C) \):

\[ \tau_{[\text{loc}]} M_1 \land \tau_{[\text{prod}]} M_2 \land (e_1,M_1) \approx_{\alpha} (e_2,M_2) \land gc(G,(e_1,M_1)) \]

\[ \Rightarrow \exists G'. \phi(G'), (e_2,M_2) \land \phi(C) \subseteq \phi(G') \subseteq \phi(G) \]

**Proof.** First, we show that \( \phi(C) \) is a set of collectible locations. Suppose it isn’t. Then let \( m^S \in G \cap \text{dom}(\phi) \) be such that \( \phi(m^S) \notin \phi(C) \) is noncollectible: \( nc(\phi(m^S),(e_2,M_2)) \). Since \( \phi(m^S) \) is high-authority and high-persistence, by induction on the derivation of \( nc(\phi(m^S),(e_2,M_2)) \), we can show that \( m^S \) must be noncollectible: \( nc(m^S,(e_1,M_1)) \). Therefore, by Lemma 20, \( G \) cannot be a collectible group, a contradiction.

Let \( G' \subseteq \text{dom}(\phi) \) be such that \( \phi(G') \) is the largest superset of \( \phi(C) \) such that from every location in \( \phi(G') \), some location in \( \phi(C) \) is reachable through a chain of hard references: \( \forall m^S_0 \in \phi(G'). \exists m^S_1 \in \phi(C). \; nc(m^S_1,(m^S_0,M_2)) \). By Lemma 21, we know \( \phi(G') \) is a collectible group.

We now show that \( \phi(G') \) is also a subset of \( \phi(G) \) by showing \( G' \subseteq G \). Suppose \( m^S \in G' \). By construction of \( G' \), let \( m_1^{S_1} \in C \) be such that \( nc(\phi(m_1^{S_1}),(m^S,M_2)) \). From this, by Lemma 23, we know \( nc(m_1^{S_1},M^S_1) \). By Lemma 24, then, we must have \( m^S \in G \). So \( G' \subseteq G \).

Lemmas 26–29 prove a few results about the preservation of the security relation under various conditions

**Lemma 26** (Relation substitution). If \( e_1 \approx_{\alpha} e_2 \) and \( e_1' =_{\alpha} e_2' \), then \( e_1 \{ e_1'/x \} \approx_{\alpha} e_2 \{ e_2'/x \} \).

**Proof.** By induction on the derivation of \( e_1 \approx_{\alpha} e_2 \).

**Lemma 27** (Security-relation preservation under auto-bracketing). Auto-bracketing preserves the security relation:

\[ e_1 \approx_{\alpha} e_2 \Rightarrow e_1 \bullet_{\alpha} \tau \approx_{\alpha} e_2 \bullet_{\alpha} \tau. \]

**Proof.** By induction on the derivation of \( e_1 \approx_{\alpha} e_2 \).

**Lemma 28.** Suppose \( e_1 \approx_{\alpha} e_2 \). Let \( m^S \) be such that \( m^S \not\in \text{locs}(e_1) \) and let \( \phi' = \phi[m^S \mapsto \ell_1] \). Then \( e_1 \approx_{\phi'} e_2 \).

**Proof.** By induction on the derivation of \( e_1 \approx_{\phi} e_2 \).
Lemma 29. Evaluating in a low-integrity context preserves the security relation on memories.

Let \( \langle e_1, M_1 \rangle \) be a well-formed configuration, wherein \( e_1 \) is well-typed in a low-integrity context. Let \( M_2 \) be a well-formed memory and suppose \( \langle e_1, M_1 \rangle \xrightarrow{\varphi} \langle e'_1, M'_1 \rangle \).

Then the following holds.

1. \( M_1 \stackrel{\varphi}{=} M_2 \Rightarrow M'_1 \stackrel{\varphi}{=} M_2 \) and
2. \( \vdash_{\varphi} \langle e_1, M_1 \rangle \land M_2 \stackrel{\varphi}{=} M_1 \Rightarrow M_2 \stackrel{\varphi}{=} M'_1 \).

Proof. If we can show this is true for the case where a single \( \xrightarrow{\varphi} \) step is taken, then the rest follows by induction on the number of \( \xrightarrow{\varphi} \) steps taken. (We know the induction hypothesis will apply because of Corollary 11 and Corollary 12.)

We show the single-step case by induction on the derivation of \( \langle e_1, M_1 \rangle \xrightarrow{\varphi} \langle e'_1, M'_1 \rangle \). The proof proceeds by cases according to the evaluation rules.

In cases SELECT, DANGLE-SELECT, SOFT-SELECT, DANGLE-ASSIGN, APPLY, EXISTS-TRUE, EXISTS-FALSE, TRY-VAL, TRY-CATCH, TRY-ESC, PARALLEL-RESULT, IF-TRUE, IF-FALSE, LET, FAIL-PROP, BRACKET-SELECT, BRACKET-SOFT-SELECT, BRACKET-ASSIGN, BRACKET-SOFT-ASSIGN, BRACKET-SOFT, BRACKET-EXISTS, BRACKET-APPLY, BRACKET-TRY, BRACKET-IF, BRACKET-LET, DOUBLE-BRACKET, and BRACKET-Fail, the result holds trivially, since \( M'_1 = M_1 \).

Case CREATE (\( \langle \{ \bar{x}_1 = \bar{v}_1 \}^S, M_1 \rangle \xrightarrow{\varphi} \langle m^S, M_1[m^S \mapsto \{ \bar{x}_1 = \bar{v}_1 \bullet \bar{\alpha} \bar{\tau}_1 \}] \rangle \), where \( m \) is fresh and \( S = \{ \bar{x}_1 : \bar{\tau}_1 \}_{(a,p)} \):

1. Suppose \( M_1 \stackrel{\varphi}{=} M_2 \). We show that \( M'_1 \stackrel{\varphi}{=} M_2 \).

   From the derivation of \( \emptyset ; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \), we know \( \vdash pc \land \text{integ}(\tau) \leq pc \) for all \( i \). Therefore, we know \( m^S \) is neither high-integrity nor high-persistence. The result then follows from the fact that \( m^S \not\models \text{dom}(\bar{\alpha}) \) and the assumption \( M_1 \stackrel{\varphi}{=} M_2 \).

2. Suppose \( \vdash_{\varphi} \langle e_1, M_1 \rangle \) and \( M_2 \stackrel{\varphi}{=} M_1 \). We show that \( M_2 \stackrel{\varphi}{=} M'_1 \).

   This follows from \( M_2 \stackrel{\varphi}{=} M_1 \).

Case ASSIGN (\( \langle m^S, x_c := v, M_1 \rangle \xrightarrow{\varphi} \langle \bullet \bar{\alpha} p, M_1[m^S \mapsto v \bullet \bar{\alpha} \bar{\tau}_c] \rangle \), where \( M_1(m^S) \neq \bot \) and \( S = \{ \bar{x}_1 : \bar{\tau}_1 \}_{(a,p)} \): Let \( \bar{v}' \) and \( \bar{u}_1 \) be such that \( M'_1(m^S) = \{ \bar{x}_1 = \bar{v}' \} \) and \( M_2(\bar{\phi}(m^S)) = \{ \bar{x}_1 = \bar{u}_1 \} \). We therefore have \( v_c' = v \bullet \bar{\alpha} \bar{\tau}_c \).

   From \( \emptyset ; pc; \mathcal{H} \vdash m^S, x_c := v : \tau, \mathcal{X} \), we know \( \vdash \tau \cap pc \leq \tau_c \), for some \( \tau' \). Therefore, we know \( \vdash \text{integ}(\tau_c) \leq pc \). So, from \( \bar{\alpha} \not\models pc \), we have \( \bar{c} \not\models \text{integ}(\tau_c) \). Therefore, \( v_c' \) is a bracketed value: there exists a \( v_c'' \) such that \( v_c' = [v_c''] \).

   From \( \vdash_{\varphi} \langle e_1, M_1 \rangle \) and \( M_2 \stackrel{\varphi}{=} M_1 \), we also know \( u_c = \) a bracketed value: \( u_c = [u_c'] \), for some \( u_c' \).

1. Suppose \( M_1 \stackrel{\varphi}{=} M_2 \). We show that \( M'_1 \stackrel{\varphi}{=} M_2 \).

   Assume \( m^S \in \text{dom}(\bar{\phi}) \) (otherwise, the result follows directly from \( M_1 \stackrel{\varphi}{=} M_2 \)). Since \( M_1 \stackrel{\varphi}{=} M_2 \), it suffices to show \( v_c' \stackrel{\varphi}{=} u_c \). This is trivial, since \( v_c' = [v_c''] \) and \( u_c = [u_c'] \).

2. Suppose \( \vdash_{\varphi} \langle e_1, M_1 \rangle \) and \( M_2 \stackrel{\varphi}{=} M_1 \). We show that \( M_2 \stackrel{\varphi}{=} M'_1 \).

   Assume \( m^S \in \text{im}(\bar{\phi}) \) (otherwise, the result follows directly from \( M_2 \stackrel{\varphi}{=} M_1 \)). Since \( M_2 \stackrel{\varphi}{=} M_1 \), it suffices to show \( u_c \stackrel{\varphi}{=} v_c' \). This is trivial, since \( u_c = [u_c'] \) and \( v_c' = [v_c''] \).

Case SOFT-ASSIGN (\( \langle \text{soft} m^S, x_c := v, M_1 \rangle \xrightarrow{\varphi} \langle e'_1 \bullet \bar{\alpha} (a \cap p), M'_1 \rangle \), where \( \langle m^S, x_c := v, M_1 \rangle \xrightarrow{\varphi} \langle e'_1, M'_1 \rangle \) and \( S = \{ \bar{x}_1 : \bar{\tau}_1 \}_{(a,p)} \): We proceed by cases according to the evaluation rules for \( \langle m^S, x_c := v, M_1 \rangle \xrightarrow{\varphi} \langle e'_1, M'_1 \rangle \).
Sub-case ASSIGN (\(\{m^\delta, x_i := v, M_1\} \xrightarrow{\delta} \{\ast \cdot \alpha p, M_1[m^\delta, x_i \mapsto v \cdot \alpha \tau_e]\}\)):

Let \(v'_e\) and \(u'_e\) be such that \(M'_1(m^\delta) = \{x_i = v'_e\}\) and \(M_2(\phi(m^\delta)) = \{x_i = u'_e\}\). We therefore have \(v'_e = v \cdot \alpha \tau_e\).

From \(\forall \alpha \cdot pc; \mathcal{H} \vdash E[x_1] \cdot \tau, \mathcal{X}\), we know \(\vdash \tau' \cap pc \leq \tau_e\), for some \(\tau'\). Therefore we know \(\vdash \text{integ}(\tau_e) \leq pc\). So, from \(\alpha \notin pc\), we have \(\alpha \notin \text{integ}(\tau_e)\). Therefore, \(v'_e\) is a bracketed value: there exists a \(v''_e\) such that \(v'_e = [v''_e]\).

From \(\tau'_M\) and \(\alpha \notin \text{integ}(\tau_e)\), we also know \(u_e\) is a bracketed value: \(u_e = [u'_e]\), for some \(u'_e\).

