Succour to the Confused Deputy

Types for Capabilities

Radha Jagadeesan, Corin Pitcher, and James Riely

DePaul University

Abstract. The possession of secrets is a recurrent theme in security literature and practice. We present a refinement type system, based on indexed intuitonist S4 necessity, for an object calculus with explicit locations (corresponding to principals) to control the principals that may possess a secret. Type safety ensures that if the execution of a well-typed program leads to a configuration with an object p located at principal a, then a possesses the capability to p. We illustrate the type system with simple examples drawn from web applications, including an illustration of how Cross-Site Request Forgery (CSRF) vulnerabilities may manifest themselves as absurd refinements on object declarations during type checking. This is an extended version of a paper that appears in APLAS 2012.

1 Introduction

Many systems depend upon a prescribed usage of secrets to enforce policies that incorporate secrecy, integrity, authentication, authorization, and auditing concerns. Nevertheless, it may be computationally expensive, or impossible in some adversarial models, to control the *use* of secrets directly. For this reason, it is common to control the *possession* of secrets instead of their use. However, invariants about the possession of secrets can fail due to inadequately-specified interfaces or a lack of agreement between software components. We illustrate some of the issues with two examples.

Object References The Java security manager permits access control checks based upon permissions assigned to code [22]. This allows control over systems composed of code from different sources. The `java.io.FileOutputStream` system class utilizes access control checks in the following manner:

- The `FileOutputStream` constructor checks for the relevant file write permission.
- For performance, `FileOutputStream` methods do not have access control checks.

The lack of access control checks after construction means that references to instances of `FileOutputStream` can be used to write a file by untrusted code, *if* the reference is made available to the untrusted code. For this reason, sensitive object references must be confined to trusted code.

Cross-Site Request Forgery and the Confused Deputy Cross-Site Request Forgery (CSRF) attacks [16] are acknowledged as an instance of the Confused Deputy problem [25]. A principal is a Confused Deputy if it uses its authority to mistakenly act on behalf of an initiating principal. The capability-based solution [25] to the Confused Deputy problem requires the initiating principal to provide a capability to the Deputy, which the Deputy requires to complete its actions. Since capabilities authorize access to a resource without further checks, their possession must be constrained programmatically. For example, one might ask:

- If the initiating principal hands a capability to the Deputy, to whom can the Deputy pass the capability?
- Is the Deputy permitted to add its own capabilities to any request from the initiating principal?

In the case of a web browser, acting as a Deputy for both user and JavaScript behavior on web pages, it is permitted to add cookies to outgoing HTTP requests based on the URLs determined from the web pages. Several browser extensions provide a more restrictive policy on forwarding cookies for cross-site requests to prevent misunderstandings in web applications vulnerable to CSRF attacks.

In this paper, we address control over the possession of secrets via the use of logical specifications embedded in the types of a distributed programming language. Static analysis is used to verify that programs comply with possession policies, yielding an upper bound on the principals that may possess a secret.

Approach. The main contribution of this paper is the application of a *refinement type system for a distributed object-oriented language*. The type system controls possession of object references, representing secrets, via specifications in a principal-indexed variant of intuitionist S4.

We specify possession of secrets using intuitionist S4 logic [9,32]. Instead of a single modality, we consider modalities indexed by principals. An indexed modality of the form $\Box_a \Phi$ represents a predicate Φ that is permissible for principal a [20,13]. We use a "may possess" predicate $\texttt{mp(s)}$ representing possession of a secret $\texttt{s}$. Thus $\Box_a\texttt{mp(s)}$ means that principal a is permitted to possess secret $\texttt{s}$.

It is key to our approach that indexed modalities allow different principals to have different possession policies. In particular, $\Box_a\texttt{mp(s)}$ and $\Box_b\texttt{mp(s)}$ are independent statements about whether the different principals a and b may possess $\texttt{s}$. The underlying logic then provides relationships between uses of modalities:

- Indexed modalities commute, i.e., $(\Box_a\Box_b\Phi) \Rightarrow (\Box_b\Box_a\Phi)$. This permits a sequence of indexed modalities to be treated as a multiset.
- The counit $(\Box_a\Phi) \Rightarrow \Phi$ allows a modality to be eliminated. The converse does not hold in general. Consequently, a right for principal a can be forgotten during logical deduction, but cannot be manufactured.
- From $\Box_a(\Phi_1 \Rightarrow \Phi_2)$ and $\Box_a\Phi_1$, we can deduce $\Box_a\Phi_2$; thus, the possessions of a principal are closed under deduction. From comultiplication, $(\Box_a\Phi) \Rightarrow (\Box_a\Box_a\Phi)$, we deduce that the deductions in the scope of a principal a includes the knowledge of a's posessions.
- If b is less secure than a and $\Box_b\Phi$ then we can deduce $\Box_a\Phi$; so, by this principle of Principal naturality, more secure principals have access to more secrets.

These relationships yield an indexed intuitionist S4 necessity modality, representing layers of permission for principals, over the underlying logic. This distinction in the logic between permissions and who has those permissions, represented by principal-indexed modalities, greatly reduces the need to quantify over principals during reasoning. For example, if a policy states that $\texttt{s2}$ may be possessed if $\texttt{s1}$ may be possessed, written $\texttt{mp(s1)} \Rightarrow \texttt{mp(s2)}$, then the indexed intuitionist S4 necessity modality structure allows this implication to be lifted to any principal a as $(\Box_a\texttt{mp(s1)}) \Rightarrow (\Box_a\texttt{mp(s2)})$.

Noninterference theorems [28] justify the use of indexed intuitionist S4 necessity modalities in this modeling. In that paper, we show that noninterference captures the idea that there is no information flow between differently indexed modalities. Let α be a modality free formula. The intuitive idea behind non interference is that if $\Box_a\alpha$ is derivable from some deductively closed set of hypothesis, then it is derivable from a subset of those hypothesis that are in the scope of the modality indexed by a, i.e. the formulas of the form $\Box_a\cdot$. In particular, noninterference implies the unprovability of the following formulas:

– $\Box_a\texttt{mp(s1)} \Rightarrow \Box_b\texttt{mp(s1)}$
– $(\Box_a(\texttt{mp(s1)} \Rightarrow \texttt{mp(s2)}) \wedge \Box_b\texttt{mp(s1)}) \Rightarrow \Box_a\texttt{mp(s2)}$

The unprovability of $\Box_a\texttt{mp(s1)} \Rightarrow \Box_b\texttt{mp(s1)}$ shows that the logical reasoning does not transfer capabilities unrestrictedly between principals. The unprovability of the second formula $(\Box_a(\texttt{mp(s1)} \Rightarrow \texttt{mp(s2)}) \wedge \Box_b\texttt{mp(s1)}) \Rightarrow \Box_a\texttt{mp(s2)}$ ensures that the acquisition of a new capability ($\texttt{s1}$) by another principal (b) does not create new capabilities for principal a by purely logical reasoning. Thus, non-interference facilitates distribution and decentralized enforcement of policies in the following sense. The reference monitor at a location uses logical reasoning to deduce whether a principal has sufficient capabilities to access the resource available at the location. Noninterference ensures that this reasoning is not dependent on other principals; so, the reference monitor at a location can function without knowledge of the principals at other locations.

We present three analyses to establish the utility of our approach:

– Sealed objects (Section 2) that demonstrate modeling of symmetric cryptography [2].
– An object encoding (Section 5) of Hardy's Confused Deputy [25].
– A web browser and server model to explore browser security policies and Cross-Site Request Forgery prevention solutions.

Related Work

Capability-based systems. Capabilities have been used to realize security policies in a variety of systems, e.g., [26,34,5] to name but a few. Distributed object languages such as E [15] illustrate the "capabilities-as-object references" paradigm where both subjects and resources are represented uniformly as objects, and classical object-oriented mechanisms are used to structure the exchange and invocation of capabilities. This viewpoint underlies Caja, a safe subset of Javascript. Caja eschews direct references to DOM objects, instead providing references to wrappers that restrict the capabilities provided on DOM objects. [29] formalize a notion of capability-safety, show that the subset Cajita satisfies this property and derive that Cajita programs have inter-component isolation.

Type systems for secrecy, confinement, and access control. In object-oriented languages, ownership and confinement types (see [14] for a survey of ownership type system) aim to delimit the portions of the object reference graph that can have references to the objects under consideration. In this paper, we generalize from confinement types to multi-party secrecy types using refinement types built on intuitionist S4 to express dependencies.

Abadi [4] describes a type system for controlling secret keys in the spi calculus, using a binary division of code as either fully trusted or untrusted. This paper explores an idea stated there: "distinguish various principals within the system, and enable us to discuss which of these principals have a given piece of data".

Language-based approaches to access control have long been studied in the setting of process calculi, though these approaches are not based explicitly in logic; two early references are [31,27]. In [18], Fournet, Gordon and Maffeis validate authorization policies statically using a specification language with "expect" assertions in a Datalog-style language. The `says` family of principal-indexed modalities is used in logics for reasoning about authorization statements made by different principals [21,1].The `says` modality has a monadic structure, as exemplified by the unit law ($\Phi \Rightarrow a\ \mathtt{says}\ \Phi$). In our prior work [12] we develop a type system based on authorization logic to capture provenance in a distributed object calculus. [17] explores the impact of compromised principals on authorization policies in a distributed setting.

In this paper, we carry out a similar program, albeit in the logical setting of intutionist S4, by reusing the infrastructure of *refinement types* [19] developed in the literature: policies (and therefore types) may quantify over object references of a given class. Object references (and variables) appear in logical formulae in equality predicates and in the "may-possess" predicate $\mathtt{mp}(.)$ described previously. Our semantics and notion of safety are from [12] and derive, ultimately, from [23] and [24].

2 Sealed Objects

In this section, we introduce the computational model and logic, by way of an example. The details of the logic can be found in a companion paper [28]; here we summarize the properties of the logic required to understand the example.

Computation is based on threads that communicate via a shared heap. Threads are "located" at the principal for which the thread is running; similarly objects are "located" at the principal that created the object. We use the terms "principal" and "location" interchangeably. For an object p, the location is available to the programmer via the pseudo-field $p.\mathsf{loc}$.

Neither threads nor objects can change location; however, object references can be communicated between threads using shared objects. A method invocation on an object leads to code execution at the location of the callee object. Thus, when the caller and callee objects are located at different locations, method invocation leads to a change of location context.

We conflate opponents, representing them all via $\perp$. Threads acting on behalf of opponents can only instantiate classes with trivial invariants, discussed below. Threads acting on behalf of non-opponents must obey a global policy. All threads must be well typed according to typical object-oriented programming rules, e.g., as in Java. Additionally, our type system controls communication of object references by non-opponent threads.

Principals are ordered by a partial order with least principal being the Opponent $\perp$. Principal naturality allows that whenever $\Box_{\perp}\Phi$ is deducible, then so is $\Box_a\Phi$, for any a. In particular, this means that any of the Opponent capabilities are available to all principals. Thus, our type system does not impose any restrictions more than those of usual object oriented programming on Opponent programs.

A program is *safe* if every object reference that is available to a principal at runtime is permitted by the global specification of permitted capabilities. Our type system ensures that safe well-typed programs remain safe under evaluation in the face of arbitrary opponent processes.

Consider `javax.crypto.SealedObject`. It permits a serializable object to be encrypted with a secret key and a symmetric-key cipher. The constructor is responsible for serialization and encryption. The resulting `SealedObject` contains only ciphertext. The original object can be recovered by passing the same secret key to `getObject`. We model `SealedObject` as:

```
class SealedObject {
  private final SecretKey key;
  private final Object contents;
  public SealedObject (SecretKey key, Object contents) {
    this.key = key; this.contents = contents;
  }
  public Object getObject (Section key) {
    if (key == this.key) return this.contents;
    else return null;
  }
}[ □⊥(mp(this.key) ⇒ mp(this.contents)) ]
```

By controlling possession of the `key`, one controls access to the `contents`. This code uses private fields guarded by object equality rather than encryption. This is sufficient since the type system enforces that the caller of `getObject` must possess `key`.

Specifications in our system are divided between a *global policy* and a set of *class invariants*. Intuitively, the combinatation of these policies indicates upper bounds on the capabilities that can be possessed by a principal. Our safety theorem shows that at any stage in the evolution of a system, even in the presence of opponents, any principal only possesses references that are provided for in the policy.

The global policy describes the distribution of initial secrets, and also any potential relationships between classes. It is informative to consider the following extremal global policies. Suppose that all class invariants are trivial (i.e., `tt`).

– The extremely permissive global policy $\forall\eta.\ \Box_{\bot}\mathtt{mp}(\eta)$ does not forbid any transmission of objects. Thus, typing under this global policy is essentially the same as standard object-oriented typing.
– The extremely restrictive global policy `tt` in the case where there are only two principals — Opponent ($\bot$) and Secret ($\top$) — forbids all transmission of objects from $\top$ to $\bot$. Thus, typing under this global policy is essentially the same as standard information flow.

The class invariant is intended to describe the private internals of a single class. The mutable state in our objects is only in the form of private instance variables. The class invariant is written at the end of each class, in square brackets. Because we are in a concurrent setting, we make the simplifying assumption that only final fields may be mentioned in the class invariant and that constructors may do nothing but assign fields — we also disallow reassignment of method parameters and local variables. Thus, the class invariant holds for every object at the point its constructor terminates.

References to `SealedObjects` can be safely sent anywhere because they do not leak their contents arbitrarily. The fact that they are allowed anywhere is exemplified by the global policy $(\forall \texttt{o}{:}\texttt{SealedObject}.\ \Box_{\bot}\texttt{mp}(\texttt{o}))$ — type-sorted quantification is shorthand for quantification using a "type" predicate on objects. This policy allows `SealedObjects` to be given to opponents; however, they can only retrieve the contents if they have the matching key. More restrictive policies are also possible.

The class invariant of `SealedObject` indicates that any principal that may possess `this.key` may also possess `this.contents`. Opponents cannot create secrets, and therefore are restricted to creating instances of "global" classes with invariants (i.e. "true") that are trivially satisfied. The invariant of `SealedObject` is nontrivial, and therefore opponents may not create instances of the class.

The invariant must be statically justifiable by any code that creates an instance of the class. For example, consider the code new SealedObject (key, acct), where `key` is an instance of `SecretKey` and `acct` is an instance of a `BankAccount` class. We must establish that every principal that may possess `key` may also possess `acct`, written $\Box_{\bot}(\texttt{mp}(\texttt{key}) \Rightarrow \texttt{mp}(\texttt{acct}))$. This might be accomplished using a global policy that allows `acct` to be possessed anywhere, written $\Box_{\bot}(\texttt{mp}(\texttt{acct}))$. Stricter policies could be specified pairwise, including $\Box_{\bot}(\texttt{mp}(\texttt{key}) \Rightarrow \texttt{mp}(\texttt{acct}))$ as a fact. More flexible arrangements are also possible, for example, using the invariant of the factory class that creates new keys. In any case, the implication must be deduced from the available policy in order to instantiate `SealedObject`. In all non-trivial cases, the initial ability to create `SealedObjects` is specified as part of the global policy; indeed, the non-interference theorems ensure that there is no possible creation of `SealedObjects` otherwise.

In order to justify safety of the `getObject` method, we first observe that the caller to `getObject` must possess the `key`, i.e., $\Box_{\mathsf{caller}}(\texttt{mp}(\texttt{key}))$. From the `SealedObject` class invariant, we know that:

$$\Box_{\bot}(\texttt{mp}(\texttt{this.key}) \Rightarrow \texttt{mp}(\texttt{this.contents}))$$

From which we can deduce that (note the principal on $\Box$):

$$\Box_{\mathsf{caller}}(\texttt{mp}(\texttt{this.key}) \Rightarrow \texttt{mp}(\texttt{this.contents}))$$

After the reference equality test `(key==this.key)`, the callee knows $\texttt{key} = \texttt{this.key}$. Moreover, equality can be lifted to comodalities, and we have $\Box_{\mathsf{caller}}(\texttt{key} = \texttt{this.key})$. From $\Box_{\mathsf{caller}}(\texttt{mp}(\texttt{key}))$ and $\Box_{\mathsf{caller}}(\texttt{key} = \texttt{this.key})$, we deduce $\Box_{\mathsf{caller}}(\texttt{mp}(\texttt{this.key}))$. In conjunction with the implication above, we find that $\Box_{\mathsf{caller}}(\texttt{mp}(\texttt{this.contents}))$. This justifies return of `this.contents` to the caller. In the case where the equality test fails, we use the property that `null` may be possessed anywhere, written $\Box_{\bot}(\texttt{mp}(\texttt{null}))$.

If the `SealedObject` class had public fields, then a higher threshold must be met to instantiate the class. In this case, one would also need to establish that any principal that may possess the object may also possess the values placed into the public fields.

It is worth noting that other symmetric cryptography schemes can be encoded as simple variants of `SealedObject`. For example, using nested conditionals, one can encode an object requiring n keys to encrypt and $k \leq n$ keys to decrypt.

3 Language

To formalize the preceding discussion, we first describe a distributed class-based language with mutable objects [12]. The operational semantics borrows heavily from [23], adding distribution [10,11] and classes. We consider typing in Section 4.

Syntax Names for classes (c, d), methods (ℓ), fields (f, g), variables (x, y, z), objects (p, q) and principals (a, b) are drawn from separate namespaces, as usual. Predicate variables (α, β) and predicate constructors (γ) occur in static annotations used during type-checking.

The reserved words of the language include: the variable name "this"; the principal "caller"; the class name Object; the predicate constructors "`tt`", "`ff`", "$\Rightarrow$", "$\wedge$", "$\vee$", "$\neg$" and "$\Box$". We write binary constructors infix.

The language is explicitly typed. Object types ($c\texttt{<}\vec{\phi}\texttt{>}$) include the actual predicate parameters $\vec{\phi}$, which we treat formally as *extended values*. Value types include objects (C), principals (Prin) and Unit. Extended value types include predicate types (P), which are resolved during typechecking. The process type (Proc) has no values.

One may write classes and methods that are generic in the predicate variables, achieving ML-style polymorphism with respect to effects. Class declarations thus include the formal predicate parameters $\vec{\alpha}$, which may occur in the effect Φ (see next table) associated with instances of the class. In addition to effects, class declarations include field and method declarations, but omit implicit constructor declarations. Fields include mutability annotations that are used in the statics. The syntax is as follows[1].

Types, Annotations, Class and Method Declarations

$C,D ::= c\texttt{<}\vec{\phi}\texttt{>}$	Object Types
$T,S ::= C \mid \mathsf{Prin} \mid \mathsf{Unit}$	Value Types
$P,Q ::= \mathsf{Pred}(\mathscr{T})$	Predicate Types
$\mathscr{T},\mathscr{S} ::= T \mid P \mid \mathsf{Proc}$	Types

$$\mu ::= \mathsf{private\ final} \mid \mathsf{private\ mutable} \mid \mathsf{public\ final}$$
$$\mathscr{D} ::= \mathsf{class}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \triangleleft D\{\vec{\mu}\ \vec{T}\ \vec{f};\ \vec{\mathscr{M}}\}[\Phi]$$
$$\mathscr{M} ::= \texttt{<}\vec{\beta} : \vec{Q}\texttt{>} S\ \ell(\vec{T}\ \vec{x})\{M\}$$

Values, Terms, Evaluation Contexts

$$V,W,U,A,B,\phi,\psi ::= x \mid p \mid a \mid \mathsf{unit} \mid \alpha \mid \gamma \mid \phi(\vec{V})$$
$$\begin{aligned} M,N,L,\Phi,\Psi ::=\ & V \mid V.f \mid V.\mathsf{loc} \\ & \mathsf{if}\ V = W\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{let}\ x = N;\ M \mid N \mathbin{\|\!\ulcorner} M \\ & V.f := W \mid \mathsf{let}\ x = \mathsf{new}\ c\texttt{<}\vec{\phi}\texttt{>}(\vec{V});\ M \mid \mathsf{let}\ x = V.\ell\texttt{<}\vec{\phi}\texttt{>}(\vec{W});\ M \\ & p{:}C\{\vec{f} = \vec{V}\} \mid (\nu p{:}C)\ M \mid a[M]^{b}_{c} \end{aligned}$$
$$\mathbb{E} ::= [\text{–}] \mid a[\mathbb{E}]^{b}_{c} \mid \mathsf{let}\ x = \mathbb{E};\ M \mid \mathbb{E} \mathbin{\|\!\ulcorner} N \mid M \mathbin{\|\!\ulcorner} \mathbb{E} \mid (\nu p)\ \mathbb{E}$$

[1] When writing definitions using classes and methods, we sometimes omit irrelevant bits of syntax, e.g., we leave out the parameters to classes when empty, such as writing Object rather than Object<$\emptyset$>. We identify syntax up to renaming of bound names, and write $M\{\!\!\{{}^{V}\!/_{x}\}\!\!\}$ for substitution of V for x in M (and similarly for other categories). We often omit type information. We use standard syntactic sugar in place of explicit sequencing. For example, we may write "$y.f.g$" to abbreviate "let $x = y.f;\ x.g$".

We use the metavariables ϕ, ψ, Φ and Ψ to represent values and terms of predicate type, and the other metavariables to represent runtime values and terms, with A and B reserved for values of principal type. Predicates are static annotations used in type-checking; they play no role in the dynamics. An *expectation* "$\mathsf{expect}\ \Phi$" as in [18] can be coded as "$\mathsf{new\ Proof{<}}\Phi\mathsf{{>}()}$", where class Proof is defined "$\mathsf{class\ Proof{<}}\alpha : \mathsf{Pred{>}\{\}}[\alpha]$".

The last three constructs in the definition of terms — $p{:}C\{\vec{f}{=}\vec{V}\}$, $(\nu p{:}C)\,M$, and $a\,[M]^{\,b}_{c}$ — are *dynamic constructs*. These constructs are not allowed in method declarations or initial code.

With the exception of $V.\mathsf{loc}$, $N \mathbin{\|\!\!\ulcorner} M$, and the terms on the last line of the definition, the constructs of the language are standard for class-based languages with generics.

The special "field" loc returns the location of an object. Concurrent composition ($\mathbin{\|\!\!\ulcorner}$) is asymmetric. In $N \mathbin{\|\!\!\ulcorner} M$, the returned value comes from M; the term N is available only for side effects. The terms on the last line are are not allowed to appear in declarations, as they model the runtime heap and call stack. These include heap elements $p{:}C\{\cdots\}$ (indicating that p is located at a with actual class C and fields $\vec{f}{=}\vec{V}$), name restriction (νp) (indicating that p is a fresh name) and frames $a\,[M]^{\,b}_{c}$ (indicating that M is running under authority of principle a and class c, with result available to b). We write irreducible frames simply as $a\,[M]$.