1. Suppose \(M_1 \equiv^\delta M_2\). We show \(M'_1 \equiv^\delta M_2\).

Assume \(m^\delta \in \text{dom}(\phi)\) (otherwise, the result follows directly from \(M_1 \equiv^\delta M_2\)). Since \(M_1 \equiv^\delta M_2\), it suffices to show \(v'_e \equiv^\delta u_e\). This is trivial, since \(v'_e = [v''_e]\) and \(u_e = [u'_e]\).

2. Suppose \(\tau'_M\) \((e_1, M_1)\) and \(M_2 \equiv^\delta M_1\). We show \(M'_2 \equiv^\delta M'_1\).

Assume \(m^\delta \in \text{im}(\phi)\) (otherwise, the result follows directly from \(M_2 \equiv^\delta M_1\)). Since \(M_2 \equiv^\delta M_1\), it suffices to show \(u_e \equiv^\delta v'_e\). This is trivial, since \(u_e = [u'_e]\) and \(v'_e = [v''_e]\).

Sub-case DANGLE-ASSIGN (\(\{m^\delta, x_i := v, M_1\} \xrightarrow{\delta} \{\bot \cdot \alpha p, M_1\}\)):

The result holds trivially, since \(M'_1 = M_1\).

Case EVAL-CONTEXT (\(\{E[e_3], M_1\} \xrightarrow{\delta} \{E[e'_3], M'_1\}\), where \(e_3, M_1 \xrightarrow{\delta} \{e'_3, M'_1\}\)):

A case analysis on the syntax of \(E[\cdot]\) shows that from the derivation of \(\forall \alpha \cdot pc; \mathcal{H} \vdash E[e_3] \cdot \tau, \mathcal{X}\), we know \(\forall \alpha \cdot pc; \mathcal{H}' \vdash e_3 \cdot \tau', \mathcal{X}'\), for some \(\mathcal{H}', \tau', \mathcal{X}'\). Therefore, we can apply the induction hypothesis and obtain the result.

Case BRACKET-CONTEXT (\(\{[e_3], M_1\} \xrightarrow{\delta} \{[e'_3], M'_1\}\), where \(e_3, M_1 \xrightarrow{\delta} \{e'_3, M'_1\}\)):

From the derivation of \(\forall \alpha \cdot pc; \mathcal{H} \vdash [e_3] \cdot \tau, \mathcal{X}\), we know \(\forall \alpha \cdot pc \cap \ell; \mathcal{H} \vdash e_3 \cdot \tau', \mathcal{X}\), where \(\alpha \notin \ell\) and \(\tau = \tau' \cap \ell\). Therefore, we can apply the induction hypothesis and obtain the result.

We can now prove Theorem 1, restated below for convenience.

**Theorem 1** (Referential integrity). Assume \(\{e_1, M_1\}\) is a well-formed configuration and \(\{e_2, M_2\}\) is a well-formed, nonadversarial, \(\phi\)-related configuration, such that \(e_1\) and \(e_2\) have type \(\tau\) and \(M_2\) is well-formed:

\[\tau'_M\} \{e_1, M_1\} \land \tau'_M\} \{e_2, M_2\} \land \{e_1, M_1\} \equiv^\phi \{e_2, M_2\}\]
\[\forall \alpha \cdot pc; \mathcal{H} \vdash e_1 \cdot \tau, \mathcal{X} \land \forall \alpha \cdot pc; \mathcal{H} \vdash e_2 \cdot \tau, \mathcal{X} \land \tau'_M\} \{e_1, M_1\} \equiv \{e_2, M_2\}\]

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

\[\{e_1, M_1\} \xrightarrow{\alpha} \{e'_1, M'_1\} \land \neg \{e_2, M_2\} \Uparrow\]
\[\exists \{e'_2, M'_2\}, \phi'. \{e_2, M_2\} \xrightarrow{\alpha} \{e'_2, M'_2\} \land \{e'_1, M'_1\} \equiv^\phi \{e'_2, M'_2\} \land \phi = \phi|_{\text{dom}(\phi)}\]

**Proof.** If we can show this is true for the case where a single \(\rightarrow^\alpha\) step is taken to reach \(\{e'_1, M'_1\}\), then the rest follows by induction on the number of \(\rightarrow^\alpha\) steps taken. (We know the induction hypothesis will apply because of Corollary 11 and the fact that \(\forall \alpha \cdot pc; \mathcal{H} \vdash e_1 \cdot \tau, \mathcal{X} \land \forall \alpha \cdot pc; \mathcal{H} \vdash e_2 \cdot \tau, \mathcal{X} \land \tau'_M\} \{e_1, M_1\} \equiv \{e_2, M_2\}\).)

We show the single-step case by induction on the derivation of \(\{e_1, M_1\} \rightarrow^\alpha \{e'_1, M'_1\}\). The proof proceeds by cases according to the evaluation rules. For each case, we need to show two things about \(e'_2, M'_2\) and \(\phi'\):

i. Related expressions: \(e'_1 \equiv^\phi e'_2\)

ii. Related memories: \(M'_1 \equiv^\phi M'_2\).
It will be obvious by its construction that \( \phi' \) is a high-integrity homomorphism that extends \( \phi \).

Case CREATE \( \{ \{ \overline{x_i} = \overline{v_i} \}^S, M_1 \} \rightarrow \alpha \{ m_1^S, M_1[ m_1^S \rightarrow \overline{x_i} = \overline{v_i} \cdot \alpha \overline{\tau}_i ] \} \), where \( m_1 \) is fresh and \( S = \{ \overline{x_i} : \overline{\tau}_i \} \):

From \( e_1 \alpha\overline{u}_1 \overline{v}_2 \), we know \( e_2 = \{ \overline{x_i} = \overline{u}_i \}^S \) with \( \overline{v}_i \alpha\overline{u}_i \) for all \( i \). By CREATE,

\[ \langle e_2, M_2 \rangle \rightarrow \{ m_2^S, M_2[ m_2^S \rightarrow \overline{x_i} = \overline{u_i} \cdot \alpha \overline{\tau}_i ] \} \],

where \( m_2 \) is fresh. Choose \( \phi' = \phi \{ m_1^S \rightarrow m_2^S \} \).

i. We need to show \( m_1^S \approx_{\alpha} m_2^S \).

This follows by construction of \( \phi' \).

ii. We need to show \( M_1 \approx_{\alpha} M_2 \), where \( M_1 = M_1[ m_1^S \rightarrow \overline{x_i} = \overline{v_i} \cdot \alpha \overline{\tau}_i ] \) and \( M_2 = M_2[ m_2^S \rightarrow \overline{x_i} = \overline{u_i} \cdot \alpha \overline{\tau}_i ] \).

First, let \( m_1^S \in \text{dom}(\phi') \) be such that \( M_1(\phi(\phi')) \neq \perp \). We show that \( M_2(\phi'(\phi')) \neq \perp \) and \( M_1'(\phi'(\phi')) \approx_{\alpha} M_2'(\phi'(\phi')) \).

If \( m_1^S = m_2^S \), then \( M_1'(\phi'(\phi')) = M_1'(\phi' \overline{\phi'}) \) and \( M_2'(\phi'(\phi')) = M_2'(\phi' \overline{\phi'}) \). So the result follows from the assumption \( e_1 \alpha\overline{u}_1 \overline{v}_2 \) via Lemmas 27 and 28. Otherwise, \( m_1^S \neq m_2^S \), so \( M_1'(\phi'(\phi')) = M_1(\phi'(\phi')) \) and \( M_2'(\phi'(\phi')) = M_2(\phi'(\phi')) \). The result therefore follows from the assumption \( M_1 \approx_{\alpha} M_2 \).

Now, let \( m_1^S \in \text{dom}(\phi') \) be such that \( \vdash \alpha \leq \text{auth}^+(\overline{S'}) \cap \text{persist}(\overline{S'}) \) and \( M_1'(\phi'(\phi')) = \perp \). We show that \( M_2'(\phi'(\phi')) = \perp \). Since \( M_1'(\phi'(\phi')) = \perp \), we must have \( m_1^S = m_1^S \), so \( M_1'(\phi'(\phi')) = M_1'(m_1^S) = M_2'(m_1^S) = M_2(\phi'(\phi')) \). The result therefore follows from the assumption \( M_1 \approx_{\alpha} M_2 \).

Case SELECT \( \{ m_1^S, x_i, M_1 \} \rightarrow \alpha \{ \overline{v}_c \cdot \alpha \overline{p}, M_1 \} \), where \( S = \{ \overline{x_i} : \overline{\tau}_i \} \) and \( M_1(\overline{x_i}) = \{ \overline{v_i} = \overline{v_i} \} \):

From \( e_1 \alpha\overline{u}_1 \overline{v}_2 \), we know \( e_2 = m_1^S \cdot x_i \), where \( \phi(m_1^S) = m_2^S \). Therefore, from \( M_1 \approx_{\alpha} M_2 \), we have \( M_2(\overline{x_i}) = \{ \overline{x_i} = \overline{u}_i \} \) for some \( \overline{u}_i \), where \( \overline{v}_i \alpha\overline{u}_i \). So, by SELECT, \( \langle e_2, M_2 \rangle \rightarrow \{ \overline{u}_c \cdot \alpha \overline{p}, M_2 \} \). Choose \( \phi' = \phi \).

i. We need to show \( \overline{v}_c \cdot \alpha \overline{p} \approx_{\alpha} \overline{u}_c \cdot \alpha \overline{p} \).

This follows via Lemma 27.

ii. We need to show \( M_1 \approx_{\alpha} M_2 \).

This is given.

Case DANGLE-SELECT \( \{ m_1^S, x_i, M_1 \} \rightarrow \alpha \{ \overline{l}_p \cdot \alpha \overline{p}, M_1 \} \), where \( S = \{ \overline{x_i} : \overline{\tau}_i \} \) and \( M_1(\overline{x_i}) = \perp \):

From \( e_1 \alpha\overline{u}_1 \overline{v}_2 \), we know \( e_2 = m_1^S \cdot x_i \). Therefore, from \( M_1 \approx_{\alpha} M_2 \), we have \( M_2(\overline{x}_i) = \{ \overline{x}_i = \overline{u}_i \} \).

If \( M_2(\overline{x}_i) = \perp \), then by DANGLE-SELECT, \( \langle e_2, M_2 \rangle \rightarrow \{ \overline{l}_p \cdot \overline{x}_i, M_2 \} \).

Otherwise, without loss of generality, assume \( M_2(\overline{x}_i) = \{ \overline{x}_i = \overline{u}_i \} \). Since \( \alpha \neq p \), we have \( \overline{u}_c \cdot \alpha \overline{p} = \{ \overline{u}_c \} \) for some \( \overline{u}_c \). So, by SELECT, \( \langle e_2, M_2 \rangle \rightarrow \{ \overline{u}_c \cdot \overline{p}, M_2 \} \). Choose \( \phi' = \phi \).

i. We need to show that \( \overline{l}_p \cdot \alpha \overline{p} \approx_{\alpha} \overline{u} \) for some \( \overline{u} \in \{ \overline{u}_c, \overline{l}_p \} \).