Structural congruence Evaluation is defined using a structural congruence on terms. Let $\equiv$ be the least congruence on terms that satisfies the following axioms. The rules for concurrent composition are from [23].. They capture properties of concurrent composition, including semi-associativity and the interaction with let. The rules for distribution are inspired by [11]. The interpretation of a value is independent of the location at which it occurs and the computation of a frame does not depend upon the location from which the frame was invoked (eg. $a\,[b\,[M]^{\,b'}_{d}]^{\,a'}_{c} \equiv b\,[M]^{\,b'}_{d}$) and axiomatize the interaction of let with distribution (eg. $a\,[\mathsf{let}\ x{=}N;\, M]^{\,a'}_{c} \equiv \mathsf{let}\ x{=}a\,[N]^{\,a'}_{c};\, a\,[M]^{\,a'}_{c}$).

Structural Congruence $(M \equiv M')$ (where $p \notin fn(M)$)

$$
\begin{array}{ll}
(M \mathbin{\|\!\!\ulcorner} N) \mathbin{\|\!\!\ulcorner} L \equiv M \mathbin{\|\!\!\ulcorner} (N \mathbin{\|\!\!\ulcorner} L) & \\
(M \mathbin{\|\!\!\ulcorner} N) \mathbin{\|\!\!\ulcorner} L \equiv (N \mathbin{\|\!\!\ulcorner} M) \mathbin{\|\!\!\ulcorner} L & a\,[\mathsf{let}\ x{=}b\,[V]^{\,a}_{d};\, M]^{\,a'}_{c} \equiv a\,[\mathsf{let}\ x{=}V;\, M]^{\,a'}_{c} \\
((\nu p)\,N) \mathbin{\|\!\!\ulcorner} M \equiv (\nu p)(N \mathbin{\|\!\!\ulcorner} M) & a\,[b\,[M]^{\,b'}_{d}]^{\,a'}_{c} \equiv b\,[M]^{\,b'}_{d} \\
M \mathbin{\|\!\!\ulcorner} ((\nu p)\,N) \equiv (\nu p)(M \mathbin{\|\!\!\ulcorner} N) & a\,[N \mathbin{\|\!\!\ulcorner} M]^{\,a'}_{c} \equiv a\,[N]^{\,a'}_{c} \mathbin{\|\!\!\ulcorner} a\,[M]^{\,a'}_{c} \\
\mathsf{let}\ x{=}(L \mathbin{\|\!\!\ulcorner} N);\, M \equiv L \mathbin{\|\!\!\ulcorner} (\mathsf{let}\ x{=}N;\, M) & a\,[\,(\nu p)\,N]^{\,a'}_{c} \equiv (\nu p)\,a\,[N]^{\,a'}_{c} \\
\mathsf{let}\ x{=}((\nu p)\,N);\, M \equiv (\nu p)(\mathsf{let}\ x{=}N;\, M) & a\,[\mathsf{let}\ x{=}N;\, M]^{\,a'}_{c} \equiv \mathsf{let}\ x{=}a\,[N]^{\,a'}_{c};\, a\,[M]^{\,a'}_{c}
\end{array}
$$

Evaluation The evaluation relation is defined with respect to an arbitrary fixed class table. The class table is referenced indirectly in the semantics through the lookup functions *fields* and *body*. Fix a global class table $\vec{\mathscr{D}}$. The fields and method lookup functions are standard.

Field Lookup $(\mathit{fields}(C) = \vec{\mu}\ \vec{T}\,\vec{f})$

$$
\frac{}{\mathit{fields}(\mathsf{Object}) = \emptyset}
\qquad
\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c{<}\vec{\alpha}{>} \lhd D\{\vec{\mu}\ \vec{T}\,\vec{f};\,\cdots\} \qquad \mathit{fields}(D\{\![\vec{\phi}/\vec{\alpha}]\!\}) = \vec{\mu}_D\ \vec{T}_D\,\vec{f}_D}{\mathit{fields}(c{<}\vec{\phi}{>}) = \vec{\mu}_D\ \vec{T}_D\,\vec{f}_D, (\vec{\mu}\ \vec{T}\,\vec{f})\{\![\vec{\phi}/\vec{\alpha}]\!\}}
$$

Method Lookup $(body(C.\ell) = \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\texttt{(}\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}})$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \triangleleft D\texttt{\{} \cdots \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\ \ell\texttt{(}\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}} \cdots \texttt{\}}}{body(c\texttt{<}\vec{\phi}\texttt{>}.\ell) = (\texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\texttt{(}\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}})\{\!\![\vec{\phi}/\vec{\alpha}]\!\!\}}$$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \triangleleft D\texttt{\{} \cdots \vec{\mathscr{M}}\texttt{\}} \quad \ell \text{ not defined in } \vec{\mathscr{M}} \qquad body(D\{\!\![\vec{\phi}/\vec{\alpha}]\!\!\}.\ell) = \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\texttt{(}\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}}}{body(c\texttt{<}\vec{\phi}\texttt{>}.\ell) = \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\texttt{(}\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}}}$$

Term Evaluation $(M \to M')$

$$\mathsf{let}\ y\texttt{=}\mathsf{new}\ C\texttt{(}\vec{V}\texttt{)}\texttt{;}\ L \to (\nu p : C)\,(p : C\texttt{\{}\vec{f}\texttt{=}\vec{V}\texttt{\}} \parallel\!\!\lceil L\{\!\![p/y]\!\!\})$$
$$\quad \text{if } \mathit{fields}(C) = \vec{f} \text{ and } |\vec{f}| = |\vec{V}|$$
$$b\texttt{[}p : C\texttt{\{} \cdots \texttt{\}]} \parallel\!\!\lceil a\texttt{[}\mathsf{let}\ y\texttt{=}p.\ell\texttt{(}\vec{W}\texttt{)}\texttt{;}\ L\texttt{]}_{d}^{a'} \to b\texttt{[}p : C\texttt{\{} \cdots \texttt{\}]} \parallel\!\!\lceil a\texttt{[}\mathsf{let}\ y\texttt{=}b\texttt{[}M'\texttt{]}_{c}^{a}\texttt{;}\ L'\texttt{]}_{d}^{a'}$$
$$\quad \text{if } body(C.\ell) = \texttt{(}\vec{x}\texttt{)}\texttt{\{}M\texttt{\}} \text{ and } |\vec{x}| = |\vec{W}| \text{ and } M' = M\{\!\![a/\mathsf{caller}]\!\!\}\{\!\![p/\mathsf{this}]\!\!\}\{\!\![\vec{W}/\vec{x}]\!\!\} \text{ and } C = c\texttt{<}\cdots\texttt{>}$$
$$b\texttt{[}p : C\texttt{\{} \cdots \texttt{\}]} \parallel\!\!\lceil p.\mathsf{loc} \to b\texttt{[}p : C\texttt{\{} \cdots \texttt{\}]} \parallel\!\!\lceil b$$
$$b\texttt{[}p : C\texttt{\{}f\texttt{=}V \cdots \texttt{\}]} \parallel\!\!\lceil p.f\texttt{:=}W \to b\texttt{[}p : C\texttt{\{}f\texttt{=}W \cdots \texttt{\}]} \parallel\!\!\lceil \mathsf{unit}$$
$$b\texttt{[}p : C\texttt{\{}f\texttt{=}V \cdots \texttt{\}]} \parallel\!\!\lceil p.f \to b\texttt{[}p : C\texttt{\{}f\texttt{=}V \cdots \texttt{\}]} \parallel\!\!\lceil V$$
$$\mathsf{if}\ V\texttt{=}V\ \mathsf{then}\ M\ \mathsf{else}\ N \to M$$
$$\mathsf{if}\ V\texttt{=}W\ \mathsf{then}\ M\ \mathsf{else}\ N \to N \text{ if } V \neq W$$
$$\mathsf{let}\ x\texttt{=}V\texttt{;}\ M \to M\{\!\![V/x]\!\!\}$$
$$\frac{M \equiv N \to N' \equiv M'}{M \to M'} \qquad \frac{M \to M'}{\mathbb{E}[M] \to \mathbb{E}[M']}$$

The new construct creates an object and returns a reference to it. The result is a concurrent composition: the new object appears on the left, the return value on the right. Method invocation happens at the callee site, and thus a new frame is introduced in the consequent $b\texttt{[}M'\texttt{]}_{c}^{a}$; the result of the method call will be made available to a. In M', the distinguished variables caller and this are bound to the calling principal and the object upon which the method is invoked respectively.

4 Types

The type system controls the distribution of object references via logical policies. We follow [18], as adapted to distributed OO languages with localities in [12].

By allowing predicates to include open values, we can reason about terms that include variables, such as x; however, we cannot reason about $x.f$. Thus we extend the type system to include equations between terms and values. Allowing any term is unsound, however, since our language includes mutability. Thus we identify a subset of *pure* terms which do not include mutable features. In addition, we require that evaluation of pure terms must terminate, and therefore we disallow method calls in pure terms. To shorten some definitions, we define a category of *identifiers*, η, which include bound names and principals.

$$\eta ::= x \mid p \mid a \mid \alpha$$

Environments have two types of data: type bindings for names (as usual) and logical phrases, including equalities and predicates. Define $dom(\Delta) = \{\eta \mid \eta : \mathscr{T} \in \Delta\}$.

$$\Delta ::= \emptyset \mid \Delta, \eta : \mathscr{T} \mid \Delta, \Phi \mid \Delta, V\texttt{=}M$$

Predicate lookup $(\mathit{effect}(C) = \Phi)$ is similar to method lookup. Here "$\Phi_D \wedge \Phi\{\![^{\vec{\phi}}/\vec{\alpha}]\!\}$" is sugar for "$\mathsf{let}\ x = \Phi_D;\ \mathsf{let}\ y = \Phi\{\![^{\vec{\phi}}/\vec{\alpha}]\!\};\ x \wedge y$".

$$\frac{}{\mathit{effect}(\mathsf{Object}) = \mathsf{true}} \qquad \frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha} : \vec{P}\texttt{>} \triangleleft D\texttt{\{} \cdots \texttt{\}}[\Phi] \quad \mathit{effect}(D\{\![^{\vec{\phi}}/\vec{\alpha}]\!\}) = \Phi_D}{\mathit{effect}(c\texttt{<}\vec{\phi}\texttt{>}) = \Phi_D \wedge \Phi\{\![^{\vec{\phi}}/\vec{\alpha}]\!\}}$$

We also define a function $(\mathit{env}_a(M) = \Delta)$ to create an environment from a term.

$$\mathit{env}_a(\eta : C\texttt{\{}\vec{f}=\vec{V}\texttt{\}}) = \Box_a\mathtt{mp}(\eta), a = \eta.\mathsf{loc}, V_1 = \eta.f_1, \ldots, V_n = \eta.f_n$$
$$\mathit{env}_a(\mathsf{let}\ x = N;\ M) = \mathit{env}_a(N) \qquad \mathit{env}_a(N \mathbin{|\!\lceil} M) = \mathit{env}_a(N), \mathit{env}_a(M)$$
$$\mathit{env}_a(b\,\texttt{[}M\texttt{]}_c^{a'}) = \mathit{env}_b(M) \quad \mathit{env}_a(\texttt{(}\nu p : C\texttt{)}\,M) = p : C, \mathit{env}_a(M) \qquad \mathit{env}_a(M) = \emptyset,\ \mathit{otherwise}$$

The type system is parameterized with respect to a semantic entailment relation $(\Delta \vDash \Psi)$. In addition to the rules arising from indexed intuitionist necessity modalities, we expect the relation to support domain specific axioms and satisfy the following properties. Let σ stand for substitutions of pure terms M for x.

1. If $\Delta \vDash \Psi$ then $\Delta\sigma \vDash \Psi\sigma$, for any substitution σ from variables to values, or from principals to principals.
2. If $\Delta, V = V, \Delta' \vDash \Psi$ then $\Delta, \Delta' \vDash \Psi$.
3. If $\Delta, x : T, x = M, \Box_a\mathtt{mp}(x), \Delta' \vDash \Psi$ and $\Delta, \Delta' \vDash \Box_a\mathtt{mp}(M)$ then $\Delta, \Delta' \vDash \Psi\{\![^M/x]\!\}$.

In examples, we assume that whenever $\Box_a\mathtt{mp}(\eta)$ and $\eta : C$ are deducible, then so is $\Box_a\mathtt{mp}(\eta.f)$ for every public field of C.

The standard judgements required for the type system are relegated to Appendix C including subtyping $(\vdash \mathscr{T}' <: \mathscr{T})$, well-formed overriding $(\vdash \texttt{<}\vec{\beta} : \vec{Q}\texttt{>}S\texttt{(}\vec{T}\texttt{)}\ \mathit{overrides}\ D.\ell)$, well-formed types $(\Delta \vdash \mathscr{T})$, and well-formed environments $(\Delta \vdash \diamond)$. The only noteworthy aspect of these definitions is that the implication of the effects for the same base class also yields subtyping:

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}\texttt{>} \quad \vec{\phi} \vDash \vec{\psi} \quad |\vec{\alpha}| = |\vec{\phi}| = |\vec{\psi}|}{\vdash c\texttt{<}\vec{\phi}\texttt{>} <: c\texttt{<}\vec{\psi}\texttt{>}}$$

The judgments for declarations have the standard format. The judgment for values include a script a, indicating that the value is well typed at a specific location. The judgment for terms carries additional structure. In $\Delta \vdash_a^{a'} M : \mathscr{T}\ \rho\ d$, a should be read as the location of the term, a' as the location of the caller, $\mathscr{T}$ as the type of the resulting value, d as the class from which the code is derived, and $\rho \in \{\mathsf{Pure}, \mathsf{Impure}\}$ as a *purity annotation*.

The effect on a class must be a pure term of type Pred. The rule for typing methods uses a standard well-formed overriding definition. The typing of the method body occurs in the context of an abstract principal a that is constrained to coincide with the location of the ambient object. Similarly, the abstract principal caller is constrained to coincide with the annotation on the typing of the body of the method. In typing the method body, one can use the logical variables of the class, the method declaration and assume that the caller was permitted to possess the arguments.

Well-Formed Declarations $(\Delta \vdash \mathscr{D})$ $(\Delta \vdash \mathscr{M}\ \mathit{in}\ c{<}\vec{\alpha}:\vec{P}{>}{\lhd}D)$

$$\frac{\begin{array}{l}\Delta,\vec{\alpha}:\vec{P}\vdash D,\vec{T} \quad \Delta,\vec{\alpha}:\vec{P},a:\mathsf{Prin},\mathsf{this}:c{<}\vec{\alpha}{>},a{=}\mathsf{this.loc},\Box_a\mathtt{mp}(\mathsf{this}) \models^{a}_{a} \Phi : \mathsf{Pred\ Pure}\ c\\ \Delta\vdash\vec{\mathscr{M}}\ \mathit{in}\ c{<}\vec{\alpha}:\vec{P}{>}{\lhd}D \quad \mathit{fields}(D)=\vec{\mu}_D\,\vec{T}_D\,\vec{f}_D \quad \vec{f}_D\cap\vec{f}=\emptyset \qquad\qquad a\notin \mathit{fn}(M)\end{array}}{\Delta\vdash\mathsf{class}\ c{<}\vec{\alpha}:\vec{P}{>}{\lhd}D\{\vec{\mu}\ \vec{T}\,\vec{f};\ \vec{\mathscr{M}}\}[\Phi]}$$

$$\frac{\begin{array}{l}\Delta,\vec{\alpha}:\vec{P},\vec{\beta}:\vec{Q}\vdash S,\vec{T} \quad \vdash S'<:S \quad \vdash{<}\vec{\beta}{>}S(\vec{T})\ \mathit{overrides}\ D.\ell\\ \Delta,\vec{\alpha}:\vec{P},\vec{\beta}:\vec{Q},\vec{x}:\vec{T},a:\mathsf{Prin},\mathsf{this}:c{<}\vec{\alpha}{>},a{=}\mathsf{this.loc},\Box_a(\mathtt{mp}(\mathsf{this})\wedge\mathtt{mp}(\vec{x})),\\ \qquad\qquad\qquad \mathsf{caller}:\mathsf{Prin},\Box_{\mathsf{caller}}\mathtt{mp}(\vec{x}) \models^{\mathsf{caller}}_{a} M:S'\ \rho\ c \quad a\notin\mathit{fn}(M)\end{array}}{\Delta\vdash{<}\vec{\beta}:\vec{Q}{>}S\ \ell(\vec{T}\ \vec{x})\{M\}\ \mathit{in}\ c{<}\vec{\alpha}:\vec{P}{>}{\lhd}D}$$

The judgment for values requires that well-formed objects satisfy their class invariants. In addition, the object value, as well as the objects held in its public fields must be permitted at the given location.

Well-Formed Values and Terms $(\Delta \models_a V:\mathscr{T})$ $(\Delta \models^{a'}_{a} M:\mathscr{T}\ \rho\ d)$ $(\rho\ ::=\ \mathsf{Pure}\ |\ \mathsf{Impure})$

$$\frac{\Delta\ni b:\mathsf{Prin}}{\Delta\models_a b:\mathsf{Prin}} \qquad \frac{\Delta\ni x:T \quad \Delta\vDash\Box_a\mathtt{mp}(x)}{\Delta\models_a x:T} \qquad \frac{\Delta\ni p:C \quad \Delta\vDash\Box_a\mathtt{mp}(p)}{\Delta\models_a p:C} \qquad \frac{}{\Delta\models_a \mathsf{unit}:\mathsf{Unit}}$$

$$\frac{\Delta\ni\alpha:\mathsf{Pred}(\vec{\mathscr{T}})}{\Delta\models_a\alpha:\mathsf{Pred}(\vec{\mathscr{T}})} \qquad \frac{\mathit{arity}(\gamma)=\vec{\mathscr{T}}}{\Delta\models_a\gamma:\mathsf{Pred}(\vec{\mathscr{T}})} \qquad \frac{\Delta\models_a\phi:\mathsf{Pred}(\vec{\mathscr{T}}) \quad \Delta\models_a\vec{V}:\vec{\mathscr{T}}}{\Delta\models_a\phi(\vec{V}):\mathsf{Pred}}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\models_a p:C \quad \mathit{fields}(C)=\vec{\mu}\ \vec{T}\,\vec{f} \quad \Delta\models_a\vec{V}:\vec{T}' \quad \vdash\vec{T}'<:\vec{T}\\ \Delta,\mathit{env}_a(p{:}C\{\vec{f}{=}\vec{V}\})\vDash\mathit{effect}(C)\{\![{}^{p}/\mathsf{this}]\!\}\end{array}}{\Delta\models^{a'}_{a}\ p{:}C\{\vec{f}{=}\vec{V}\}:\mathsf{Proc}\ \rho\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\vdash C \quad \Delta,x{:}C\vDash\Box_a\mathtt{mp}(x) \quad \mathit{fields}(C)=\vec{\mu}\ \vec{T}\,\vec{f} \quad \Delta\models_a\vec{V}:\vec{T}' \quad \vdash\vec{T}'<:\vec{T}\\ \Delta,x{:}C,\mathit{env}_a(x{:}C\{\vec{f}{=}\vec{V}\})\vDash\mathit{effect}(C)\{\![{}^{x}/\mathsf{this}]\!\} \qquad \Delta,x{:}C,\mathit{env}_a(x{:}C\{\vec{f}{=}\vec{V}\})\models^{a'}_{a} M:\mathscr{T}\ \rho\ d\end{array}}{\Delta\models^{a'}_{a}\ \mathsf{let}\ x{=}\mathsf{new}\ C(\vec{V});M:\mathscr{T}\ \mathsf{Impure}\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta,\Box_a\mathtt{mp}(x)\models_a p:C \quad \mathit{fields}(C)=\vec{\mu}\ \vec{T}\,\vec{f} \quad \Delta\models_a\vec{V}:\vec{T}' \quad \vdash\vec{T}'<:\vec{T}\\ \Delta,\mathit{env}_a(p{:}C\{\vec{f}{=}\vec{V}\})\vDash\mathit{effect}(C)\{\![{}^{p}/\mathsf{this}]\!\}\end{array}}{\Delta\models^{a'}_{a}\ p{:}C\{\vec{f}{=}\vec{V}\}:\mathsf{Proc}\ \rho\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\vdash C \quad \mathit{fields}(C)=\vec{\mu}\ \vec{T}\,\vec{f} \quad \Delta\models_a\vec{V}:\vec{T}' \quad \vdash\vec{T}'<:\vec{T}\\ \Delta,x{:}C,\mathit{env}_a(x{:}C\{\vec{f}{=}\vec{V}\})\vDash\mathit{effect}(C)\{\![{}^{x}/\mathsf{this}]\!\} \qquad \Delta,x{:}C,\mathit{env}_a(x{:}C\{\vec{f}{=}\vec{V}\})\models^{a'}_{a} M:\mathscr{T}\ \rho\ d\end{array}}{\Delta\models^{a'}_{a}\ \mathsf{let}\ x{=}\mathsf{new}\ C(\vec{V});M:\mathscr{T}\ \mathsf{Impure}\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\models_a V:C \quad \mathit{body}(C.\ell)={<}\vec{\beta}:\vec{Q}{>}S(\vec{T}) \quad \Delta\models_a\vec{\phi}:\vec{Q} \quad \Delta\models_a\vec{W}:\vec{T}' \quad \vdash\vec{T}'<:\vec{T}\{\![{}^{\vec{\phi}}/\vec{\beta}]\!\}\\ \Delta,b{:}\mathsf{Prin},b{=}V.\mathsf{loc}\vDash\Box_b\mathtt{mp}(\vec{W}) \quad b\notin\mathit{dom}(\Delta) \quad \Delta,x{:}S\{\![{}^{\vec{\phi}}/\vec{\beta}]\!\},\Box_a\mathtt{mp}(x)\models^{a'}_{a} M:\mathscr{T}\ \rho\ d\end{array}}{\Delta\models^{a'}_{a}\ \mathsf{let}\ x{=}V.\ell{<}\vec{\phi}{>}(\vec{W});M:\mathscr{T}\ \mathsf{Impure}\ d}$$

$$\frac{\Delta\vdash\diamond \quad \Delta\models_a V:d{<}\vec{\phi}{>} \quad \mathit{fields}(d{<}\vec{\phi}{>})=\vec{\mu}\ \vec{T}\,\vec{f} \quad \mu_i=\mathsf{private\ mutable} \quad \Delta\models_a W:T' \quad \vdash T'<:T_i}{\Delta\models^{a'}_{a}\ V.f_i{:=}W:\mathsf{Unit\ Impure}\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\models_a V:c{<}\vec{\phi}{>} \quad \mathit{fields}(c{<}\vec{\phi}{>})=\vec{\mu}\ \vec{T}\,\vec{f} \quad \text{If } \mu_i\ni\mathsf{private}\text{ then } c=d\\ \text{If } \mu_i\ni\mathsf{mutable}\text{ then } \rho=\mathsf{Impure}\end{array}}{\Delta\models^{a'}_{a}\ V.f_i:T_i\ \rho\ d}$$

$$\frac{\begin{array}{l}\Delta\vdash\diamond \quad \Delta\models_a V:T \quad \Delta\models_a W:S \quad \Delta,V{=}W\models^{a'}_{a} M:\mathscr{T}\ \rho\ d \quad \Delta\models^{a'}_{a} N:\mathscr{T}'\ \rho\ d\\ \text{Either }\vdash\mathscr{T}'<:\mathscr{T}=\mathscr{T}''\text{ or }\vdash\mathscr{T}<:\mathscr{T}'=\mathscr{T}''\end{array}}{\Delta\models^{a'}_{a}\ \mathsf{if}\ V{=}W\ \mathsf{then}\ M\ \mathsf{else}\ N:\mathscr{T}''\ \rho\ d}$$