This is trivial.

ii. We need to show that \( M_1 \approx_{\alpha} M_2 \).

This is given.

Case SOFT-SELECT \( \{ (\text{soft } m_1^S) \cdot x_i, M_1 \} \rightarrow \alpha \langle v', M_1 \rangle \), where \( m_1^S, x_i, M_1 \rightarrow \alpha \{ v, M_1 \}, S = \{ \overline{x_i} : \overline{\tau}_i \} \), and \( v = v \cdot \alpha (a \cap p) \):

From \( e_1 \alpha\overline{u}_1 \overline{v}_2 \), we know \( e_2 = ((\text{soft } m_2^S) \cdot x_i) \). If we can find \( u \) so that \( m_2^S \cdot x_i, M_2 \rightarrow \alpha \{ u, M_2 \} \), then by SOFT-SELECT, we have \( \langle e_2, M_2 \rangle \rightarrow \{ (\text{soft } m_2^S) \cdot x_i, M_2 \} \). Choose \( \phi' = \phi \).

We proceed by cases according to the evaluation rules for \( m_1^S \cdot x_i, M_1 \rightarrow \alpha \{ v, M_1 \} \).
Sub-case \( \text{SELECT} \ (v = \psi_c \cdot a \ p, \text{where } M_1(m_1^1) = \{ \overrightarrow{x_v = v_i} \}) \):

From \( M_1 \approx_\alpha^0 M_2 \), we know \( M_2(m_2^1) = \{ \overrightarrow{x_i = u_i} \} \) for some \( u_i \), where \( v_i \approx_\alpha^0 u_i \). So, by \( \text{SELECT} \), we have \( \{ m_2^1, x_c \cdot M_2 \} \xrightarrow{\psi_c \cdot a \ p} \{ u_c \cdot a \ p, M_2 \} \). Therefore, \( u = u_c \cdot a \ p \).

i. We need to show \( (v_c \cdot a \ p) \cdot a (\alpha \cap p) \approx_\alpha^0 (u_c \cdot a \ p) \cdot a (\alpha \cap p) \).

This follows via Lemma 27.

ii. We need to show \( M_1 \approx_\alpha^0 M_2 \).

This is given.

Sub-case \( \text{DANGLE-SELECT} \ (v = \bot \cdot a \ p, \text{where } M_1(m_1^2) = \bot) \):

First, suppose \( \vdash \alpha \leq a \cap p \). Then \( \psi_c \cdot a (\alpha \cap p) = \bot \cdot a, \ u \cdot a \cap p = u, \text{ and } \vdash a \leq p \). Therefore, we have \( m_1^2 \in \text{dom}(\phi) \), and so, \( \phi(m_1^2) = m_2^2 \). From \( M_1 \approx_\alpha^0 M_2 \), then, we know \( M_2(m_2^2) = \bot \). So, by \( \text{SELECT} \), we have \( \{ m_2^2, x_c \cdot M_2 \} \xrightarrow{\bot \cdot a \ p, M_2} \). Therefore, \( u = \bot \).

i. We need to show \( \bot \cdot a \approx_\alpha^0 \bot \cdot a \).

This is trivial.

ii. We need to show \( M_1 \approx_\alpha^0 M_2 \).

This is given.

Now, suppose \( \vdash \alpha \neq a \cap p \). Then \( \psi_c \cdot a (\alpha \cap p) = [\bot \cdot a] \). If \( M_2(m_2^2) = \bot \), then by \( \text{DANGLE-SELECT} \), \( \{ m_2^2, x_c \cdot M_2 \} \xrightarrow{\bot \cdot a \ p, M_2} \). Otherwise, without loss of generality, assume \( M_2(m_2^2) = \{ \overrightarrow{x_i = u_i} \} \). So, by \( \text{SELECT} \), \( \{ m_2^2, x_c \cdot M_2 \} \xrightarrow{\{ u_c \cdot a \ p, M_2 \}} \). Therefore, \( u \cdot a \cap p \in \{ \{ \bot \cdot a \cdot a \ p \} \cdot a \cap p \cdot a \cap p \cdot a \cap p \cdot a \cap p \} \) for some \( u_c \).

i. We need to show \( \{ \bot \cdot a \} \approx_\alpha^0 \{ u' \} \) for \( u' \in \{ u_c \cdot \bot \cdot a \} \).

This is trivial.

ii. We need to show \( M_1 \approx_\alpha^0 M_2 \).

This is given.

Case \( \text{ASSIGN} \ (\{ m_1^1, x_c := v, M_1 \} \rightarrow a \cdot \psi_c \cdot p, M_1[\{ m_1^1, x_c := \psi_c \cdot p, M_1 \}] \cdot a \cdot \tau_c] \), where \( M_1(m_1^1) = \bot \) and \( S = \{ \overrightarrow{x_v = v_i} \} (a \cdot p) \):

From \( e_1 = e_2 \) \( \text{ASSIGN} \), we know \( e_2 = (m_2^2, x_c := u) \) with \( \phi(m_2^2) = m_2^2 \) and \( \psi_c = \psi_c = u \). From \( M_1 \approx_\alpha^0 M_2 \), we have \( M_2(m_2^2) = \bot \). So, by \( \text{ASSIGN} \), \( \{ m_2^2, x_c := u, M_2 \} \rightarrow \{ a \cdot \psi_c \cdot p, M_2[\{ m_2^2, x_c := \psi_c \cdot p, M_2 \}] \cdot a \cdot \tau_c] \} \). Choose \( \psi_c = \psi_c \).

i. We need to show \( a \cdot \psi_c \cdot p \approx_\alpha^0 a \cdot \psi_c \cdot p \).

This follows via Lemma 27.

ii. We need to show \( M_1[\{ m_1^1, x_c := \psi_c \cdot p, M_1 \}] \approx_\alpha^0 M_2[\{ m_2^2, x_c := \psi_c \cdot p, M_2 \} \cdot a \cdot \tau_c] \).

First, let \( m_0^0 \in \text{dom}(\phi) \), such that \( M_1'(\phi(m_0^0)) \neq \bot \). We show that \( M_1'(\phi(m_0^0)) \neq \bot \) and \( M_1'(m_0^0) \approx_\alpha^0 M_2'(\phi(m_0^0)) \).

If \( m_0^0 = m_1^2 \), then \( \phi(m_0^0) = m_2^2 \). Let \( \overrightarrow{x' v} \) and \( \overrightarrow{x' v} \) be such that \( M_1'(m_1^1) = \{ \overrightarrow{x_i = v_i} \} \) and \( M_1'(m_2^2) = \{ \overrightarrow{x_i = u_i} \} \). Since \( \psi_c \approx_\alpha^0 u \), by Lemma 27, we have \( \psi_c \cdot a \cdot \tau_c = \psi_c \cdot a \cdot \tau_c \) and \( \psi_c \cdot a \cdot \tau_c = \psi_c \cdot a \cdot \tau_c \). Therefore, we have \( \psi_c \cdot a \cdot u' \). Therefore, from \( M_1 \approx_\alpha^0 M_2 \), it follows that \( \{ \overrightarrow{x_i = v_i} \} \approx_\alpha^0 \{ \overrightarrow{x_i = u_i} \} \).

Otherwise, \( m_0^0 = m_2^2 \), so \( M_1'(m_0^0) = M_1'(m_0^0) \) and \( M_2'(\phi(m_0^0)) = M_2'(\phi(m_0^0)) \). The result therefore follows from \( M_1 \approx_\alpha^0 M_2 \).

Now, let \( m_0^0 \in \text{dom}(\phi) \) be such that \( \vdash \alpha \leq \text{auth}^+ (S_0) \cdot \text{persist}(S_0) \) and \( M_1'(m_0^0) = \bot \). We show that \( M_1'(\phi(m_0^0)) = \bot \). This follows from \( M_1 \approx_\alpha^0 M_2 \) by construction of \( M_1' \) and \( M_2' \).

Case \( \text{DANGLE-ASSIGN} \ (\{ m_1^1, x_c := v, M_1 \} \rightarrow a \bot \cdot a \cdot p, M_1 \) \), where \( S = \{ \overrightarrow{x_v = v_i} \} (a \cdot p) \) and \( M_1(m^1) = \bot \):

From \( a \bot \cdot a \cdot p \), \( \{ m_1^1, x_c := v, M_1 \} \cdot nc(m_1^1, m_1^1, x_c := v, M_1 \}) \), and \( M_1(m^1) = \bot \), we know \( \vdash \alpha \neq p \). Therefore, \( \vdash \alpha \neq a \cdot p = [\bot \cdot a] \).

From \( e_1 = e_2 \), we know \( e_2 = m_2^2, x_c := u \) with \( \phi(m_2^2) = m_2^2 \) and \( \psi_c = \psi_c = u \). If \( M_2(m_2^2) = \bot \), then by \( \text{DANGLE-ASSIGN} \), \( (e_2, M_2) \rightarrow \{ \{ \bot \cdot a \cdot p \}, [M_2] \} \).

46
Otherwise, \( M_2(m^2_S) \neq 1 \). Since \( \alpha \neq p \), we have \( \star \cdot \alpha p = [*] \). So, by \ ASSIGN, \( e_2, M_2 \to ([*], M_2[m^3_S x_c \to u \cdot \alpha \tau_c]) \).

Choose \( \phi' = \phi \).

i. We need to show \( \downarrow_p =^\phi u' \) for \( u' \in \{*, \downarrow_p \} \).

This is trivial.

ii. We need to show \( M'_1 =^\phi M_2' \).

First, let \( m^0_S \in \text{dom} (\phi) \) be such that \( M'_1(m^0_S) \neq 1 \). We show that \( M'_2(\phi(m^0_S)) \neq 1 \) and \( M'_1(m^0_S) =^\phi M'_2(\phi(m^0_S)) \).

Since \( M'_1 = M_1 \) and \( M'_1 (m^0_S) \neq 1 \), we must have \( m^0_S \neq m^1_S \), so \( M'_1 (m^0_S) = M_1 (m^0_S) \) and \( M'_2(\phi(m^0_S)) = M_2(\phi(m^0_S)) \). The result therefore follows from \( M'_1 =^\phi M_2 \).

Now, let \( m^0_S \in \text{dom} (\phi) \) be such that \( \uparrow \alpha \leq \text{auth}^* (S_0) \cap \text{persist} (S_0) \) and \( M'_1 (m^0_S) = 1 \). We show that \( M'_2(\phi(m^0_S)) = 1 \). This follows from \( M_1 =^\phi M_2 \) by construction of \( M'_2 \).

Case \texttt{SOFT-ASSIGN} \( \langle (\text{soft } m^3_S)x_c := v, M_1 \rangle \to \langle v' \cdot (a \land p), M'_1 \rangle \), where \( \langle m^3_S x_c := v, M_1 \rangle \xrightarrow{\sim} (v', M'_1) \) and \( S = \{x_i := \tau_c\}(a,p)\):

From \( e_1 =^\phi e_2 \), we know \( e_2 = \langle ((\text{soft } m^3_S)x_c := u) \text{ with } \phi (m^0_S) = m^2_S \text{ and } v =^\phi u \rangle \). If we can find a configuration \( \langle u', M'_2 \rangle \) so that \( \langle m^3_S x_c := u, M_2 \rangle \xrightarrow{\sim} \langle u', M'_2 \rangle \), then by \texttt{SOFT-ASSIGN}, we have \( e_2, M_2 \to \langle (\text{soft } m^3_S)x_c := u, M_2 \rangle \to \langle u' \cdot \alpha (a \land p), M'_2 \rangle \). Choose \( \phi' = \phi \).