$$\frac{\Delta \models^{a}_{a} N : T\ \mathsf{Impure}\ d \qquad \Delta, env_a(N), x{:}T, \Box_a\mathtt{mp}(x) \models^{a'}_{a} M : \mathscr{T}\ \rho\ d}{\Delta \models^{a'}_{a} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \mathsf{Impure}\ d} \qquad \frac{right(N) = N' \qquad \Delta \models^{a}_{a} N : T\ \rho\ d \qquad \Delta, env_a(N) \models^{a}_{a} N' : T\ \mathsf{Pure}\ d \qquad \Delta, env_a(N), x{:}T, x{=}N', \Box_a\mathtt{mp}(x) \models^{a'}_{a} M : \mathscr{T}\ \rho\ d}{\Delta \models^{a'}_{a} \mathsf{let}\ x{=}N;\ M : \mathscr{T}\ \rho\ d}$$

$$\frac{\Delta, env_a(M) \models^{a''}_{a} N : \mathscr{T}'\ \rho\ d \qquad \Delta, env_a(N) \models^{a'}_{a} M : \mathscr{T}\ \rho\ d}{\Delta \models^{a'}_{a} N \parallel\!\lceil M : \mathscr{T}\ \rho\ d} \qquad \frac{\Delta, p{:}C \models^{a'}_{a} M : \mathscr{T}\ \rho\ d}{\Delta \models^{a'}_{a} (\nu p{:}C)\, M : \mathscr{T}\ \rho\ d}$$

$$\frac{\Delta \vdash \diamond \qquad \Delta \models_{a'} V : \mathscr{T}}{\Delta \models^{a'}_{a} V : \mathscr{T}\ \rho\ d} \qquad \frac{\Delta \vdash \diamond \qquad \Delta \models_{a} V : C}{\Delta \models^{a'}_{a} V.\mathsf{loc} : \mathsf{Prin}\ \rho\ d} \qquad \frac{\Delta \models_{a} b : \mathsf{Prin} \qquad \Delta \models^{b'}_{b} M : \mathscr{T}\ \rho\ c}{\Delta \models^{a'}_{a} b[M]^{b'}_{c} : \mathscr{T}\ \rho\ d}$$

The typing rules for terms are designed to establish several invariants, which we now discuss.
Well located. The rules for terms use the value judgment to ensure that value occurrences are available at a given location. The rule for located terms switches principals as expected.
Purity annotations. Field updates and constructor calls are impure because they mutate the heap. Field accesss to mutable fields are impure because they rely on mutable state. Method invocations are impure because they might not terminate. In all other cases, the purity annotation is constructed inductively. For example, a let is pure only if both terms involved are pure. Similarly for concurrent composition and conditionals.
Equations. The rule for pure let terms uses the function *right*. Intuitively, for any term N, $right(N)$ returns the rightmost subterm of N after it has been rewritten to a normal form. Routine details are omitted. Conditionals and let expression on pure terms introduce equations to the environment. Equations are also generated by the rules for heap objects ($\eta : C\{\cdots\}$) and new.
Caller annotations. The caller annotation is carried inductively through all rules but two. In the rule for the concurrent composition, only the right term is constrained; the value of the left term is ignored. The purpose of the caller annotation is revealed by the rule for values which appear as terms — these are the return values. The rule ensures that the caller principal is permitted to have a reference to the value.
Checking effects and the $\mathtt{mp}(\cdot)$ *predicate.* The rule for new illustrates the methodology. (The rule for heap objects enforces similar proof obligations.) In this rule, the hypothesis for typing fields is standard. The lookup of the effect obligation via $effect(C)$ yields a conjunction of the effects for this class and all its superclasses. The proof obligations ensure that the created object conforms to the class predicate, and that the reference and its public fields are permitted to be at the principal at which the object is located. The facts used to discharge this proof obligation are derived from the environment via Δ which accumulates the benefits derivable from the objects declared in the environment and the equations accumulated in the environment via lets and conditionals. The parameters to the constructor have to be available at the current location a.

In field update and lookup, the class annotation on the typing judgment is ensured to be the class of the object if the field is private.

In the rule for "generic" methods, we substitute concrete formulas for the logical variables being carried in the method definition. Since methods are executed at the location of the callee, we check to ensure that the location of the callee object possesses the right to hold references to the objects being passed in as actual parameters.

Conjoining specifications The rule for concurrent composition reflects the ideas from conjoining specifications of concurrent systems [3] — each component can assume the information exposed by the other component.

Results. An *initial program* is one that contains no dynamic constructs.

An *opponent class* is one whose effect is trivial, i.e., `tt`. An *opponent program* is one that can be typed only allowing the constructor rule for opponent classes. In typing opponents, we allow the assumption $\forall\eta.\ \Box_{\perp}\texttt{mp}(\eta)$. Thus opponents are typed using a restricted class table, but under a permissive policy. This permissive policy is essentially the same as standard object-oriented typing.

An opponent can instantiate opponent classes. By Principal Naturality, the opponent can unconstrainedly pass arguments or return results in method invocations. Thus, the opponent typability requirement in the following safety result means only that the opponent program is typable in the sense of classic object-oriented programming.

Recall that a frame is a term of the form $a\,\texttt{[}M\texttt{]}_{c}^{b}$.

Definition 1. A term M is *safe for Δ* if whenever $M \to^* N$, $N \equiv \mathbb{E}[a\,\texttt{[}N'\texttt{]}_{c}^{b}]$, N' contains no frames and $p \in fn(N)$ then $\Delta, env_{\perp}(N) \vDash \Box_{a}\texttt{mp}(p)$. □

Proposition 2. Suppose that $\Delta \vdash_{\perp} M$ and $\Delta, env_{\perp}(M), (\forall\eta.\ \Box_{\perp}\texttt{mp}(\eta)) \vdash_{\perp} N$ for an initial opponent program N. Then $N \,\Vert\!\!\!\ulcorner M$ is safe for Δ. □

The safety result ensures that well-typed trustworthy programs are safe when combined with arbitrary (typed but untrustworthy) opponents.

5 The Confused Deputy

In this section, we examine how to typecheck code that addresses the Confused Deputy problem using object references as capabilities.

Hardy [25] discusses a system with a compiler invoked by a user. The compiler writes two files, in addition to any generated code. The first is a statistics file. The name of the statistics file is hardcoded into the compiler, and the compiler is explicitly granted permission to write to that file. The second is a debugging file, chosen by the user. In order to write to the user's choice of debugging file, the compiler must be granted a broad permission. Hardy describes an occasion when a user selected a sensitive file—subsequently overwritten by the compiler—and dubs the compiler a Confused Deputy.

Hardy's solution requires the user to obtain a capability to write to the debugging file, and to send that capability to the compiler. The compiler can use the user's capability to write to the debugging file.

Modeling. We model Hardy's solution using the code in Figure 1. Following the object references as capabilities paradigm, the capabilities to write to files are represented by a `FileOutputStream` class (as in Java).

The `User` class invokes a compiler, passing the `FileOutputStream` contents of the `fDebug` field. The `User` class allows its `fDebug` field to be updated via a method `setDebug`—we examine the typing consequences below.

```
class User {
  private final Compiler compiler;
  private FileOutputStream fDebug;
  void action () {
    ...
    this.compiler.compile (this.fDebug, ...); // Invoke with current fDebug.
  }
  void setDebug (FileOutputStream fDebug) {
    this.fDebug = fDebug;
  }
}[ ∀o : FileOutputStream.□_{this.loc}mp(o) ⇒ □_{this.compiler.loc}mp(o) ]

class Compiler {
  private final FileOutputStream fStats;
  public void compile (FileOutputStream fDebug, String source) {
    ...
    this.fStats.write (...); // Write statistics to fStats.
    fDebug.write (...); // Write debugging output to fDebug.
  }
}
```

Fig. 1. User and Compiler Code

The `Compiler` class must be initialized with a final `FileOutputStream` field `fStats` at construction. When it compiles, it uses its own `fStats` field and the `fDebug` method parameter supplied by the caller to write to the statistics and debugging files.

Controlling capabilities. The capability solution improves upon the Confused Deputy situation, with respect to the principle of least privilege, because the compiler lacks the broad permission to write to many files in the object/capability solution. Hardy observes that achieving a comparable system with a traditional access control policy for the compiler is challenging, e.g., because the compiler may be invoked by different users with access to different files.

However, the capability solution is not entirely satisfactory. As discussed in the introduction, an untrustworthy compiler might forward capabilities that it receives to objects at different locations (principals).

Type assignment. We now consider how to typecheck the code of Figure 1 in a way that allows the `Compiler` to receive the `fDebug` object reference but not forward it to another location. We omit discussion of the source code given to the compiler, and any executable output, for reasons of space.

The typing of the compiler's use of `FileOutputStream` references is straightforward. The compiler receives permission to possess the field `fStats` implicitly. More generally, our type system implicitly allows every object to access its own fields. On the other hand, code that constructs a `Compiler` instance is responsible for ensuring that the chosen location of the compiler is able to possess `fStats`. That is, if a newly created `Compiler` instance is referenced via `c`, then the proof obligation is $\Box_{\texttt{c.loc}}\texttt{mp}(\texttt{c.fStats})$. Similarly, our type system implicitly grants the compiler permission to possess the method parameter `fDebug`, and the obligation lies with the caller to ensure that the location of the callee may possess `fDebug`.

Typechecking the body of the `compile` method does not introduce proof obligations involving `fStats` or `fDebug`, because those values are passed as `this`, and the type system automatically validates $(\forall \texttt{o}.\Box_{\texttt{o.loc}}\texttt{mp}(\texttt{o}))$. That is, a location may possess a reference to any object stored at that location.

The user has a more interesting policy, because it has to permit forwarding of `fDebug` to the compiler. The form of the policy hinges upon the mutability of the `fDebug` field. For example, if `fDebug` was a final field, it could be referred to in the class invariant, e.g., with form:

$$\Box_{\texttt{this.compiler.loc}}\Box_{\texttt{this.loc}}\texttt{mp}(\texttt{this.fDebug})$$

which entails:

$$(\Box_{\texttt{this.loc}}\texttt{mp}(\texttt{this.fDebug})) \land (\Box_{\texttt{this.compiler.loc}}\texttt{mp}(\texttt{this.fDebug}))$$

To demonstrate a more flexible alternative, we chose to make `fDebug` non-final (mutable) in the code of Figure 1. Since we can no longer refer to the field in a class invariant, we instead state that all `FileOutputStream` references that may be possessed at the location of `User` may also be possessed at the location of the corresponding compiler. This class invariant is written:

$$\forall \texttt{o} : \texttt{FileOutputStream}.\Box_{\texttt{this.loc}}\texttt{mp}(\texttt{o}) \Rightarrow \Box_{\texttt{this.compiler.loc}}\texttt{mp}(\texttt{o})$$

With this class invariant, typechecking justifies forwarding of `this.fDebug` to `this.compiler` using implication together with the facts that: (1) `this.loc` may possess `this.fDebug` (the location of an object implicitly possesses its fields); and (2) `this.fDebug` is declared to be an instance of `FileOutputStream`.

Finally, code that constructs an instance of `User` has an obligation to show that the associated `Compiler` instance is usable with any `FileOutputStream` object reference that the `User` receives.

6 Conclusion

The control of the possession and transmission of secrets is a recurrent theme in security literature and practice. The policies on possession in this paper describe an upper bound on the principals who can possess a secret. We describe a static analysis to ensure programs in a distributed object-oriented language comply with such policies. Our static analysis takes the form of a refinement type system, based on indexed necessity modalities from intuitionist S4, for an object calculus with locations. The safety result ensures that in the configurations that arise from the execution of well-typed programs, objects are only accessible to principals who are permitted to do so by the system policy, even in the presence of attackers who try to subvert the policies by inserting malicious objects and code into the system. Our results suggest that type systems are a practical tool to debug secrecy errors in the design of user-defined APIs in distributed systems.

Acknowledgements We thank the referees of a previous version of this paper for useful comments. This research was supported by NSF CCF-0915704.

References

1. Abadi, M.: Access control in a core calculus of dependency. ENTCS. 172, 5–31 (2007)
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148, 36–47 (1999)
3. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. 17(3), 507–535 (1995)
4. Abadi, M.: Secrecy by typing in security protocols. Journal of the ACM 46, 611–638 (1998)
5. Anderson, M., Pose, R.D., Wallace, C.S.: A password-capability system. Comput. J. 29(1), 1–8 (1986)
6. Barth, A.: RFC 6265: HTTP State Management Mechanism (2011)
7. Barth, A.: RFC 6454: The Web Origin Concept (2011)
8. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. CCS'08. pp. 75–88 (2008)
9. Bierman, G.M., de Paiva, V.C.V.: On an intuitionistic modal logic. Studia Logica 65 (2001)
10. Cardelli, L.: A language with distributed scope. POPL. pp. 286–297 (1995)
11. Castellani, I.: Process algebras with localities. Handbook of Process Algebra, chap. 15, pp. 945–1045 (2001)
12. Cirillo, A., Jagadeesan, R., Pitcher, C., Riely, J.: TAPIDO: Trust and authorization via provenance and integrity in distributed objects. ESOP. pp. 208–223 (2008)
13. DeYoung, H., , Pfenning, F.: Reasoning about the consequences of authorization policies in a linear epistemic logic. Tech. Rep. 1213, CMU (2009)
14. Drossopoulou, S.: Ten years of ownership types or the benefits of putting objects into boxes (2008), invited talk at BCS. Talk available at http://www.doc.ic.ac.uk/~scd/BCS.pdf
15. E: Open source distributed capabilities, http://www.erights.org
16. Feil, R., Nyffenegger, L.: Evolution of cross site request forgery attacks. Journal in Computer Virology 4(1), 61–71 (Nov 2007)
17. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization in distributed systems. CSF (2007)
18. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. ACM Trans. Program. Lang. Syst. 29(5) (2007)
19. Freeman, T., Pfenning, F.: Refinement types for ML. pp. 268–277. PLDI '91, ACM, New York, NY, USA (1991)
20. Garg, D., Bauer, L., Bowers, K.D., Pfenning, F., Reiter, M.K.: A linear logic of authorization and knowledge. ESORICS. LNCS, vol. 4189, pp. 297–312 (2006)
21. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. CSFW. pp. 283–296 (2006)
22. Gong, L., Mueller, M., Prafullch, H.: Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. USENIX Symposium on Internet Technologies and Systems. pp. 103–112 (1997)
23. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: Reduction and typing. Proceedings HLCL'98 (1998)
24. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security 11(4), 451–520 (2003)

25. Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev. 22, 36–38 (October 1988)
26. Hardy, N.: KeyKOS architecture. SIGOPS Oper. Syst. Rev. 19, 8–25 (October 1985)
27. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Information and Computation 173, 2002 (1998)
28. Jagadeesan, R., Pitcher, C., Riely, J.: Non interference for intuitionist necessity. Tech. Rep. 12-003, School of Computing, DePaul University (2012)
29. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. IEEE Symposium on Security and Privacy. pp. 125–140 (2010)
30. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. Financial Cryptography. LNCS, vol. 5628, pp. 238–255 (2009)
31. Nicola, R.D., Ferrari, G., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 24, 315–330 (1997)
32. Pfenning, F., Wong, H.C.: On a modal λ-calculus for S4. Proceedings of MFOS. New Orleans, Louisiana (Mar 1995), *ENTCS*, Volume 1, Elsevier
33. Ryck, P.D., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: Csfire: Transparent client-side mitigation of malicious cross-domain requests. ESSoS. LNCS, vol. 5965, pp. 18–34 (2010)
34. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system. SIGOPS Oper. Syst. Rev. 33, 170–185 (Dec 1999)
35. Zalewski, M.: Browser Security Handbook, part 2. Google Inc. (2008), http://code.google.com/p/browsersec/wiki/Part2

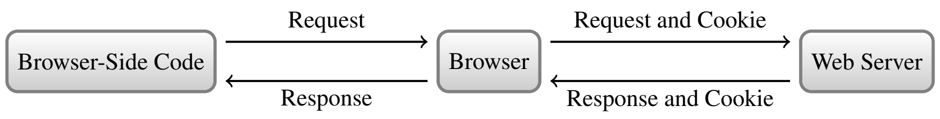

Fig. 2. Interaction Between Browser-Side Code, Web Browser, and Web Server

A Browser Security Policy and Cross-Site Request Forgery

In this section, we continue to investigate "'Deputy" problems, by considering how typing could have been used to derive elements of browser security policy and a typical solution to Cross-Site Request Forgery (CSRF). Our approach is to, first, build a crude object-oriented model that captures interaction between browser-side code, web browsers, and web servers. In particular, we model the browser addition of cookies to outgoing requests. Next, we consider policies that control a password-protected secret stored on a web server. The effect of these policies is to prevent unauthorized reads of the secret by cross-site requests, and unauthorized state changes on the web server, e.g., a password change from a CSRF attack.

A.1 Overview

The basic interaction that we consider is diagrammed in Figure 2. The distinct locations/principals are categorized as pieces of browser-side code (representing both JavaScript code and the behavior of a user interacting with a web page), a web browser, and a web server. There can be multiple web servers, each associated with a domain name. Similarly, for browser-side code. The distinction between browser-side code and the browser reflects the usual browser security model [35].

In Figure 2, browser-side code initiates an HTTP request via the browser. The browser-side code is responsible for choosing the URL, HTTP method, HTTP body of the request, etc. The browser maintains cookies that it receives from web servers. When a browser forwards a request from browser-side code to a web server, it also sends any cookies that it possesses for that web server.

A web server response to the browser's request includes, amongst other items, cookies and content in the HTTP body of the response. Depending on how the request is made, a browser might, e.g., return the results to browser-side code, add the response to the DOM, or create a fresh page. We focus on results returned to browser-side code.

Security mechanisms. For our purposes, there are two relevant security mechanisms. The first is the association of an origin with each page to implement the *same-origin* policy [35,7]. Web pages and their browser-side code can make requests for resources with various mechanisms, e.g., `img` elements, `script` elements, programmatic insertion of such elements into a page via the Document Object Model (DOM) from JavaScript, programmatic form submission from JavaScript, and use of the `XMLHttpRequest` object. Browsers place limitations on requests, and the visibility of responses, partly based on the page's origin [35].

The second mechanism is the use of cookies by web servers to indicate that a user (or some browser-side code) has authenticated successfully. Cookies are added even to cross-site requests, where a request is cross-site if the web page (or browser side code) is associated with a web server different from the target of the request. This permits cross-site requests to web servers where the user is authenticated.

Security concerns. The unconstrained addition of cookies to requests makes it challenging for web servers to determine the origin of a web page, or equivalently to detect cross-site requests, without further infrastructure [8]. It is in this sense that the browser is a Confused Deputy—the browser upgrades the authority of a request from a web page without understanding the contents of the request. There are two security concerns with being unable to identify cross-site requests:

1. A secret held on a web server may be returned to browser-side code with a different origin. Hence, there is potential for untrustworthy code to receive the secret.
2. The integrity of information on the web server may be harmed if state changes are based upon requests that are mistakenly thought to be from trustworthy code, but are instead cross-site requests from untrustworthy code.

(1) is primarily addressed by the browser security policy. (2) is a key ingredient for Cross-Site Request Forgery (CSRF) attacks—a number of mitigation strategies have been proposed, see, e.g., [8,30,33].

We now begin to develop an object-oriented model, with the intention of controlling the possession of secrets, such as passwords and cookies.

A.2 The Web Server Model

We first present a web server model with a very simple policy for releasing secret information and updating sensitive web server state. In Section A.3, we show that a more sophisticated policy is necessary to support a web browser that runs browser-side code in different protection domains.

The code for the web server model, and its supporting classes, is defined in Figure 3. The `Password`, `Cookie`, and `Secret` classes function as secrets, and have no methods or fields. They are simply unforgeable references.

HTTP requests and responses. In this model, client requests to a web server are distributed method calls, and the HTTP request is a parameter to the method. To focus on the novel components of policies for possession of object references, we make simplifying assumptions, e.g., transport is secure and reliable, there are no proxies, etc.

The concrete subclasses of the classes `Request` and `Response` model different requests and responses. We do not attempt to explicitly represent, e.g., the path component of a URL, HTTP method (and the semantics with respect to state), HTTP headers, HTTP response codes, content types, or parsing of message bodies, etc.

The `RequestLogin` class represents a request for a password login, `RequestRead` a request for a secret held on the server, and `RequestPasswd` a request to update the password used for logging in. For simplicity, we assume that each web server has just one user. The `ResponseLogin` class represents an HTTP response for a successful password login. The `cookie` field inherited from `Response` is expected to contain a reference to a `Cookie` that is used to authenticate subsequent requests. The `ResponseRead` and `ResponsePasswd` classes represent the results of successful requests to read the server's secret and to update the password respectively. We model failure of requests using the null reference, and terminate execution if null is dereferenced.

We omit constructors that initialize all final fields with arguments given to constructor of the same type. Immutable public fields are used for the Request and Response classes and their subclasses since they represent data rather than code, and because some fields are referenced in class invariants. Notably, we assume that classes are typechecked in the context of a global policy stating that instances of each subclass of Request and Response can be seen if their fields can be seen, e.g. for the ResponseRead class: $\forall \mathtt{o} : \mathtt{ResponseRead}.(\Box_{\bot}(\mathtt{mp}(\mathtt{o.cookie}) \wedge \mathtt{mp}(\mathtt{o.secret}) \Rightarrow \mathtt{mp}(\mathtt{o})))$.

Web server state. To support password updates, the state of the web server must be mutable. However, we disallow references to mutable fields in logical assertions because they may be invalidated by assignment. For this reason, the state of the web server is held in an instance of another class WebServerState, with immutable fields that may be used in logical assertions. The WebServer has a mutable field state containing a reference to an instance of WebServerState. To update the state, WebServer creates a fresh instance of WebServerState, with the obligation to establish the new class invariants for the fresh instance, and assigns the new instance to the mutable state field (line 38 of Figure 3). This yields a (non-atomic) swap of the state used by WebServer. In order to use the assertions on the data in WebServerState, the object reference in the mutable state field must be copied to a final local variable (line 24 of Figure 3). The assertions from WebServerState are available for typechecking purposes whilst the final local variable is in scope.

The first invariant of the WebServerState class states that the cookie can be held if the password can be held:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{this.password}) \Rightarrow \mathtt{mp}(\mathtt{this.cookie}))$$

and the second invariant states that the secret can be held if the cookie can be held:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{this.cookie}) \Rightarrow \mathtt{mp}(\mathtt{this.secret}))$$

We now examine the use of these invariants in the web server. When the process method is invoked with a Cookie (possibly null) and a Request, the request type is examined using instanceof (lines 25, 30, and 34) and then downcast to the more precise type (lines 26 and 35). Each request type is considered individually below.