We proceed by cases according to the evaluation rules for \( \langle m^3_S x_c := v, M_1 \rangle \xrightarrow{\sim} v' \).

Sub-case \texttt{ASSIGN} \( v' = * \cdot \alpha p \) and \( M'_1 = M_1[m^3_S x_c \to v \cdot \alpha \tau_c] \), where \( M_1(m^1_S) \neq 1 \):

From \( M_1 =^\phi M_2 \), we know \( M_2(m^3_S) \neq 1 \). So, by \ ASSIGN, we have \( \langle m^3_S x_c := u, M_2 \rangle \xrightarrow{\sim} \langle * \cdot \alpha p, M_2[m^3_S x_c \to u \cdot \alpha \tau_c] \rangle \).

So \( u' = * \cdot \alpha p \) and \( M'_2 = M_2[m^3_S x_c \to u \cdot \alpha \tau_c] \).

i. We need to show \( * \cdot \alpha p \cdot (a \land p) \xrightarrow{\sim} (\star \cdot \alpha p) \cdot (a \land p) \).

This follows via Lemma 27.

ii. We need to show \( M_1[m^3_S x_c \to v \cdot \alpha \tau_c] =^\phi M'_2[m^3_S x_c \to u \cdot \alpha \tau_c] \).

First, let \( m^0_S \in \text{dom} (\phi) \) be such that \( M'_1(m^0_S) \neq 1 \). We show that \( M'_2(\phi(m^0_S)) \neq 1 \) and \( M'_1(m^0_S) =^\phi M'_2(\phi(m^0_S)) \).

If \( m^0_S = m^1_S \), then \( \phi(m^0_S) = m^2_S \). Let \( v_i' \) and \( u_i' \) be such that \( M'_1(m^1_S) = \{x_i = v_i'\} \) and \( M'_2(m^2_S) = \{x_i = u_i'\} \). Since \( v =^\phi u \), by Lemma 27, we have \( v \cdot \alpha \tau_c =^\phi u \cdot \alpha \tau_c \). Since \( v_i' = v \cdot \alpha \tau_c \) and \( u_i' = u \cdot \alpha \tau_c \), we therefore have \( v_i' =^\phi u_i' \). Therefore, from \( M'_1 =^\phi M_2 \), it follows that \( \{x_i = v_i'\} =^\phi \{x_i = u_i'\} \). Otherwise, \( m^0_S = m^3_S \), so \( M'_1(m^0_S) = M_1(m^0_S) \) and \( M'_2(\phi(m^0_S)) = M_2(\phi(m^0_S)) \). The result therefore follows from \( M'_1 =^\phi M_2 \).

Now, let \( m^0_S \in \text{dom} (\phi) \) be such that \( \star \cdot \alpha \leq \text{auth}^* (S_0) \cap \text{persist} (S_0) \) and \( M'_1(m^0_S) = 1 \). We show that \( M'_2(\phi(m^0_S)) = 1 \). This follows from \( M_1 =^\phi M_2 \) by construction of \( M'_1 \) and \( M'_2 \).

Sub-case \texttt{DANGLE-ASSIGN} \( (v' = \downarrow_p \cdot \alpha p \) and \( M'_1 = M_1 \), where \( M_1(m^1_S) = 1 \)

First, suppose \( \downarrow \alpha \leq a \land p \). Then \( v' \cdot (a \land p) = \downarrow_p \) and \( u' \cdot (a \land p) = u' \), and from \( M'_1 =^\phi M_2 \), we know \( M_2(m^3_S) = 1 \). So, by \texttt{DANGLE-ASSIGN}, we have \( \langle m^3_S x_c := u, M_2 \rangle \xrightarrow{\sim} \langle \downarrow_p \cdot \alpha p, M_2 \rangle \). Therefore, \( u' = \downarrow_p \cdot \alpha p = \downarrow_p \).

i. We need to show \( \downarrow_p =^\phi \downarrow_p \).

This is trivial.

ii. We need to show \( M_1 =^\phi M_2 \).

This is given.
Now, suppose $\alpha \not\leq a \cap p$. Then $\nu' \cdot_\alpha (a \cap p) = [\bot_p]$. If $M_2(m_2^*) = \bot$, then by DANGLED-ASSIGN, $\{m_2^*, x_1 := u, M_2\} \overset{\perp}{\rightarrow} (\bot_p \cdot_\alpha p, M_2)$. Otherwise, $M_2(m_2^*) \neq \bot$, and by ASSIGN, $\{m_2^*, x_1 := u, M_2\} \overset{\perp}{\rightarrow} (\bot_p \cdot_\alpha p, M_2[m_2^* \rightarrow \bot_p \cdot_\alpha \tau])$. Therefore, $u \cdot_\alpha (a \cap p) \in \{(\bot_p \cdot_\alpha p) \cdot_\alpha (a \cap p), (\bot_p \cdot_\alpha p) \cdot_\alpha (a \cap p)\} = \{[\bot_p], [\bot_p]\}$. 

i. We need to show $[\bot_p] \approx_\alpha [u']$ for $u' \in \{\bot_p\}$. This is trivial.

ii. We need to show $M_1' \approx_\alpha M_2'$.

First, let $m_{0'} \in \text{dom}(\phi)$ be such that $M_1'(m_{0'}) \neq \bot$. We show that $M_2'(\phi(m_{0'})) \neq \bot$ and $M_1'(m_{0'}) \approx_\alpha M_2'(\phi(m_{0'}))$.

Since $M_1' = M_1$ and $M_1'(m_{0'}) \neq \bot$, we must have $m_{0'} \neq m_2^*$. Therefore, $\phi(m_{0'}) = M_1'(m_{0'})$ and $M_2'(\phi(m_{0'})) = M_2(\phi(m_{0'}))$. The result therefore follows from $M_1 \approx_\alpha M_2$.

Now, let $m_{0'} \in \text{dom}(\phi)$ be such that $\bot \not\approx\phi \approx \tau \not\approx_s \text{persist}(S_0)$ and $M_1'(m_{0'}) = \bot$. We show that $M_2'(\phi(m_{0'})) = \bot$. This follows from $M_1 \approx_\alpha M_2$ by construction of $M_2'$.

Case APPLY ($(\lambda(x: \tau)[pc; H], e_3, v_1, M_1) \rightarrow_\alpha (e_3\{v_1/x\}, M_1)$):

From $e_1 \approx_\alpha e_2$, we know that $e_2 = ((\lambda(x: \tau)[pc; H], e_4) v_2)$, where $e_3 \approx_\alpha e_4$ and $v_1 \approx_\alpha v_2$. By APPLY, we have

$((\lambda(x: \tau)[pc; H], e_4) v_2, M_2) \rightarrow (e_4\{v_2/x\}, M_2)$.

Choose $\phi' = \phi$.

i. We need to show $e_3\{v_1/x\} \approx_\alpha e_4\{v_2/x\}$.

This follows by Lemma 26.

ii. We need to show $M_1 \approx_\alpha M_2$.

This is given.

Case EXISTS-TRUE ($(\exists x: m_2^* \text{ as } x : e_3 \text{ else } e_4, M_1) \rightarrow_\alpha (e_3\{m_2^*/x\}) \cdot_\alpha (a \cap p), M_1)$, where $M_1(m_2^*) \neq \bot$ and $S = \{x_i : \overline{x_i}\}(a, p)$:

From $e_1 \approx_\alpha e_2$, we know that $e_2 = \exists x: m_2^* \text{ as } x : e_5 \text{ else } e_6$, where $\phi(m_2^*) = m_2^*$ and $e_3 \approx_\alpha e_5$. Since $M_1 \approx_\alpha M_2$ and $M_1(m_2^*) \neq \bot$, we know $M_2(m_2^*) \neq \bot$, so by EXISTS-TRUE, we have

$(\exists x: m_2^* \text{ as } x : e_5 \text{ else } e_6, M_2) \rightarrow (e_5\{m_2^*/x\}) \cdot_\alpha (a \cap p), M_2)$

Choose $\phi' = \phi$.

i. We need to show $(e_3\{m_2^*/x\}) \cdot_\alpha (a \cap p) \approx_\alpha (e_5\{m_2^*/x\}) \cdot_\alpha (a \cap p)$.

This follows by Lemmas 26 and 27.

ii. We need to show $M_1 \approx_\alpha M_2$.

This is given.

Case EXISTS-FALSE ($(\exists x: m_2^* \text{ as } x : e_3 \text{ else } e_4, M_1) \rightarrow_\alpha (e_3\{m_2^*/x\}) \cdot_\alpha (a \cap p), M_1)$, where $S = \{x_i : \overline{x_i}\}(a, p)$ and $M_1(m_2^*) = \bot$:

From $e_1 \approx_\alpha e_2$, we know that $e_2 = \exists x: m_2^* \text{ as } x : e_5 \text{ else } e_6$, where $\phi(m_2^*) = m_2^*$ and $e_4 \approx_\alpha e_6$. We proceed by cases according to whether $\not\leq a \cap p$.

Sub-case $\not\leq a \cap p$:

We therefore know $e_4 \cdot_\alpha (a \cap p) = e_4$ and $\not\leq a \cap p$. So, from $M_1 \approx_\alpha M_2$ and $M_1(m_2^*) = \bot$, we know $M_2(m_2^*) = \bot$, so by EXISTS-FALSE, we have $(\exists x: m_2^* \text{ as } x : e_5 \text{ else } e_6, M_2) \rightarrow (e_6, M_2)$. Choose $\phi' = \phi$.

i. We need to show $e_4 \approx_\alpha e_6$.

This is given.

ii. We need to show $M_1 \approx_\alpha M_2$.

This is given.
Sub-case $\alpha \neq a \cap p$:
We therefore have $e_4 \bullet a (\alpha \cap p) = [e_4']$ for some $e_4'$. If $M_2(m_2^2) = \bot$, then by EXISTS-\textsc{False}, we have
\(\exists x : e_5 \text{ else } e_6, M_2 \rightarrow \{[e_5'], M_2\}, \) for some $e_5'$. Otherwise, by EXISTS-\textsc{True}, we have
\(\exists x : e_5 \text{ else } e_6, M_2 \rightarrow \{[e_5'], M_2\}, \) for some $e_5'$. Choose $\phi' = \phi$.

i. We need to show $[e_5'] \approx_\alpha [e_5'']$ for $e_5' \in \{e_5', e_6\}$.
This is trivial.

ii. We need to show $M_1 \approx_\alpha M_2$.
This is given.