Typing the login request. In the case of a login request, it is assumed from the method invocation that $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{request})$, i.e., the caller may possess the request. Note that this holds for both the method parameter request and the local variable request shadowing the method parameter, because downcasting introduces an equality between the two. For this reason, we do not distinguish between the method parameter and the local variable. Next, since password is a public and final field of RequestLogin, we have:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{request}) \Rightarrow \mathtt{mp}(\mathtt{request.password}))$$

In conjunction with $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{request})$, we deduce that $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{request.password})$.

The equality test on line 27 of Figure 3 establishes that $(\mathtt{request.password} = \mathtt{state.password})$, and it follows that $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{state.password})$.

```
class Password {}   class Cookie {}   class Secret {}

abstract class Request                {}
class RequestLogin  extends Request   { public final Password password; }
class RequestRead   extends Request   {}
class RequestPasswd extends Request   {}

abstract class Response               { public final Cookie cookie; }
class ResponseLogin  extends Response {}
class ResponseRead   extends Response { public final Secret secret; }
class ResponsePasswd extends Response { public final Password passwordNew; }

class WebServerState {
  public final Password password;
  public final Cookie cookie;
  public final Secret secret;
}[   $\Box_{\bot}(\mathtt{mp}(\mathtt{this.password}) \Rightarrow \mathtt{mp}(\mathtt{this.cookie}))$
   $\wedge \Box_{\bot}(\mathtt{mp}(\mathtt{this.cookie}) \Rightarrow \mathtt{mp}(\mathtt{this.secret}))$
   $\wedge \Box_{\bot}(\mathtt{mp}(\mathtt{this}) \Rightarrow (\mathtt{mp}(\mathtt{this.password}) \wedge \mathtt{mp}(\mathtt{this.cookie}) \wedge \mathtt{mp}(\mathtt{this.secret})))$   ]

class WebServer {
  private WebServerState state;
  Response process (Cookie cookie, Request request) {
    WebServerState state = this.state;
    if (request instanceof RequestLogin) {
      RequestLogin request = (RequestLogin) request;
      if (request.password == state.password) {
        return new ResponseLogin (state.cookie); // Password accepted.
      }
    } else if (request instanceof RequestRead) {
      if (cookie == state.cookie) {
        return new ResponseRead (null, state.secret); // Cookie accepted
      }
    } else if (request instanceof RequestPasswd) {
      RequestPasswd request = (RequestPasswd) request;
      if (cookie == state.cookie) {
        // Cookie accepted, update state with new password.
        this.state = new WebServerState (request.passwordNew, state.cookie, state.secret);
        return new ResponsePasswd (null);
      }
    }
    return null; // The request failed.
  }
}
```

Fig. 3. Web Server Code

The first component of the `WebServerState` class invariant yields:

$$\Box_{\perp}(\mathtt{mp}(\mathtt{state.password}) \Rightarrow \mathtt{mp}(\mathtt{state.cookie}))$$

Hence $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{state.cookie})$. Now, the omitted class invariant for the `ResponseLogin` class is:

$$\Box_{\perp}(\mathtt{mp}(\mathtt{this.cookie}) \Rightarrow \mathtt{mp}(\mathtt{this}))$$

This justifies returning the result of the constructor invocation `new ResponseLogin (state.cookie)` on line 28, because $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{state.cookie})$, and completes the discussion of typechecking for the `RequestLogin` case.

Typing the read secret request. The handling of `RequestRead`, starting at line 30 of Figure 3, is similar to that of `RequestLogin`, except that the justification for returning the secret originates from $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{cookie})$. rather than $\Box_{\mathsf{caller}}\mathtt{mp}(\mathtt{request.password})$. A `null` value is returned instead of a cookie to indicate that the client should not change the cookie that it has stored for this web server (the actual cookie could be returned without affecting the system's behavior significantly).

But typechecking the password update request fails! The `RequestPasswd` handler starts at line 34. The request is authenticated by checking the cookie as with `RequestPasswd`. However, this test does not convey useful static information about the new password in the request.

In particular, the creation of the new instance of `WebServerState` on line 38 fails to typecheck, because the first conjunct of the class invariant cannot be proved for the arguments to the constructor:

$$\Box_{\perp}(\mathtt{mp}(\mathtt{request.passwordNew}) \Rightarrow \mathtt{mp}(\mathtt{state.cookie}))$$

If we ignore this typing failure, the integrity of the web server state can be violated. A subsequent request to read the secret stored at the web server can also be unjustified, because the reasoning above relies on the class invariant for the web server state.

This scenario is precisely that of a CSRF attack, where malicious cross-site code uses an authenticated browser session to update a user's password on a web server without having to steal the original password. The web server can then be accessed using the new password.

Our simple model identifies the integrity failure from a CSRF attack as a typechecking error. In the next section we show how a typical CSRF solution typechecks.

A.3 The Web Browser Model

We now consider the interactions of the browser (the Deputy) with browser-side code and a web server. In particular, we examine the policy justification demanded by our type system for the browser to forward requests and responses between browser-side code and a web server, including code and invariant changes used to avoid a CSRF attack.

First, objects representing the web browser and browser-side code must be placed at separate locations. Additionally, different pieces of browser-side code with different origins must be placed at separate locations. In this setting, the web browser and browser-side code can have different policies for possession of secrets, reflecting the use of separate protection domains in an implementation.

Forwarding requests. We model a `Browser` class using code of the form (the logical formula preceding the method declaration is a precondition described below):

```
class Browser {
  [ $\Box_{ws.loc}$mp(request) ]
  Response process (WebServer ws, Request request) {
    Cookie cookie = ... // Retrieve cookie for ws.
    Response response = ws.process (cookie, request);
    ...
  }
}
```

The `process` method is invoked by browser-side code to request a resource from a web server. This `process` method is a simple abstraction of the different methods available to scripts to cause HTTP requests. The browser code above adds a cookie when forwarding a request to the web server.

In typechecking the body of this method, the first step is to ensure that the target web server `ws` may possess both the cookie and the request.

Forwarding of the cookie reference to the web server must be justified by the data structure from which the cookie is retrieved. A linked list of pairs of `Cookie` and `WebServer` references suffices, where each pair states that the cookie can be sent to the web server, i.e., the class invariant for the pair class is $\Box_{\texttt{this.ws.loc}}\mathrm{mp}(\texttt{this.cookie})$. Cookies returned in responses from a web server can be paired with the web server reference, whilst satisfying the same invariant, simply because the server must have possessed the cookie in order to return it.

To justify sending the request object reference to `ws`, the `process` method requires the precondition $\Box_{\texttt{ws.loc}}\mathrm{mp}(\texttt{request})$ to be satisfied by the caller. In context, the precondition means that browser-side code cannot ask a browser to send requests with secrets to web servers that should not possess those secrets.

Such a precondition relating arguments for a method can be reduced to a class invariant by first introducing a wrapper class:

```
class PreBrowser {
  public final WebServer ws;
  public final Request request;
}[ $\Box_{this.ws.loc}$mp(this.request) ]
```

In this reduction, the `process` method is altered to accept a single parameter of type `PreBrowser` instead of the two original `ws` and `request` parameters. The class invariant may then be assumed in the body of the `process` method.

The browser should store `response.cookie` from any `response` received from `ws` with the invariant discussed above for retrieved cookies.

Returning `response` to the web page that invoked the browser `process` is more interesting. Previously, `WebServer` provided responses because the caller (such as `Browser`) was known to possess a password or a cookie, and was thus entitled to the response. In order to pass the response to others, i.e., back to the requesting page, we adopt a new policy for `WebServer`'s `process` method that is in essence:

$$\Box_{\perp}(\mathtt{mp}(\mathtt{cookie}) \wedge \mathtt{mp}(\mathtt{request}) \Rightarrow \mathtt{mp}(\mathtt{result}))$$

That is, any principal may possess the result if they may possess the `cookie` and `request` arguments. This policy is interesting because, in general, the right to possess the arguments to a method does not convey the right to possess the results of the method.

The policy is a postcondition for the `process` method that refers to the (implicitly bound) `result`. This postcondition can also be reduced to a class invariant on a wrapper class, by introducing a `PostWebServer` class that returns the arguments to `process` paired with the original result:

```
class PreWebServer {
  public final Cookie cookie;
  public final Request request;
}
class PostWebServer {
  public final PreWebServer pre;
  public final Response result;
}[ □⊥(mp(this.pre) ⇒ mp(this.result)) ]
```

As with the `Request` and `Response` subclasses, we omit the policies that these classes may be possessed if all of their fields may be possessed. In order to gain the postcondition in the caller's code after the return from the `WebServer` `process` method, the actual parameter to `process` must be compared with the `pre` field.

With this policy for a `result`, the browser need only determine whether the web page that invoked the `process` method may possess `cookie`, because the web page sent the `request` and may therefore possess it. Policies for possession of cookies are naturally based on associating an origin with a cookie and comparing it with the origin of a web page (this reflects the usual policy for allowing JavaScript to access cookies). Then the `Browser` `process` method can compare the origin of a cookie and a requesting web page. If they match, the `result` may be returned to the web page. Otherwise, returning a non-trivial `result` cannot be justified (this is also in line with the usual browser behavior that prevents responses from being made available to pages with different origins).

Forwarding responses. Returning the web server response to the browser-side code via the browser's `process` method is more interesting. In Section A.2, the web server justified returning responses with secrets by deducing that the caller was known to possess a password or a cookie, and was thus entitled to the response. This is possible, because the web server has information about the content and policy for each request type. However, a browser does not examine, or understand the invariants of, the contents of requests and responses that it forwards. Consequently, the browser is not able to deduce that a response can be returned to the caller with the policy seen so far, and code that attempts to do so will not typecheck. We proceed by extending the web server's interface with invariants that allow the browser to forward the response.

A natural first attempt is to say that, if a principal may possess the request to the browser, then they may also possess the response (here it is important that the `Request` subclasses, representing the body of an HTTP request, do not include the cookie, which is sent in HTTP headers). This would permit the browser to return the response immediately. However, this policy is too permissive: the web server's handling of requests to read the secret cannot establish the relationship between `RequestRead` and `ResponseRead` references, because the latter possesses a secret and the former does not. The problem eventually leads to a failure to typecheck the web server with this permissive policy.

In Section A.2, the handler instead established that the immediate caller may possess `ResponseRead` based upon the caller's possession of the cookie argument instead of the request argument. As a second attempt, we might say that a principal may possess the response if they may possess the cookie. This policy too is problematic for the web server when the request is secret but the cookie is not, e.g., when a password login takes place before a non-null cookie is established.

The solution is to combine the two approaches, so a principal may possess the result if they may possess both the cookie and request arguments. We state this as a postcondition for the web server's `process` method. The postcondition is written after the method, and refers to both the arguments and the (implicitly bound) result of the method.

```
class WebServer {
  private WebServerState state;
  Response process (Cookie cookie, Request request) {
    ...
  }[ □⊥(mp(cookie) ∧ mp(request) ⇒ mp(result)) ]
}
```

The first two request handling cases of the web server of Section A.2 typecheck with this postcondition, because the implication can be established. In the `RequestPasswd` case, the hypothesis for possession of the request is used. In the `RequestRead` case, the hypothesis for possession of the cookie is used. The case for handling password updates requires additional infrastructure described with CSRF solutions below.

Finally, the browser need only determine whether the browser-side code that invoked the browser's `process` method may possess the cookie, because the browser-side code sent the request originally and may therefore possess it. Policies for possession of cookies are naturally based on associating an origin with a cookie and comparing it with the origin of browser-side code (this reflects the usual policy for allowing JavaScript to access cookies). Thus the browser `process` method can compare the origin of a cookie and a requesting web page. If they match, the result may be returned to the web page. Otherwise, the browser cannot justify returning the result to the browser-side code with a different origin. This sketch yields browser code that typechecks.

The origin test in typechecked browser code provides justification for the standard browser security policy that makes HTTP responses generally unavailable to pages with different origins. We leave an exploration of models for notable exceptions such as JSONP and cross-origin sharing for future work.

Cross-Site Request Forgery and web server integrity. The failure of typechecking for password update handling seen in Section A.2 is due to the lack of static information about the new password. For example, the web server cannot assume that a principal that may possess the new password may possess the cookie argument added by the browser. If this assumption is made, it is possible to typecheck the web server in its entirety. However, this assumption cannot be established by a browser that permits cross-site requests freely, and thus it would not be possible to typecheck such a browser.

A typical solution to CSRF is to require that certain requests contain an additional secret, known only to trusted browser-side code. This secret serves to authenticate the browser-side code that initiates the request, rather than the browser that is forwarding the request. We model this solution by revising the definition of `RequestPasswd` class to include, e.g., the original password, in addition to the new password:

```
class RequestPasswd extends Request   {
  public final Password password;
  public final Password passwordNew;
}[ □⊥(mp(this.passwordNew) ⇒ mp(this.password)) ]
```

The class invariant states that a principal may possess the original password if they may possess the new passsword, i.e., fewer principals know the new password. Then the `RequestPasswd` handling code from the `process` method of the `WebServer` class is also revised to verify the old password:

```
 else if (request instanceof RequestPasswd) {
   RequestPasswd request = (RequestPasswd) request;
   if (cookie == state.cookie && request.password == state.password) { ... }
```

Now, from the `WebServerState` class invariant for local variable `state`, we find:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{state.password}) \Rightarrow \mathtt{mp}(\mathtt{state.cookie}))$$

Also, from the new `RequestPasswd` class invariant for the local variable `request`:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{request.passwordNew}) \Rightarrow \mathtt{mp}(\mathtt{request.password}))$$

And the equality test between the old passwords in the web server state and the request yields $(\mathtt{request.password} = \mathtt{state.password})$, hence:

$$\Box_{\bot}(\mathtt{mp}(\mathtt{request.passwordNew}) \Rightarrow \mathtt{mp}(\mathtt{state.cookie}))$$

This provides the necessary justification for the creation of a new instance of `WebServerState` by satisfying the class invariant, where `request.passwordNew` replaces `state.password`, and the `cookie` and `secret` fields are copied from the original value of `state` to the new instance. Writing the new instance of `WebServerState` to the mutable field `state` does not require any further justification for the purposes of typechecking. Therefore, the revised web server typechecks with the changes made to requests—and those changes are based on typical anti-CSRF measures, together with a careful choice of invariants.

B Standalone Web Client

Cookies are intended for web browsers [6], and web services intended for non-browser web clients may use other authentication schemes, e.g., HTTP Digest or TLS-based client-side certificate authentication. However, we first consider a standalone web client that uses cookies as a stepping stone to our discussion of browser behavior below. A standalone web client represents a single location, and so policies need not state how references received by the client can be shared with any location other than the web server. The client and web server locations are distinct in general.

Initially, we assert that the client has a `password` field, and can obtain a `passwordNew` satisfying the policy $\Box_{\perp}(\mathtt{mp}(\mathtt{this.passwordNew}) \Rightarrow \mathtt{mp}(\mathtt{this.password}))$. The `passwordNew` field could be a fresh password that is created with the policy above. Additionally, the client code must assert the web server is able to possess `passwordNew` (and hence `password` by the first policy). If the web server is referenced through a field `ws` of the client, this policy is $\Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{this.passwordNew})$.

To login, the client executes:

```
Request req = new RequestLogin (null, this.password);
Response resp = this.ws.process (req);
```

The transmission of `req` to `this.ws` is justified by using the policies above to first deduce that $\Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{req.password})$, and then conclude $\Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{req})$.

Justification for sending other requests is similar. The subtlety lies in the justification for sending a `Cookie` reference received in one response back to the web server in a subsequent response. The simplest argument is that $\Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{resp})$, since `resp` was returned from a method call to `this.ws`. In addition, `cookie` is a public, final field of `ResponseLogin`, so $\Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{resp.cookie})$, as required. However, this approach does not allow `resp.cookie` to be distributed to any principal other than `this.ws.loc` and `this.loc`. We present an alternative policy in Section A.3.

The following code represents a sample web client.

```
class Client {
  private final WebServer ws;
  private final Password password;
  private final Password passwordNew;

  void run () {
    // Must be able to send request with password.
    Request request1 =
      new RequestLogin (null, this.password);
    ResponseLogin response1 =
      (ResponseLogin) this.ws.process (request1);

    // Must be able to send a request with the cookie.
    Request request2 =
      new RequestRead (response1.cookie);
    ResponseRead response2 =
      (ResponseRead) this.ws.process (request2);

    // Must be able to send a request with the
    // original and new password.
    Request request3 = new RequestPasswd
      (response1.cookie, this.password, this.passwordNew);
    ResponsePasswd response3 =
      (ResponsePasswd) this.ws.process (request3);
  }
}[
```
$\Box_{\perp}(\mathtt{mp}(\mathtt{this.passwordNew}) \Rightarrow \mathtt{mp}(\mathtt{this.password}))$

$\wedge\ \Box_{\mathtt{this.ws.loc}}\mathtt{mp}(\mathtt{this.passwordNew})$

```
]
```

C Auxiliary definitions

C.1 Operational semantics

Fix a global class table $\vec{\mathscr{D}}$. The fields and method lookup functions are standard.

Field Lookup $(\mathit{fields}(C) = \vec{\mu}\ \vec{T}\ \vec{f})$

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \emptyset} \qquad \frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}\texttt{>} \triangleleft D\{\vec{\mu}\ \vec{T}\ \vec{f};\ \cdots\} \quad \mathit{fields}(D\{\![\vec{\phi}/\vec{\alpha}]\!\}) = \vec{\mu}_D\ \vec{T}_D\ \vec{f}_D}{\mathit{fields}(c\texttt{<}\vec{\phi}\texttt{>}) = \vec{\mu}_D\ \vec{T}_D\ \vec{f}_D, (\vec{\mu}\ \vec{T}\ \vec{f})\{\![\vec{\phi}/\vec{\alpha}]\!\}}$$

Method Lookup $(\mathit{body}(C.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\ \vec{x}\texttt{)}\,\{M\})$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}:\vec{P}\texttt{>} \triangleleft D\{\cdots \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\ \ell\,\texttt{(}\vec{T}\ \vec{x}\texttt{)}\,\{M\}\cdots\}}{\mathit{body}(c\texttt{<}\vec{\phi}\texttt{>}.\ell) = (\texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\ \vec{x}\texttt{)}\,\{M\})\{\![\vec{\phi}/\vec{\alpha}]\!\}}$$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}:\vec{P}\texttt{>} \triangleleft D\{\cdots \vec{\mathscr{M}}\} \quad \ell \text{ not defined in } \vec{\mathscr{M}} \quad \mathit{body}(D\{\![\vec{\phi}/\vec{\alpha}]\!\}.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\ \vec{x}\texttt{)}\,\{M\}}{\mathit{body}(c\texttt{<}\vec{\phi}\texttt{>}.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\ \vec{x}\texttt{)}\,\{M\}}$$

C.2 Typing

Subtyping $(\vdash \mathscr{T}' <: \mathscr{T})$

$$\frac{}{\vdash \mathscr{T} <: \mathscr{T}} \qquad \frac{\vdash \mathscr{T}' <: \mathscr{T}'' \quad \vdash \mathscr{T}'' <: \mathscr{T}}{\vdash \mathscr{T}' <: \mathscr{T}}$$

$$\frac{}{\vdash C <: \mathsf{Object}\texttt{<}\emptyset\texttt{>}} \qquad \frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}\texttt{>} \triangleleft D}{\vdash c\texttt{<}\vec{\phi}\texttt{>} <: D\{\![\vec{\phi}/\vec{\alpha}]\!\}}$$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}\texttt{>} \quad \vec{\phi} \vDash \vec{\psi} \quad |\vec{\alpha}| = |\vec{\phi}| = |\vec{\psi}|}{\vdash c\texttt{<}\vec{\phi}\texttt{>} <: c\texttt{<}\vec{\psi}\texttt{>}}$$

Well-Formed Overriding $(\vdash \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\texttt{)}\ \mathit{overrides}\ D.\ell)$

$$\frac{\mathit{body}(D.\ell) \text{ not defined}}{\vdash \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\texttt{)}\ \mathit{overrides}\ D.\ell} \qquad \frac{\mathit{body}(D.\ell) = \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S\,\texttt{(}\vec{T}\texttt{)} \quad \vdash S' <: S}{\vdash \texttt{<}\vec{\beta}:\vec{Q}\texttt{>}S'\,\texttt{(}\vec{T}\texttt{)}\ \mathit{overrides}\ D.\ell}$$

Well-Formed Types $(\Delta \vdash \mathscr{T})$

$$\frac{}{\Delta \vdash \mathsf{Unit}} \qquad \frac{}{\Delta \vdash \mathsf{Prin}} \qquad \frac{}{\Delta \vdash \mathsf{Object}\texttt{<}\emptyset\texttt{>}} \qquad \frac{\Delta \vdash \vec{\mathscr{T}}}{\Delta \vdash \mathsf{Pred}\,\texttt{(}\vec{\mathscr{T}}\texttt{)}}$$

$$\frac{\vec{\mathscr{D}} \ni \mathsf{class}\ c\texttt{<}\vec{\alpha}:\vec{P}\texttt{>} \quad a \in \mathit{dom}(\Delta) \quad \Delta \vDash_a \vec{\phi} : \vec{P}}{\Delta \vdash c\texttt{<}\vec{\phi}\texttt{>}}$$

Well-Formed Environments $(\Delta \vdash \diamond)$

$$\frac{\begin{array}{l}
\Delta \ni a : \mathsf{Prin} \\
\Delta \ni x : \mathscr{T} \text{ implies } \mathscr{T} = \mathsf{Pred} \text{ or } (\exists T)\ \mathscr{T} = T \text{ and } \Delta \vdash T \\
\Delta \ni p : \mathscr{T} \text{ implies } (\exists C)\ \mathscr{T} = C \text{ and } \Delta \vdash C \\
\Delta \ni b : \mathscr{T} \text{ implies } \mathscr{T} = \mathsf{Prin} \\
\Delta \ni \alpha : \mathscr{T} \text{ implies } (\exists P)\ \mathscr{T} = P \text{ and } \Delta \vdash P \\
\Delta \ni V{=}M \text{ implies } (\exists a, c, T)\ \Delta \models_{a} V : T \text{ and } \Delta \models^{a}_{a} M : T\ \mathsf{Pure}\ c \\
\text{each element in } dom(\Delta) \text{ appears exactly once}
\end{array}}{\Delta \vdash \diamond}$$

Defining right. A term M has *no gratuitous frames* if whenever $a\,[M']^{a'}_{c}$ occurs as a subterm of M, then this subterm is M itself, or it occurs in an immediate context of the form $\mathsf{let}\ x = [-];\ N$. That is, all frames of M are either (the one) top-level or in subterms of the form $\mathsf{let}\ x = a\,[M']^{a'}_{c};\ N$. A term is *simple* if it has no gratuitous frames and it contains no subterms containing $\Vert\!\ulcorner$ or ν.

For any term M, there exists a unique simple N such that $M \equiv (\nu\vec{p}:\vec{C})\ M' \Vert\!\ulcorner N$. Define the function $right(M)$ to return this N.