Case TRY-\textsc{Val} $\langle \text{try } v_1 \text{ catch } p : e_3, M_1 \rangle \rightarrow_\alpha \langle v_1, M_1 \rangle$, where $\forall p', v_1 \neq v_1', \forall v_1 \neq [v_1']$:
From $e_1 \approx_\alpha e_2$, we know that $e_2 = (\text{try } v_2 \text{ catch } p : e_4)$, where $v_1 \approx_\alpha v_2$ and $e_3 \approx_\alpha e_4$. From this, it follows that by TRY-\textsc{Val}, we have $\langle \text{try } v_2 \text{ catch } p : e_4, M_2 \rangle \rightarrow \langle v_2, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $v_1 \approx_\alpha v_2$.
This is given.

ii. We need to show $M_1 \approx_\alpha M_2$.
This is given.

Case TRY-\textsc{Catch} $\langle \text{try } \bot, p' : e_3, M_1 \rangle \rightarrow_\alpha \langle e_3, M_1 \rangle$, where $\Rightarrow p \leq p'$:
From $e_1 \approx_\alpha e_2$, we know that $e_2 = (\text{try } \bot, p' : e_4)$, where $e_3 \approx_\alpha e_4$. By TRY-\textsc{Catch}, we have $\langle \text{try } \bot, p' : e_4, M_2 \rangle \rightarrow \langle e_4, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $e_3 \approx_\alpha e_4$.
This is given.

ii. We need to show $M_1 \approx_\alpha M_2$.
This is given.

Case TRY-\textsc{Esc} $\langle \text{try } \bot, p' : e_3, M_1 \rangle \rightarrow_\alpha \langle \bot, p', M_1 \rangle$, where $p \neq p'$:
From $e_1 \approx_\alpha e_2$, we know that $e_2 = (\text{try } \bot, p' : e_4)$. By TRY-\textsc{Esc}, we have $\langle \text{try } \bot, p' : e_4, M_2 \rangle \rightarrow \langle \bot, p', M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\bot, p' \approx_\alpha \bot, p'$.
This is trivial.

ii. We need to show $M_1 \approx_\alpha M_2$.
This is given.

Case PARALLEL-\textsc{Result} $\langle \text{par } v_1 \parallel v_2, M_1 \rangle \rightarrow_\alpha \langle *, M_1 \rangle$:
From $e_1 \approx_\alpha e_2$, we know that $e_2 = u_1 \parallel u_2$, where $v_i \approx_\alpha u_i$ for $i \in \{1, 2\}$. By PARALLEL-\textsc{Result}, we have $\langle u_1 \parallel u_2, M_2 \rangle \rightarrow \langle *, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $* \approx_\alpha *$.
This is trivial.

ii. We need to show $M_1 \approx_\alpha M_2$.
This is given.

Case IF-\textsc{True} $\langle \text{if true then } e_3 \text{ else } e_4, M_1 \rangle \rightarrow_\alpha \langle e_3, M_1 \rangle$:
From $e_1 \approx_\alpha e_2$, we know that $e_2 = (\text{if true then } e_5 \text{ else } e_6)$, where $e_3 \approx_\alpha e_5$. By IF-\textsc{True}, we have $\langle \text{if true then } e_5 \text{ else } e_6, M_2 \rangle \rightarrow \langle e_5, M_2 \rangle$. Choose $\phi' = \phi$.
i. We need to show \( e_3 \approx^\alpha e_5 \).
   This is given.
ii. We need to show \( M_1 \approx^\alpha M_2 \).
   This is given.

Case If-False (\( (\text{if false then } e_3 \text{ else } e_4 , M_1) \rightarrow^\alpha (e_4 , M_1) \)):

From \( e_1 \approx^\alpha e_2 \), we know that \( e_2 = (\text{if false then } e_5 \text{ else } e_6) \), where \( e_4 \approx^\alpha e_6 \). By If-False, we have \( (\text{if false then } e_5 \text{ else } e_6, M_2) \rightarrow (e_6, M_2) \). Choose \( \phi' = \phi \).

i. We need to show \( e_4 \approx^\alpha e_6 \).
   This is given.
ii. We need to show \( M_1 \approx^\alpha M_2 \).
   This is given.

Case Let (\( (\text{let } x = v_1 \text{ in } e_3 , M_1) \rightarrow^\alpha (e_3 \{v_1/x\} , M_1) \)), where \( \forall p. \ v_1 \neq i_p \text{ and } \forall v'_1, \ v_1 \neq [v'_1] \):

From \( e_1 \approx^\alpha e_2 \), we know that \( e_2 = (\text{let } x = v_2 \text{ in } e_4) \), where \( v_1 \approx^\alpha v_2 \) and \( e_3 \approx^\alpha e_4 \). By Let, we have \( (\text{let } x = v_2 \text{ in } e_4, M_2) \rightarrow (e_4 \{v_2/x\} , M_2) \). Choose \( \phi' = \phi \).

i. We need to show \( e_3 \{v_1/x\} \approx^\alpha e_4 \{v_2/x\} \).
   This follows by Lemma 26.
ii. We need to show \( M_1 \approx^\alpha M_2 \).
   This is given.

Case Eval-Context (\( (E[e_5] , M_1) \rightarrow^\alpha (E[e_3] , M_1') \)), where \( \langle e_3 , M_1 \rangle \rightarrow^\alpha (e_3', M_1') \):

We proceed by cases according to the syntax of \( E[\cdot] \). We only show the case \( E[\cdot] = \text{try } [\cdot] \text{ catch } p : e_4 \); the other cases follow similarly.

Sub-case \( E[\cdot] = (\text{try } [\cdot] \text{ catch } p : e_4) \):

From \( e_1 \approx^\alpha e_2 \), we know \( e_2 = (\text{try } e_5 \text{ catch } p : e_6) \), where \( e_3 \approx^\alpha e_3 \) and \( e_4 \approx^\alpha e_6 \). To apply the induction hypothesis, we need:

- \( \vdash_{[\text{wf}]} \langle e_3 , M_1 \rangle \) and \( \vdash_{[\text{wf}]} \langle e_5 , M_2 \rangle \)
  These follow from
  \[ \vdash_{[\text{wf}]} (\text{try } e_3 \text{ catch } p: e_4 , M_1) \]
  and
  \[ \vdash_{[\text{wf}]} (\text{try } e_5 \text{ catch } p: e_6 , M_2) . \]
- \( \vdash_{[\text{wf}]} (\text{catch } p: e_4) \).
  These follow from the typing derivations for \( e_3 \) catch \( p : e_4 \) and \( e_5 \) catch \( p : e_6 \).
- \( \vdash_{[\text{wf}]} M_2 \) and \( \langle e_3 , M_1 \rangle \approx^\alpha \langle e_5 , M_2 \rangle \).
  These are given.

Therefore, we can apply the induction hypothesis to get a configuration \( \langle e_3', M_1' \rangle \) and a high-integrity homomorphism \( \phi' \) from \( M_1' \) to \( M_2' \) that extends \( \phi \), such that \( \langle e_5 , M_2 \rangle \rightarrow (e_5', M_2') \) and
\[
\langle e_3', M_1' \rangle \approx^\alpha \langle e_5', M_2' \rangle . \tag{14}
\]

So, by Eval-Context, we have
\[
\langle \text{try } e_5 \text{ catch } p: e_6 , M_2 \rangle \rightarrow (\langle \text{try } e_5 \text{ catch } p: e_6 , M_2' \rangle .
\]

From (14), we know \( \langle \text{try } e_5 \text{ catch } p: e_4 , M_1' \rangle \approx^\alpha \langle \text{try } e_5 \text{ catch } p: e_6 , M_2' \rangle \), as desired.
In case $E[\cdot] = \text{soft}[\cdot]$, where $e_2 = \text{soft} e_4$, to apply the induction hypothesis, we need the additional fact that since $e_3$ is not a value, neither is $e_4$, and therefore $e_3 
eq e_4$. Then, by GC, we have $\vdash_{\text{soft}} \langle e_3, M_1 \rangle$ and $\vdash_{\text{soft}} \langle e_4, M_2 \rangle$ imply $\vdash_{\text{soft}} \langle e_3, M_1 \rangle$ and $\vdash_{\text{soft}} \langle e_4, M_2 \rangle$.

Case FAIL-PROP ($\langle F[\perp_p], M_1 \rangle \rightarrow_{\alpha} \langle \perp_p, M_1 \rangle$):

From $e_1 \equiv_{\alpha} e_2$, we know that $e_2 = F'[\perp_p]$. By FAIL-PROP, we have $\langle F'[\perp_p], M_2 \rangle \rightarrow \langle \perp_p, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $\vdash_{\alpha} \perp_p$.
    This is trivial.
ii. We need to show $M_1 \equiv_{\alpha} M_2$.
    This is given.

Case GC ($\langle e_1, M_1 \rangle \rightarrow_{\alpha} \langle e_1, M_1[G \rightarrow \perp] \rangle$, where $\text{gc}(G, \langle e_1, M_1 \rangle)$):

By Lemma 25, let $G'$ be such that $C \subseteq G' \subseteq \phi(G)$ and $\text{gc}(G', \langle e_2, M_2 \rangle)$, where

$$C = \{ \phi(m^\delta) : m^\delta \in G \cap \text{dom}(\phi) \land \alpha \leq \text{auth}^+(S) \cap \text{persist}(S) \}$$

and

$$\phi(G) = \{ \phi(m^\delta) : m^\delta \in G \cap \text{dom}(\phi) \}.$$

Then, by GC, we have $\langle e_2, M_2 \rangle \rightarrow \langle e_2, M_2[G' \rightarrow \perp] \rangle$. Choose $\phi' = \phi$.

i. We need to show $e_1 \equiv_{\alpha} e_2$.
    This is given.
ii. We need to show $M_1[\perp] \equiv_{\alpha} M_2[G' \rightarrow \perp]$.
    First, let $m_0^S \in \text{dom}(\phi)$ be such that $M'_1(m_0^S) \neq \perp$. We show that $M'_2(\phi(m_0^S)) \neq \perp$ and $M'_1(\phi(m_0^S)) \equiv_{\alpha} M'_2(\phi(m_0^S))$.
    Since $M'_1(\phi(m_0^S)) \neq \perp$, we know $m_0^S \notin G$ and $\phi(m_0^S) \notin G'$. Therefore, $M'_1(\phi(m_0^S)) = M_1(m_0^S)$ and $M'_2(\phi(m_0^S)) = M_2(\phi(m_0^S))$, so the result follows from the assumption $M_1 \equiv_{\alpha} M_2$.
    Now, let $m_0^S \in \text{dom}(\phi)$ be such that $\vdash_{\alpha} \perp \leq \text{auth}^+(S) \cap \text{persist}(S_0)$ and $M'_1(m_0^S) = \perp$. We show that $M'_2(\phi(m_0^S)) = \perp$. If $m_0^S \in G$, then $\phi(m_0^S) \notin G'$, and the result follows by construction of $M'_2$. Otherwise, $m_0^S \notin G$, and so, $M'_2(\phi(m_0^S)) = G'$. Therefore, $M'_1(m_0^S) = M_1(m_0^S)$ and $M'_2(\phi(m_0^S)) = M_2(\phi(m_0^S))$, so the result follows from the assumption $M_1 \equiv_{\alpha} M_2$.

Cases Bracket-Select and Bracket-Soft-Select

$\langle [v].x_c, M_1 \rangle \rightarrow_{\alpha} \langle [v.x_c], M_1 \rangle$, where $v \in \{ m_1^{S_1}, \text{soft } m_1^{S_1} \}$.

From $e_1 \equiv_{\alpha} e_2$, we know that $e_2 = [u].x_c$. From the grammar, we must have $u \in \{ m_2^{S_2}, \text{soft } m_2^{S_2} \}$. If $u = m_2^{S_2}$, then by Bracket-Select, $\langle m_2^{S_2}, x_c, M_2 \rangle \rightarrow \langle m_2^{S_2}.x_c, M_2 \rangle$. Otherwise, $u = \text{soft } m_2^{S_2}$, and by Bracket-Soft-Select, $\langle \text{soft } m_2^{S_2}, x_c, M_2 \rangle \rightarrow \langle \text{soft } m_2^{S_2}, x_c, M_2 \rangle$. Choose $\phi' = \phi$.

i. We need to show $[v.x_c] \equiv_{\alpha} [u.x_c]$.
    This is trivial.
ii. We need to show $M_1 \equiv_{\alpha} M_2$.
    This is given.

Case Bracket-Assign

$\langle [v_1].x_c := v_2, M_1 \rangle \rightarrow_{\alpha} \langle [v_1.x_c := v_2], M_1 \rangle$.

From $e_1 \equiv_{\alpha} e_2$, we know that $e_2 = [u_1].x_c := u_2$. By Bracket-Assign, we have $\langle [u_1].x_c := u_2, M_2 \rangle \rightarrow \langle [u_1.x_c := u_2], M_2 \rangle$. Choose $\phi' = \phi$. 51
i. We need to show \( v_1.x := v_2 \models^\phi [u_1.x := u_2] \).
    This is trivial.

ii. We need to show \( M_1 \models^\phi M_2 \).
    This is given.

Case **Bracket-Soft** (\( (\text{soft } [v], M_1) \rightarrow^\alpha (\text{soft } v, M_1) \)):

From \( e_1 \models^\phi e_2 \), we know that \( e_2 = \text{soft } [e_3] \). Since \( \langle e_2, M_2 \rangle \) is assumed to not diverge, \( \langle e_3, M_2 \rangle \) cannot diverge either. So, by Corollary 14, there is a configuration \( \langle u, M_2' \rangle \) such that \( \langle e_3, M_2 \rangle \xrightarrow{\text{soft } u} \langle u, M_2' \rangle \). Then by EVAL-CONTEXT, Bracket-Context, and Bracket-Soft, we have

\[
(\text{soft } [e_3], M_2) \rightarrow^*_1 (\text{soft } [u], M_2') \rightarrow \langle \text{soft } u, M_2' \rangle.
\]

Choose \( \phi' = \phi \).

i. We need to show \( \text{soft } v \models^\phi \text{soft } u \).
    This is trivial.

ii. We need to show \( M_1 \models^\phi M_2' \).
    This follows by Lemma 29.

Case **Bracket-Exists**

\( (\exists [v] \text{ as } x : e_3 \text{ else } e_4, M_1) \rightarrow^\alpha (\exists [v \text{ as } x : e_3 \text{ else } e_4], M_1) \):

From \( e_1 \models^\phi e_2 \), we know that \( e_2 = \exists [u] \text{ as } x : e_5 \text{ else } e_6 \). By Bracket-Exists, we have

\[
(\exists [u] \text{ as } x : e_5 \text{ else } e_6, M_2) \rightarrow (\exists [u \text{ as } x : e_5 \text{ else } e_6], M_2).
\]

Choose \( \phi' = \phi \).

i. We need to show \( \exists [v \text{ as } x : e_3 \text{ else } e_4] \models^\phi \exists [u \text{ as } x : e_5 \text{ else } e_6] \).
    This is trivial.

ii. We need to show \( M_1 \models^\phi M_2 \).
    This is given.

Case **Bracket-Apply** (\( ([v_1] v_2, M_1) \rightarrow^\alpha ([v_1] v_2, M_1) \)):

From \( e_1 \models^\phi e_2 \), we know that \( e_2 = ([u_1] u_2) \). By Bracket-Apply, we have \( ([u_1] u_2, M_2) \rightarrow ([u_1 u_2], M_2) \).

Choose \( \phi' = \phi \).

i. We need to show \( [v_1] v_2 \models^\phi [u_1 u_2] \).
    This is trivial.

ii. We need to show \( M_1 \models^\phi M_2 \).
    This is given.

Case **Bracket-Try** (\( (\text{try } [v] \text{ catch } p : e_3, M_1) \rightarrow^\alpha (\text{try } v \text{ catch } p : e_3, M_1) \)):

From \( e_1 \models^\phi e_2 \), we know that \( e_2 = (\text{try } e_4) \text{ catch } p : e_3 \). Since \( \langle e_2, M_2 \rangle \) is assumed to not diverge, \( \langle e_4, M_2 \rangle \) cannot diverge either. So, by Corollary 14, there is a configuration \( \langle u, M_2' \rangle \) such that \( \langle e_4, M_2 \rangle \xrightarrow{\text{try } u \text{ catch } p : e_3} \langle u, M_2' \rangle \). Then by EVAL-CONTEXT, Bracket-Context, and Bracket-Try, we have

\[
(\text{try } e_4 \text{ catch } p : e_5, M_2) \rightarrow^*_1 (\text{try } [u] \text{ catch } p : e_5, M_2') \rightarrow (\text{try } u \text{ catch } p : e_5, M_2')
\]

Choose \( \phi' = \phi \).

i. We need to show \( \text{try } v \text{ catch } p : e_3 \models^\phi \text{try } u \text{ catch } p : e_3 \).
    This is trivial.
ii. We need to show \( M_1 \approx_\alpha M_2 \).

This follows by Lemma 29.

Case \text{\textsc{Bracket-If}} (\langle \text{if } [v] \text{ then } e_3 \text{ else } e_4, M_1 \rangle \rightarrow_\alpha \langle \text{if } v \text{ then } e_3 \text{ else } e_4, M_1 \rangle) :

From \( e_1 \approx_\alpha e_2 \), we know that \( e_2 = (\text{if } u \text{ then } e_5 \text{ else } e_6) \). By \text{\textsc{Bracket-If}}, we have

\[
\langle \text{if } u \text{ then } e_5 \text{ else } e_6, M_2 \rangle \rightarrow \langle \text{if } u \text{ then } e_5 \text{ else } e_6, M_2 \rangle.
\]

Choose \( \phi' = \phi \).

i. We need to show \( [\text{if } v \text{ then } e_3 \text{ else } e_4] \approx_\alpha [\text{if } u \text{ then } e_5 \text{ else } e_6] \).

This is trivial.

ii. We need to show \( M_1 \approx_\alpha M_2 \).

This is given.

Case \text{\textsc{Bracket-Let}} (\langle \text{let } x = [v] \text{ in } e_3, M_1 \rangle \rightarrow_\alpha \langle [e_3[[x]/x]], M_1 \rangle) :

From \( e_1 \approx_\alpha e_2 \), we know that \( e_2 = (\text{let } x = [u] \text{ in } e_4) \). If \( \forall p. u \neq \bot_p \), then by \text{\textsc{Bracket-Let}}, we have

\[
\langle \text{let } x = [u] \text{ in } e_4, M_2 \rangle \rightarrow \langle [e_4[[u]/x]], M_2 \rangle.
\]

Otherwise, let \( p \) be such that \( u = \bot_p \). Then, by \text{\textsc{Bracket-Fail}}, we have

\[
\langle \text{let } x = [u] \text{ in } e_4, M_2 \rangle \rightarrow \langle [\bot_p], M_2 \rangle.
\]

Choose \( \phi' = \phi \).

i. We need to show \( [e_3[[v]/x]] \approx_\alpha [e_2'] \), where \( e_2' \in \{e_4[[u]/x], \bot_p\} \).

This is trivial.

ii. We need to show \( M_1 \approx_\alpha M_2 \).

This is given.

Case \text{\textsc{Double-Bracket}} (\langle [v], M_1 \rangle \rightarrow_\alpha \langle [v], M_1 \rangle) :

From \( e_1 \approx_\alpha e_2 \), we know that \( e_2 = [e_2'] \), so we trivially have \( [e_2'], M_2 \) \rightarrow^* \langle [e_2'], M_2 \rangle \). Choose \( \phi' = \phi \).

i. We need to show \( v \approx_\alpha e_2' \).

This is trivial.

ii. We need to show \( M_1 \approx_\alpha M_2 \).

This is given.

Case \text{\textsc{Bracket-Context}} (\langle e_3, M_1 \rangle \rightarrow_\alpha \langle e_3', M_1' \rangle) :

From \( e_1 \approx_\alpha e_2 \), we know that \( e_2 = [e_4] \), and we trivially have \( \langle e_4, M_2 \rangle \rightarrow^* \langle [e_4], M_2 \rangle \). Choose \( \phi' = \phi \).

i. We need to show \( e_3' \approx_\alpha [e_4] \).

This is trivial.

ii. We need to show \( M_1' \approx_\alpha M_2 \).

This follows by Lemma 29.

Case \text{\textsc{Bracket-Fail}} (\langle F[[\bot_p]], M_1 \rangle \rightarrow_\alpha \langle [\bot_p], M_1 \rangle) :

From \( e_1 \approx_\alpha e_2 \), an easy case analysis on the syntax of \( F[\cdot] \) shows that \( e_2, M_2 \) \rightarrow \langle [\bot_p], M_2 \rangle \) via \text{\textsc{Bracket-Fail}}.

Choose \( \phi' = \phi \).  

53
i. We need to show \([\perp, \rho] \approx^\phi_\alpha [\perp, \rho]\).
   This is trivial.

ii. We need to show \(M_1 \approx^\phi_\alpha M_2\).
   This is given.

Case \(\alpha\text{-CREATE} (\langle e_1, M_1 \rangle \rightarrow \alpha \{ e_1, M_1[m^S \mapsto \{x_i = [v_j]\}] \}),\) where \(m^S\) is fresh, \(\emptyset; \tau; \tau \vdash \{x_i = [v_j]\}^S : R_{\tau, \tau}, \neg^\alpha_{\text{wf}}\)
\(M[m^S \mapsto \{x_i = [v_j]\}],\) and \(\alpha \notin \text{persist}(S)\):
We trivially have \(\langle e_2, M_2 \rangle \rightarrow^* \langle e_2, M_2 \rangle\). Choose \(\phi' = \phi\).

i. We need to show \(e_1 \approx^\phi_\alpha e_2\).
   This is given.

ii. We need to show \(M_1[m^S \mapsto \{x_i = [v_j]\}] \approx^\phi_\alpha M_2\).

Since \(m^S \notin \text{dom}(\phi)\), this follows from the assumption \(M_1 \approx^\phi_\alpha M_2\) by construction of \(M'_1\).

Case \(\alpha\text{-ASSIGN} (\langle e_1, M_1 \rangle \rightarrow \alpha \{ e_1, M_1[m^S.x_c \mapsto [v]] \}),\) where \(m^S \in \text{dom}(M_1)\), \(M_1(m^S) \neq \perp, S = \{x_i : \tau_c, \emptyset; \tau; \tau \vdash [v]: \tau_c, \tau, \) and \(\neg^\alpha_{\text{wf}} M_1[m^S.x_c \mapsto [v]]\):
We trivially have \(\langle e_2, M_2 \rangle \rightarrow^* \langle e_2, M_2 \rangle\). Choose \(\phi' = \phi\).

i. We need to show \(e_1 \approx^\phi_\alpha e_2\).
   This is given.

ii. We need to show \(M_1[m^S.x_c \mapsto [v]] \approx^\phi_\alpha M_2\).

If \(m^S \notin \text{dom}(\phi)\), then this follows from the assumption \(M_1 \approx^\phi_\alpha M_2\) by construction of \(M'_1\).
Suppose \(m^S \in \text{dom}(\phi)\). Then it suffices to show that \(M'_1(m^S) \approx^\phi_\alpha M_2(\phi(m^S))\), since the rest follows from \(M_1 \approx^\phi_\alpha M_2\).

Let \(\vec{v}_i, \vec{v}'_i, \) and \(\vec{u}_i\) be such that \(M_1(m^S) = \{x_i = \vec{v}_i\}\)
\(M'_1(m^S) = \{x_i = \vec{v}'_i\}\) and \(M_2(m^S) = \{x_i = \vec{u}_i\}\). From \(M_1 \approx^\phi_\alpha M_2\), we know \(v_i \approx^\phi_\alpha u_i\) for all \(i\).
By construction of \(M'_1\), we have \(v'_i = [v]\) and \(v'_i = v_i\) for \(i \neq c\).
Therefore, it remains to be shown that \(\vec{v} \approx^\phi_\alpha \vec{u}_c\).

From \(\emptyset; \tau; \tau \vdash [v]: \tau_c, \tau\), we know that \(\alpha \notin \text{integ}(\tau_c)\). Therefore, from \(\neg^\alpha_{\text{wf}} M_2\), we must have \(u_c = [u]\), for some \(u\). The result \(\vec{v} \approx^\phi_\alpha \vec{u}_c\) then follows trivially.

Case \(\alpha\text{-FORGET} (\langle e_1, M_1 \rangle \rightarrow \alpha \{ e_1, M_1[m^S \mapsto \perp] \}),\) where \(m^S \in \text{dom}(M_1)\) and \(\alpha \notin \text{persist}(S)\):
We trivially have \(\langle e_2, M_2 \rangle \rightarrow^* \langle e_2, M_2 \rangle\). Choose \(\phi' = \phi\).

i. We need to show \(e_1 \approx^\phi_\alpha e_2\).
   This is given.

ii. We need to show \(M_1[m^S \mapsto \perp] \approx^\phi_\alpha M_2\).

This follows from the assumption \(M_1 \approx^\phi_\alpha M_2\), since \(\alpha \notin \text{persist}(S)\).

\(\square\)

8.6 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 non-collectible. Lemma 30 says that this is also implied by Theorem 1.
Lemma 30. 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:

\[
\vdash_{\text{_wf}}^\alpha \langle e_1, M_1 \rangle \land \vdash_{\text{_wf}}^\alpha \langle e_2, M_2 \rangle \land \langle e_1, M_1 \rangle \equiv^\alpha \langle e_2, M_2 \rangle
\]

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

To prove this, we first prove a preliminary result about garbage-collection roots.

Lemma 31. Let \( e_1 \) and \( e_2 \) be well-typed expressions, both of type \( \tau \), that are related via a high-integrity homomorphism \( \phi \). If \( m^\delta \) is a high-authority GC root in \( e_1 \), then \( \phi(m^\delta) \) is also a GC root in \( e_2 \).

\[
\Gamma; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \land \Gamma; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \land \vdash_{\text{wf}}^\alpha \mathcal{M}_2
\]

\[
\Gamma; pc; \mathcal{H} \vdash \text{root}(\phi(m^\delta), e_2)
\]

Proof. By induction on the derivation of \( \text{root}(m^\delta, e_1) \). The proof proceeds by cases according to the syntax of \( e_1 \). The result holds vacuously in cases \( e_1 = x \), \( e_1 = \text{true} \), \( e_1 = \text{false} \), \( e_1 = * \), and \( e_1 = \text{soft} m_1^\delta \), since \( \neg \text{root}(m^\delta, e_1) \) in these cases.

Case \( e_1 = m_1^\delta \): Since \( \text{root}(m^\delta, m_1^\delta) \), we must have \( m_1^\delta = m^\delta \). Therefore, from \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = \phi(m^\delta) \), and the result follows via Rule R1.

Case \( e_1 = \lambda(x : \tau') [pc' : \mathcal{H}']. e_3 \): From \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = \lambda(x : \tau') [pc' : \mathcal{H}']. e_4 \), where \( e_1 \equiv^\alpha e_4 \). From the typing of \( e_1 \) and \( e_2 \), we have \( \Gamma, x : \tau' ; pc' ; \mathcal{H} \vdash e_3 : \tau'', \mathcal{H}' \) and \( \Gamma, x : \tau' ; pc' ; \mathcal{H} \vdash e_4 : \tau'', \mathcal{H}' \), for some \( pc' \), \( \mathcal{H}' \), and \( \tau'' \). From the derivation of \( \text{root}(m^\delta, e_1) \), we know \( \text{root}(m^\delta, e_3) \). Therefore, we can apply the induction hypothesis to obtain \( \text{root}(\phi(m^\delta), e_4) \), and the result follows via Rule R6.

Case \( e_1 = v_1 v_2 \): From the derivation of \( \text{root}(m^\delta, e_1) \), we know \( \text{root}(m^\delta, v_1) \) or \( \text{root}(m^\delta, v_2) \). Without loss of generality, assume \( \text{root}(m^\delta, v_1) \). From \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = v_1 v_4 \), where \( v_1 \equiv^\alpha v_3 \). From the typing of \( e_1 \) and \( e_2 \), we have \( \Gamma, pc; \mathcal{H} \vdash v_1 : \tau', \tau \) and \( \Gamma, pc; \mathcal{H} \vdash v_3 : \tau', \tau \), for some \( \tau' \). Therefore, we can apply the induction hypothesis to obtain \( \text{root}(\phi(m^\delta), v_1) \), and the result follows via Rule R7.

Case \( e_1 = \text{if } e_1 \text{ then } e_3 \text{ else } e_4 \): From the derivation of \( \text{root}(m^\delta, e_1) \), we know \( \text{root}(m^\delta, e_k) \) for some \( k \in \{3, 4, 5\} \). From \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = \text{if } e_3' \text{ then } e_4' \text{ else } e_5' \), where \( e_i \equiv^\alpha e_i' \) for \( i \in \{3, 4, 5\} \). From the typing of \( e_1 \) and \( e_2 \), we have \( \Gamma, pc ; \mathcal{H} \vdash e_k : \tau', \mathcal{X}' \) and \( \Gamma, pc ; \mathcal{H} \vdash e_k' : \tau', \mathcal{X}' \), for some \( pc' \), \( \mathcal{X}' \), and \( \mathcal{X}'' \). Therefore, we can apply the induction hypothesis to obtain \( \text{root}(\phi(m^\delta), e_k') \), and the result follows via Rule R10.

Case \( e_1 = (x_i = v_i)^{S_1} \): From the derivation of \( \text{root}(m^\delta, e_1) \), we know \( \text{root}(m^\delta, v_k) \) for some \( k \). From \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = (x_i = u_i)^{S_1} \), where \( v_i \equiv^\alpha u_i \) for all \( i \). From the typing of \( e_1 \) and \( e_2 \), we have \( \Gamma, pc ; \mathcal{H} \vdash u_i : \tau', \tau \) and \( \Gamma, pc ; \mathcal{H} \vdash e_k' : \tau', \tau \), for some \( \tau' \). Therefore, we can apply the induction hypothesis to obtain \( \text{root}(\phi(m^\delta), u_k) \), and the result follows via Rule R3.

Case \( e_1 = v_1 x \): From the derivation of \( \text{root}(m^\delta, e_1) \), we know \( \text{root}(m^\delta, v) \). From \( e_1 \equiv^\alpha e_2 \), we have \( e_2 = u_1 x \), where \( v \equiv^\alpha u \). From the typing of \( e_1 \) and \( e_2 \), we have \( \Gamma, pc ; \mathcal{H} \vdash v : \tau', \tau \) and \( \Gamma, pc ; \mathcal{H} \vdash u : \tau', \tau \), for some \( \tau' \). Therefore, we can apply the induction hypothesis to obtain \( \text{root}(\phi(m^\delta), u) \), and the result follows via Rule R4.
Case $e_1 = v_1.x := v_2$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, v_k)$ for some $k$. From $e_1 \equiv_{\alpha} e_2$, we have $e_2 = u_1.x := u_2$, where $v_i \equiv_{\alpha} u_i$ for $i \in \{1, 2\}$. From the typing of $e_1$ and $e_2$, we have $\Gamma; pc; \mathcal{H} \vdash v_k : \tau', \top$ and $\Gamma; pc; \mathcal{H} \vdash u_k : \tau', \top$, for some $\tau'$. Therefore, we can apply the induction hypothesis to obtain $\text{root}(\phi(m^\delta), u_k)$, and the result follows via Rule R5.

Case $e_1 = \text{soft} e_3$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, e_3)$ and $e_3$ is not a memory location. From $e_1 \equiv_{\alpha} e_2$, we have $e_2 = \text{soft} e_4$, where $e_3 \equiv_{\alpha} e_4$; therefore, $e_4$ is not a memory location. From the typing of $e_1$ and $e_2$, we have $\Gamma; pc; \mathcal{H} \vdash e_3 : \tau', \mathcal{X}$ and $\Gamma; pc; \mathcal{H} \vdash e_4 : \tau', \mathcal{X}$, for some $\tau'$. Therefore, we can apply the induction hypothesis to obtain $\text{root}(\phi(m^\delta), e_4)$, and the result follows via Rule R9.

Case $e_1 = e_3 \parallel e_4$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, e_k)$ for some $k \in \{3, 4\}$. From $e_1 \equiv_{\alpha} e_2$, we have $e_2 = e_3 \parallel e_4$, where $e_3 \equiv_{\alpha} e_4$ for $i \in \{3, 4\}$. From the typing of $e_1$ and $e_2$, we have $\Gamma; \tau \vdash e_k : \tau', \top$ and $\Gamma; \tau \vdash e_k : \tau', \top$, for some $\tau'$. Therefore, we can apply the induction hypothesis to obtain $\text{root}(\phi(m^\delta), e_k)$, and the result follows via Rule R11.

Case $e_1 = \text{let} x = e_3$ in $e_4$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, e_k)$ for some $k \in \{3, 4\}$. From $e_1 \equiv_{\alpha} e_2$, we have $e_2 = \text{let} x = e_3'$ in $e_4'$, where $e_1 \equiv_{\alpha} e_4'$ for $i \in \{3, 4\}$. From the typing of $e_1$ and $e_2$, we have $\Gamma; \tau \vdash e_3 : \tau', \mathcal{X}'$ and $\Gamma; \tau \vdash e_4 : \tau', \mathcal{X}'$, for some $\tau'$, $\mathcal{X}'$, and $\tau'$. Therefore, we can apply the induction hypothesis to obtain $\text{root}(\phi(m^\delta), e_k)$, and the result follows via Rule R8.

Case $e_1 = \text{try} e_3$ catch $p$: $e_4$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, e_k)$ for some $k \in \{3, 4\}$. From $e_1 \equiv_{\alpha} e_2$, we have $e_2 = \text{try} e_3'$ catch $p$: $e_4'$, where $e_1 \equiv_{\alpha} e_4'$ for $i \in \{3, 4\}$. From the typing of $e_1$ and $e_2$, we have $\Gamma; \tau \vdash e_3 : \tau', \mathcal{X}'$ and $\Gamma; \tau \vdash e_4 : \tau', \mathcal{X}'$, for some $\mathcal{X}'$. Therefore, we can apply the induction hypothesis to obtain $\text{root}(\phi(m^\delta), e_k)$, and the result follows via Rule R12.

Case $e_1 = [e_3]$:

From the derivation of $\text{root}(m^\delta, e_1)$, we know $\text{root}(m^\delta, e_3)$. From the typing of $e_1$, we know $\Gamma; pc \cap \ell; \mathcal{H} \vdash e_3 : \tau, \mathcal{X}$ for some $\ell$ and $\tau'$, where $\alpha \not\equiv \ell$ and $\vdash \text{auth}^+(\tau') \not\equiv pc \cap \ell$. Therefore, by Lemma 15, we must have $\alpha \not\equiv \text{auth}^+(S)$, which contradicts the assumption $\vdash \alpha \equiv \text{auth}^+(S)$. So, this case holds vacuously.

We now prove Lemma 30, restated here for convenience.

**Lemma 30.** 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:

$$
\vdash_{\alpha} \langle e_1, M_1 \rangle \land \vdash_{\alpha} \langle e_2, M_2 \rangle \land \langle e_1, M_1 \rangle \equiv_{\alpha} \langle e_2, M_2 \rangle \\
\land \varnothing; pc; \mathcal{H} \vdash e_1 : \tau, \mathcal{X} \land \varnothing; pc; \mathcal{H} \vdash e_2 : \tau, \mathcal{X} \land \vdash_{\alpha} M_2
$$

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

$$
\vdash \alpha \equiv \text{auth}^+(S) \land \text{nc}(m^\delta, \langle e_1, M_1 \rangle) \Rightarrow \text{nc}(\phi(m^\delta), \langle e_2, M_2 \rangle)
$$
Proof. By induction on the derivation of $nc(m^S, (e_1, M_1))$.

Case NC1:
We have root($m^S, e_1$). By Lemma 31, we therefore have root($\phi(m^S), e_2$), and the result follows by NC1.

Case NC2:
We have root($m^S, e_1$) and $nc(m^S, (v_c, M_1))$ for some $m^S$, where $M_1(m^S) = \{ x_i = v_i \}$. It therefore follows that $nc(m^S, (m^S, M_1)), \rightarrow_{[\alpha]} (m^S, M_1)$, and $\rightarrow_{[\alpha]} (v_c, M_1)$.
Assume $m^S = m^S$. (Otherwise, we would have root($m^S, e_1$), and the argument for Case NC1 applies.)

Since $m^S$ is high-authority and $nc(m^S, (m^S, M_1))$, by Corollary 18, we know $m^S$ is high-integrity, high-authority, and high-persistence.

Since root($m^S, e_1$) and $m^S$ is high-authority, by Lemma 31, we know root($\phi(m^S), e_2$).

From $(e_1, M_1) \rightarrow_{[\alpha]} (e_2, M_2)$, we know $M_1 \rightarrow_{[\alpha]} M_2$. So, from $M_1(m^S) \nleq \bot$, we have $M_2(\phi(m^S)) = \{ x_i \mapsto \widetilde{u}_i \}$, where $v_c \rightarrow_{[\alpha]} \widetilde{u}_i$, for some $\widetilde{u}_i$.
Since $\rightarrow_{[\alpha]} M_1$ and $\rightarrow_{[\alpha]} M_2$, we also know $\emptyset; T \vdash v_c : \tau_c, \top$ and $\emptyset; T \vdash u_c : \tau_c, \top$, for some $\tau_c$. Therefore, we can apply the induction hypothesis to get $nc(\phi(m^S), (u_c, M_2))$.

The result follows via NC2.

\[\square\]

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, 12]. 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 [13] 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 [14] 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 [16] 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 \( \lambda_{\text{persis}} \) 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


A Appendix

A.1 Full syntax of \( \lambda_{\text{persist}} \)

<table>
<thead>
<tr>
<th>Variables</th>
<th>( x, y \in \text{Var} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory locations</td>
<td>( m \in \text{Mem} )</td>
</tr>
<tr>
<td>Labeled record types</td>
<td>( S \ ::= { x_j : \tau_j }_j )</td>
</tr>
<tr>
<td>Labeled reference types</td>
<td>( R \ ::= { x_j : \tau_j }_j )</td>
</tr>
<tr>
<td>Base types</td>
<td>( b ::= \text{bool} \mid \tau_1 \mid R \mid \text{soft } R )</td>
</tr>
<tr>
<td>Values</td>
<td>( v, \mu ::= x \mid \text{true} \mid \text{false} \mid v \mid m^\tau \mid \text{soft } m^\tau \mid \lambda(x: \tau)[pc; \mathcal{H}].e \mid (| p) )</td>
</tr>
<tr>
<td>Terms</td>
<td>( e ::= v \mid v_1. v_2 \mid \text{if } v_1 \text{ then } e_2 \text{ else } e_3 \mid { x_i = \nu_i } \mid v.x \mid v_1. x ::= v_2 \mid \text{soft } e \mid e_1 \parallel e_2 \mid \text{exists } v \text{ as } x : e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2 )</td>
</tr>
</tbody>
</table>

A.2 Full small-step operational semantics for nonadversarial execution of \( \lambda_{\text{persist}} \)

\[
\begin{align*}
\text{APPLY} & \quad ((\lambda(x: \tau)[pc; \mathcal{H}].e) \ v, M) \xrightarrow{\text{APPLY}} (e(v/x), M) \\
\text{LET} & \quad \forall p, v \neq \perp_p \\
\text{IF-TRUE} & \quad (\text{if true then } e_1 \text{ else } e_2, M) \xrightarrow{\text{IF-TRUE}} (e_1, M) \\
\text{IF-FALSE} & \quad (\text{if false then } e_1 \text{ else } e_2, M) \xrightarrow{\text{IF-FALSE}} (e_2, M) \\
\text{CREATE} & \quad \text{let } x = \nu_1. v_2. M = \text{newloc}(M) \\
\text{PARALLEL-RESULT} & \quad (v_1. v_2, M) \xrightarrow{\text{PARALLEL-RESULT}} (v, M) \\
\text{SELECT} & \quad M(m^\tau) = 1 \xrightarrow{\text{SELECT}} (v, M) \\
\text{ASSIGN} & \quad M(m^\tau) = 1 \xrightarrow{\text{ASSIGN}} (\text{let } x = v. M) \\
\text{DANGLE-SELECT} & \quad M(m^\tau) = 1 \xrightarrow{\text{DANGLE-SELECT}} (\text{let } x = v. M) \\
\text{DANGLE-ASSIGN} & \quad M(m^\tau) = 1 \xrightarrow{\text{DANGLE-ASSIGN}} (\text{let } x = v. M) \\
\text{EXISTSS-TRUE} & \quad (\exists x : e_1 \text{ else } e_2, M) \xrightarrow{\text{EXISTSS-TRUE}} (e_1, M) \\
\text{EXISTSS-FALSE} & \quad (\exists x : e_1 \text{ else } e_2, M) \xrightarrow{\text{EXISTSS-FALSE}} (e_2, M) \\
\text{EVAL-CONTEXT} & \quad (e, M) \xrightarrow{\text{EVAL-CONTEXT}} (e', M') \\
\text{FAIL-PROP} & \quad (F \| p), M \xrightarrow{\text{FAIL-PROP}} (\perp_p, M) \\
\text{SOFT-SELECT} & \quad (\text{soft } m^\tau, x, M) \xrightarrow{\text{SOFT-SELECT}} (v, M) \\
\text{SOFT-ASSIGN} & \quad (\text{soft } m^\tau, x := v, M) \xrightarrow{\text{SOFT-ASSIGN}} (v', M') \\
\text{TRY-VAL} & \quad (v \circ \text{catch } p : e, M) \xrightarrow{\text{TRY-VAL}} (v, M) \\
\text{TRY-ESC} & \quad (v \circ \text{catch } p : e, M) \xrightarrow{\text{TRY-ESC}} (\perp_p, M) \\
\text{PROG-STEP} & \quad (e, M) \xrightarrow{\text{PROG-STEP}} (e', M') \\
\text{GC} & \quad \text{gc}(G, (e, M)) \\
\end{align*}
\]
A.3 Full subtyping rules for \( \lambda_{\text{persist}} \)

\[
\begin{align*}
[S1] & \quad \vdash \{ x_1 : \tau_1, \ldots, x_n : \tau_n \} \leq \{ x_1 : \tau_1, \ldots, x_m : \tau_m \} \\
& \quad \vdash \tau_2 \leq \tau_1 \\
& \quad \vdash \tau_1 \leq \tau_2' \\
& \quad \vdash \tau_1 \vdash \tau_2 \leq \tau_1' \\
& \quad \vdash \tau_2 \vdash \tau_1 \vdash \tau_1' \\
[S2] & \quad \vdash R_1 \leq R_2 \\
& \quad \vdash \text{soft } R_1 \leq \text{soft } R_2 \\
[S3] & \quad \vdash b_1 \leq b_2 \\
& \quad \vdash w_2 \leq w_1 \\
& \quad \vdash (b_1)_{w_1} \leq (b_2)_{w_2}
\end{align*}
\]

A.4 Full typing rules for \( \lambda_{\text{persist}} \)

\[
\begin{align*}
[T-\text{Bool}] & \quad b \in \{ \text{true, false} \} \quad \Gamma ; p ; \mathcal{H} \vdash b : \text{bool} ; \top \\
[T-\text{Unit}] & \quad \Gamma ; p ; \mathcal{H} \vdash * : \top, \top \\
[T-\text{Var}] & \quad \Gamma(x) = \tau \\
[T-\text{Bottom}] & \quad p \neq \top \quad \vdash \mathcal{H} \leq p \\
[T-\text{Loc}] & \quad \vdash \text{of } S : \text{rectype } S = (x : \tau_i) \quad \Gamma ; p ; \mathcal{H} \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Parallel}] & \quad \Gamma ; p ; \mathcal{H} \vdash e_1 : \tau_1, \top \\
[T-\text{Soft}] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Rec}e] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Select}] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Assign}] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Try}] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
[T-\text{Subsume}] & \quad \vdash \text{of } \tau_i \leq \tau_j \\
\end{align*}
\]